Empirical Software Engineering (2019) 24:2056–2101
https://doi.org/10.1007/s10664-019-09689-7

The Android OS stack and its vulnerabilities: an empirical study

Alejandro Mazuera-Rozo[1,2] · Jairo Bautista-Mora[1] · Mario Linares-Vásquez[1] · Sandra Rueda[1] · Gabriele Bavota[2]

Published online: 20 February 2019

Abstract
The wide and rapid adoption of Android-based devices in the last years has motivated the usage of Android apps to support a broad range of daily activities. In that sense, being the most popular mobile platform makes it an attractive target for security attacks. In fact, 1,489 security vulnerabilities have been reported in the last three years (2015-2017) for the Android OS (which is the underlying platform for Android-based devices). While there is a plethora of approaches and tools for detecting malware and security issues in Android apps, few research has been done to identify, categorize, or detect vulnerabilities in the Android OS. In this paper we present the largest study so far aimed at analyzing software vulnerabilities in the Android OS. In particular, we analyzed a total of 1,235 vulnerabilities from four different perspectives: vulnerability types and their evolution, CVSS vectors that describe the vulnerabilities, impacted Android OS layers, and their survivability across the Android OS history. Based on our findings, we propose a list of future actions that could be performed by researchers and practitioners to reduce the number of vulnerabilities in the Android OS as well as their impact and survivability.

Keywords Vulnerabilities · Android · Empirical study · Operating system

Communicated by: Lin Tan

✉ Mario Linares-Vásquez
m.linaresv@uniandes.edu.co

Alejandro Mazuera-Rozo
a.mazuera@uniandes.edu.co

Jairo Bautista-Mora
je.bautista10@uniandes.edu.co

Sandra Rueda
sarueda@uniandes.edu.co

Gabriele Bavota
gabriele.bavota@usi.ch

[1] Universidad de los Andes, Bogotá, Colombia

[2] Università della Svizzera italiana, Lugano, Switzerland

1 Introduction

In the last few years, mobile apps have powered a whole new economy that substantially impacted the software market. The cultural popularity of mobile devices, the new monetization/revenue models the apps' market propose, and the capillary distribution infrastructure represented by app stores, are only some of the driving factors making apps an attractive market for software developers. Also, the need for "enterprise apps" that support startups or serve as a new front-end for traditional companies is pushing software-related professionals to embrace mobile technologies (VisionMobile 2014).

From the users' perspective, mobile apps and devices are a mechanism for achieving ubiquity, allowing them to perform multiple tasks and daily activities from anywhere, and to always have available at the touch of their hands important/sensitive information. Consequently, the security of mobile apps and of the underlying platforms on which they run has become a big concern for researchers and practitioners, due to the impact that security issues affecting mobile platforms might have on the private life of individuals (e.g., allowing to stole private files) as well as on companies (e.g., allowing to intercept strategic business decisions) (LLC 2014; Stefanko 2015; Anderson et al. 2016).

Recently, the impact of those vulnerabilities in everyday life has been more evident to society due to public announcements of malware and vulnerabilities in mobile platforms that compromise sensitive information and/or computational resources in the affected devices. In 2015 mobile malware reached a tremendous +153% (Android) and +235% (iOS) in the number of reported threats as compared to the previous year (Anderson et al. 2016). Representative examples of mobile malware with notorious impact are games such as "Cowboy adventure" and "Jump chess" that infected about one million devices (Beres 2015), the Locker trojan (Stefanko 2015; Anderson et al. 2016) for Android, and the XcodeGhost malware that infected 40k+ apps from the Apple App Store (Anderson et al. 2016). Also, according to the CVE details portal (MITRE 2017q), 1,489 vulnerabilities in the Android OS were reported between 2015 and 2017. One of those famous vulnerabilities is "Stagefright" (Burgess 2016; Nickinson 2015; Wikipedia 2017c) that affected 95% of the Android devices in 2015.

As a contribution from the research community, substantial effort has been recently invested in the analysis and detection of malware and vulnerabilities at the applications level (*e.g.,* Bagheri et al. 2015; Ahmad et al. 2016; Sadeghi et al. 2015, 2016, 2017; Garcia et al. 2017). However, (i) few works have focused on the vulnerabilities at the OS level (Thomas 2015a, b; Jimenez et al. 2016; Bagheri et al. In Press; Cao et al. 2015), the underlying platform on which any app runs, and (ii) most of the studies have focused on a limited number of vulnerabilities.

In this paper, we present an empirical study aimed at analyzing Android-related vulnerabilities from different perspectives and with a specific focus on those affecting the Android Operating System (Android OS). In particular, we study (i) the types of vulnerability and their evolution over time, (ii) the most common security scoring vectors in the vulnerabilities, (iii) the layers and subsystems from the Android OS affected by vulnerabilities, and (iv) the survivability of vulnerabilities (i.e., the number of days between the vulnerability introduction and its fixing). While previous studies have focused the attention on a small set of vulnerabilities (e.g., 11 in Thomas et al. (2015b), 1 in Thomas (2015a), and 32 in Jimenez et al. 2016), we mined all the vulnerabilities (1,235) available in the official Android bulletins and the CVE-details portal up to August 2017. The vast majority of our study has been carried out via manual analysis of vulnerability-related documents available on issue

trackers, versioning system, official Android security bulletins, and information available on the National Vulnerability Database (NIST 2017).

Knowing the types of security vulnerabilities affecting the Android devices and their characteristics can help to guide (i) apps developers, in focusing their verification & validation activities toward the identification of the most frequently reported types of vulnerability, (ii) researchers, in investing in vulnerabilities detection tools targeting the most diffused types of Android-related vulnerabilities (thus being particularly valuable to increase the security of Android devices), and (iii) language/API developers, to design/improve mechanisms for secure coding of Android apps and the underlying platform.

As a result of our study we defined a detailed taxonomy of OS-related vulnerabilities affecting Android devices (Figs. 3 and 4) based on the Common Weaknesses Enumeration (MITRE 2017p), and identified the most impacted layers/subsystems of the Android OS (Fig. 11). We found that most of the vulnerabilities, while not requiring a complex attack to be exploited, have a severe impact on the device's confidentiality, integrity, and availability; most of the vulnerabilities (69,97%) are exploitable remotely and do not require authentication on the system. The hardware drivers in the lowest level of the Android OS stack (i.e., Linux kernel) and the native libraries are the layers mostly impacted by security vulnerabilities, and the lack of secure coding practices for restricting operations in the bounds of memory buffers is the main source of vulnerabilities. In addition, we found that Android-related security vulnerabilities survive for very long time (at least 770 days, on average). Finally, we observed an increasing trend of the vulnerabilities affecting the Android OS over time.

This paper is an extension of a previous paper published at the 2017 Conference on Mining Software Repositories (Linares-Vásquez et al. 2017). The extension includes: (i) a larger set of vulnerabilities (1,235 *vs* 660), which accounts for 87,12% of additional data; (ii) additional analyses aimed at identifying the evolution of vulnerabilities since 2008 and their characteristics as reported in the Common Vulnerability Scoring System (CVSS) vectors; (iii) an analysis of the vulnerability patches to understand the most common fixes; (iv) additional material in the background and related work section; (v) more vulnerabilities examples discussed qualitatively; and (vi) an extensive online appendix that includes the complete dataset.

Structure of the paper We provide some background on the Android OS security vulnerabilities and discuss the related literature in Section 2. Section 3 presents the design of our study, while our findings are discussed in Section 4. Section 5 lists the threats that could affect the validity of our results. Finally, Section 6 outlines the learned lessons and future work.

2 Background and Related Work

The wide and rapid adoption of Android-based devices in the last years has motivated the usage of Android apps to support a broad range of daily activities. In that sense, being the most popular mobile platform makes it an attractive target for security attacks (Huang et al. 2015). In fact, the number and complexity of the attacks to Android-based systems is increasing drastically (Huang et al. 2015). Since 2008, Android-related vulnerabilities have been reported including critical issues such as the "Heartbleed" (Wikipedia 2017b) flaw in the SSL library, the "Stagefright" flaw in the media framework (Wikipedia 2017c;

Nickinson 2015; Burgess 2016) that has infected 95% of the Android devices in 2015, and the "BlueBorne" vulnerability that allows attackers to remotely penetrate and control devices via bluetooth (Armis 2017). As a consequence, the industry has improved the security mechanisms and services in the Android ecosystem (Google 2016) and designed mobile-specific malware detectors.

Researchers have also contributed to improve the security of the Android ecosystem by analyzing security vulnerabilities and proposing improvements to current security models (Bagheri et al. In Press; Huang et al. 2015; Xu et al. 2016; Cao et al. 2015; You et al. 2016; Wang et al. 2016; Wu et al. 2013; Ahmad et al. 2016; Zhou and Jiang 2012b; Sufatrio Tan et al. 2015; Sadeghi et al. 2016; Fahl et al. 2012; Zuo et al. 2015; Backes et al. 2016; Kantola et al. 2012). However, while the focus of the academic research has been the security of the applications—the closest component to the user—, the core of the Android ecosystem (i.e., the Android OS) has received little attention, and it is the focus of our work.

In the following subsections we briefly discuss previous work related to vulnerabilities in Android applications and the Android OS. Also, we contextualize the reader with background concepts and information that are instrumental for our work.

2.1 Background Concepts

A vulnerability is typically defined as a weakness in procedures or system configurations that could lead to a failure of confidentiality, integrity or availability (U.S. National Institute of Standards and Technology - NIST 2012b; for Standardization 2011; Mell et al. 2007). Organizations like MITRE and the US National Institute of Standards and Technology (NIST) have created databases of known vulnerabilities: The Common Vulnerabilities and Exposures (CVE) dataset (Corporation 2017) and the NIST National Vulnerabilities Database (NVD) (U.S. National Institute of Standards and Technology - NIST 2012a) are the result of these efforts.

Both CVE and NVD use a standard set of attributes to describe vulnerabilities. This includes: a unique identifier (*e.g.,* CVE-2016-2439), names of the affected product (as defined by the Common Platform Enumeration Dictionary), product's vendor, and vulnerability features (as defined by the Common Vulnerability Scoring System FIRST Organization 2019) such as the confidentiality, integrity, and availability impact of the vulnerability.

The Common Vulnerability Scoring System (CVSS) uses three groups of attributes describing a vulnerability: Base, Temporal and Environmental (see Table 1). The Base group represents the "intrinsic and fundamental characteristics that are constant over time and user environments" (Mell et al. 2007). This includes, for example, how complex it is to exploit the vulnerability. The Temporal group represents the characteristics "that change over time but not among user environments" (e.g., whether a patch exists for the vulnerability) (Mell et al. 2007). Finally, the Environmental group represents the characteristics "that are unique to a particular user's environment" (e.g., the target distribution of the vulnerability) (Mell et al. 2007). Two versions of CVSS are currently used by the vulnerabilities databases: CVSS 2.0 and 3.0. While CVSS 3.0 is the most recent, it is not available for vulnerabilities discovered before 2016. For this reason, in our study we use the CVSS 2.0 version, including the attributes in Table 1[1]. Note that each group of attributes in CVSS is assigned with (i) a numeric score representing how critical the vulnerability is, and (ii) a vector

[1]We used the base group attributes because it is the only mandatory group.

Table 1 CVSS 2.0 Attributes

Base Group Attributes and Description:
Access vector (AV): represents the access means to exploit a vulnerability. Possible values are (L)ocal, (A)djacent network, (N)etwork
Access complexity (AC): represents the knowledge and resources required to exploit a vulnerability, after having gained access to a system. Possible values are (H)igh, (M)edium, (L)ow
Authentication (Au): reflects the number of times a malicious agent must authenticate in order to exploit a vulnerability. Possible values are (M)ultiple, (S)ingle, (N)one
Confidentiality Impact (C), Integrity Impact (I), Availability Impact (A): measure the adverse impact on confidentiality, integrity and availability respectively, if a vulnerability were successfully exploited.
Temporal group Attributes and Description:
Exploitability (E): measures the current state of exploit techniques or code availability. Possible values are (U)nproven, proof-of-concept (POC), (F)unctional, (H)igh, not defined (ND).
Remediation Level (RL): reflects the current stage to generate a patch that solves the vulnerability. Possible vales are official fix (OF), temporary fix (TF), (W)orkaround, (U)navailable, and not defined (ND).
Report Confidence (RC): measures the degree of confidence in the existence of the vulnerability. Possible values are unconfirmed (UC), uncorroborated (UR), (C)onfirmed, and not defined (ND).
Environmental group Attributes and Description:
Collateral Damage Potential (CDP) : reflects the potential of collateral losses, including lives, assets, and economic loss of productivity or revenue. Possible values are (N)one, (L)ow, low-medium (LM), medium-high (MH), (H)igh, and (N)ot (D)efined
Target Distribution (TD): approximates the percentage of systems that could be affected by the vulnerability. Possible values are (N)one, (L)ow, (M)edium, (H)igh, and (N)ot (D)efined
Security Requirements (CR, IR, AR): allow the analyst to weight the relevance of confidentiality, integrity and availability requirements respectively, for an organization. Possible values are (L)ow, (M)edium, (H)igh and not defined (ND).

reporting the features used to compute the score. In particular, the vector is a collection of *<attribute>:<value>* pairs (e.g., AC:H is a pair meaning Access Complexity:High).

2.2 Security in Android Applications

Android malware and vulnerabilities in Android apps are characterized by a novel set of flaws that exploit user level weaknesses and the issues in security mechanisms of the Android OS. For instance, Android-specific attacks include (i) privileges/permissions escalation through pairs of infected apps that exploit inter-application communication or mis-configured apps (Kantola et al. 2012; Sbîrlea et al. 2013; Sadeghi et al. 2015; Bagheri et al. 2015; Ahmad et al. 2016), (ii) applications tapjacking/hijacking by apps repackaging and substitution (You et al. 2016), (iii) information leaking through covert channels (Gasior and Yang 2012; Novak et al. 2015), (iv) SSL vulnerabilities in hybrid (Zuo et al. 2015) and native apps (Fahl et al. 2012), (v) activity/task hijacking (Ren et al. 2015; Lee et al. 2017), (vi) security issues introduced by third party libraries (Backes et al. 2016), and (vii) security issues introduced by OS customizations (Wu et al. 2013). These novel attacks, in addition to classic security attacks induced by malware (e.g., DoS), have been widely studied by the community at the application level, however few attention has been given to vulnerabilities in the Android OS.

Various approaches have been proposed for detection and mitigation of vulnerabilities in Android apps. Some works use static analysis, like JoDroid (Graf and Hecker 2015), Chex

(Lu et al. 2012), DidFail (Bhosale 2014), User-aware control project (Xiao et al. 2012), FlowDroid (Arzt et al. 2014), COVERT (Sadeghi et al. 2015; Bagheri et al. 2015), MudFlow (Avdiienko et al. 2015), Chabada (Gorla et al. 2014), Kirin (Enck et al. 2009) and Permission-based profiles (Morales and Rueda 2015). Other works use techniques based on dynamic behavior, like TaintDroid (Enck et al. 2010), Q-Floid (Castellanos et al. 2016), CopperDroid (Fattori et al. 2014), AppInspector (Gilbert et al. 2011), and Malware detection based on system calls (Dimjaševic et al. 2015; Park and Reeves 2013). There are also hybrid approaches, such as Andrubis (Weichselbaum et al. 2012).

Other resources, like the Android Malware Genome Project (Malgenome) (Zhou and Jiang 2012a), aim at characterizing Android malware families by describing installation methods, activation mechanisms, and malicious payloads. The Malgenome project includes 1,200 malware samples collected from August 2010 to October 2011.

Beyond the detection, research efforts have been also devoted to automatically generate exploits for specific vulnerabilities at the applications level and to generate test cases that are security features oriented. Representative examples are LetterBomb by Garcia et al. (2017) and PATDroid by Sadeghi et al. (2017). The former is an approach for the automatic generation of exploits, in particular for vulnerabilities in inter-component communication via intents. The latter (PATDroid) minimizes/prioritizes the combinations of permissions that should be granted/revoked with an existing test suite.

For more details on the existing techniques for detection of vulnerabilities in Android apps, we refer the interested reader to the works by Zhou and Jiang (2012b) and Sufatrio Tan et al. (2015) that widely describe Android malware and detection techniques, and a recent work by Sadeghi et al. (2016) presenting a survey of static analysis techniques for detecting Android malware.

In addition to malware and vulnerabilities, recent work cataloged a set of 28 code smells related to the security of Android apps. Security smells are sets of instructions in the source code that should be avoided by developers, since they likely indicate the presence of one or more vulnerabilities (Ghafari et al. 2017). The 28 smells reported by Ghafari et al. (2017) are categorized in five different categories: (i) insufficient attack protection, (ii) security invalidation, (iii) broken access control, (iv) sensitive data exposure, and (v) lax input validation.

The aforementioned papers analyze vulnerabilities at the application level, while our study focuses on vulnerabilities at the OS level.

2.3 Android OS Vulnerabilities

The Android OS is an open source mobile operative system developed by Google and based on the Linux Kernel. It is composed of a set of architectural layers that follows a software stack model (i.e., there is no communication between layers that are not neighbors in the stack), having the *Linux Kernel* as the foundation, and an *Applications* layer as the closest interaction point for the end users. Each layer is composed of subsystems/components mostly implemented in Java and C/C++.

Figure 1 depicts the Android OS software stack (including predominant implementation languages). Some of those components are developed by third-party contributors of the Android open source project (AOSP), such as original equipment manufacturers (OEM) and Linux contributors; for instance, the *Libraries* layer contains an Android's standard C library named Bionic, and the *Runtime* layer includes an independent Java Library implementation (Apache harmony or an OpenJDK-based implementation for newer OS versions), both developed by third-party contributors. For more details of the Android OS architecture

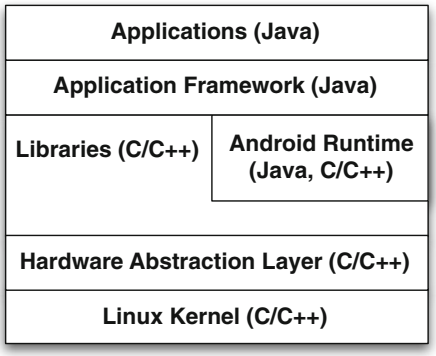


Fig. 1 Architectural layers in the Android OS software stack

we point the interested reader to the following sources: (Brady 2008; Google 2017b; Drake et al. 2014).

The Android OS stack is composed of the following layers:

- *Applications:* this layer contains software running on the device that uses the Android APIs to implement specific features, like geo-localization. The components in this layer are the mobile "apps" (mostly developed by Google) that are shipped with the Android OS and we use daily such as Browser, Calendar, Telephony, Messaging, and Settings; these apps are mostly written in Java and the Android API, although some apps can include native C/C++ code.
- *Android Framework:* provides apps (and developers) with the building blocks and common tasks required for exposing/using device- and Android-specific features such as managing UI elements and sensors. The Android Framework contains the Android APIs used by Android apps, and the Android managers (*a.k.a.,* services) that implement the services and features exposed by the APIs; examples of these services are the View System and the Activity Manager that are in charge of managing the different views and view groups available in the GUI, and of controlling applications life cycle and the activity/task stack, respectively. This layer is mostly implemented in Java.
- *Runtime:* contains the Virtual Machine (Dalvik/ART) and the core libraries required for the execution of apps and services on the device. The Runtime layer is required for ensuring apps portability across different devices. Examples of the core libraries in the Runtime layer are the independent implementation of Java used by Android (e.g., Apache Harmony) and the Bouncy castle library for cryptography.
- *Native Libraries:* provide low level functionalities and computational intensive services required by the Android Framework and the Runtime, such as the Bionic libc library, the WebKit browser engine, OpenGL, SSL, and the Media Framework. The libraries are written in C/C++.
- *Hardware Abstraction Layer (HAL):* it is the bridge between the high level representations of the hardware used in the libraries, and low level representations used by the kernel. It is a set of interfaces for hardware-specific software that needs to be implemented by OEMs and hardware manufacturers. Components in the HAL are written in C/C++.
- *Linux Kernel:* it provides the Android OS with core OS systems infrastructure, a security model, networking, and memory and process management, among others. Android uses a modified version of Linux tailored to mobile devices; this tailored version

includes changes/enhancements such as the Android Binder, Logger, ashmem, power management, wakelocks, and mechanisms for memory management (e.g., Android Low Memory Killer). More details of the Android-specific contributions added to the Linux kernel are described in (wiki. 2015).

A few previous studies focused on the analysis of specific components of the Android OS and their security issues. Bagheri et al. (In Press) analyzed the vulnerabilities of the permission system in the OS; Cao et al. (2015) analyzed input validation mechanisms in the services/managers of the *Android Framework*; Huang et al. (2015) found four vulnerabilities (*a.k.a.,* Android Stroke Vulnerabilities) in two services of the *Android Framework* (i.e., Activity and Window Manager) that can be used for DoS attacks and for inducing OS soft-rebooting; Wang et al. (2016) also analyzed the *Android Framework* layer and found six unknown vulnerabilities in three of its services (i.e., Activity Manager, Location Manager, Mount Service), and two apps from the *Applications* layer (i.e., SystemUI and Phone).

Closer to our study are the previous works aimed at analyzing security vulnerabilities by following a mining-based approach. Some of those studies are Android-specific (Thomas 2015a, b; Jimenez et al. 2016; Linares-Vásquez et al. 2017) while others are more general in the sense that they aim at characterizing security bugs (Zaman et al. 2011; Lal and Sureka 2012).

Thomas et al. (2015b) mined the OS updates installed on 20k+ Android devices to measure the delivery time of security updates for eleven vulnerabilities, and to establish a scoring model of insecure devices; the results suggest that, on average, 87.7% of the devices are exposed to at least one of the analyzed vulnerabilities. Thomas (2015a) investigated the Cve-2012-6636 (2017) vulnerability on the JavaScript-to-Java interface of the WebView API; 102k+ APKs were statically analyzed to measure the number of apps in which the vulnerability could be exploited. In addition, the lifetime of the vulnerability was analyzed using an approach similar to Thomas et al. (2015b).

Jimenez et al. (2016) analyzed 42 vulnerabilities from the CVE database (MITRE 2017r) to identify the issues, involved components, complexity of the patches, and complexity of the code methods/functions involved in the vulnerability.

Finally, Linares-Vásquez et al. (2017) analyzed a larger set (660) of Android OS vulnerabilities (mined from CVE) to analyze their survivability, the subsystems and components of the Android OS involved in the vulnerabilities, and defined a taxonomy of security issues based on the Common Weakness Enumeration (CWE) hierarchy of vulnerabilities (MITRE 2017p). As explained in Section 1, this paper extends the work in Linares-Vásquez et al. (2017).

3 Study Design

The *goal* of the study is to investigate security vulnerabilities reported over the past 10 years in the Android OS (*i.e.,* 2008 - 2017). The *purpose* is to (i) define a taxonomy highlighting the types of Android-related vulnerabilities, their evolution over time, the most common fixes applied by developers, and the Android OS subsystems more exposed to security issues; (ii) investigate what the most frequent attributes from the CVSS vectors are; and (iii) investigate the survivability of vulnerabilities in Android.

The *context* consists of 1,235 vulnerabilities mined from CVE Details (MITRE 2017r; 2017q), the NVD database (NIST 2017), and the official vulnerability bulletins released by

Google. All the data used in the study are available in our online appendix (Mazuera-Rozo et al. 2017).

This study addresses the following research questions:

RQ_1: *Which types of security vulnerabilities affect the Android OS?* This research question aims at identifying the types (*e.g.,* inadequate encryption strength) of Android-related vulnerabilities reported over the past ten years. Note that with "Android-related" we refer to both vulnerabilities directly affecting code components belonging to the Android OS, i.e., the components of the Android software stack developed by Google, as well as those components in the OS authored by third-party contributors (e.g., hardware drivers, apps shipped with the devices) threatening the security of Android devices. In RQ_1 we also report the evolution of the types of security vulnerabilities over six-month periods through the whole history of the Android OS, in particular the time period between 2008 and August 2017 (the last month in which we collected the data). This analysis will provide indications useful to understand whether the security of the Android OS is improving over time and what the major issues are nowadays.

RQ_2: *What are the most common CVSS vectors that describe the Android OS vulnerabilities?* This research question aims at studying the characteristics of the CVSS vectors describing the vulnerabilities and will serve to further characterize the security issues affecting the Android OS.

RQ_3: *Which are the Android subsystems more affected by security vulnerabilities?* The third research question sheds light on the Android subsystems more frequently affected by security vulnerabilities. Note that in this case our focus will be on the architecture of the Android OS, while less emphasis will be given to vulnerabilities affecting third-party components. Indeed, our goal is to point out to developers (both apps developers as well as contributors of the Android OS) which are the riskier services, APIs, apps, *etc.* in the OS. This information can be used to better focus verification & validation activities as well as to develop better Android-specific tools for vulnerability detection and secure coding.

RQ_4: *How long does it take to fix security vulnerabilities in Android?* This research question studies the survivability of the security vulnerabilities subject of our study. In particular, we assess the number of days between the vulnerability introduction and its fixing. RQ_4's findings could help in assessing the usefulness of effective vulnerability detection tools able to immediately catch an introduced vulnerability (i.e., a long survivability of the vulnerabilities would indicate the urge for such tools).

3.1 Data Extraction

The context of the study consists of 1,235 vulnerabilities mined from the official Android Security Bulletins (Google 2017a), the CVE details portal (MITRE 2017r) and the NVD database(U.S. National Institute of Standards and Technology - NIST 2012a). Every month Google releases a security bulletin that describes the most recent discovered and fixed vulnerabilities that have been reported in the NVD. For some of the vulnerabilities links to the patches are also provided. Since the first bulletin was published in August 2015, not all the vulnerabilities relevant for our study are covered in the Android bulletins. For this reason, we also mined the CVE details portal, a mirror of the NVD database providing pre-defined tags for searching purpose. For example, the link https://www.cvedetails.com/product/19997/Google-Android.html lists vulnerabilities related to the Android OS. Some of the vulnerabilities relevant for our study are categorized in CVE details under different

tags, such as "Linux kernel". This is why mining only the CVE details portal is not sufficient, and we combined the list of CVE ids extracted from the Android bulletins with those categorized in CVE details under the "Google-Android" tag.

To collect this information, we built a Java web-based scraper. The scraper firstly mined all the security bulletins published by Google looking for their CVE ids. Then, it extracted the details of each vulnerability by using its CVE id to get the corresponding data from the CVE details portal and the NVD web site.

A second web-based scraper was used to extract from the CVE details portal all vulnerabilities in the Android category. In total, we collected 1,430 CVE ids. We found that 62 of the vulnerabilities were reported as reserved, meaning that the details of the vulnerability were not publicly available yet, and were consequently discarded. Additionally, information regarding other 132 vulnerabilities could not be collected because of different reasons (e.g., timeout error, absence of important data, *etc.*). Finally, one vulnerability was discarded since it was related to iOS. We ended up with 1,235 CVE ids having valid information stored in *JSON* files as depicted in Fig. 2. Note that there are two groups of information related to CVSS 2.0, because there are two sources of information: the CVE Details and the NVD website. The `gainedAccess` field is marked in red in Fig. 2 because this element is only used in CVE Details and it is not defined in the CVSS 2.0 documentation. Indeed, the authentication field from the CVE Details portal uses the CVSS 1.0 nomenclature. For this reason, while using the general description information provided in CVE Details (see top part of Fig. 2), we exploit the CVSS information collected from NVD rather than the one from CVE Details portal. It is worth noting that some of the collected vulnerabilities do not have all the CVSS-related data because some information is not mandatory.

We complemented/updated the data collected for these vulnerabilities via manual analysis. In particular, once extracted the information in Fig. 2, two of the authors manually analyzed each vulnerability to:

1. *Check and complement the vulnerability type automatically inferred by CVE Details, and obtain its hierarchy*. CVE Details uses a keywords-based mechanism to automatically infer the type of each vulnerability according to the Common Weakness

General description from CVE Details
cveId: the id of the vulnerability
description: a textual description of the vulnerability
publishDate: vulnerability publication date
patchLinks: [Links to the patch(es) aimed at fixing this vulnerability]
vulnerabilityTypes: [The type(s) of the vulnerability automatically inferred]

CVEDetails CVSSv2.0:
score: a score from 0 to 10 indicating the severity of the vulnerability
severity: score categorical value (Low/Med./High)
vector: vector string
gainedAccess: whether or not an attacker has gained access to the target system (None/User/Admin)
accessComplexity: complexity of the attack required to exploit the vulnerability once an attacker has gained access to the target system (High/Med./Low)
authentication: whether or not an attacker needs to be authenticated to the target system in order to exploit the vulnerability (Required/Not Required)
confidentialityImpact: impact on confidentiality of a successfully exploited vulnerability (None/Partial/Complete)
integrityImpact: impact to integrity of a successfully exploited vulnerability (None/Partial/Complete)
availabilityImpact: impact to availability of a successfully exploited vulnerability (None/Partial/Complete)

NVD CVSSv2.0:
baseScore: a score from 0 to 10 indicating the severity of the vulnerability
severity: base score categorical value (Low/Med./High)
vector: vector string
attackVector: reflects how the vulnerability is exploited (Local/Adjacent Network/Network)
accessComplexity: complexity of the attack required to exploit the vulnerability once an attacker has gained access to the target system (High/Med./Low)
authtentication: number of times an attacker must authenticate to a target in order to exploit a vulnerability (Multiple/Single/None)
confidentialityImpact: impact on confidentiality of a successfully exploited vulnerability (None/Partial/Complete)
integrityImpact: impact to integrity of a successfully exploited vulnerability (None/Partial/Complete)
availabilityImpact: impact to availability of a successfully exploited vulnerability (None/Partial/Complete)

Fig. 2 Information stored for the mined vulnerabilities

Enumeration (CWE) dictionary, version 2.11 (MITRE 2017p). Such an automatic process can introduce imprecisions in the data. For this reason, the authors analyzed all the information available about each vulnerability (i.e., its page on the National Vulnerability Database, fixing patches when publicly available, official vulnerability bulletins, the Android issue tracker, *etc.*) to verify the type of the vulnerability, identify the CWE hierarchy, and change/complement the classification according to the CWE dictionary.

Note that a vulnerability can belong to multiple types having hierarchical relationships between them. For example, a vulnerability can be classified in any of the types of the following hierarchy (the hierarchy is shown from the least to the most specialized vulnerability type):

> Weaknesses that Affect Memory → Improper Restriction of Operations within the Bounds of a Memory Buffer → Out-of-bounds Read → Buffer Over-read

Overall, the manual analysis led to the change of the type provided by CVE Details for 81.4% of the analyzed vulnerabilities.

2. *Identify the subsystems affected by the vulnerability*. We also analyzed the information available in the NVD Database and CVE details (including, when available, the patches fixing the vulnerability) as well as on-line documentation (e.g., the Android issue tracker) to identify the code components in the Android OS affected by the vulnerability at two-levels: first, what layer of the Android Stack is affected by the vulnerability (e.g., *Android runtime*), and second, which component (subsystem) of the layer was affected (e.g., *Dalvik VM*).

The above described manual analysis was performed in three rounds. First, the authors manually analyzed half of the 1,235 vulnerabilities each. Then, an author A_i checked the vulnerability types and the impacted architectural layers/subsystems assigned by A_j and *vice versa*. In cases in which the evaluators were undecided about the specific type of vulnerability and/or about the subsystem affecting the vulnerability, an "unclear" tag was assigned. Before solving conflicts, the authors achieved an agreement ratio of 93%, i.e., only in 86 cases the authors disagreed. To determine whether such agreement level was due to chance, we computed the Fleiss' kappa inter-rater agreement coefficient (Li 2010), obtaining an almost perfect agreement between the raters with a kappa score of 0.92.

One example of the conflicts arisen during the manual coding is the CVE-2017-0735 vulnerability, described in CVE details as a denial of service in the Android media framework (libavc). In this case the patches fixing the vulnerability were available and they could be inspected by the authors.[2,3] Analyzing the code committed, two alternatives were discussed regarding the type of vulnerability: (i) *Improper Initialization*, i.e., the software does not initialize or incorrectly initializes a resource, which might leave the resource in an unexpected state when it is accessed or used; (ii) *Unchecked Error Condition*, i.e., ignoring exceptions and other error conditions may allow an attacker to induce unexpected behavior unnoticed. Given the patches and their commits messages (i.e., *"Postponed the initializations to decoder context till the end of the parse sps function, after all the error checks are done"* and *"Fixed hang in the case of multiple sps id")*, it can be appreciated considering the first commit[2] that despite the inappropriate initialization of particular parameters in the code, the main factor to be considered is the postponement of the initialization of those

[2] https://tinyurl.com/yadtggsr
[3] https://tinyurl.com/y9u5odrv

parameters after error checks are done. If these checks are not considered before the initializations, error conditions may allow an attacker to induce unexpected behavior unnoticed. Moreover, contemplating the second commit[3] it is noticeable that an uncommon scenario was not being foreseen and a validation was introduced in order to handle it. Therefore, the type of vulnerability was defined as:

> Improper Check or Handling of Exceptional Conditions → Unchecked Error Condition.

Another example of conflict is CVE-2017-8267, described in CVE details as "*in all Qualcomm products with Android releases from CAF using the Linux kernel, a race condition exists in an IOCTL handler potentially leading to an integer overflow and then an out-of-bounds write*". The patch for this vulnerability is not publicly available, constraining the decision about its classification to the information available in the CVE details description. The two taggers discussed two alternatives (i) *Exposed IOCTL with Insufficient Access Control*, i.e., the software implements an IOCTL (Input/Output ConTroL) with functionality that should be restricted, but it does not properly enforce access control for the IOCTL; (ii) *Race Condition within a Thread*, i.e., two threads use a resource simultaneously, with the possibility that resources may be used while invalid, in turn making the state of execution undefined. Since there was no code to review and the IOCTL system call could not be analyzed, the type of vulnerability was defined as:

> Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition') → *Race Condition within a Thread*

3.2 Analysis Method

We answer $\mathbf{RQ}_1$ by creating a taxonomy of the types of vulnerabilities identified in the manual analysis. Also, we analyze the evolution of the vulnerabilities by grouping them in semester-based batches according to the vulnerabilities publication date (e.g., we have one batch for the vulnerabilities published during the first semester of 2008, one batch for the vulnerabilities published during the second semester of 2008, *etc.*). To analyze the evolution, we identified the changes in the frequency of vulnerability types across different semesters.

We address $\mathbf{RQ}_2$ by analyzing the features in the CVSS vectors to characterize the most common vectors and the vulnerabilities related to the vectors. As part of the analysis, we first computed the distribution of the values for the base group attributes depicted in Fig. 2 (i.e., attack complexity, severity, user interaction, confidentiality impact, integrity impact, and availability impact); we did not include the authentication attribute in the analysis, because for all vulnerabilities a "None" value is reported for this attribute. Then, we identified the most common CVSS 2.0 vectors and the types of vulnerability related to each vector.

Concerning $\mathbf{RQ}_3$, we report a heat map showing the distribution of vulnerabilities across the Android layers subsystems. We also analyzed the top ten subsystems/components in the Android OS layers more frequently impacted by vulnerabilities in each of the semester-based batches considered in RQ_1. We complement our discussion for RQ_3 with qualitative examples of the vulnerabilities.

To answer $\mathbf{RQ}_4$ we need information not available neither in the CVE Details nor in the NVD sites. In particular, we need to identify the commits in which each vulnerability has been introduced and fixed. As for the commit fixing each vulnerability, we mined it from

the Android Security Bulletins. The vulnerability-fixing commit is not available for all the 1,235 Android-related vulnerabilities we collected, because (i) some vulnerabilities were reported before the first available bulletin, and (ii) the fixing commit (see e.g., http://tinyurl.com/hrod7q9) is only available for the subset of vulnerabilities fixed in the Android open source project (e.g., it is not available for vulnerabilities related to third-party components such as drivers). Note also that, although the CVE reports include in some cases the bug id, the ids are for the internal bug trackers of Google and hardware manufactures, which are not publicly available. For these reasons, the analysis for RQ_4 is limited to a set of 331 vulnerabilities for which we identified the fixing commit. Once identified the fixing commit, we used the SZZ algorithm (Sliwerski et al. 2005) to identify the commit introducing the vulnerability. The algorithm relies on the annotation/blame feature of versioning systems. Given a vulnerability-fixing commit VF_k (where k identifies the vulnerability), the approach works as follows:

1. For each file f_i, involved in VF_k and fixed in its revision $rel\text{-}fix_{i,k}$, we extract the file revision just *before* the vulnerability fixing ($rel\text{-}fix_k - 1$).
2. Starting from the revision $rel\text{-}fix_k - 1$, for each source code line in f_i changed to fix the vulnerability k, the *blame* feature of Git is used to identify the file revision where the last change to that line occurred.

In doing that, blank lines are identified and ignored. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions $rel\text{-}vulnerability_{i,j,k}$, $j = 1 \ldots n_{i,k}$.

Since more than one commit can be indicated by the SZZ algorithm as responsible for inducing the vulnerability-fix, there are time vulnerability ranges defined by lower (minimum survivability) and upper bounds (maximum survivability). Therefore, we answer $\mathbf{RQ_4}$ by following a meta analysis-based procedure (Hedges and Olkin 1985; Cumming 2011):[4] the minimum and the maximum survivability of the vulnerabilities (i.e., number of days between the vulnerability introduction and fixing) are plotted using forest plots with confidence intervals, and a central tendency measure of the survivability is computed by using the random effects model (Cumming 2011) (based on the recommendations by Cumming 2011). The minimum survivability is the one observed when considering the most recent commit identified by the SZZ algorithm as the one that induced the vulnerability-fix. *Vice versa*, the maximum survivability is observed when considering the least recent commit identified by the SZZ algorithm as the one that induced the vulnerability-fix. The forest plots are depicted by considering a 95% confidence interval.

We also verified whether vulnerabilities having different severity levels have different survivability. For this analysis, we used the severity classification available in CVSS 2.0 (*low*, *medium*, *high*). In particular, we compared the distributions of the survivability of the different categories of vulnerabilities (e.g., *low vs. medium*) via (i) forest plots, and (ii) statistical tests. For the latter we exploited the Mann-Whitney test (Conover 1998) with results intended as statistically significant at $\alpha = 0.05$. To control the impact of multiple pairwise comparisons (e.g., the survivability of the vulnerabilities having *low* severity is compared against the survivability of those having *medium* and *high* severity), we adjust p-values using the Holm's correction (Holm 1979). We also estimate the magnitude of the differences by using the Cliff's Delta (d), a non-parametric effect size measure (Grissom and Kim 2005) for ordinal data. We follow well-established guidelines to interpret the effect

[4]Meta-analysis is a statistical inference technique aimed at consolidating results from more than one study or experiment.

size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ (Grissom and Kim 2005). Finally, to ease the interpretation of the achieved results, we also compare the survivability of the vulnerabilities with that of other types of bugs, unrelated to security issues, and extracted from the same repositories in which the vulnerabilities were fixed. In particular, for each vulnerability-fixing commit, we randomly selected, from the same repository in which the vulnerability was fixed, three bug-fixing commits having their commit note matching the regular expression `fix(es)(ed) bug(s)`. Then, knowing the possible imprecisions introduced by this procedure, the first author manually analyzed all candidate bug-fixing commits discarding the ones that were related in some way to security issues **or** that were not bug-fixing activities. This left us with 385 valid bug-fixing commits not related to security vulnerabilities. From this set, we randomly selected 331 of them, to compare the survivability of the bugs they fix to that of the vulnerabilities fixed in the 331 vulnerability-fixing commits. Also in this case, we compute the minimum and the maximum survivability using the same procedure previously described, and we statistically compare the survivability of vulnerabilities with that of other types of bugs by using the Mann-Whitney test and the Cliff's Delta.

4 Results

This section discusses the quantitative results achieved in our study according to the four formulated RQs. Also, we complement quantitative data with qualitative examples, by referring to specific vulnerabilities identified with their CVE id (e.g., `CVE-2016-2439`). The reader can access the page detailing a vulnerability by visiting "https://web.nvd.nist.gov/view/vuln/detail?vulnId=" followed by the CVE id. We provide in our online appendix (Mazuera-Rozo et al. 2017) the complete list of vulnerabilities considered in our study, including their categorization by subsystem, component, and vulnerability type. Also, we created visualizations aimed at helping the reader when browsing the vulnerabilities list (Mazuera-Rozo et al. 2017).

4.1 RQ_1: *Which Types of Security Vulnerabilities Affect the Android OS?*

Figures 3 and 4 show the taxonomy reporting the vulnerability types we found in the 1,235 manually inspected vulnerabilities. The vulnerability types are depicted as several hierarchies (each color represents a hierarchy) by following the categorization provided in the CWE dictionary (MITRE 2017p). The hierarchies have been split into two figures for the sake of readability. Note that the two figures only report the classification for 1,066 vulnerabilities. This is due to the fact that we were not able to infer the type of 169 vulnerabilities during our manual analysis. Also, the root categories (i.e., the ones reporting the overall number of vulnerabilities in each hierarchy) have been defined by us to allow an easier navigation of the different vulnerability types. Finally, it is worth noting that changes in the taxonomy are present as compared to the work by Linares-Vásquez et al. (2017) that this paper extends. This is due to (i) new categories identified as a result of the larger set of vulnerabilities analyzed, and (ii) the fact that we updated the previous classification in Linares-Vásquez et al. (2017) to the newest version of the CWE dictionary (2.11 as compared to the 2.10 used in Linares-Vásquez et al. 2017).

The vulnerabilities most frequently affecting Android devices are those related to permissions, privileges, and access control with 243 instances (23%). They include, for example, weaknesses due to *improper access control* and to *permission issues*, like the

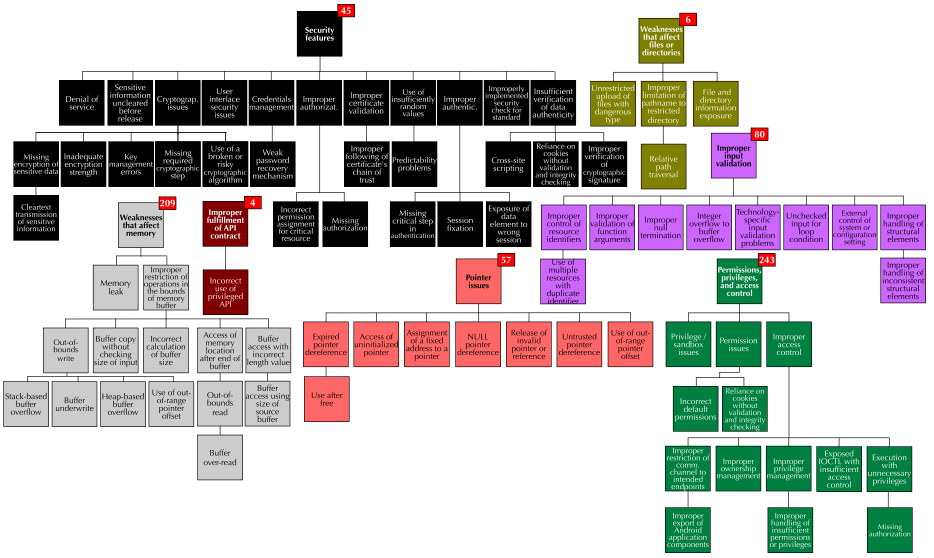


Fig. 3 Types of Android-related vulnerabilities (Part I)

reliance on cookies without validation and integrity checking vulnerability discussed in `CVE-2008-7298`:

> The Android browser cannot properly restrict modifications to cookies established in HTTPS sessions, which allows man-in-the-middle attackers to overwrite or delete arbitrary cookies via a Set-Cookie header in an HTTP response [...]

Vulnerabilities related to weaknesses that affect the memory, are represented in our taxonomy with 209 instances (20%). These weaknesses include all vulnerabilities related to the *improper restriction of operations in the bounds of memory buffer*, like *out-of-bounds*

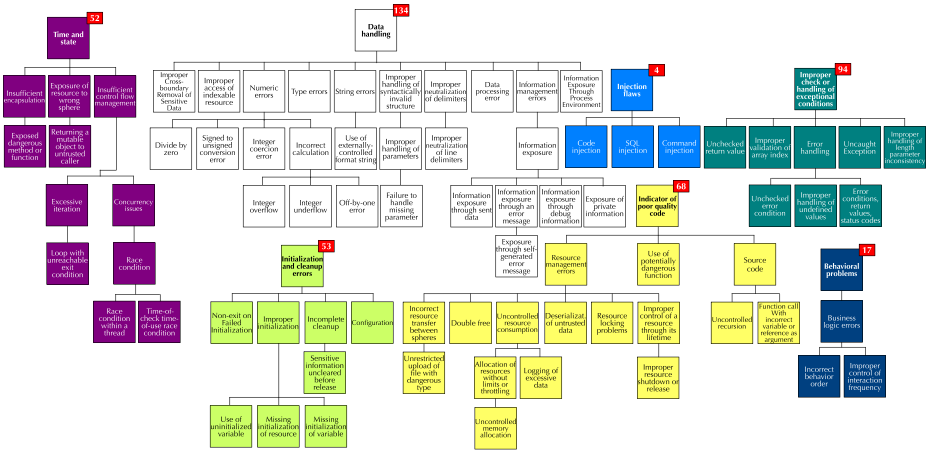


Fig. 4 Types of Android-related vulnerabilities (Part II)

read/write. One vulnerability falling in this category is `CVE-2016-2439`, described as follows:

> Buffer overflow in btif_dm.c in Bluetooth [...] allows attackers to execute arbitrary code via a long PIN value

The vulnerability was fixed in commit `9b534de`, modifying the conditional statement checking a PIN-related error from `if (pin_code == NULL)` to `if (pin_code == NULL || pin_len > PIN_CODE_LEN)`. The rationale behind the change is also documented by the developer in the commit message: *If a malicious client sets a pin that was too long it would overflow the pin code memory*.

Very frequent are also vulnerabilities related to **data handling**, typically found in functionalities that process data (MITRE 2017p) (134 instances—13%). These include, for example, *type errors* like the one related to `CVE-2016-3918`:

> AttachmentProvider.java in AOSP Mail in Android [...] does not ensure that certain values are integers, which allows attackers to read arbitrary attachments [...]

The vulnerability was fixed in commit `6b2b0bd` that, as reported in the commit message: *Limits account id and id to longs [...] Both id and account id are now parsed into longs (and if either fails, an error will be logged and null will be returned)*. Note that the *data handling* category includes several different sub-categories such as *numeric errors* (e.g., *signed to unsigned conversion error*) and *information exposure* (e.g., *exposure of private information*)—see Fig. 4.

Improper input validation (80 instances—8%) includes vulnerabilities caused by a missing or improper validation of inputs that can affect the control/data flow of the program (MITRE 2017p). Vulnerabilities in this category include (but are not limited to—see Fig. 3) *unchecked input for loop condition* and *improper validation of function arguments*.

The latter are the most popular in this category and, while their fixing is generally simple (e.g., the addition of a missing/improper argument validation), they can result in severe attacks like the one possible by exploiting `CVE-2016-3910` (9.3 out of 10 in severity score) which was fixed by adding size checks on the sound model data structure.

Initialization and cleanup errors and improper check or handling of exceptional conditions are the cause for 53 (5%) and 94 (9%), respectively, of the categorized vulnerabilities. These categories include, among others, the *missing initialization of a variable* and *uncaught exceptions*. An example for the latter is the vulnerability `CVE-2017-0394`, allowing to perform a denial of service attack crashing the phone by exploiting an uncaught exception. The vulnerability has been fixed in commit `1cdced5`, reporting the message: *Catch SIP exceptions which can crash Phone process on answer*, where SIP refers to the Session Initiation Protocol allowing apps to integrate Internet telephony features.

Indicator of poor quality code (68 instances—6%) includes weaknesses that can lead to unpredictable behavior. For example, cases of *uncontrolled recursion* such as the one described in `CVE-2017-0692`, fixed in commit `6db4826` (commit note: *fix infinite recursion*) by adding the condition `if (offset <= nodeOffset)` inside a `for` loop checking for the need to interrupt the recursion with a `return` statement.

Time and state vulnerabilities represent 52 (5%) instances of our taxonomy. They represent *"weaknesses related to the improper management of time and state in an environment*

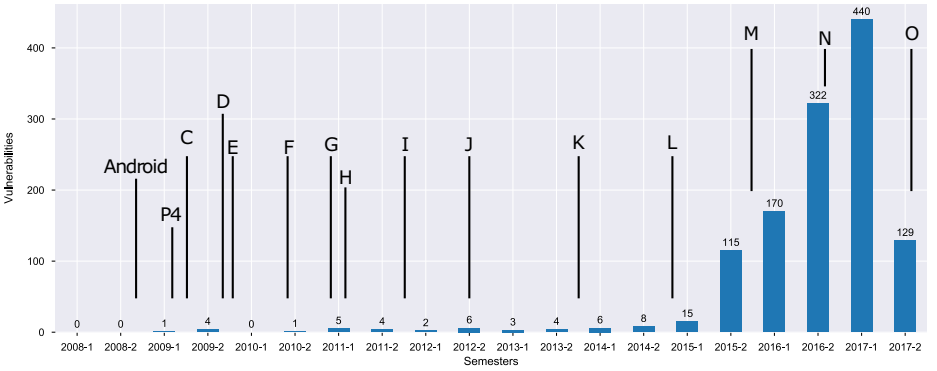


Fig. 5 Total number of Android vulnerabilities across OS versions. The black horizontal lines denote the time when a new version of Android was released (e.g., P4 version, Android C, Android D, etc.)

that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads" (MITRE 2017p). Classic examples of this category are race conditions between threads such as `CVE-2017-0387`, fixed by the developers in commit `675e212` by implementing a mutex (mutual exclusion) lock mechanism.

Security features are involved in 45 vulnerabilities (4%) related to *cryptographic issues*, *user interface security issues*, *credentials management* problems, etc. (see Fig. 3). For example, `CVE-2011-2344` reports a vulnerability due to *inadequate encryption strength* possibly causing severe attacks allowing the stealing of private pictures:

> Android Picasa in Android [...] uses a cleartext HTTP session when transmitting the authToken obtained from ClientLogin, which allows remote attackers to gain privileges and access private pictures and web albums by sniffing the token from connections with picasaweb.google.com

Finally, other less diffused vulnerabilities are those falling in the categories: behavioral problems (17), weaknesses that affect files or directories (6), injection flaws (4), and improper fulfillment of API contract (4). A description of these categories can be found in the CWE dictionary (MITRE 2017p), while Android-related examples from our dataset are available in our online appendix (Mazuera-Rozo et al. 2017).

Evolution of the vulnerabilities over time Figure 5 shows (i) the total number of vulnerabilities published by semester, since 2008 (first Android release), and the moment (black horizontal lines tagged with the version id) in which every Android version was released (Wikipedia 2017a).

The number of reported vulnerabilities increased drastically after the 2015-1 "Stagefright Vulnerability" issue and the creation of the Google Project Zero in July 2014. Thus, for the analysis we distinguish two time periods: (i) from the first semester of 2008 to the first semester of 2015 (included), and (ii) from the second semester of 2015 to the first semester of 2017 (included).[5] From the first semester of 2008 to the first semester of 2015

[5]We found 129 vulnerabilities for the 2017-2 period, but we excluded this semester from the RQ_5 analysis because we do not have complete data for 2017-2.

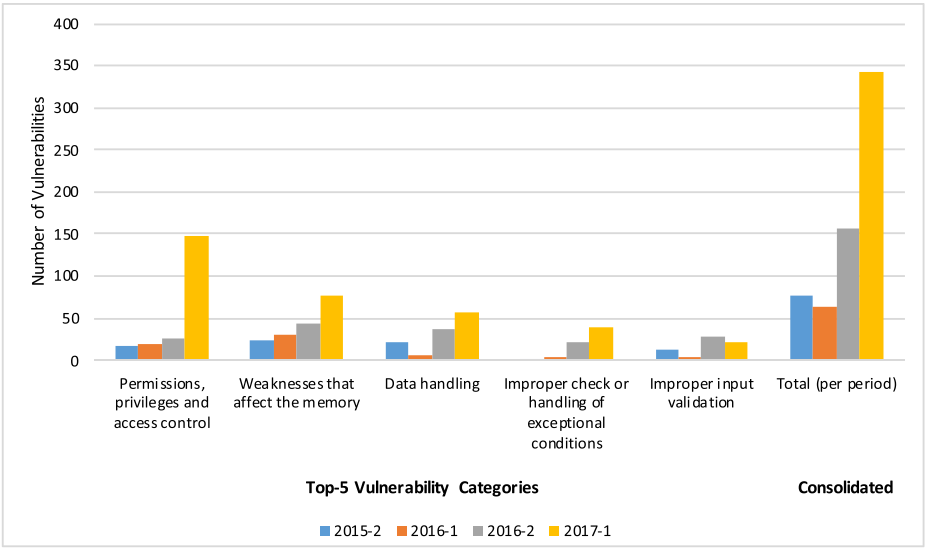


Fig. 6 Changes over time for the top-5 vulnerability categories

(included) only 59 vulnerabilities were identified. Regarding the affected layers (during this period) of the OS, 32 vulnerabilities (54.2%) impacted the application layer while the rest is homogeneously distributed in the remaining layers.[6] Among the 32 vulnerabilities affecting the application layer, 12 are related to Adobe Flash Player. In relation to the vulnerabilities category, about 50% of the vulnerabilities belong to three categories: security features (12), permissions, privileges, and access control (9), and weaknesses that affect the memory (7).

We now analyze the evolution of the types of vulnerabilities over time, considering 885 vulnerabilities out of the 1,047 related to the second time period, since there are 162 vulnerabilities for which we were not able to manually identify the type. Figure 6 depicts changes in the frequencies of the five most frequent types of vulnerabilities in the Android OS: (i) permissions, privileges and access control, (ii) weaknesses that affect memory, (iii) data handling, (iv) improper management of exceptions, and (v) improper input validation. Figure 6 shows that there is an increasing tendency in the number of vulnerabilities for the five categories, being permissions, privileges and access control the category with the highest increment, in particular from 2016-2 to 2017-1. Most of these vulnerabilities belong to kernel programs/subsystems and are due to the mishandling of access control decisions, enabling behaviors like the execution of arbitrary code within the context of the kernel.

Qualitative analysis of patches We manually analyzed a sample of patches for the vulnerabilities to identify the underlying code issues. Note that in our dataset we only found

[6]We were not able to classify the layer affected by 5 vulnerabilities.

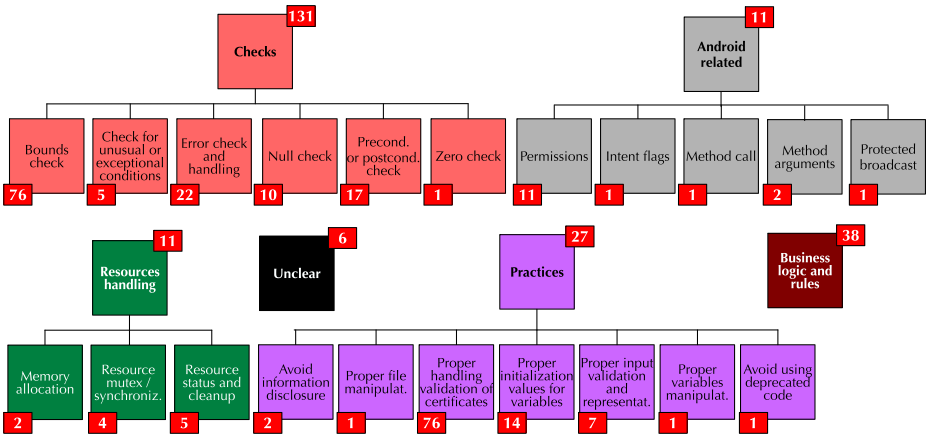


Fig. 7 Types of changes performed by Android OS developers to fix 178 security vulnerabilities. It is worth noting that a patch could have more than one type of change, thus, the sum of the instances in the categories is greater than 178

331 vulnerabilities with a fixing commit (as mentioned in Section 3.1), therefore, we randomly selected 178 vulnerabilities (which is a representative sample of the 331 fixing commits with 95%±5 confidence level) for the manual analysis of the underlying code issues.

Each evaluated patch was analyzed with the purpose of reporting whether there are common patterns in the changes conducted by developers to fix vulnerabilities. Figure 7 depicts the common patterns grouped into five different categories:[7]

- **Checks (131 instances).** Changes that have the purpose of properly checking for unusual or exceptional conditions that are not expected to occur frequently, or preconditions required for the proper execution of a code block.
- **Business logic and rules (38 instances).** Changes oriented toward correctly guaranteeing the business processes, application logic, and sequences of behaviors. This type of code changes is very specific to the part of the system that is fixed and, therefore, there are no specific "common change pattern" as in the case of the "Checks" category.
- **Practices (27 instances).** Changes that introduce the usage of secure coding practices.
- **Resources handling (11 instances)**. Changes addressing the proper management of system resources.
- **Android-related (16 instances).** Changes aimed at fixing issues in mechanisms that are specific to the Android platform such as fixes in permissions and inter process communication.

Most of the fixes are categorized as Checks, in particular, Bounds checks added to the code with the purpose of validating whether a variable is within the expected

[7]Note that there is also an **Unclear** category because in six cases we were not able to identify the type of change because of the patches complexity . For instance, in `CVE-2016-3751` several changes were done over 207 files, thus it was not easy to categorize the changes.

bounds of a structure or collection. Examples of vulnerabilities in this subcategory are CVE-2015-3871 and CVE-2015-3864, described respectively as follows:

> libstagefright in Android before 5.1.1 LMY48T allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted media file

> Integer underflow in the MPEG4Extractor::parseChunk function in MPEG4 Extractor.cpp in libstagefright in mediaserver in Android before 5.1.1 LMY48M allows remote attackers to execute arbitrary code via crafted MPEG-4 data

The CVE-2015-3871 vulnerability was fixed in the commit **c570778**[8] by adding an extra clause to an existing condition that validates the maximum size of a buffer:

```
@@ -2217,7 +2217,7 @@
}

        status_t MPEG4Extractor::parseITunesMetaData(off64_t offset,
             size_t size) {
-       if (size < 4) {
+       if (size < 4 || size == SIZE_MAX) {
                return ERROR_MALFORMED;
         }
```

The case of CVE-2015-3864 is very similar. The patch with id 6fe85f7,[9] shows that the vulnerability was fixed by adding a bound check for the chunk size:

```
@@ -1893,7 +1893,7 @@
        size = 0;
}

-       if (SIZE_MAX - chunk_size <= size) {
+       if ((chunk_size > SIZE_MAX) || (SIZE_MAX - chunk_size <=
    size)) {
        return ERROR_MALFORMED;
}
```

Another remarkable example represents the second largest category, Business logic and rules. The vulnerability identified by CVE-2015-3844 and fixed in commit e3cde78 is described as follows:

> The getProcessRecordLocked method in services/core/java/com/android/server/am/ActivityManagerService.java in ActivityManager in Android before 5.1.1 LMY48I allows attackers to trigger incorrect process loading via a crafted application, as demonstrated by interfering with use of the Settings application

As discussed by the Android Security team,[10] the method ActivityManagerService.getProcessRecordLocked() may load a system UID application into the wrong process.

[8] https://tinyurl.com/y777kx6s
[9] https://tinyurl.com/yblfv22p
[10] https://tinyurl.com/yaehg5b2

This method does not properly verify that an application's process name matches the corresponding package name and in some cases, this can allow ActivityManager to load the wrong process for certain tasks. The patch required to fix this issue prevents the system uid component from running in an app process:

```
@@ -2691,9 +2691,14 @@
// should never happen).
SparseArray<ProcessRecord> procs = mProcessNames.getMap().get(
    processName);
if (procs == null) return null;
-             final int N = procs.size();
-             for (int i = 0; i < N; i++) {
-                 if (UserHandle.isSameUser(procs.keyAt(i), uid))
    return procs.valueAt(i);
+             final int procCount = procs.size();
+             for (int i = 0; i < procCount; i++) {
+                 final int procUid = procs.keyAt(i);
+                 if (UserHandle.isApp(procUid) || !UserHandle.
    isSameUser(procUid, uid)) {
+                     // Don't use an app process or different user
    process for system component.
+                     continue;
+                 }
+                 return procs.valueAt(i);
}
}
ProcessRecord proc = mProcessNames.get(processName, uid);
```

In the category Practices most of the patches belong to the Proper initialization values for variables subcategory, e.g., the `CVE-2016-2499` vulnerability described as follows:

> AudioSource.cpp in libstagefright in mediaserver in Android 4.x before 4.4.4, 5.0.x before 5.0.2, 5.1.x before 5.1.1, and 6.x before 2016-06-01 does not initialize certain data, which allows attackers to obtain sensitive information via a crafted application

In order to prevent information leakage, several variables were initialized inside the `media/libstagefright/AudioSource.cpp` file as it can be appreciated in commit dd35467[11] and directly in the snippet below:

```
@@ -55,8 +55,12 @@
        : mStarted(false),
        mSampleRate(sampleRate),
        mOutSampleRate(outSampleRate > 0 ? outSampleRate : sampleRate
            ),
+       mTrackMaxAmplitude(false),
+       mStartTimeUs(0),
+       mMaxAmplitude(0),
        mPrevSampleTimeUs(0),
        mFirstSampleTimeUs(-1ll),
+       mInitialReadTimeUs(0),
        mNumFramesReceived(0),
        mNumClientOwnedBuffers(0) {
      ALOGV("sampleRate: %u, outSampleRate: %u, channelCount: %u",
```

[11] https://tinyurl.com/ya8rbl7n

Concerning the Resources handling category, we found an interesting case of Resource status and cleanup subcategory: `CVE-2017-0726` is described in CVE details as *a denial of service vulnerability in the Android media framework*, and it was fixed in commit 8995285[12] preventing a potential memory leak by deallocating memory as shown below:

```
@@ -2826,8 +2826,10 @@
        int32_t delay, padding;
        if (sscanf(mLastCommentData,
                " %*x %x %x %*x", &delay, &padding) == 2) {
-           if (mLastTrack == NULL)
+           if (mLastTrack == NULL) {
+               delete[] buffer;
                return ERROR_MALFORMED;
+       }

        mLastTrack->meta->setInt32(kKeyEncoderDelay, delay);
        mLastTrack->meta->setInt32(kKeyEncoderPadding, padding);
```

Finally, with respect to the Android-related category, `CVE-2017-0423` is an example of the Permissions subcategory. The vulnerability is described as follows:

> An elevation of privilege vulnerability in Bluetooth could enable a proximate attacker to manage access to documents on the device. This issue is rated as Moderate because it first requires exploitation of a separate vulnerability in the Bluetooth stack.

This vulnerability was fixed in commit 4c1f39e[13] removing the `MANAGE_DOCUMENTS` permission from the `AndroidManifest.xml` file as it was not needed.

```
@@ -60,7 +60,6 @@
  <uses-permission android:name="android.permission.WRITE_SMS" />
  <uses-permission android:name="android.permission.READ_CONTACTS"
      />
  <uses-permission android:name="android.permission.
      MEDIA_CONTENT_CONTROL" />
- <uses-permission android:name="android.permission.MANAGE_DOCUMENTS
    " />
  <uses-permission android:name="android.permission.
      UPDATE_APP_OPS_STATS" />
  <uses-permission android:name="android.permission.VIBRATE" />
  <uses-permission android:name="android.permission.DEVICE_POWER" />
```

4.2 RQ_2: What are the Most Common CVSS Vectors that Describe the Android OS Vulnerabilities?

To answer this question, we first analyzed the frequency of the values for the base group features of the CVSS vectors. Figure 8 shows the base group attributes and corresponding values for the 1,235 analyzed vulnerabilities. In particular, the top part of Fig. 8 shows their severity level and access complexity (i.e., how difficult it is to exploit the vulnerability). The first thing that leaps to the eyes is that the severity of the vulnerabilities is generally high (68% of cases), while it is not so difficult to exploit them (only 17% of vulnerabilities

[12] https://tinyurl.com/ychpav57
[13] https://tinyurl.com/y6v7me9z

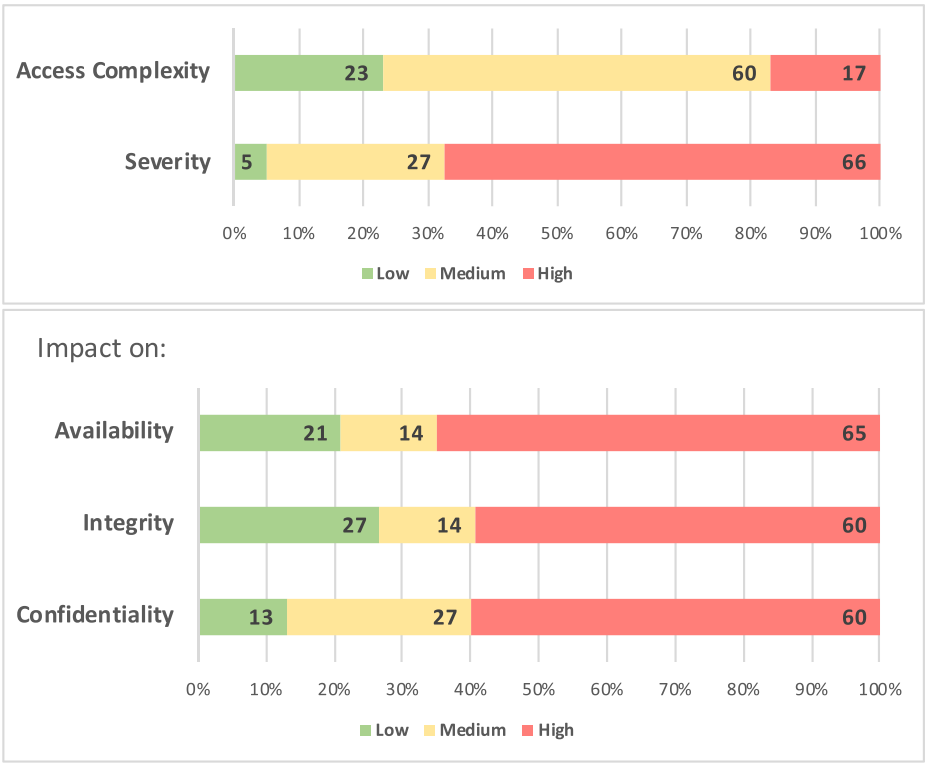


Fig. 8 Base group features of Android-related vulnerabilities

have a high access complexity which means that specialized access conditions are required). The high severity of the vulnerabilities is also confirmed by their confidentiality, integrity, and availability impact (bottom part of Fig. 8), classified in most of cases as "complete" indicating, for example, the possibility to make completely unavailable the computational resources of the affected device (availability impact: complete).

In the set of 1,235 analyzed vulnerabilities, we found 49 different CVSS 2.0 vectors summarizing all characteristics of the vulnerabilities.[14] The top-5 CVSS vectors are reported in Table 2, represent 69,79% of the 1,235 vulnerabilities, and for four out of the five vectors, the Android Kernel is the most impacted layer (see Fig. 10b). The distribution of vulnerabilities for each of the 49 vector types is reported in Fig. 9. The distribution is extremely skewed, with an average of 25 vulnerabilities per vector type, and a median of 2. The upper bound (i.e., the most frequent CVSS vector) includes 399 vulnerabilities and corresponds to the vector **AV:N/AC:M/Au:N/C:C/I:C/A:C** (Fig. 10).

Note that in the top-5 vectors the value for the Access Vector attribute is "Network" (AV:N), and the Authentication value is "None" (Au:N); this means that at least 69,79% of the Android OS vulnerabilities are remotely exploitable and authentication is not required

[14]For the base group attributes in CVSS 2.0, there are 729 possible combinations of attribute values. Therefore, the 1,235 analyzed vulnerabilities cover 6.72% (49 out of 729) of all the CVSS 2.0 vectors for the base group attributes.

Table 2 Top-5 CVSS 2.0 vectors. The table lists the values for the Access Vector (AV), Access Complexity (AC), Authentication (Au), Confidentiality Impact (C), Integrity Impact (I), and Availability Impact (A) attributes

Top	Instances	AV	AC	Au	C	I	A
1	399	Network	Medium	None	Complete	Complete	Complete
2	150	Network	High	None	Complete	Complete	Complete
3	128	Network	Medium	None	Partial	None	None
4	114	Network	Low	None	Complete	Complete	Complete
5	71	Network	Medium	None	Partial	Partial	Partial

to exploit the vulnerabilities. In most of the vectors (Top-1, Top-2, and Top-4) the confidentiality, integrity and availability of the system are completely compromised (C:C/I:C/A:C); this means that in at least 53,68% of the vulnerabilities (663 out of 1,235) there is total disclosure of the system files, total compromise of the system integrity (i.e., the attacker is able to modify any file (NIST 2015)), and there can be a total shutdown of the affected resource (or the complete system).

Top 1: AV:N/AC:M/Au:N/C:C/I:C/A:C This vector represents 399 instances; in 80 cases we assigned "unclear" as vulnerability type. In the remaining cases, 45.8% are in the following categories: permissions, privileges, and access control (77); weaknesses that affect the memory (71); and improper check or handling of exceptional conditions (35). A case related to this vector is the vulnerability `CVE-2017-0331` that belongs to the permissions, privileges, and access control category and affects the video driver in the `Kernel`:

> An elevation of privilege vulnerability in the NVIDIA video driver could enable a local malicious application to execute arbitrary code within the context of the kernel.

The most affected Android OS layer is the `Kernel` with 212 instances, where the most impacted subsystem is the `video driver` (26). The second most affected layer is `Native Libraries` with 131 instances, 104 of which belong to the `Stagefright (media framework)` subsystem.

Top 2: AV:N/AC:H/Au:N/C:C/I:C/A:C 150 instances were identified with this vector, and more than 50% of them belong to the category permissions, privileges, and access control (79). Again, the most affected layer is the **Kernel** (94%), with the most impacted subsystem being the Wi-Fi driver (34). The vulnerabilities in this vector are the most complicated to exploit because require specialized access conditions (AC:H), e.g., suspicious or atypical user actions, or usage of social engineering; however, when exploited, the vulnerabilities with this vector have complete impact on confidentiality, integrity, and availability of the system (C:C/I:C/A:C).

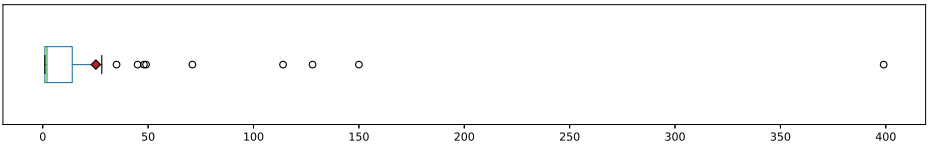


Fig. 9 Distribution of number of vulnerabilities per CVSS vs. 2.0 vector. The red diamond represents the mean, and the green line is the median

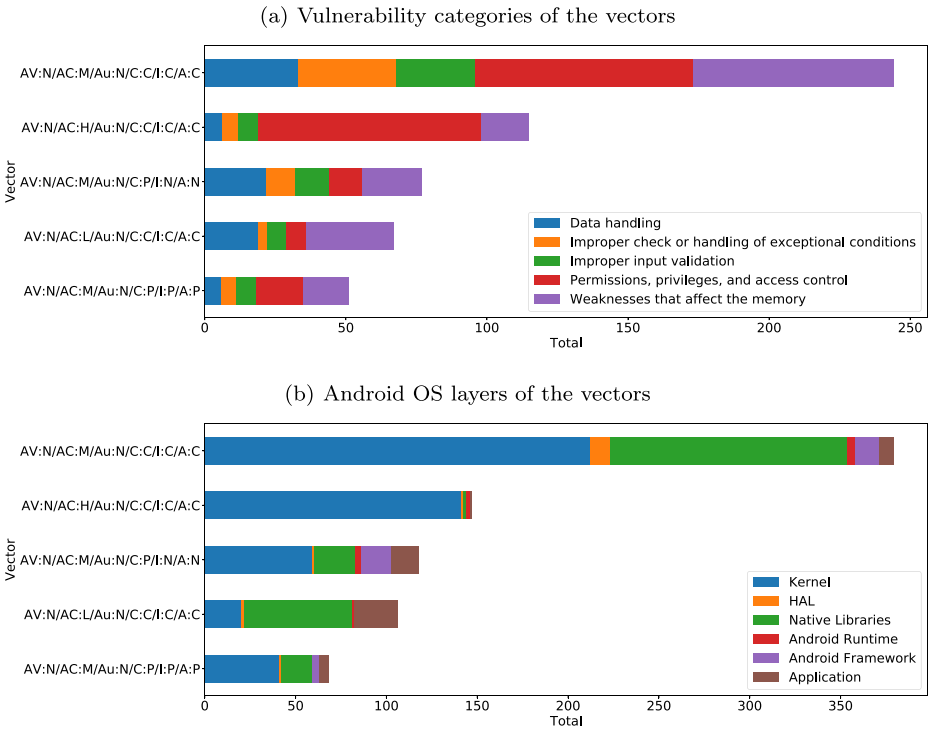


Fig. 10 Top CVSS 2.0 vectors and their distributions in terms of **a** vulnerability categories, and **b** Android OS layers

Top 3: AV:N/AC:M/Au:N/C:P/I:N/A:N More than 50% of the instances are in the following categories: data handling (22); weaknesses that affect the memory (21); permissions, privileges, and access control (12); and improper input validation (12). The vulnerabilities with this vector neither impact the integrity nor the availability of the system (I:N/A:N) and the information disclosure is partial without having control over the obtained information (C:P). With respect to the Android OS layers, 59 out of these 128 vulnerabilities are related to the kernel, being thus the most affected layer, followed by Native Libraries with 23 vulnerabilities, where 20 of these concern the Stagefright (media framework) subsystem. An example of vulnerability with this vector is `CVE-2017-0397`:

> An information disclosure vulnerability in id3/ID3.cpp in libstagefright in Mediaserver could enable a local malicious application to access data outside of its permission levels.
>
> Developers fixed this vulnerability in commit `7a3246b`, identifying that:
>
> several points in stagefrights mp3 album art code used strlen() to parse user-supplied strings that may be unterminated, resulting in reading beyond the end of a buffer.

Top 4: AV:N/AC:L/Au:N/C:C/I:C/A:C This vector is related to 114 vulnerabilities; 43,8% of these instances are grouped in the following categories: weaknesses that affect the memory (31); and data handling (19). In this case the two most affected layers are Native Libraries (59) and Application (24), and the most affected subsystems are Stagefright

(media framework) (52) and Adobe flash player (22), respectively. This is the only vector in the top-5 list that does not require specialized access conditions to exploit the vulnerability (AC:L); therefore, the vulnerabilities with this vector are the most dangerous in the dataset because have complete impact on confidentiality, integrity and availability (C:C/I:C/A:C), without requiring authentication (Au:N), and without the need for specialized access conditions (AC:L). `CVE-2016-2506` in the media framework has this vector, which means that the (i) vulnerability can be exploited without having specialized access to the system, (ii) there is total disclosure of the files, (iii) the system integrity is compromised, and (iv) there is a total shutdown of the system when the vulnerability is exploited:

> DRMExtractor.cpp in libstagefright in mediaserver in Android 4.x before 4.4.4, 5.0.x before 5.0.2, 5.1.x before 5.1.1, and 6.x before 2016-07-01 does not validate a certain offset value, which allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted media file.

Top 5: AV:N/AC:M/Au:N/C:P/I:P/A:P There are 71 vulnerabilities associated to this vector; 46,4% of these are grouped in the following categories: permissions, privileges, and access control (17); and weaknesses that affect the memory (16). Examples for this vector, from the permissions, privileges, and access control category, are identified with the ids `CVE-2017-0740`, CVE-2017-0741, `CVE-2017-0742`. The three vulnerabilities are an elevation of privilege vulnerability but in three different drivers: Broadcom networking driver, MediaTek gpu driver and MediaTek video driver.

4.3 RQ_3: Which are the Android Subsystems More Affected by Security Vulnerabilities?

Figure 11 depicts (using a heat-map style), the manually identified layers and subsystems of the Android OS impacted by 1,174 vulnerabilities. For building the heatmap, from the 1,235 vulnerabilities dataset, we excluded 41 in which we were not able to manually identify the layer, and 20 that were categorized as multi-layer vulnerabilities (later in this subsection we will talk about the unclear and the multi-layer vulnerabilities).

For the heatmap we used two color schemes: white-to-red for the layers, with white representing the lowest value and red the highest one; and white-to-yellow for the subsystems (i.e., internal boxes), with full yellow meaning that a subsystem is responsible for 100% of the vulnerabilities in the corresponding layer. Note that the subsystems' colors are normalized on the basis of the total vulnerabilities affecting a layer. Figure 11 also reports the percentage of vulnerabilities affecting each layer and subsystem/component.

Top 1: The Kernel It is the most frequently affected layer, with 615 of the 1,235WOUnclML vulnerabilities (52.39%).[15] It is worth noting that the `Kernel` layer in the Android Open Source Project is a fork of the original Linux Kernel; Android-specific changes have been made to the Linux Kernel to enable mobile features such as the memory "viking killer" and the Android Binder. We assigned "unclear" to the impacted subsystem/component for 32 of the `Kernel` vulnerabilities.

Most of the vulnerabilities in the Android OS Kernel are in the drivers for hardware components. In particular, we found that 471 (76.59%) of the `Kernel` vulnerabilities are related

[15] Compared to Linares-Vásquez et al. (2017), in our dataset we observed 354 new vulnerabilities in the Kernel that have been reported from November 2016 to August 2017.

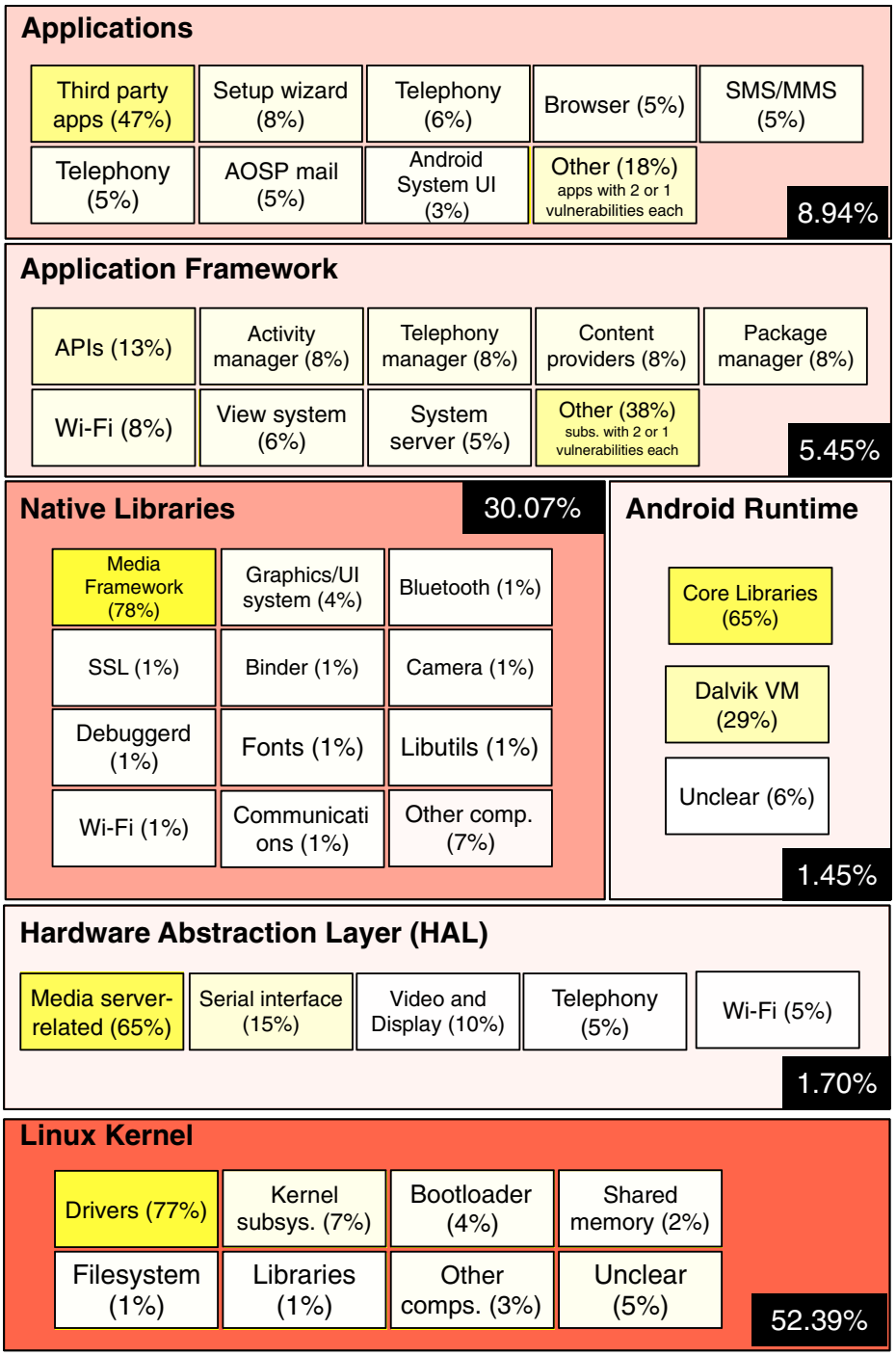


Fig. 11 RQ_3: Heat map of vulnerabilities in the Android layers/subsystems

to the drivers developed by Google and third-parties such as Qualcomm, Broadcom, Mediatek, and NVIDIA. The top-5 components impacted by vulnerabilities in kernel drivers are Wi-Fi (81 vulnerabilities), Video (59), Camera (49), Sound (49) and GPU (36). Concerning the vulnerability categories in the drivers, permissions, privileges, and access control is the most frequent type with 138 instances, followed by 73 instances of weaknesses that affect the memory; the top-3 position is for data handling with 63 instances.

The next two categories are improper input validation (32) and time state (24). We assigned "unclear" to the type of 64 vulnerabilities. The remaining 77 vulnerabilities were distributed across seven types: pointer issues, improper input validation, security features, improper check or handling of exceptional conditions, initialization and cleanup errors, indicator of poor quality code, behavioral problems and injection flaws.

The `Kernel` and its subsystems (e.g., sockets, profiling/performance, security, networking, ION) account together for 46 vulnerabilities. Permissions, privileges, and access control is again the most frequent category (15 vulnerabilities), followed by pointer issues (8) and weaknesses that affect the memory (5). Android-specific components (i.e., Google contributions to the kernel) such as Binder, ashmem/Shared memory, and aboot/Boot loader are the top-3 with 37 vulnerabilities. The most frequent vulnerabilities in Android-specific contributions to the kernel are data handling (10), improper check or handling of exceptional conditions (6), and permissions, privileges, and access control (6).

Top 2: The Native Libraries As described in Section 2, the `native libraries` layer contains Android-specific libraries like libstagefright (*a.k.a.,* Media framework) and third party libraries such as libc, bionic, and SSL. Both, Google and third-party native libraries have 353 out of 1,174 vulnerabilities (30.07%). This is mostly due to the `Media Framework` subsystem that has suffered of 274 vulnerabilities, including the set of issues known as "Stagefright" (Wikipedia 2017c; Nickinson 2015; Burgess 2016) that are sourced in the `Stagefright` library (libstagefright). Most of the vulnerabilities in the `Media Framework` are related to weaknesses that affect the memory (69), such as arrays access/writing, and memory management that lead to any type of overflow/underflow when accessing, writing, creating, and copying buffers (MITRE 2017a, b, c), and when performing integer operations (MITRE 2017d). Other frequent categories of vulnerabilities in the Media framework are improper check or handling of exceptional conditions (40) and **initialization and cleanup errors** (30).

The vulnerability `CVE-2015-3834`, reported as fixed in the August 2015 bulletin, is a representative example of weakness that affect the memory in the Media framework, and has the following CVE description:

> Multiple integer overflows in the BnHDCP::onTransact function in libstagefright allow attackers to execute arbitrary code via a crafted application that uses HDCP encryption, leading to a heap-based buffer overflow [...]

This vulnerability was fixed with the commit `c82e31a` that modifies the `IHDCP.cpp` file. The lack of buffer size validation when computing a buffer size, was leading to heap-based buffer overflows (MITRE 2017c) when creating an input buffer with the calculated size. Another example of a security issue in the `Media Framework` related to improper validation/restriction of operations within the bounds of a memory buffer is `CVE-2016-0815`:

> The MPEG4Source::fragmentedRead function in MPEG4Extractor.cpp in libstagefright in mediaserver [...] allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted media file [...]

The issue can be summarized as an out-of-bounds write (MITRE 2017k) generated when an array offset goes beyond the buffer size. The vulnerability was fixed in commit 5403587. Another example of vulnerability in the Media framework, but related to initialization and clean-up errors is CVE-2016-0829, reported in the March 2016 bulletin:

> The BnGraphicBufferProducer::onTransact function in libs/gui/IGraphicBufferConsumer.cpp in mediaserver in Android 4.x before 4.4.4, 5.x before 5.1.1 LMY49H, and 6.x before 2016-03-01 does not initialize a certain output data structure, which allows attackers to obtain sensitive information, and consequently bypass an unspecified protection mechanism, by triggering a QUEUE_BUFFER action [...]

The CVE-2016-0829 vulnerability is an example of missing initialization of resource (MITRE 2017n), in which an output buffer is not initialized with the memset C function.

The second most affected subsystem in the native libraries layer is the one related to the Graphics/UI system (14 vulnerabilities), which includes libraries such as libpng, libgui, OpenGL, and OpenJPEG. The Graphics/UI system in the native libraries layer has been affected by eight categories of vulnerabilities, having weaknesses that affect the memory as the top-one type with 5 vulnerabilities, followed by data handling (2). One example of vulnerability in the Graphics/UI system is CVE-2016-8332:

> A buffer overflow in OpenJPEG 2.1.1 causes arbitrary code execution when parsing a crafted image. An exploitable code execution vulnerability exists in the jpeg2000 image file format parser as implemented in the OpenJpeg library. A specially crafted jpeg2000 file can cause an out of bound heap write resulting in heap corruption leading to arbitrary code execution. For a successful attack, the target user needs to open a malicious jpeg2000 file. The jpeg2000 image file format is mostly used for embedding images inside PDF documents and the OpenJpeg library is used by a number of popular PDF renderers making PDF documents a likely attack vector.

Bluetooth and OpenSSL are top-3 in the native libraries with five vulnerabilities each. An interesting example here is an instance of indicator of poor quality code in OpenSSL reported in the May 2016 bulletin with the id CVE-2016-0705:

> Double free vulnerability in the dsa_priv_decode function in crypto/dsa/dsa_ameth.c in OpenSSL 1.0.1 before 1.0.1s and 1.0.2 before 1.0.2g allows remote attackers to cause a denial of service (memory corruption) or possibly have unspecified other impact via a malformed DSA private key.

The description reports a "double free", which means a double call to the free C function on the same memory address (i.e., with the same argument) (MITRE 2017j).

Top 3: The Applications Layer This layer contains apps shipped by Google and third-parties, as part of the AOSP project. The applications layer has been affected by 105 vulnerabilities (8.94%) located in 31 applications.[16] Concerning the third-party applications, the Adobe Flash Player is the most vulnerable app with 29 vulnerabilities; the next more vulnerable app is Firefox (five vulnerabilities); Nvidia Profiler, Widevine QSEE trustzone and Samsung OMACP are the top three apps with three vulnerabilities each. Concerning the apps developed by Google, Browser, Telephony, and Bluetooth have been the most vulnerable with five vulnerabilities each.

[16] Note that we only report numbers for vulnerabilities in the AOSP apps and reported as vulnerabilities in the NVD database.

Permissions, privileges, and access control issues are the main source of vulnerabilities in this layer with 27 instances. The top-2 category of vulnerabilities in the applications layer is security features (12 vulnerabilities), while the top-3 are improper check or handling of exceptional conditions and pointer issues with seven each. Vulnerabilities in the `Applications` layer are diverse in terms of types. For instance, `CVE-2011-2344` is an example from the security features category that affected the Android Picasa app, in particular with an *inadequate encryption strength* (MITRE 2017h) on sensitive data:

> Android Picasa in Android 3.0 and 2.x through 2.3.4 uses a cleartext HTTP session when transmitting the authToken obtained from ClientLogin, which allows remote attackers to gain privileges and access private pictures and web albums by sniffing the token from connections with picasaweb.google.com.

Other example of vulnerability in the applications layer, but from the behavioral problems category, is `CVE-2011-0680`, which affected the Android `Browser` and impacted 6 different versions of the OS (before 2.3.4):

> data/WorkingMessage.java in the Mms application in Android before 2.2.2 and 2.3.x before 2.3.2 does not properly manage the draft cache which allows remote attackers to read SMS messages intended for other recipients in opportunistic circumstances via a standard text messaging service.

The `CVE-2011-0680` vulnerability is an example of *information exposure through sent data* (MITRE 2017e) because of the lack of URIs validation in the `Browser` app.

Top 4: The Android Framework It is the layer that provides services and APIs to the application layer. The `Android framework` layer has been affected by 64 vulnerabilities (5.45%) belonging to 27 different subsystems; most of the vulnerabilities are instances of permissions, privileges, and access control (14), **security features** (10), and improper check or handling of exceptional conditions (7). Conversely to the `Libraries` layer that exhibits a non-diverse set of vulnerabilities (in terms of the type), the `Android Framework` has been affected by a diverse set of vulnerabilities including code injection (MITRE 2017o), overflows (MITRE 2017d), permission issues (MITRE 2017f), business logic errors (MITRE 2017l), missing authorizations (MITRE 2017m), and use of a risky cryptographic algorithm (MITRE 2017i), among others.

The Framework APIs (i.e., Android API) is the most impacted subsystem with eight vulnerabilities, followed by the activity manager, the telephony manager, the content providers, and the Wi-Fi with five vulnerabilities each. The package manager is the next in the list with four vulnerabilities. An example of vulnerability in the `Android Framework` from the category behavioral problems is `CVE-2016-2500`:

> Activity Manager in Android [...] does not properly terminate process groups, which allows attackers to obtain sensitive information via a crafted application [...]

This vulnerability was a consequence of an invocation to the `killProcessGroup` method in the `ActivityManagerService.java` file using wrong parameters. Another example of vulnerability introduced by business logic errors in the `Android Framework` layer is `CVE-2016-3923`, in particular in the `Accessibility Services`, that:

> mishandle motion events, which allows attackers to conduct touchjacking attacks and consequently gain privileges via a crafted application.

Top 5: The Hardware Abstraction Layer 20 of the 1,174 vulnerabilities are in the HAL (1.70%). 13 of these vulnerabilities belong to media server-related interfaces; 3 to the serial interface; and the remaining vulnerabilities (4) are in the telephony, wifi, and video interfaces. The vulnerabilities are representative of the indicator of poor quality code (4), weaknesses that affect the memory (4), pointer issues (3), initialization and cleanup errors (2), and improper check or handling of exceptional conditions (1); for 6 of the vulnerabilities we were not able to identify the subsystem.

Top 6: The Runtime Only 17 vulnerabilities are related to the `Runtime` layer (1.45%). The top category of vulnerability here is security features with five instances. The next categories in the list are behavioral problems (3); Data handling (2); improper input validation (2); indicator of poor quality code (2); permissions, privileges, and access control (2); and weaknesses that affect the memory (1).

Most of the vulnerabilities in the `Runtime` affect core libraries such as `Conscrypt` (5), `Apache Harmony/Java SE` (3), `Dalvik VM` (3), and `Bouncy Castle` (2). `Conscrypt` is a Java Security Provider used by Android as one of its core libraries hosted with the `Runtime` layer. An example of vulnerability in `Conscrypt` is `CVE-2016-0818`, which is an *improper following of a certificate's chain of trust* (MITRE 2017g):

> The caching functionality in the TrustManagerImpl class in TrustManagerImpl.java in Conscrypt in Android 4.x before 4.4.4, 5.x before 5.1.1 LMY49H, and 6.x before 2016-03-01 mishandles the distinction between an intermediate CA and a trusted root CA, which allows man-in-the-middle attackers to spoof servers by leveraging access to an intermediate CA to issue a certificate [...]

Another example, also in `Conscrypt`, is CVE-2016-2462. This vulnerability is a *business logic error* (MITRE 2017l), in particular because of missing an assignment statement in the `OpenSSLCipher.java` file:

> OpenSSLCipher.java in Conscrypt in Android 6.x before 2016-05-01 mishandles updates of the Additional Authenticated Data (AAD) array, which allows attackers to spoof message authentication via unspecified vectors, aka internal bug 27371173.

Multi-layer and Unclear-layer Vulnerabilities Finally, we assigned "unclear" to the corresponding layer in 41 cases, and 20 vulnerabilities (not included in the heatmap) were manually assigned to different layers because the files modified in the patches belong to different layers of the Android OS stack; those multi-layer vulnerabilities are reported in Table 3.

Vulnerabilities impacting Android OS layers over time Figure 12 shows the reported vulnerabilities (classified according to the Android OS layer they belong to) and grouped by semester (as done in RQ_1). Note that we excluded 16 vulnerabilities affecting multiple Android OS Layers at the same time and we also discarded 33 vulnerabilities for which we were not able to identify the impacted Android OS Layer.

`Kernel` and `Native Libraries` are the layers with the higher number of vulnerabilities across all the semesters under analysis (except for 2015-2 in the case of the kernel); they represent between 67.8% and 91.8% of the total number of vulnerabilities every semester. Also note that the kernel is the only layer showing a continuous increasing trend, while this does not hold for the native libraries layer. The other four layers tend to be affected by a stable low number of vulnerabilities. Additional analyses looking at the evolution of vulnerabilities at subsystem level are available in our online appendix (Mazuera-Rozo et al. 2017).

Table 3 Vulnerabilities with patches modifying files in different Android OS layers (Multi-layer Vulnerabilities)

CVE-ID	Category	Type	Layers
CVE-2010-4832	Security features	CWE-320: Key Management Errors	Android Framework, Native Libraries
CVE-2015-3843	Data handling	CWE-190: Integer Overflow or Wraparound	Application, Android Framework
CVE-2016-2496	Permissions, privileges, and access control	CWE-862: Missing Authorization	Application, Android Framework
CVE-2016-3760	Unclear	Unclear	HAL, Native Libraries, Application
CVE-2016-3889	Permissions, privileges, and access control	CWE-862: Missing Authorization	Application, Android Framework
CVE-2016-10335	Permissions, privileges, and access control	CWE-265: Privilege / Sandbox Issues	Kernel, Native Libraries
CVE-2016-10339	Data handling	CWE-200: Information Exposure	Kernel, Android Framework
CVE-2017-0382	Permissions, privileges, and access control	CWE-269: Improper Privilege Management	Android Framework, Native Libraries
CVE-2017-0390	Weaknesses that affect the memory	CWE-125: Out-of-bounds Read	Android Framework, Native Libraries
CVE-2017-0399	Weaknesses that affect the memory	CWE-127: Buffer Under-read	Native Libraries, HAL
CVE-2017-0400	Weaknesses that affect the memory	CWE-127: Buffer Under-read	Native Libraries, HAL
CVE-2017-0401	Weaknesses that affect the memory	CWE-127: Buffer Under-read	Native Libraries, HAL
CVE-2017-0402	Weaknesses that affect the memory	CWE-127: Buffer Under-read	Native Libraries, HAL
CVE-2017-0418	Weaknesses that affect the memory	CWE-787: Out-of-bounds Write	Android Framework, Native Libraries, HAL
CVE-2017-0494	Weaknesses that affect the memory	CWE-122: Heap-based Buffer Overflow	Application, Native Libraries
CVE-2017-0499	Unclear	Unclear	Android Framework, Native Libraries
CVE-2017-0554	Permissions, privileges, and access control	CWE-275: Permission Issues	Application, Android Framework
CVE-2017-0665	Improper check or handling of exceptional conditions	CWE-391: Unchecked Error Condition	Android Framework, Native Libraries
CVE-2017-0666	Unclear	Unclear	Android Framework, Native Libraries
CVE-2017-0738	Weaknesses that affect the memory	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	Android Framework, Native Libraries, HAL

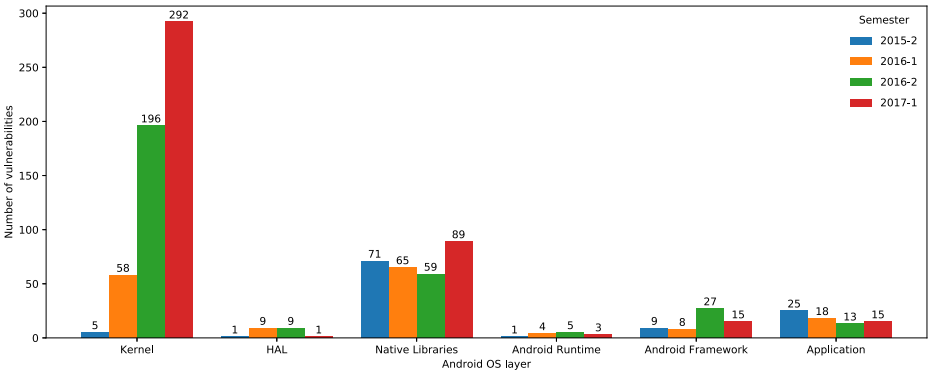


Fig. 12 Changes in number of vulnerabilities according to the Android OS layer

4.4 RQ$_4$: How Long does it Take to Fix Security Vulnerabilities in Android?

Figure 13 depicts the forest plots reporting the survivability of Android-related vulnerabilities (i.e., the number of days between the vulnerability introduction and its fixing). As explained in Section 3.1, we report the minimum (green) and the maximum (red) survivability intervals as computed with the SZZ algorithm. In each forest plot the square represents the average value of the distribution, while the line passing through it depicts the 95% confidence interval. Figure 13 shows the survivability intervals when considering all the analyzed vulnerabilities together (top part of Fig. 13) as well as when grouping them by severity (low, medium, and high). The black line shown for the overall set of vulnerabilities depicts the results of the random effects model (Christensen 2011), used in meta-analysis to combine the results of different studies in a single result outcome. In our case, the set of "different

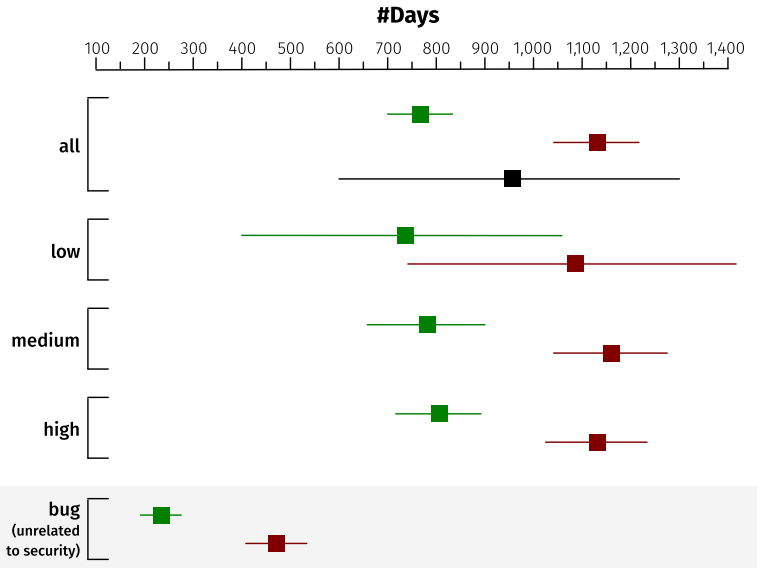


Fig. 13 Survivability in days of Android-related vulnerabilities. Green (red) depicts minimum (maximum) estimates at 95% confidence interval. Black shows the results of the random effect model

studies" includes the survivability estimates when considering the minimum (*study I*) and the maximum (*study II*) survivability. Finally, the bottom grey part of Fig. 13 depicts the minimum and maximum survivability intervals for the 331 bugs unrelated to security issues that we selected in order to compare the survivability of software vulnerabilities with that of other types of bugs in Android.

The first thing that leaps to the eyes from the analysis of Fig. 13 is the very long survivability of the analyzed Android-related vulnerabilities. Indeed, even when considering the most conservative results (i.e., the minimum estimated survivability—green line), the number of days needed to fix an introduced vulnerability is, on average, 770 (it grows to 951 for the random effects model, and to 1,131 for the maximum estimated survivability). A famous case to highlight here is the set of media server vulnerabilities know as "Stagefright", which has been recognized by the community as the worst ever discovered issue in the Android OS; about 95% of Android devices were in risk of remote execution because of stagefright. Although the stagefright issue was originally reported with 8 vulnerabilities (`CVE-2015-1538`, `CVE-2015-1539`, `CVE-2015-3824`, `CVE-2015-3826`, `CVE-2015-3827`, `CVE-2015-3828`, `CVE-2015-3829`, `CVE-2015-3864`), almost 300 patches have been generated for issues in the media server and libstagefright components. From the original 8 patches, we were able to identify survivability times for seven of the vulnerabilities using the SSZ algorithm. On average, the survivability time for the seven vulnerabilities is 631 days (bot the minimum and the maximum, since a single introducing commit was identified for all of them), with a maximum of 1,359 days for `CVE-2015-1539`.

While discussing the results related to the survivability of the vulnerabilities, it is important to note that this is not the number of days needed to fix a vulnerability after *it has been reported*, but after *it has been introduced*. This means that a vulnerability could remain unnoticed in the system for years before being identified, possibly exploited, and then fixed. While it would have been interesting to also analyze the time actually needed for the vulnerability fixing (i.e., the number of days between the vulnerability reporting and fixing), we did not find a way to reliably identifying the reporting date.

The very long survivability of the Android-related vulnerabilities was surprising for us at a first sight, especially due to the young age of the Android OS. Thus, we manually inspected 30 randomly selected vulnerabilities in order to verify whether strong imprecisions of the SZZ algorithm were there affecting our findings. Note that such a sample is not statistically significant, but just meant to show qualitative examples about the extracted data.

Overall, we found the estimates provided by the SZZ algorithm to be quite precise. In particular, in 17 of the inspected cases the SZZ identified a single commit as the vulnerability-fix-inducing one. In 15 cases the identified commit was correct. In the other two cases we found imprecisions that can be explained by discussing the case of the vulnerability `CVE-2015-3867`, caused by a possible integer overflow. The vulnerability has been fixed in commit `7e9ac35` and the SZZ algorithm identifies as the only bug-inducing change, a commit performed four days before that introduced the line of code then fixed in `7e9ac35`. The problem here is that this line of code (i.e., `if (chunk_data_size >= SIZE_MAX - 1)`) was a first attempt to fix the `CVE-2015-3867` vulnerability that already affected the system. Thus, the SZZ is underestimating the survivability of the vulnerability, considering the first fixing attempt as the bug-inducing commit. While we expect other imprecisions of this type in our dataset, it is worth noting that this means that the very long survivability we observed is likely to be an underestimation.

In the remaining 13 cases we manually analyzed, multiple commits were identified by the SZZ algorithm as the possible responsible for the vulnerability introduction. In all

these cases, either the minimum or the maximum vulnerability estimate was correct. In the following, we discuss some examples of manually inspected vulnerabilities.

The vulnerability `CVE-2015-1538` has been reported in the August 2015 security bulletin and is described as follows:

> Integer overflow in the SampleTable::setSampleToChunkParams function in libstagefright in Android before 5.1.1 LMY48I allows remote attackers to execute arbitrary code [...]

Such a vulnerability has been fixed in the commit `cf1581c` made on the 8th April 2015, having commit message *Fix several ineffective integer overflow checks* and modifying the file `libstagefright/SampleTable.cpp`. By inspecting the diff of such a commit, three lines were changed to fix the integer overflows (Aosp commit 2015). The SZZ algorithm correctly identifies the commit `edd4a76e` performed on the 28th July 2014 as the vulnerability-inducing commit (thus, the vulnerability survived in the system for 254 days). Indeed, in such a commit the three lines causing the integer overflows and then fixed were introduced all together, as it can be seen from the commit diff (Aosp commit 2018b). Note that this is one of those cases in which the SZZ algorithm identified a single commit as the responsible for inducing the vulnerability-fix. This was the case for 181 out of the 331 vulnerabilities (55%) considered in this research question.

For the vulnerability `CVE-2015-6608` we identified instead multiple commits as the possible responsible for the vulnerability introduction. This vulnerability is described as follows:

> [...] allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) [...]

The vulnerability has been fixed in the commit `8ec845c` (commit note: *stagefright: check IMemory::pointer() before using the allocation*) made on the 15th May 2015 and modifying two lines (Aosp commit 2018a) in `media/libstagefright/ACodec.cpp`. These two lines were modified for the last time by two different commits, one performed on the 21st February 2012 (i.e., `5778822`) and one performed on the 2nd May 2013 (i.e., `054e734`). Each of these commits introduced one of the two lines then fixed in `8ec845c` thus, they were both correctly identified as vulnerability-inducing commits.

In this case, the commit `054e734` contributes to the "minimum survivability distribution" depicted in green in Fig. 13 (the survivability is 742 days), while `5778822` contributes to the "maximum survivability distribution" depicted in red in Fig. 13 (survivability=1,179 days). Clearly, in this case the correct survivability estimate is 1,179, since the vulnerability was there (at least in part) since the 21st February 2012.

When looking for the survivability of vulnerabilities having different severity levels, we were not able to identify any clear trend: It is not possible to assert that vulnerabilities having a higher severity have a higher/lower survivability with respect to those having a lower severity (or *vice versa*). This is visible both from the forest plots (see Fig. 13) and confirmed by the statistical analysis, in which we did not observe any significant difference, with all the adjusted p-values higher than 0.05 (see Table 4).[17]

Finally, when comparing the survivability of vulnerabilities to that of other types of bugs unrelated to security, we found that the former have a much longer survivability (see

[17] Note that all p-values equal 1.0 after the holm correction procedure. Before that they were in any case all higher than 0.7.

Table 4 RQ_4: Survivability of vulnerabilities having different severity levels: Mann-Whitney test (adj. p-value) and Cliff's Delta (d)

Test	adj. p-value	d
Minimum Estimates		
Low *vs* medium	1.00	−0.055 (Negligible)
Low *vs* high	1.00	−0.045 (Negligible)
Medium *vs* high	1.00	0.018 (Negligible)
Maximum Estimates		
Low *vs* medium	1.00	0.015 (Negligible)
Low *vs* high	1.00	−0.035 (Negligible)
Medium *vs* high	1.00	−0.05 (Negligible)

Fig. 13). The difference is statistically significant for both minimum and maximum estimates (p-values < 0.001) with a large effect size ($d = 0.67$ for minimum and $d = 0.59$ for maximum estimates). Our conjecture is that vulnerabilities are more difficult to spot than other types of bugs, thus exhibiting longer survivability. This is not the case for other types of bugs, especially for the functional ones, that are easier to spot during testing activities or when running the system.

5 Threats to Validity

Threats to construct validity concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important kind of threat for our study, and is related to:

RQ_1, RQ_2, and RQ_3: Subjectivity in the manual classification. We identified through manual analysis the types of vulnerabilities (RQ_1) and the subsystems (RQ_3) they affect, and also use this data in the results discussion of RQ_2. To mitigate subjectivity bias in such a process, the authors manually analyzed the vulnerabilities in couples to allow cross validation. For instance, A_1 checked the vulnerability types and the impacted subsystems assigned by A_2 and *vice versa*. Finally, the authors discussed the cases of disagreement, reaching an agreement on the correct classification needed. Also, when the type of the vulnerability and/or the impacted subsystem was unclear, we preferred to exclude the vulnerability from the study rather than risking to introduce imprecisions.
RQ_4: Approximations due to identifying bug-inducing commits using the SZZ algorithm (Kim et al. 2008). We used heuristics to limit the number of false positives, for example excluding blank and comment lines from the set of bug-inducing changes. Also, we computed both the minimum and the maximum survivability estimates on the basis of the SZZ outcome, showing that in any case the main outcome of our study did not change: Android-related vulnerabilities survive for long time. Moreover, the manual analysis performed on some vulnerabilities confirmed the validity of our experimental design to assess the survivability of vulnerabilities.
RQ_4: Imprecision due to tangled code changes (Herzig and Zeller 2013). We cannot exclude that some vulnerability-fixing commits grouped together tangled code changes,

of which just a subset was focusing on the vulnerability fix. This would result in imprecisions when running the SZZ algorithm on the fixing commit. Again, by presenting both the minimum and the maximum survivability estimates such a risk is mitigated.

Threats to internal validity concern external factors we did not consider that could affect the variables and the relations being investigated. When analyzing the survivability of vulnerabilities (RQ_4) we considered the severity of the vulnerability as a confounding factor to be controlled.

We are aware that many other factors could influence the survivability, and we plan to analyze them in future work. To reinforce the internal validity, when possible, we integrated the quantitative analysis with a qualitative one.

Threats to conclusion validity concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences. In the case of RQ_4 we used meta-analysis to avoid having a biased conclusion towards the maximum or minimum survivability times reported by the SZZ algorithm.

Threats to external validity concern the generalization of results. All RQs but RQ_4 considered 1,235 vulnerabilities, while the RQ_4's findings are based on the analysis of 331 vulnerabilities due to the need for identifying the vulnerability-fixing commit (see Section 3.1 for details). Clearly, the number of Android-related vulnerabilities that can be studied will increase in the future, and larger replications of our study will be possible.

6 Learned Lessons and Future Work

We analyzed 1,235 Android-related vulnerabilities from five different perspectives: (i) the types of the vulnerabilities and their hierarchical relationships; (ii) the most frequent CVSS vectors that describe the vulnerabilities; (iii) the layers and components from the Android software stack impacted by the vulnerabilities; (iii) the survivability of the vulnerabilities (i.e., the time required to fix a vulnerability since its introduction); and (iv) the evolution of vulnerabilities across the Android OS history. To the best of our knowledge, this is the largest empirical study of Android OS vulnerabilities not only in terms of vulnerabilities but also in terms of aspects analyzed in the vulnerabilities dataset. Compared to previous studies on Android OS vulnerabilities such as Jimenez et al. (2016) and Linares-Vásquez et al. (2017) there is a increase of +1193 and +575 respectively, in the number of analyzed vulnerabilities. Moreover, ours is the first study that analyzes the evolution of the vulnerabilities and the features of the top CVSS vectors.

In terms of findings, while Jimenez et al. (2016) report that Android OS vulnerabilities are rooted on incorrect implementation of features and incorrect handling of authorization and input, our study — with a more comprehensive taxonomy — shows a diverse set of issues at different granularity levels such as *permissions, privileges, and access control*, *weaknesses that affect memory*, *data handling*, and *improper check or handling of exceptional conditions*. Also, Jimenez et al. (2016) reported the vulnerable Android OS

components for the 42 vulnerabilities but in terms of the roles/features provided by the components; in our case, we report the most vulnerable OS layers and internal components, which provides a more detailed view of the OS internals that require more attention from the Android OS developers in terms of security. Jimenez et al. (2016) also analyzed the type of changes in the patches at statement level (e.g., add if-then-else). We report in this study the changes organized by higher level categories (see Fig. 7).

The main findings of our study can be summarized as follows:

1. Most of the Android OS vulnerabilities are related to improper access control, improper restriction of operations in the bounds of memory buffers, issues processing data (*e.g.,* numeric, type, and string errors), and improper input validations (from $\mathbf{RQ}_1$).
2. Most of the vulnerabilities (at least 69,79%) are exploitable remotely and do not require authentication of the system; in addition, in at least 53,68% of the vulnerabilities there is total disclosure of the system files, total compromise of the system integrity (i.e., the attacker is able to modify any file NIST 2015), and there can be a total shutdown of the affected resource or of the complete system (from $\mathbf{RQ}_2$).
3. The *kernel* and the *native libraries* layers are the ones most affected by the vulnerabilities (82.46% of the analyzed vulnerabilities impact these two layers). *Kernel drivers* and the *media framework* are the top affected subsystems (from $\mathbf{RQ}_3$).
4. A vulnerability remains unnoticed in the system for 770 days, on average (considering the conservative model) before being identified, possibly exploited, and then fixed (from $\mathbf{RQ}_4$), without considering those vulnerabilities that are neither discovered nor reported.
5. The number of reported vulnerabilities has been increasing since 2015. It is clear that the kernel layer has kept a continuously increasing tendency in the number of reported vulnerabilities since 2015. Moreover, the vulnerabilities in the categories permissions, privileges, and access control, weaknesses that affect the memory, and data handling categories are the top-3 in terms of increasing tendency since 2015 (from $\mathbf{RQ}_1$).

Based on the aforementioned findings, researchers and practitioners should invest in designing approaches and implementing processes for the early detection of vulnerabilities in the Android OS. As presented in Section 2, there is a plethora of approaches and tools for detecting vulnerabilities at the Applications level; however, little effort has been devoted to the identification of vulnerabilities at the Android OS level. This study is a first effort to present the magnitude of the problem and a potential research/action agenda aimed at reducing not only the volume of vulnerabilities in the Android OS but also their impact and survivability. In particular, based on the main findings of our study, we propose the following set of actions:

1. Our findings indicate that third-party hardware drivers are the components with most of the vulnerabilities in the Android OS, thus suggesting the need to strengthen verification & validation tasks. The usage of automated vulnerabilities detection tools could be enforced as part of the continuous integration pipeline before committing to the AOSP repository.
2. We showed that Android vulnerabilities survive for long time in the code base. This stresses the importance for researchers to invest effort in the development of automatic vulnerability detectors tailored for the mobile world. The taxonomy of vulnerabilities presented in this paper can be used as a reference for the definition of the types of vulnerabilities such detectors should target. In terms of efforts prioritization, the

implementations can be focused on providing detectors for (i) the vulnerabilities in the most impacted layers (i.e., kernel and native libraries), (ii) the vulnerabilities belonging to the top CVSS vectors reported in this paper, or (iii) the most frequent vulnerability types we found. The design and implementation of effective vulnerability detection tools for mobile OS/apps is part of our future research agenda, in particular the ones for detecting vulnerabilities in the permissions, privileges, and access control and weaknesses that affect the memory categories.

3. The manual analysis of the patches implemented to fix the vulnerabilities highlighted as many of them can be easily addressed with simple code transformations, for example by adding pre-condition checks (e.g., boundary and null checks), or by properly handling permissions in the manifest files. Given the wide availability of data in code sharing platforms such as GitHub, researchers could apply advanced machine learning techniques such as Deep Learning to automatically "learn" and apply these code transformations, similarly to what has been recently done by Tufano et al. (2018) for automatic bug-fixing. The authors showed that a Neural Machine Translation (NMT) model can automatically learn from real bug-fixing activities how to fix bugs in the same way as humans do. The model works in 9% of real buggy code provided as input. We believe that a technique specialized in fixing vulnerabilities (thus only learning from commits fixing vulnerabilities) could achieve even higher performance, given the simplicity of the code transformations often applied to fix them (see Fig. 7).
4. The vulnerability categories in the Android OS layers show some interesting patterns. While permissions, privileges, and access control issues are traversal to all the layers, the vulnerabilities are more related to security features and improper check handling of exceptional conditions when the layer is closer to the user; when the layer is closer to the hardware, weaknesses that affect memory and pointer issues are the most frequent categories; and, in the middle of the stack, **initialization and cleanup errors** are the most common. These "patterns" are directly related to the choice of the implementation language used in a layer and the type of features implemented. Thus, the usage of secure coding practices could be promoted/supported/enforced according to the Android OS layer a developer is working on.
5. If teaching/training secure coding practices is not an option, the secure coding practices could be enforced, for example, via *just-in-time* quality control techniques statically analyzing the code contributed to the Android OS in each commit activity. Therefore, future work could be devoted to develop secure coding-oriented linters and static analyzers that could be specialized according to the OS layer the developers are contributing to. Also, mobile OS developers could consider the usage of modern programming languages embedding mechanisms for secure coding (e.g., Rust 2013).
6. Finally, it is worth highlighting the most common features of the vulnerabilities as reported by the top-5 CVSS vectors. Most of the vulnerabilities (i) are only exploited remotely, (ii) do not require authentication, and (iii) have complete impact on integrity/confidentiality/availability. This could be a "warning signal" for the OS designers. Thus, new efforts aimed at redesigning or updating the Android OS architecture (or a new OS for Android) could be oriented to implement mechanisms that control or avoid the aforementioned issues.

Given the wide diffusion of Android devices nowadays and the impact that security vulnerabilities can have on private data, we strongly believe in the need of investigating the

above listed action items, and we hope that our paper can represent a stepping stone in this direction.

References

Aosp commit cf1581c66c2ad8c5b1aaca2e43e350cf5974f46d (2017a) http://tinyurl.com/hxqdp7f
Aosp commit 8ec845c8fe0f03bc57c901bc484541bdd6a7cf80 (2017b) http://tinyurl.com/hvndh7r
Aosp commit edd4a76eb4747bd19ed122df46fa46b452c12a0d (2017c) http://tinyurl.com/hkw399d
Ahmad W, Kästner C, Sunshine J, Aldrich J (2016) Inter-app communication in android: Developer challenges. In: Proceedings of the 13th international conference on mining software repositories, MSR '16. ACM, New York, pp 177–188. https://doi.org/10.1145/2901739.2901762
Anderson B et al (2016) Hpe security research. cyber risk report 2016. Tech. rep., Hewlett Packard
Armis (2017) The attack vector "blueborne" exposes almost every connected device. https://www.armis.com/blueborne/
Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14. ACM, New York, pp 259–269. https://doi.org/10.1145/2594291.2594299
Avdiienko V, Kuznetsov K, Gorla A, Zeller A, Arzt S, Rasthofer S, Bodden E (2015) Mining apps for abnormal usage of sensitive data. In: ICSE'15, pp 426–436. http://dl.acm.org/citation.cfm?id=2818754.2818808
Backes M, Bugiel S, Derr E (2016) Reliable third-party library detection in android and its security applications. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, CCS '16. ACM, New York, pp 356–367. https://doi.org/10.1145/2976749.2978333
Bagheri H, Kang E, Malek S, Jackson D (In Press) A formal approach for detection of security flaws in the android permission system. Springer Journal on Formal Aspects of Computing
Bagheri H, Sadeghi A, Garcia J, Malek S (2015) Covert: compositional analysis of android inter-app permission leakage. IEEE Trans Softw Eng 41(9):866–886. https://doi.org/10.1109/TSE.2015.2419611
Beres D (2015) 'cowboy adventure' game infects up to 1 million android users with malware. http://www.huffingtonpost.com/2015/07/10/android-security_n_7765842.html
Bhosale A (2014) Precise static analysis of taint flow for android application sets. Master's thesis, Heinz College Carnegie Mellon University
Brady P (2008) Anatomy & physiology of an android. https://sites.google.com/site/io/anatomy–physiology-of-an-android
Burgess M (2016) Millions of android devices vulnerable to new stagefright exploit. http://www.wired.co.uk/article/stagefright-android-real-world-hack
Cao C, Gao N, Liu P, Xiang J (2015) Towards analyzing the input validation vulnerabilities associated with android system services. In: Proceedings of the 31st annual computer security applications conference, ACSAC 2015. ACM, New York, pp 361–370. https://doi.org/10.1145/2818000.2818033
Castellanos JH, Wuchner T, Ochoa M, Rueda S (2016) Q-floid: Android malware detection with quantitative data flow graphs. In: Singapore cyber-security conference (SG-CRC). IOS Press, pp 13–26
Christensen R (2011) Plane Answers to Complex Questions: The Theory of Linear models, 4th edn. Springer Texts in Statistics Springer, Berlin
Conover WJ (1998) Practical Nonparametric Statistics, 3rd edn. Wiley, New York
Corporation M (2017) Cve common vulnerabilities and exposures. http://cve.mitre.org
Cumming G (2011) Introduction to the new Statistics: Effect sizes, confidence intervals, and Meta-Analysis. Routledge, Evanston
Cve-2012-6636 (2017) https://www.cvedetails.com/cve/cve-2012-6636
Dimjaševic M, Atzeni S, Ugrina I, Rakamaric Z (2015) Android malware detection based on system calls
Drake JJ, Lanier Z, Mulliner C, Fora PO, Ridley SA, Wicherski G (2014) Android hacker's handbook. Wiley, New York
Enck W, Gilbert P, Chun BG, Cox L, Jung J, McDaniel P, Sheth AN (2010) Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX

conference on operating systems design and implementation, OSDI'10. USENIX Association, Berkeley, pp 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971
Enck W, Ongtang M, McDaniel P (2009) On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on computer and communications security, CCS '09. ACM, New York, pp 235–245. https://doi.org/10.1145/1653662.1653691
Fahl S, Harbach M, Muders T, Baumgärtner L., Freisleben B, Smith M (2012) Why eve and mallory love android: an analysis of android ssl (in)security. In: Proceedings of the 2012 ACM conference on computer and communications security, CCS '12. ACM, New York, pp 50–61. https://doi.org/10.1145/2382196.2382205
Fattori A, Tam K, Khan SJ, Cavallaro L, Reina A (2014) CopperDroid: On the Reconstruction of Android Malware Behaviors. Tech. rep. Royal Holloway University of London
FIRST Organization (2019) Common vulnerability scoring system sig. https://www.first.org/cvss
for Standardization IO (2011) Iso 27005 information security risk management
Garcia J, Hammad M, Ghorbani N, Malek S (2017) Automatic generation of inter-component communication exploits for android applications. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017. ACM, New York, pp 661–671. https://doi.org/10.1145/3106237.3106286
Gasior W, Yang L (2012) Exploring covert channel in android platform. In: 2012 international conference on cyber security, pp 173–177. https://doi.org/10.1109/CyberSecurity.2012.29
Ghafari M, Gadient P, Nierstrasz O (2017) Security smells in android. In: 2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM), pp 121–130. https://doi.org/10.1109/SCAM.2017.24
Gilbert P, Chun BG, Cox LP, Jung J (2011) Vision: automated security validation of mobile apps at app markets. In: Proceedings of the second international workshop on mobile cloud computing and services, MCS '11. ACM, New York, pp 21–26. https://doi.org/10.1145/1999732.1999740
Gorla A, Tavecchia I, Gross F, Zeller A (2014) Checking app behavior against app descriptions. In: ICSE'14, pp 1025–1035. https://doi.org/10.1145/2568225.2568276
Google (2016) Android security 2015 year in review. https://static.googleusercontent.com/media/source.android.com/en//security/reports/Google_Android_Security_2015_Report_Final.pdf
Google (2017a) Android security bulletins. https://source.android.com/security/bulletin/
Google (2017b) Platform architecture. https://developer.android.com/guide/platform/index.html
Graf J, Hecker MMM (2015) Jodroid: Adding android support to a static information flow control tool. In: Working conference on programming languages
Grissom RJ, Kim JJ (2005) Effect sizes for research: a broad practical approach, 2nd edn. Lawrence Earlbaum Associates, New Jersey
Hedges LV, Olkin I (1985) Statistical methods for Meta-Analysis. Academic Press, New York
Herzig K, Zeller A (2013) The impact of tangled code changes. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, pp 121–130
Holm S (1979) A simple sequentially rejective Bonferroni test procedure. Scand J Stat 6:65–70
Huang H, Zhu S, Chen K, Liu P (2015) From system services freezing to system server shutdown in android: All you need is a loop in an app. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security, CCS '15. ACM, New York, pp 1236–1247. https://doi.org/10.1145/2813606
Jimenez M, Papadakis M, Bissyandé TF, Klein J (2016) Profiling android vulnerabilities. In: 2016 IEEE International conference on software quality, reliability and security (QRS), pp 222–229. https://doi.org/10.1109/QRS.2016.34
Kantola D, Chin E, He W, Wagner D (2012) Reducing attack surfaces for intra-application communication in android. In: Proceedings of the second ACM workshop on security and privacy in smartphones and mobile devices, SPSM '12. ACM, New York, pp 69–80. https://doi.org/10.1145/2381934.2381948
Kim S, James Whitehead Jr E, Zhang Y (2008) Classifying software changes: clean or buggy? IEEE Trans Softw Eng 34(2):181–196
Lal S, Sureka A (2012) Comparison of seven bug report types: a case-study of google chrome browser project. In: 2012 19th asia-pacific software engineering conference, vol 1, pp 517–526. https://doi.org/10.1109/APSEC.2012.54
Lee S, Hwang S, Ryu S (2017) All about activity injection: Threats, semantics, and detection. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017. IEEE Press, Piscataway, pp 252–262. http://dl.acm.org/citation.cfm?id=3155562.3155597
Li GK (2010) Computing inter-rater reliability and its variance in the presence of high agreement. Br J Math Stat Psychol 61(1):29–48. https://doi.org/10.1348/000711006X126600

Linares-Vásquez M, Bavota G, Escobar-Velásquez C (2017) An empirical study on android-related vulnerabilities. In: Proceedings of the 14th international conference on mining software repositories, MSR '17. IEEE Press, Piscataway, pp 2–13. https://doi.org/10.1109/MSR.2017.60

LLC PI (2014) The security impact of mobile device use by employees. Tech. rep., Ponemon Institute

Lu L, Li Z, Wu Z, Lee W, Jiang G (2012) Chex: statically vetting android apps for component hijacking vulnerabilities. In: ACM Conference on computer and communications security, pp 229–240

Mazuera-Rozo A, Bautista-Mora J, Linares-Vásquez M, Rueda S, Bavota G (2017) Replication package: "The Android OS Stack and its Vulnerabilities: An Empirical Study". http://ml-papers.gitlab.io/android.vulnerabilities-2017/appendix/

Mell P, Scarfone K, Romanosky S (2007) A Complete Guide to the Common Vulnerability Scoring System Version 2.0, 2.0 edn

MITRE (2017a) Cwe-120: Buffer copy without checking size of input ('classic buffer overflow'). https://cwe.mitre.org/data/definitions/120.html

MITRE (2017b) Cwe-121: Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html

MITRE (2017c) Cwe-122: Heap-based buffer overflow. https://cwe.mitre.org/data/definitions/122.html

MITRE (2017d) Cwe-190: Integer overflow or wraparound. https://cwe.mitre.org/data/definitions/190.html

MITRE (2017e) Cwe-201: Information exposure through sent data. https://cwe.mitre.org/data/definitions/201.html

MITRE (2017f) Cwe-275: Permission issues. https://cwe.mitre.org/data/definitions/275.html

MITRE (2017g) Cwe-296: Improper following of a certificate's chain of trust. https://cwe.mitre.org/data/definitions/296.html

MITRE (2017h) Cwe-326: Inadequate encryption strength. https://cwe.mitre.org/data/definitions/326.html

MITRE (2017i) Cwe-327: Use of a broken or risky cryptographic algorithm. https://cwe.mitre.org/data/definitions/327.html

MITRE (2017j) Cwe-415: Double free. https://cwe.mitre.org/data/definitions/415.html

MITRE (2017k) Cwe-787: Out-of-bounds write. https://cwe.mitre.org/data/definitions/787.html

MITRE (2017l) Cwe-840: Business logic errors. https://cwe.mitre.org/data/definitions/840.html

MITRE (2017m) Cwe-862: Missing authorization. https://cwe.mitre.org/data/definitions/862.html

MITRE (2017n) Cwe-909: Missing initialization of resource. https://cwe.mitre.org/data/definitions/909.html

MITRE (2017o) Cwe-94: Improper control of generation of code ('code injection'). https://cwe.mitre.org/data/definitions/94.html

MITRE (2017p) Common weakness enumeration http://cwe.mitre.org/

MITRE (2017q) Cve details Android vulnerabilities. https://www.cvedetails.com/product/19997/Google-Android.html

MITRE (2017r) Cve details. https://www.cvedetails.com/

Morales LV, Rueda SJ (2015) Meaningful permission management in android. IEEE Lat Am Trans 13(4):1160–1166. https://doi.org/10.1109/TLA.2015.7106371

Nickinson P (2015) The 'stagefright' exploit: what you need to know. http://www.androidcentral.com/stagefright

NIST (2015) Common vulnerability scoring system calculator version 2. https://nvd.nist.gov/vuln-metrics/cvss/v2-calculator

NIST (2017) Nvd data feeds http://nvd.nist.gov/download.cfm#{RSS}

Novak E, Tang Y, Hao Z, Li Q, Zhang Y (2015) Physical media covert channels on smart mobile devices. In: Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing, UbiComp '15. ACM, New York, pp 367–378. https://doi.org/10.1145/2750858.2804253

Park Y, Reeves DS (2013) Deriving common malware behavior through graph clustering. Comput Secur 39(PART B):419–430. https://doi.org/10.1016/j.cose.2013.09.006

Ren C, Zhang Y, Xue H, Wei T, Liu P (2015) Towards discovering and understanding task hijacking in android. In: Proceedings of the 24th USENIX conference on security symposium, SEC'15. USENIX Association, Berkeley, pp 945–959. http://dl.acm.org/citation.cfm?id=2831143.2831203

Rust (2013) https://www.rust-lang.org

Sadeghi A, Bagheri H, Malek S (2015) Analysis of android inter-app security vulnerabilities using covert. In: ICSE'15, pp 725–728. http://dl.acm.org/citation.cfm?id=2819009.2819149

Sadeghi A, Bagheri H, Garcia J, Malek S (2016) A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. IEEE Trans Softw Eng PP(99):1–1. https://doi.org/10.1109/TSE.2016.2615307

Sadeghi A, Jabbarvand R, Malek S (2017) Patdroid: Permission-aware gui testing of android. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017. ACM, New York, pp 220–232. https://doi.org/10.1145/3106237.3106250

Sbîrlea D, Burke MG, Guarnieri S, Pistoia M, Sarkar V (2013) Automatic detection of inter-application permission leaks in android applications. IBM J Res Dev 57(6):2:10–2:10. https://doi.org/10.1147/JRD.2013.2284403

Sliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories

Stefanko L (2015) Aggressive android ransomware spreading in the usa. http://www.welivesecurity.com/2015/09/10/aggressive-android-ransomware-spreading-in-the-usa/

Sufatrio Tan DJJ, Chua TW, Thing VLL (2015) Securing android: a survey, taxonomy, and challenges. ACM Comput Surv 47(4):58:1–58:45. https://doi.org/10.1145/2733306

Thomas DR (2015a) The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface (Transcript of Discussion). Springer International Publishing, Cham, pp 139–144. https://doi.org/10.1007/978-3-319-26096-9_14

Thomas DR, Beresford AR, Rice A (2015b) Security metrics for the android ecosystem. In: Proceedings of the 5th annual ACM CCS workshop on security and privacy in smartphones and mobile devices, SPSM '15. ACM, New York, pp 87–98. https://doi.org/10.1145/2808117.2808118

Tufano M, Watson C, Bavota G, Di Penta M, White M, Poshyvanyk D (2018) An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ASE 2018. ACM, New York, pp 832–837. https://doi.org/10.1145/3238147.3240732

U.S. National Institute of Standards and Technology - NIST (2012) National vulnerability database. http://nvd.nist.gov

U.S. National Institute of Standards and Technology - NIST (2012) Sp 800-30 guide for conducting risk assessments

VisionMobile: Developer economics q1 2014 (2014) State of the developer nation. Tech. rep.

Wang K, Zhang Y, Liu P (2016) Call me back!: Attacks on system server and system apps in android through synchronous callback. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, CCS '16. ACM, New York, pp 92–103. https://doi.org/10.1145/2976749.2978342

Weichselbaum L, Neugschwandtner M, Lindorfer M, Fratantonio Y, Veen VVD, Platzer C (2012) ANDRUBIS: Android Malware Under The Magnifying Glass. Tech. rep., Vienna University of Technology. https://www.iseclab.org/papers/andrubis_techreport.pdf

wiki. L (2015) Android kernel features. http://elinux.org/Android_Kernel_Features

Wikipedia (2017a) Android version history https://en.wikipedia.org/wiki/Android_version_history

Wikipedia (2017b) Heartbleed https://en.wikipedia.org/wiki/Heartbleed

Wikipedia (2017c) Stagefright https://en.wikipedia.org/wiki/Stagefright_(bug)

Wu L, Grace M, Zhou Y, Wu C, Jiang X (2013) The impact of vendor customizations on android security. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security, CCS '13. ACM, New York, pp 623–634. https://doi.org/10.1145/2508859.2516728

Xiao X, Tillman N, Fahndrich M, DeHalleux J, Moskal M (2012) User-aware privacy control via extended static-information-flow analysis. In: IEEE/ACM international conference on automated software engineering

Xu M, Song C, Ji Y, Shih MW, Lu K, Zheng C, Duan R, Jang Y, Lee B, Qian C, Lee S, Kim T (2016) Toward engineering a secure android ecosystem: a survey of existing techniques. ACM Comput Surv 49(2):38:1–38:47. https://doi.org/10.1145/2963145

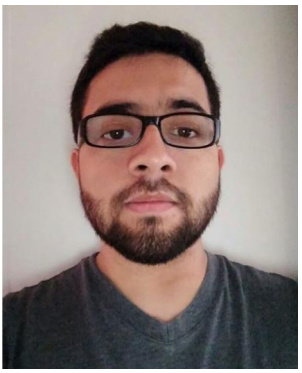
You W, Liang B, Shi W, Zhu S, Wang P, Xie S, Zhang X (2016) Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices. In: Proceedings of the 38th international conference on software engineering, ICSE '16. ACM, New York, pp 959–970. https://doi.org/10.1145/2884781.2884863

Zaman S, Adams B, Hassan AE (2011) Security versus performance bugs: a case study on firefox. In: Proceedings of the 8th working conference on mining software repositories, MSR'11. ACM, New York, pp 93–102. https://doi.org/10.1145/1985441.1985457

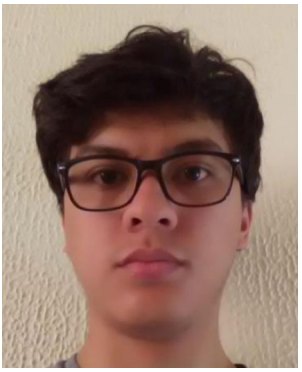
Zhou Y, Jiang X (2012) Android malware genome project. http://www.malgenomeproject.org/

Zhou Y, Jiang X (2012) Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on security and privacy, pp 95–109. https://doi.org/10.1109/SP.2012.16

Zuo C, Wu J, Guo S (2015) 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. ACM, New York, pp 591–596. https://doi.org/10.1145/2714576.2714583



Alejandro Mazuera-Rozo is a Ph.D. student in the Faculty of Informatics at the Università della Svizzera italiana (USI), Switzerland. He received his M.S. in Information Security from Universidad de los Andes in 2018, and his B.S. in Telematics Engineering from Universidad Icesi in 2015. His research interests include network and system security, information security, security of software systems, software quality and mobile development.



Jairo Bautista-Mora is pursuing a M.S in Software Engineering at Universidad de Los Andes. He received his B.S in Systems and Computing Engineering from Universidad de Los Andes in 2018. His research interests include software architecture, task automatization for software development, and application of data mining and machine learning techniques to increase the value of web and mobile applications.

Mario Linares-Vásquez is an Assistant Professor at Universidad de los Andes in Colombia. He received his Ph.D. degree in Computer Science from the College of William and Mary in 2016. He received his B.S. in Systems Engineering from Universidad Nacional de Colombia in 2005, and his M.S. in Systems Engineering and Computing from Universidad Nacional de Colombia in 2009. He is leading the software design lab at Uniandes. His research interests include software evolution and maintenance, software architecture, mining software repositories, application of data mining and machine learning techniques to support software engineering tasks, and mobile development. He received three ACM SIGSOFT Distinguished Paper awards at ESEC-FSE 2015, ICPC 2016, and ASE 2017.

Sandra Rueda is an Assistant professor at Universidad de los Andes. Bogotá, Colombia. She holds an M.S. degree from Universidad de los Andes and a Ph.D. from The Pennsylvania State University, Pennsylvania. Her research interests include security of software systems, access control, policy analysis, and policy generation.

Gabriele Bavota is an Assistant Professor at the Università della Svizzera italiana (USI), Switzerland. He received the PhD degree in computer science from the University of Salerno, Italy, in 2013. His research interests include software maintenance, empirical software engineering, and mining software repository. He is the author of over 90 papers appeared in international journals, conferences and workshops. He received four ACM SIGSOFT Distinguished Paper awards at ASE 2013, ESEC-FSE 2015, ICSE 2015, and ASE 2017, the best paper award at SCAM 2012, and three distinguished reviewer awards at WCRE 2012, SANER 2015, and MSR 2015. He served as a Program Co-Chair for ICPC'16, SCAM'16, and SANER'17. He also serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, FSE, ASE, ICSME, MSR, SANER, ICPC, SCAM, and others.