GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

Doctoral Thesis

Enhancing Trustability of Android Applications via Flexible Permissions

PhD Program in Computer Science: XXX cycle

Supervisor
Prof. Paola Inverardi
paola.inverardi@univaq.it

Author:
Gian Luca Scoccia
gianluca.scoccia@gssi.it

Co-supervisor
Prof. Ivano Malavolta
i.malavolta@vu.nl

Co-supervisor
Prof. Marco Autili
marco.autili@univaq.it

June 2019

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

Abstract

Recent years have seen a global widespread adoption of smart mobile devices, notably, smartphones and tablets. Coupled with them is the even more explosive diffusion of mobile apps. The diffusion of such devices provides end users with previously unimaginable capabilities, and sensitive tasks such as purchasing products, managing bank accounts or keeping track of vital health information are now possible with just the flick of a finger. This increased reliance on smart mobile devices and apps is not without risks. They have an unprecedented access to sensitive personal information that is increasingly collected and used by companies.

To counteract this issue, the European Commission launched the *Next Generation Internet* (NGI) initiative, with the ultimate goal of ensuring the creation of an internet that respects human and societal values, privacy, participation and diversity. *Privacy* and *Trust* play a key role, as NGI will inherently contain technical capabilities to support the data sovereignty of the end user, who should have the authority to decide how and by whom her data are used.

In this dissertation, we investigate how mobile apps can be made more in line with the NGI vision, shifting to a more human-centric approach to privacy protection by giving control back to the user. Specifically, focusing on the Android platform, we investigate existing issues in its current security- and privacy-preserving mechanisms that result in a negative impact on users' trust on the whole platform. Building on the results of this investigation, we propose a new permissions model that enables end-users with more control over their personal data and, at the same time, provides a better understanding of how and why such data are used.

The contributions of this dissertation are: (*i*) an up-to-date map of the state of the art in static analysis of mobile apps, complete with an evaluation of the potential for industrial adoption; (*ii*) the identification of a number of existing issues in the current Android permissions system from the end user perspective; (*iii*) an empirical investigation on the introduction by developers of permissions-related issues in open-source Android apps, complete with a characterization of their frequency and decay time; (*iv*) the realization and evaluation of *Android Flexible Permissions* (AFP), a new flexible permissions model that empowers end users to specify and enact flexible permissions for Android apps.

Contents

List of Figures

Chapter 1

Introduction

1.1 Smartphones and privacy

Recent years have seen a global widespread adoption of smart mobile devices (*i.e.,* smartphones and tablets). This trend is expected to continue, with the total worldwide smartphone subscriptions projected to grow from 4.3 billion in 2019 to 7.2 billion in 2023 [1]. Coupled with the diffusion of mobile devices is the even more explosive diffusion of mobile apps, *i.e.,* software that is specifically designed to run on mobile devices. Global mobile app revenues are currently estimated at 87 billion US$ and further grow is forecasted, with a predicted estimate of 129 billion for 2021 (as shown in Figure 1.1) [2]. The Android operating system represents a key player in this landscape, as it is the smartphone OS market leader among all age segments in the US, UK, and other countries [3]. The Android apps market, represented by the well known Google Play Store, now counts more than two millions apps, downloaded billions of times per year [4].

The tremendous diffusion of smartphone and mobile apps enables end-users with previously unimaginable capabilities, as sensitive tasks such as purchasing products, managing their bank account and keeping track of vital health information are now possible with just the flick of a finger. Unsurprisingly, this increased reliance on smarthpone and mobile apps is not without risk, as mobile apps have an unprecedented access to sensitive personal information.

A 2014 survey conducted among adult Americans [5] found that 91% of participants believe that consumers have lost control over how personal information is collected and used by companies and that most participants would like to do more to protect their personal information online. In a similar manner, a study among American mobile phone users found that more than half of app users have uninstalled or decided to not install an app due to concerns about personal information [6].

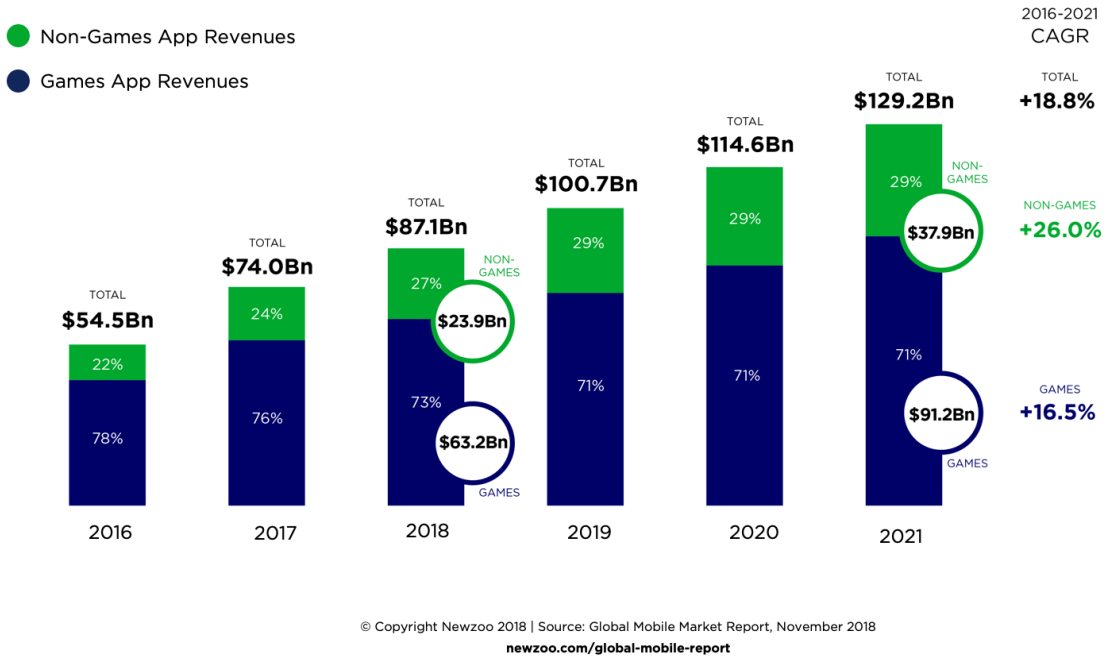


FIGURE 1.1: Past and predicted global app revenues

1.2 A new perspective and its challenges

The Internet raison d'être is enabling the exchange of data in its multiple forms, whether it is about sending an email, watching videos, listening to music, chatting or posting pictures. A big portion of our life happens on the internet where a huge amount of data is generated and elaborated. This, in an increasingly digital world, raises serious concerns as citizens' lack of control over their personal data can lead to inequalities, manipulation of information and restriction in freedom of choices [7].

To address these issues, the *Next Generation Internet* (NGI) initiative has been launched by the European Commission with the ambition to ensure the creation of an internet that respects human and societal values, privacy, participation and diversity, and offers new functionalities to support people's real needs, and addresses global sustainability challenges [7]. The overall vision is grounded on the idea that via increased connectivity and the progressive adoption of advanced concepts and technologies better value will be delivered to the people and to the society [8].

For this vision to be realised, several open questions and challenges need to be addressed at economic, political, legal and technological levels. Among them, *Privacy* and *Trust* play a key role, as the NGI must inherently contain technical capabilities to support the data sovereignty of the end user. The end user should have the power to decide how and by whom her data are used [8, 9]. This will allow citizens to take back control of the

internet and make it a powerful tool to improve their lives in areas as diverse as health, democracy, environment and mobility.

1.3 Research questions

In this dissertation, we investigate how mobile apps can be made more in line with the NGI vision, shifting to a more human-centric approach to privacy protection by giving control back to the user. Specifically, we focus on the Android platform, given its dominant position in the smartphone OS market, its open-source nature, and the fact that it has been the platform of choice for past research work on mobile software.

In order to accomplish our goal, multiple aspects have to be considered. On the one hand, current Android security- and privacy-preserving mechanisms can provide a valuable source of information, as lessons can be learned by identifying and understanding issues that affect them and that have a negative impact on users' trust on the whole platform. On the other hand, Android security and privacy-preserving mechanisms have to be rethought, as to enable end-users with more control over their personal data and, at the same time, provide a better understanding of how and why such data is used.

RQ1 *What is the state on the art on static analysis for mobile applications?*

The redesign of Android security- and privacy-preserving mechanisms will be heavily based on static analysis. Hence, overviewing the current state of the art is necessary to understand which techniques can be adopted to answer RQ3. This research question will be answered in Chapter 4.

RQ2 *Are there any existing issues in current Android security- and privacy-preserving mechanisms that negatively affect the users' trust on the whole platform?*

The identification and analysis of existing issues in current Android security- and privacy-preserving mechanisms can provide us with key insights on how these mechanisms are perceived by users. This information will be key to conceive a more user-centric redesign of aforementioned mechanisms. This research question will be answered in Chapter 5 and Chapter 6.

RQ3 *How can we rethink Android security- and privacy-preserving mechanisms in order to make them more user-centric and in line with the NGI vision?*

Rethinking Android security- and privacy-preserving mechanisms comes with multiple challenges. On the one hand, solutions to the issues identified in RQ2 must be user-friendly and accepted by end-users. On the other hand, in order to make

proposed solutions adoptable in practice, modifications to the existing Android platform and any additional effort required from app developers must be minimized. Considering both these aspects, the answer to this question is provided in Chapter 7.

1.4 Research approach and method

Given the dual nature of the goal at hand, a variety of investigation and analysis techniques have been used to gather insights and to empirically validate the proposed modifications to security- and privacy-preserving mechanisms.

For this purpose, the following qualitative and quantitative investigation and analysis techniques were used:

- **Mining of software repositories (MSR)** describes a broad class of investigations into the examination of software repositories, in the broad sense of artifacts that are produced and archived during software evolution. They include sources such as the information stored in source code version-control systems (*e.g.,* the Subversion versioning system), requirements/bug-tracking systems (*e.g.,* Bugzilla), and communication archives (*e.g.,* e-mail). These repositories hold a wealth of information and provide a unique view of the actual evolutionary path taken to realize a software system. The premise of MSR is that empirical and systematic investigations of repositories will shed new light on the process of software evolution and the changes that occur over time by uncovering pertinent information, relationships, or trends about a particular evolutionary characteristic of the system [10].

- **App store analysis** is a form of software repository mining. Unlike other software repositories traditionally used in MSR work, app stores usually do not provide source code. However, they do provide a wealth of other information in the form of pricing and customer reviews. Hence, data mining can be used to analyse apps' technical, customer and business aspects [11].

- **Systematic Mapping Study (SMS)** or scoping studies is a qualitative research method designed to give an overview of a research area through classification and counting contributions in relation to the categories of that classification [12, 13]. The outcome of a mapping study is an inventory of papers on the topic area, mapped to a classification. Hence, a mapping study provides an overview of the scope of the area, and allows to discover research gaps and trends.

- **Experiment** and **Quasi-experiment**. An experiment in software engineering is an empirical inquiry that manipulates one factor or variable of the studied setting. Based on randomization, different treatments are applied to or by different subjects, while keeping other variables constant, and measuring the effects on outcome variables. The effect of the manipulation is measured, and based on this a statistical analysis can be performed. A quasi-experiment is a similar inquiry in which, unlike experiments, the assignment of treatments to subjects cannot be based on randomization, but emerges from the characteristics of the subjects or objects themselves [12].

1.5 Contributions

Concrete contributions of this dissertation are provided below.

- a semi-automatic classification pipeline to classify Android reviews according to specific concerns. Accuracy of the pipeline, and of the underlying machine learning techniques, has been evaluated in the field in the context of an empirical study focused on Android run-time permissions that analyzed 18,326,624 reviews about 15,124 apps.
- the identification of a number of potential issues of the run-time permission system from the end user perspective. Such issues are identified starting from app reviews that mention the Android run-time permission system, have been organized into a taxonomy and how they can be addressed in the future has is discussed.
- an empirical study that investigates the introduction of *permission-related issues* (PRIs) by developers in their apps. In the context of 1,059 GitHub repositories of open-source Android apps, we analyze the occurrences of four types of PRIs across the lifetime of the apps, characterize their *frequency* and their *decay time* and provide an objective assessment of whether PRI introduction and fix correlates with the experience of the developer performing it.
- a classification framework for categorizing, comparing, and evaluating approaches for static analysis of mobile apps according to a number of parameters (*e.g.,* analysis goal, supported platforms, type and number of needed inputs, types of supported analysis).
- an up-to-date map of the state of the art in static analysis of mobile apps, complete with an evaluation of the potential for industrial adoption of existing research, and a discussion of the emerging challenges and their implications for future research.

- the realization of *Android Flexible Permissions* (AFP), a new flexible permission model for Android apps that empowers end users to specify and enact flexible permissions. AFP has been evalutaed by means of four different experiments and it's implementation is made publicly available.

1.6 Structure of this dissertation

Figure 1.2 provides an outline of the structure of this dissertation.

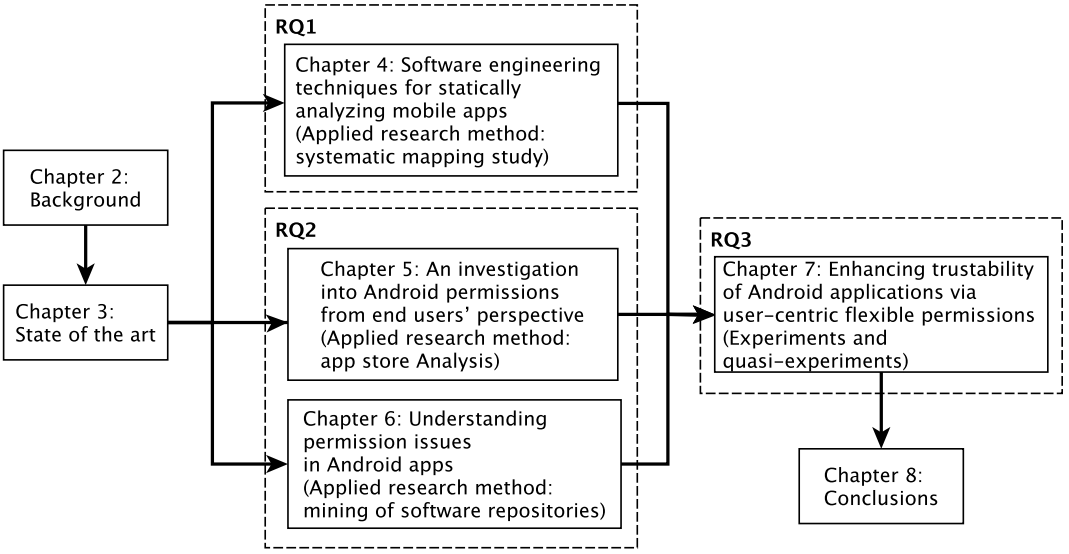


FIGURE 1.2: Structure of the dissertation

In the following, a brief description of the contents of each chapter:

- *Chapter 2* introduces the necessary background concepts about mobile apps, the Android OS and static program analysis.
- *Chapter 3* provides an overview of the current state of the art for each of the research fields related to the thesis.
- *Chapter 4* introduces a mapping study on existing approaches for static analysis of mobile apps. Such techniques are fundamental for the work of Chapter 7, hence this chapter serves as a preliminary investigation of the state of the art.

  Parts of this chapter were previously submitted as:

  Marco Autili, Ivano Malavolta, Alexander Perucci, Gian Luca Scoccia, Roberto Verdecchia. Software Engineering Techniques for Statically Analyzing Mobile Apps: Research Trends, Characteristics, and Potential for Industrial Adoption. Journal of Systems and Software. Journal of Systems and Software.

- *Chapter 5* reports on an preliminary empirical study, focused on the analysis of mobile apps user reviews, performed in order to identify existing issues in the Android permission system from the end-user perspective.

  Parts of this chapter were previously published as:

  Gian Luca Scoccia, Stefano Ruberto, Ivano Malavolta, Marco Autili, Paola Inverardi. An Investigation into Android Run-time Permissions from the End Users' Perspective. 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems, 2018.

- *Chapter 6* presents an empirical study, focused on the analysis of repositories of open-source mobile apps, performed in order to identify existing issues in the Android permission system from the developer perspective.

  Parts of this chapter were previously published as:

  Gian Luca Scoccia, Anthony Peruma, Virginia Pujols, Ivano Malavolta, Daniel E. Krutz. An Empirical History of Permission Requests and Mistakes in Open Source Android Apps. International Conference on Mining Software Repositories, 2019.

  Parts of this chapter were previously submitted as:

  Gian Luca Scoccia, Anthony Peruma, Virginia Pujols, Ivano Malavolta, Daniel E. Krutz. Permission Issues in Open-source Android Apps: An Exploratory Study. International Symposium on Empirical Software Engineering and Measurement, 2019.

- *Chapter 7* reports on the design and the experimentation of Android Flexible Permissions (AFP), a user-centric approach to flexible permissions management aimed at empowering end users to play an active role with respect to Android permissions.

  Parts of this chapter were previously published as:

  Gian Luca Scoccia, Ivano Malavolta, Marco Autili, Amleto Di Salle, Paola Inverardi. User-centric Android flexible permissions. IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017

  Parts of this chapter were previously submitted as:

  Gian Luca Scoccia, Ivano Malavolta, Marco Autili, Amleto Di Salle, Paola Inverardi. Enhancing Trustability of Android Applications via User-Centric Flexible Permissions. IEEE Transactions on software engineering.

- *Chapter 8* provides concluding remarks and possible future work.

Chapter 2

Background

This section gives a brief explanation of background concepts that will be mentioned in the chapters to come.

2.1 The mobile apps ecosystem

A *mobile app* (short for *mobile application*) is a computer program designed to run on mobile devices such as smartphones and tablet computers. Mobile apps were originally offered for general productivity and information retrieval, including email, calendar, contacts, stock market and weather information. However, public demand drove rapid expansion into other categories and nowadays are used by millions of people, who use them for their everyday activities like purchasing products, messaging, entertainment, etc. [14]. Analyst's reports estimate that the app economy creates revenues of more than 10 billion € per year within the European Union, while 1.8 million jobs have been created in 28 EU states due to the growth of the app market [15]. Apps that are not pre-installed are usually available through application distribution platforms, which began appearing in 2008. App stores are typically operated by the owner of the mobile operating system (such as the Apple App Store[1], Google Play[2] or the Windows Phone Store[3]). Generally, mobile apps are downloaded directly from the distribution platform to a target mobile device.

Mobile apps fall broadly into three categories: native, web-based, and hybrid [16]. Native applications run on a device's operating system and have direct access to services provided by their underlying mobile platform, via means of Application Programming

[1] https://www.apple.com/it/ios/app-store/
[2] https://play.google.com/
[3] https://www.microsoft.com/store/apps/windows-phone

Interfaces (API). Thanks to platform-specific APIs and tools, developers can create native mobile apps with rich user experiences, advanced graphics, and high performance. However, the use of platform-specific technologies leads to the phenomenon of mobile platform *fragmentation*, *i.e.,* code written for one mobile platform (*e.g.,* the Java code of an Android app) cannot be used on another (*e.g.,* the Objective-C code of an Apple iOS app). Fragmentation results in potentially higher development time, higher testing and maintenance costs, and low portability.

Web-based apps are developed with web technologies, hosted on remote servers, served via standard protocols, and accessed via a web browser on a mobile device. Since the code of mobile web apps conforms to standard languages, a single app delivers a uniform experience across multiple platforms, resulting in faster development, simpler maintenance, and full application portability. Nonetheless, even if the browser is increasingly becoming a fully-fledged software platform (as new web APIs allow for geolocation, accessing the camera, microphone, etc.), as of today mobile web apps struggle in handling heavy graphics and still lack straightforward means to access low level features (*e.g.,* background services management). Finally, as mobile web apps are hosted and served like usual websites, they cannot be distributed via app stores.

Hybrid apps are web-based apps hosted inside a native application. They are developed via standard web technologies and they can be distributed for any supported mobile platform, like Android or iOS. More specifically, a hybrid development framework (*e.g.,* Apache Cordova) allows developers to create a cross-platform hybrid mobile app by providing (i) a native wrapper for containing the web-based code, and (ii) a JavaScript API that bridges all the service requests from the web-based code to the corresponding platform API. Thanks to the native wrapper, a hybrid mobile app can be packaged and distributed for any supported platform. Existing knowledge of web developers can be reused also for developing mobile apps, and the development process is simplified, as a single code base can be used for all platforms. On the negative side, hybrid mobile apps can access the platform APIs only via the bridge provided by the hybrid development framework, which considers only a subset of all the possible APIs provided by each platform and imposes an additional performance overhead when accessing platform APIs.

At the time of writing, the mobile operating systems market is dominated by two main platforms: Android and iOS. Combined, these two platforms make up over 99% of smartphone sales worldwide [17].

2.2 The Android operating system

Android is an open-source, Linux-based mobile device operating system developed by Google. The Linux kernel was chosen due to its proven driver model, existing drivers, memory and process management, networking support along with other core operating system services [18]. On top of the Linux kernel, various layers, libraries and apps are built in order to support higher functionality. The complete Android software stack is presented in Figure 2.1.

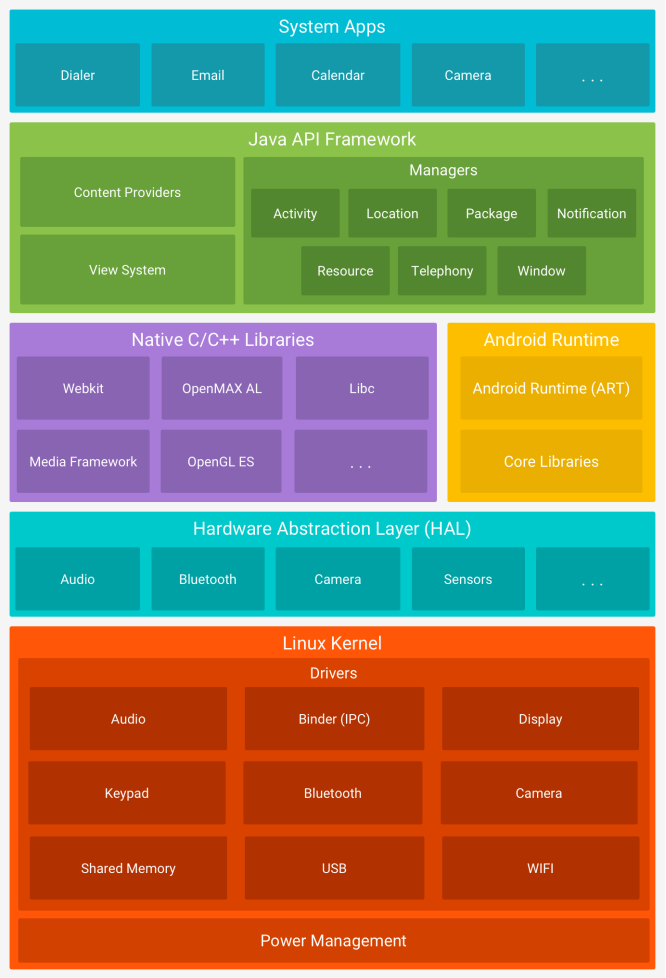


FIGURE 2.1: The Android software stack

Android apps are written in the Java programming language and are compiled into an *APK file*, short for *Android package*. An APK contains all the compiled code plus any data and resource files that are required by the application. Of particular note, inside the apk resides the app *manifest file*, which provides essential information about the app to the Android system, such as required libraries and APIs. For a given app, the related APK can be distributed, directly or through app stores, to Android-powered devices in order to install the app. However, differently from standard Java applications,

instead of the class files being run in a J2ME virtual machine, the code is translated after compilation into a *"Dex file"* that can be run on an ad-hoc virtual machine named *Dalvik*. Compared with regular Java class files, Dex files are optimized to be smaller in size. The virtual machine itself is optimized to perform well on mobile devices with a slow CPU, limited memory, no operating system swap space and most importantly limited battery power. As to enable a higher level of security and privacy, every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently.

Each Android app is built by composing four kinds of essential application components: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers*. An Activity is the entry point for interacting with the user and represents a single screen with a user interface. A Service is a component that runs in the background to perform long-running operations. A Broadcast Receiver is a component that enables the system to deliver events to apps outside of regular user flow. It allows apps to respond to system-wide events, even if they are not cur- rently running. A Content Provider manages a shared set of app data. Through it, other apps can query or modify the data if the content provider allows it.

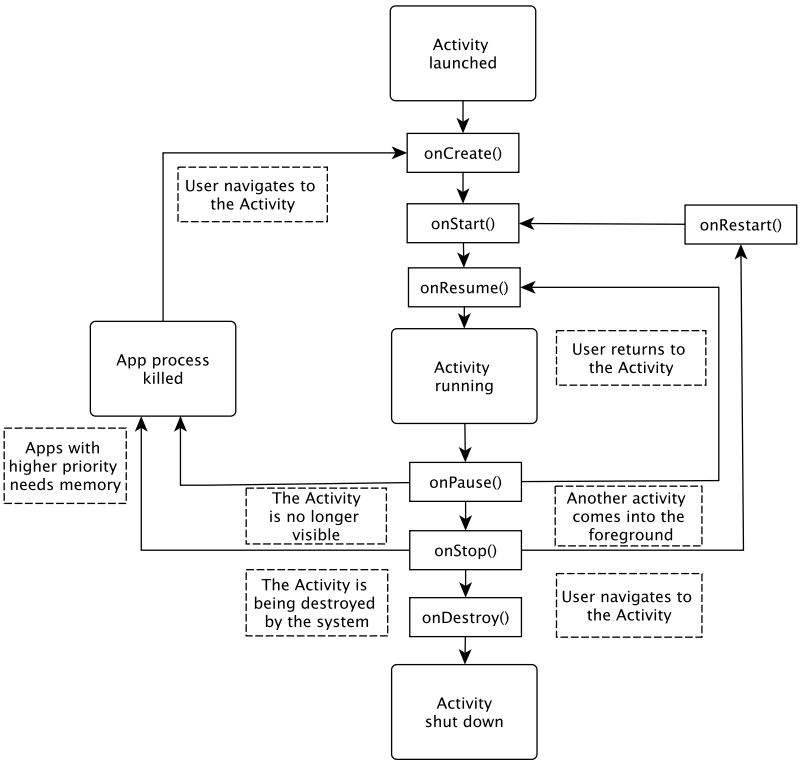


FIGURE 2.2: Android Activity lifecycle[4]

During the lifetime of an application, as the user interacts with it, activities and services that compose the app go through a set of well defined states, collectively named as

[4]Image source: https://developer.android.com/guide/platform/

lifecycle. Figure 2.2 shows the lifecycle of an activity. Developers can insert their own logic into each state change by hooking into one or more callback methods that are invoked during each transition. Inside an Android application, components communicate with each other through the use of *Intents*. An Intent is an abstract description of an operation to be performed, and can be used to activate and request an action from other app components.

2.3 Permission systems

Operating systems provide development platforms that support thriving markets for third-party applications. However, third-party code creates risks for the user: some third-party authors are malicious or can unintentionally introduce vulnerabilities. In order to protect users from the threats associated with third-party code, modern platforms make use of *permission systems* to control access to security- and privacy- relevant parts of the platform [19]. These systems limit damages from security breaches by enacting the *principle of least privilege* [20]. According to this principle, every program and every privileged user should operate using the least amount of privileges necessary to complete the job at hand. The goal of the principle is to reduce the number of potential interactions among privileged programs to the minimum necessary, so that one may develop confidence that unintentional, unwanted, or improper uses of privilege do not occur. When the principle is followed, effects of accidents are reduced and the number of programs which must be audited to discover an accident's root causes is minimized.

Nowadays, newer *application-based* permission systems, where permissions are granted individually to each application, are gaining popularity over traditional *user-based* permission systems, where privileges are granted to users and apply to all user applications. The former can be further divided into *usage-time* permission systems, that prompt users to approve permissions as needed by applications at runtime, and *install-time* permission systems, that ask developers to declare their applications permission requirements upfront so that users can grant them during installation [19]. Most recent Android releases (*i.e.,* from Android API version 23 onwards) use a usage-time permission system while former versions rely on an install-time permission system.

2.3.1 Android install-time permission system

Up to version 5.1.1 (i.e., Android API level 22), Android makes use of an install-time permissions system to regulate access to sensitive APIs of the platform. Developers have to declare upfront (in the app manifest file) if their app requires access to security- and

privacy-relevant parts of the platform. During app installation, the user is notified of required resources and possible risks by a dialog shown on the screen, such as the one in Figure 2.3. She can then choose whether to continue with the installation, thus granting the app access to all requested resources, or to abort the installation process. Once granted, permissions cannot be removed, except by completely uninstalling the app.

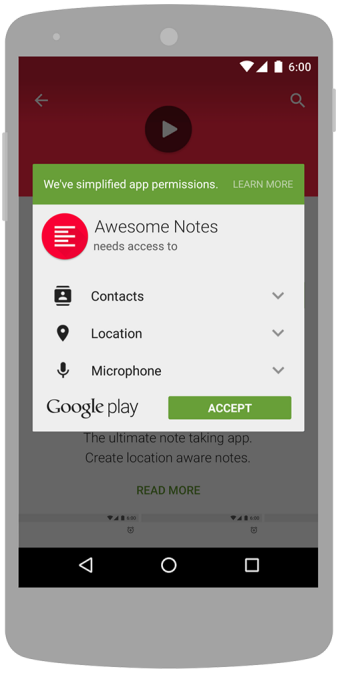


FIGURE 2.3: Install-time permission request dialog[5]

2.3.2 Android usage-time permission system

Starting with Android 6 (i.e., Android API level 23), access to privacy- and security-relevant parts of the platform is enforced by a usage-time permission system. While the app is running, whenever access to a restricted part of the platform is attempted, the system checks whether the app has the required permissions. If necessary, it prompts the users for confirmation with a dialog, such as the one in Figure 2.4, from which the permissions can be granted or denied. In the latter case, app execution continues with possibly (partially) degraded functionalities.

Unlike the previous system, users can selectively grant permissions, *i.e.,* they can choose to grant only a subset of requested permissions. Nonetheless, permissions are still granted on a whole-app basis, *i.e.,* once granted the permission is valid for the entire app. Once granted, permissions be can dynamically removed at a later time accessing the device

[5]Image source: https://developer.android.com/guide/topics/permissions/overview
[6]Image source: https://developer.android.com/guide/topics/permissions/overview

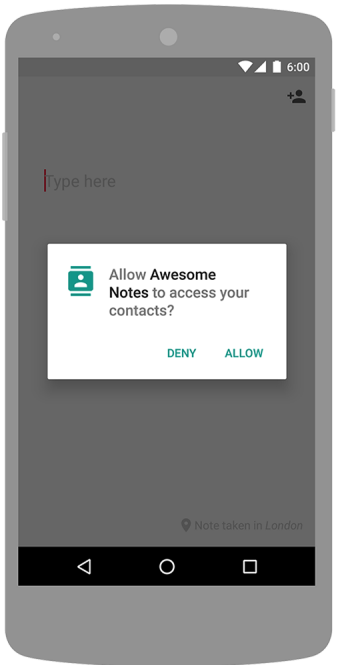


FIGURE 2.4: Usage-time permission request dialog[6]

system settings and manually changing the app permission settings. Developers still have to list all permissions required by an app in its manifest file.

In addition, permissions are grouped into *permission groups*. A permission group is a set of two or more permissions that reference the same resource. For instance, the permissions READ_CONTACTS and WRITE_CONTACTS both belong to the CONTACTS permission group. Whenever an app tries to access a protected resource, users are prompted for confirmation only if no other permission in the belonging permission group is already granted.

2.3.3 Static program analysis

Static program analysis [21] allows for predicting (precise or approximated) quantitative and qualitative properties related to the run-time behavior of a program without actually executing it. For instance, static analysis techniques allow for statically inferring cost-related properties (such as the estimation of the maximal number of loop iterations and the related worst-case execution time), as well as properties related to resource consumption (such as memory/heap usage and energy consumption). Many kinds of theoretical and practical approaches exist in the literature, *e.g.,* structural and control-flow analysis, data-flow and state-based analysis, taint analysis, etc. Practical static analysis is usually carried out by automatic tools - the analyzers - and can be performed against program models or intermediate representations (*e.g.,* Carmel intermediate representation of Java Card byte code), or against the actual (source or binary) program code.

Models enable static analysis approaches which are independent from any technological solution. When performed against these models, static analysis is able to compute under- or over-approximations of the actual program behavior. When performed against the actual program code, it allows for directly obtaining more precise values or refining their approximations.

In the literature, static analysis of mobile apps has been applied with variety of goals in mind, ranging from malware and privacy leaks detection to detection of bugs in the app source, to reduction of energy and memory consumption [22–26]. Seeking to attain these goals, researchers have experimented with a variety of different static analysis techniques. Among the ones worth mentioning, *data-flow analysis* considers a program as a graph: nodes are elementary blocks and edges describe how control passes from one block to another [21]. *Taint Analysis* is a special case of data-flow analysis that aims to detect the existence of a data flow from sensitive data sources, often simply referred as *sources*, to untrusted program statements, called *sinks* [24]. *Type Analysis* aims to verify the type safety of a program, *i.e.,* if we can guarantee that the eventual value of any expression in the program will not violate that expression's static type. In other words, type analysis aims to detect type errors in a program source code. *Abstract interpretation* is a sound approximation of the semantics of a program, based on monotonic functions over ordered sets. It is able to extract information about the semantics of a program without performing all the calculations. *Program slicing* aims to compute the set of program statements, referred to as the program slice, which may affect the values at some point of interest, referred to as a slicing criterion.

An in depth overview of state of the art on static program analysis for mobile apps is provided in Chapter 4.

Chapter 3

State of the Art

This chapter discusses the state of the art of the research fields that are relevant to our work. Section 3.1 briefly touches on the literature that studied users' behavior regarding privacy decisions. Section 3.2 discusses the literature that investigated end users perception of mobile app permissions. Section 3.3 gives an overview of existing literature that proposes enhancements and modifications to the Android permission system in order to address its shortcomings.

3.1 Users behavior on privacy decisions

Permissions-related decisions belong to the broader category of privacy and security choices to which individuals are confronted in their daily on-line activities. Users' behavior in relation to those decisions has been a subject of interest for researchers of different fields, such as human computer interaction, behavioral economics and usability research.

Initial efforts to formalize users' behavior in relation to these choices relied on the *theory of rational choice* [27]. According to this model, users are assumed to be fully rational agents that, when confronted with a decision, choose the option that maximizes their utility, in accordance with their preferences.

Over the years, limitations of such a view became more evident, leading to the emergence of new models in which users exhibit a *"bounded rationality"*. Rather than viewing users as purely rational decision makers, Simon [28] views individuals as agents with limited computational capabilities, unable to perform a thorough exploration of all possible outcomes of a choice. Hence, heuristics are used instead as shortcuts in decision making. In addition, a wide variety of cognitive and behavioral biases may factor their influence into the decision making process [29];

Akerlof [30] modeled scenarios in which the two parties involved in a transaction have access to information that is, *incomplete*, that is when parts of it are lacking for both sides, or *asymmetrical*, when availability of it is different between the two sides. Information unavailability hinders involved subjects ability to perform correct decision making. The aforementioned scenarios are the norm in the on-line privacy domain as threats and technologies continuously evolve and as user interfaces purposely hide relevant information [31].

Böhme *et al.* propose a model in which user attention is seen as a scarce resource, quickly depleted by repeated requests for intervention. Since lack of attention can lead to privacy and security breaches, the former should be best allocated to the primary task and the decisions that really matter [32]. Such model is consistent with a phenomenon known as *"warning fatigue"*, *i.e.,* users may become insensitive to warning dialogs after being exposed to an excessive amount, that has been observed in privacy-relevant applications [33, 34].

3.2 Effectiveness of Android permissions

With the growth in popularity of Android since its first release in 2008, a comprehensive body of research has focused on its permission system and its effectiveness, with the purpose of identifying shortcomings and possible points for improvement.

Seminal work on Android permissions has been conducted by Felt *et al.* that, in [35], investigated the effectiveness of the (at the time novel) Android install-time permission system warnings, focusing on users' understanding and attention to permission information provided during app installation. Towards this goal, the authors performed two usability studies, in the form of an on-line survey involving 308 Android users and a series of semi structured live interviews with 25 Android users. Results of both studies evidenced the existence of several usability issues in the system, with only 17% of participants that paid attention to permissions during installation, and only 3% of Internet survey respondents that correctly answered questions aimed at assessing their understanding of permissions. From the data and the observations collected during interviews, the authors highlighted several issues experienced by users, such as confusing permission names, lack of a proper risks explanation and warning fatigue affecting some users.

Similar issues have been found by Kelley *et al.* in a concurrent study [36] carried on as a set of semi-structured interviews with 20 participants. Although permission-related warnings were generally found to be viewed and read, at the same time comprehension

and understanding of them was found to be lacking. In addition, users were found to be largely uninformed about the existence of malware and malicious applications.

Mylonas *et al.* [37] observed that despite the fact that smartphone users increasingly download and install applications from official application repositories, vetting mechanisms are often not in place. Instead the user is delegated with the critical task of authorizing which functionality and protected resources can by accessed by applications. In this respect, a little or badly informed user can easily become prey of malicious app developers. Hence, they conducted 458 structured interviews with smarthpone users that routinely install applications from official repositories to assess their security awareness. From the analysis of the data collected during interviews, a general security complacency emerges, with most users believing that downloading apps from official repositories poses no security risk and only a minority that routinely scrutinizes warning messages. As a first step towards overcoming highlighted issues, the authors developed a prediction model to identify users who blindly trust app repositories.

In [38], Kelley *et al.* investigated how providing users with additional privacy information, alongside used permissions, affects their decision during the app selection process. For this purpose they developed a "Privacy Facts" display that, in the form of a short checklist, informs users about privacy and security risks associated with a given application. Effectiveness of said display was evaluated in a laboratory study and an on-line study involving respectively 20 and 366 participants. In each study participants were asked to select applications for a friend who has just gotten their first Android phone and were assigned to use either the privacy checklist or the standard Android permissions display. Results of both studies reveal that both the timing and the content of the displayed privacy information may impact the extent to which users pay attention to the information. A similar beneficial effect of providing users with additional information was noticed by Egelman and colleagues [39] that, performing a survey among 483 Android users, found that when participants were comparison shopping between multiple applications that performed similar functionality, a quarter responded that they were willing to pay a $1.50 premium for the application that requested the fewest permissions, when viewing the permissions requested by each application side-by-side. Such results suggest that many smartphone users are concerned with their privacy and are willing to pay premium for applications that are less likely to request access to personal information.

Following the changes to the permissions system, effective from Android version 6, researchers have started investigating the effectiveness and issues of the new run-time system. Andriotis *et al.* collected and analyzed anonymous data from 50 participants in order to understand users' adaptation to the new run-time permission model. In [40]

they conclude that users adapted positively to the new model and that most of them prefer to use the new system over the previous. Moreover, they highlight that users are generally willing to allow access to permissions directly related to an app main functionality and that users' behavior is consistent regarding the resources they allow different social media applications to access. In [41] they suggest that although people are more reluctant to allow access to resources such as their cameras or microphones, they tend to grant these permissions to specific app categories. Furthermore, they point that users should be informed about the least required resources an app needs to provide its basic functionality before installation. In [42] they report the results of a second data collection round held a month after the first one, that found that 50% of the study participants did not change a single permission on their devices and only 2.26% of installed applications presented altered permission settings.

Peruma *et al.* [43] conducted an in-person study, involving 185 participants, to understand users perception of the current run-time Android permissions model, former install-time model and a third proposed one, where users are also informed about other apps on the device that use a given permission whenever it is requested. Their results highlight that the current Android runtime model does not make users feel more secure in comparison with the older install-time model, although it is significantly beneficial in helping users to recall the requested permissions.

3.3 Extensions of Android permissions

Over the years a diverse amount of work has investigated various possible extensions of the Android permissions system. In the following, existing works are presented, grouping together works that employed similar strategies.

3.3.1 Finer-grade extensions

Many approaches aim to offer end-users a finer-grain control over permissions granting and enforcement. For this purpose, a wide array of strategies have been employed.

One of the first approaches of this kind is Apex [44]. Apex addressed one of the main problems of the original Android install-time permission system (see Section 2.3.1), *i.e.,* users could not selectively grant permissions: if she wishes to use an application, all requested permissions had to be granted. Moreover, Apex allows restricting the usage of resources based on runtime constraints such as the location of the device or the number of times a resource has been previously used. Apex is implemented via small changes

to the existing Android source code and provides end-users with a modified applications installer that allows users to specify their constraints for each permission at install time using a simple and usable interface.

Similarly, Aurasium [45] allows the definition of fine-grained policies to prevent privacy violations such as attempts to retrieve a user's sensitive information, send SMS covertly, or access malicious IP addresses. Aurasium does not require modifications to the Android OS. Rather, to achieve its goals, applications are preemptively repackaged to insert user-level code that implements sandboxing and policy enforcement. Evaluated over a large sample of both benign and malicious Android applications, repackaging and sandboxing success rate was close to 100 percent, although repackaged apps suffer from some performance and space overhead.

A related, yet more flexible, solution is Crèpe [46, 47] (Context-Related Policy Enforcing), an extension of Android security mechanisms that is able to enforce fine-grained policies that depend on the smartphone context. Crèpe considers a broader concept of context, that can be defined either by the status of some variables (e.g. location, time, temperature, noise, and light), presence of other devices, particular interactions between the user and the smartphone, or a combination of these. Furthermore, Crèpe allows context-related policies to be defined either by the user or by trusted third parties.

Similarly, AppGuard [48, 49] allows for the definition and enforcement of user-defined security policies on untrusted Android applications, without requiring alterations to the Android operating system. To achieve its goals AppGuard takes an untrusted app and a user-defined security policy as input and embeds a security monitor in the app, thus obtaining a self-monitoring app. In the latter, sensitive API calls are redirected to the embedded security monitor that checks whether executing the call is allowed by the user-defined security policy. Unlike similar approaches, AppGuard supports fully-automatic on-the-phone instrumentation of third-party apps and automatic update of rewritten apps. Evaluated on 25,000 apps, taken from two different app markets, over 99% of the experimental subjects were found to be stable after the instrumentation although they were not checked for semantic equivalence.

Jeon *et al.* [50, 51] observed that Android permissions can be divided into broad categories according to the kind of resource that is being protected and that a few, general strategies for deriving finer-grade variants can be applied to all permissions in the same category. Hence, rather than allow users to define their own security policies, they focus on deriving finer-grade variants of existing Android permissions. For instance, *ContactCol(c)* is a finer-grade version of the READ_CONTACTS permission that allows access only to a particular field c of an accessed contact in the user's address book. To this purpose, they developed a suite of tools that allow these fine-grained permissions to be inferred

on existing apps, to be enforced by developers on their own apps, and to be retrofitted by users on existing apps. A set of 14 apps from the Google Play store were used as case study, evidencing that finer-grade permissions can be inserted into existing apps with minimal side effects or loss of functionality.

Shen *et al.* [52, 53] propose an extension of Android permissions named *"Flow Permissions"*, directly inferred from an existing app using static taint analysis techniques. The goal of flow permissions is to inform users about the existence of data-flows that could potentially leak sensitive information to the outside, *e.g.,* transmitting the users' phone number over the network, but are not actively enforced at runtime. In order to support Flow Permissions on Android, the authors developed a static analysis engine that detects flows within an Android application and between two communicating applications. Evaluated by means of a survey over 540 participants, Flow Permissions were found to be able to significantly impact user's decisions on whether to install an app when the users are unbiased, *i.e.,* when users do not have any preconceived notions about the app or its developer.

Russello *et al.* devised YAASE [54], a modification of Android that extends its security model to support fine-grained access control policies. YAASE allows users to define labels (such as public, private, confidential, etc.) and use them to mark data and enforce security decisions on how data has to be disseminated within the device (application to application) or the outside world (through internet connections). Only a preliminary evaluation performance was performed and YAASE appears to introduce a substantial, yet tolerable, overhead.

Jaebaek *et al.* developed FLEXDROID [55], an extension to the Android permission system that allows app developers to control access to a user's private information by third-party libraries. To this end, FLEXDROID provides an interface, as a part of the app manifest, for app developers to specify a set of different permissions granted to each third-party library. Upon any request for user's information, FLEXDROID identifies the source of the request (either an app or third-party libraries) via a new security mechanism, called *inter-process stack inspection* and then accordingly allows or denies the request. Evaluated on a set of 32 Android apps, all but 5 of them run as normal under FLEXDROID's permission system, without any code modification except for the manifest, and with minimal performance over-heads.

3.3.2 Mock-based extensions

Several works in the literature present approaches that aim to increase user's control of sensitive resources without depriving them of application's functionality. These approaches are built upon the concept of *"mocking"* of sensitive information, *i.e.,* when, according to user's preferences, access to a sensitive resource has to be denied, anonymized information is instead provided to the requesting app.

Among the first works that explored anonymization, Beresford *et al.* devised Mockdroid [56], a modified version of the Android operating system which allows a user to mock an application's access to a resource. Said resource is subsequently reported as empty or unavailable whenever the application requests access. The authors speculate that existing applications continue to work on MockDroid, even if with reduced functionality, because applications are written to tolerate resource failure, such as network unavailability or lack of a GPS signal. Evaluation on a set of 23 applications was performed, manually exercising each application after enabling mocking for internet and/or gps resources. All applications were found to continue to function, although with severely reduced functionality in case of apps that made major use of internet access.

Similarly Zhou *et al.* developed TISSA [57], a lightweight modification of Android that introduces a novel *"privacy mode"* in the operating system. When enabled, such mode allows to fine tune an app's access to private information stored on the phone, such as device ID, contacts, call log, and location. When access to a given resource has to be denied, three options can be employed to provide an answer to the requesting application: *"empty"* that simply returns an empty result to the requesting app, *"anonymized"* that provides an anonymized version from the original information and *"bogus"* that provides a fake result of the requested information. Evaluation was performed on a set of 24 Android apps that were known to leak (*i.e.,* transmit without user consent) personal information to third parties. All apps were found to maintain full functionality without raising security exceptions when enabling TISSA's privacy mode.

Hornyack *et al.* [58] proposed AppFence, a modification of the Android OS that combines anonymization with taint analysis, to introduce advanced privacy controls that allow for blocking of network transmissions containing sensitive data. In order to evaluate the effectiveness of their approach, the authors devised an automated testing methodology that records screenshots of application executions with and without AppFence's privacy controls. Then, it automatically highlights the visual differences between executions. Evaluated on a set of 50 apps, for 66% of them AppFence was able to introduce its privacy controls without visible side-effects.

Fawaz *et al.* [59] developed LP-Guardian, a framework for protection against location tracking and profiling. LP-Guardian utilizes decision logic, taking into account the user's preferences, to determine an appropriate anonymization strategy on a per-app basis, as to preserve functionalities as much as possible. According to participants of a user study, the loss of app functionality was perceived to be tolerable by the majority of them.

Brutschy *et al.* conceived SHAMDROID [60], a transformation algorithm that rewrites an Android app to eliminate dependencies on sensitive resources. SHAMDROID's goal is to, simultaneously, disable access to sensitive resources specified by the user while, at the same time, retaining, as much as possible, application's functionalities that depends on non-sensitive resources. These two requirements combined translate to substituting sensitive inputs with mock values that exhibit *"maximal utility"*, *i.e.,* they can drive execution along a maximal path. Hence, the authors propose an app-sensitive mocking algorithm, in which mock synthesis is governed by the particular behaviors of the subject app. When compared on a dataset of 27 apps, the SHAMDROID anonymization mechanism was found to cause abnormal behavior in only 1 app.

Although not fully focused on improving mocking techniques, some works [55, 61] make use of mocking to increase app stability. Indeed, experimental evaluation has evidenced that many existing apps can't handle the forceful revocation of a permission [62], and hence mocking techniques are used in place of full revocation.

3.3.3 Context-based extensions

An emerging line of research is investigating the possibility of constructing an automatic permission granting mechanism. Such mechanism leverages run-time contextual information (such as device connectivity, user's location, recently used applications) to systematically determine when to grant (and when to deny) permission requests without user intervention.

Preliminary work on the topic was performed by Wijesekera and colleagues that, in [63], instrumented the Android platform to collect data regarding how often and under what circumstances smartphone applications access protected resources regulated by permissions. After collecting and analyzing data from 36 participants of a field study, they found that apps rarely respect "contextual integrity" [64], i.e., applications often access protected resources when users are not expecting it. Moreover, in exit interviews, participants stated that at least 80% of them would have preferred to prevent at least one permission request, and overall, declared a desire to block over a third of all requests.

SmarPer [65] was the first attempt to utilize contextual information for automatic permission granting. SmarPer adopted a Bayesian linear regression model [66] to mimic user decisions, obtaining a mean correct classification rate of 80% after training the model with data collected from 41 Android users. Wijesekera *et al.* built upon such results and investigated the effectiveness of such techniques on a larger scale [67], obtaining a 96.8% precision in a 131-person field study. A second 31-person field experiment was performed in [61], and showed that, with slight model modifications, performance issues and practical limitations are dealt with, while suffering an acceptable reduction in precision.

Towards further improving existing results, researchers are focusing on identifying and incorporating in such decision systems other relevant sources of contextual information. Votipka *et al.* [68] conducted a 2,198-participant fractional-factorial vignette study, showing that both *when* and *why* a resource is accessed are important to users' comfort. Moreover, they identified different meaningful classes of accesses for each of these factors, showing that not all background accesses are regarded equally by users.

Chapter 4

Software engineering techniques for statically analyzing mobile apps: research trends, characteristics, and potential for industrial adoption

Static program analysis allows for predicting (precise or approximated) quantitative and qualitative properties related to the run-time behaviour of a program without actually executing it [21]. For instance, static analysis techniques allow for statically inferring cost-related properties (such as the estimation of the maximal number of loop iterations and the related worst-case execution time), as well as properties related to resource consumption [69] (such as memory/heap usage and energy consumption).

Under this perspective, static analysis of mobile apps can be of interest for both app developers, app store moderators and, indirectly, end users. App developers can use it to quickly get non-trivial insights about their apps, such as the presence of subtle security issues (as the ones discussed in Chapter 6), energy hotspots, programming antipatterns, and inefficient use of hardware sensors. App store moderators, can leverage static analysis for systematically assessing the level of quality of apps they distribute, possibly identifying those apps with an unacceptable level of quality due to the presence of well-known security flaws, requesting suspicious permissions, and with strong energy inefficiencies. Leveraging static analysis, end users can be empowered with new approaches that allow them to specify and validate their own requirements to which an application must comply to be considered as trustable.

Static analysis of mobile apps is gaining a growing interest in both academia and industry. Literally hundreds of (often overlapping) kinds of (theoretical and practical)

static analysis approaches exist in the literature, ranging from structural and control-flow analysis, to data-flow and state-based analysis, interval analysis (used in optimizing compilers) and so on [21]. Such approaches exploit static analysis techniques from different perspectives and belong to extremely different research areas of software engineering, such as software analytics, security, testing, verification, etc. Industrial tools are also emerging and 65 being maintained by key players in the technological panorama. For example, Facebook's Infer1 applies separation logic and bi-abduction for inter-procedural analysis [70] and it is used by Facebook itself, Spotify, Mozilla, the Amazon Web Services division, etc.

The **goal** of this study is to precisely characterize existing software engineering research on static analysis of mobile apps from three different perspectives, namely: (*i*) research trends, (*ii*) the characteristics of the proposed approaches, and (*iii*) their potential for industrial adoption.

In order to achieve this goal, we applied the systematic mapping study methodology [12, 13]. The aim of this methodology is to provide an objective, replicable, and unbiased approach to answer a set of research questions about the state of the art on a given topic. In this paper, we systematically selected 140 primary studies from over 8,000 potentially relevant publications on static analysis of mobile apps. Then, we defined a classification framework for categorizing the selected approaches, and we rigorously applied it to the 140 primary studies. Finally, we synthesized the obtained data to let emerge a crystal-clear snapshot of the state of the art on static analysis of mobile apps.

4.1 Study design

This research was organized into three main phases, which are well-established when it comes to systematic literature studies [12, 71]: *planning*, *conducting*, and *documenting*.

Planning. After establishing the need for performing a review on static analysis of mobile apps, we identified the main research questions (Section 4.1.1), and we defined the protocol to be followed by the involved researchers.

Conducting. We performed the mapping study by following all the steps defined in our research protocol, namely: (*i*) search and selection of primary studies, *i.e.,* the relevant research articles on static analysis methods and techniques of mobile apps (Section 4.1.2), (*ii*) extraction of relevant data from each primary study according to a rigorously-defined classification framework (Section 4.1.3), and (*iii*) synthesis of main findings emerging from the analysis and summary of the data extracted in the previous activity (Section 4.1.4).

Documenting. The main activities performed in this phase are: (*i*) a thorough elaboration of the data extracted in the previous phase, with the main goal of setting the obtained results in their context, (*ii*) the discussion of possible threats to validity, specially to the ones identified during the definition of the review protocol (in this activity new threats to validity may emerge too), and (*iii*) the writing of a final report (*i.e.,* this article) describing the performed mapping study.

A complete *replication package* is publicly available to allow interested researchers to independently replicate and verify our study[1]. It includes the review protocol, the list of both searched and selected studies, a detailed data extraction form, the raw extracted data, and the R scripts for data analysis.

4.1.1 Research questions

We formulate the goal of this study by using the Goal-Question-Metric perspectives (i.e., purpose, issue, object, viewpoint [72]). Table 4.1 shows the result of the above mentioned formulation.

Purpose	Identify, classify, and evaluate
Issue	trends, characteristics and potential for industrial adoption
Object	of existing research in static analysis of mobile apps
Viewpoint	from a researcher's and practitioner's point of view.

TABLE 4.1: Goal of this research

The results of this study are targeted to both (*i*) researchers willing to further contribute to this research area, and (*ii*) practitioners willing to understand existing research on static analysis approaches of mobile apps and thereby to be able to adopt those solutions that better fit with their needs. We refined our abstract goal into the following research questions:

RQ1 - *What are the **research trends** on static analysis of mobile apps?*

Rationale: a multitude of researchers are investigating static analysis for mobile apps over time with different degrees of independence and different methodologies. By answering this research question, we aim at characterizing the scientific interest on static analysis approaches of mobile apps, the relevant venues where academics are publishing their results on the topic, and their contribution type.

[1] `http://cs.gssi.it/mobileStaticAnalysisReplicationPackage`

RQ2 - *What are the **characteristics** of existing approaches for static analysis of mobile apps?*

Rationale: static analysis of mobile apps is a multi-faceted research topic, where researchers can focus on very different aspects (*e.g.,* energy consumption, security), applying very different research methodologies (e.g., industrial case studies, empirical evaluations), providing different types of contributions (e.g., tools for automating development activities, techniques for analyzing a specific aspect of the mobile app). By answering this research question, we aim at providing (*i*) a solid foundation for classifying existing (and future) research on static analysis of mobile apps, and (*ii*) an understanding of current research trends and gaps in the state of the art on static analysis of mobile apps.

RQ3 - *What is the **potential for industrial adoption** of existing research on static analysis of mobile apps?*

Rationale: while it is well known that mobile apps have their roots in industry, many research groups focus on them from an academic perspective. Therefore, it is natural to ask ourselves how the produced research findings and contributions can be actually transferred back to industry. By answering this research question we aim at assessing how and if the current state of the art on static analysis of mobile apps is ready to be adopted in industry.

4.1.2 Search and selection process

Our first choice for searching potentially relevant studies was to perform an automatic search on known data sources (e.g., IEEE Xplore, the ACM Digital Library, SCOPUS). However, from the results of a preliminary study [73], we understood that the research topic of mobile static analysis resulted to be extremely heterogeneous; for example, many keywords like "program analysis" resulted to be profoundly overloaded, leading to imprecise and inaccurate automatic search results. In order to prevent biases associated to automatic searches, we adopted two complementary manual search activities. This decision is supported by the evidence that automatic searches and backward snowballing activities lead to similar results, and that the decision on which to prefer is context specific [74]. Our search strategy was divided into two subsequent and complementary steps. The first step was carried out by manually inspecting all the publications of the top-level software engineering venues. The papers identified through this first step were then subsequently utilized as input for a backward and forward snowballing[2] process [75].

[2]Inspection of the studies referenced by a paper (*backward snowballing*) and of the studies referencing it (*forward snowballing*)

In order to ensure the correctness of the adopted manual approach, the backward snowballing activity was based exclusively on the papers selected from the top-level software engineering venues. Furthermore, the backward snowballing results were further contemplated by adopting a forward snowballing process, that ensured soundness and relevance of the set of the selected primary studies.

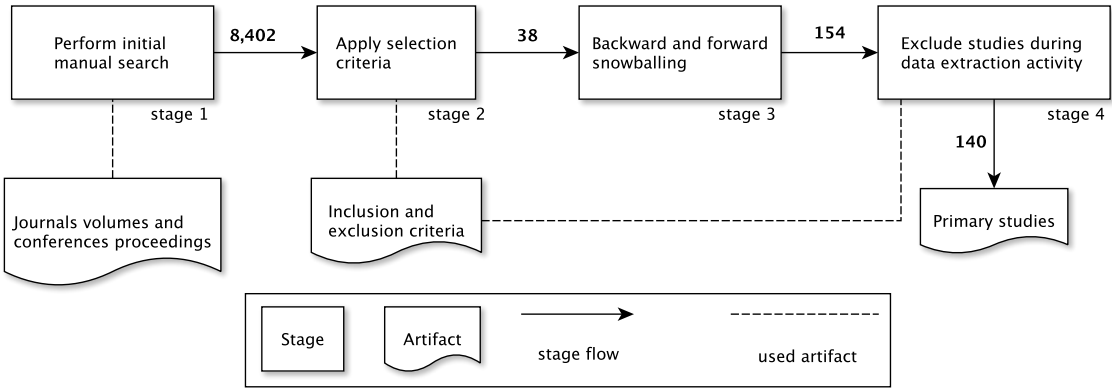


FIGURE 4.1: The search and selection process of this study

Figure 4.1 shows our search and selection process, whose main steps are detailed in the following. Our search and selection process is designed as a multi-stage process in order to have full control on the number and characteristics of the studies being either selected or excluded during the various stages.

1. Perform initial manual search. We performed a manual search by considering exclusively articles published in the top-level software engineering conferences[3] and international journals[4] according to well known sources in the field. Table 4.2 shows the considered conferences and journals. The time span of our search ranges from January 2007[5] to December 2016. This step resulted in a total of 8,402 potentially relevant studies distributed across more than 9 years of research in software engineering.

2. Apply selection criteria. Each study was filtered according to a set of well-defined selection criteria. The adopted criteria are detailed in Section 4.1.2.1. An adaptive reading depth was applied in order to carry out the selection process in a time-efficient and objective manner [76]. This step resulted in a total of 38 potentially relevant studies. This significant reduction of the number of potentially relevant studies is due to the fact that (*i*) we considered exclusively top-level venues in the field of software engineering, and (*ii*) the considered venues are quite general, with static analysis of mobile apps being only one of the many topics of interest of those venues.

[3] `http://goo.gl/auU7su`

[4] `http://www.webofknowledge.com`

[5] Given that the concept of mobile app exists only since 2007

Conferences	**#Studies**	**Journals**	**#Studies**
International Conference on Software Engineering (ICSE)	810	IEEE Transactions on Software Engineering (TSE)	616
European Software Engineering Conference (ESEC)\ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)	638	ACM Transactions on Software Engineering and Methodology (TOSEM)	205
International Conference on Fundamental Approaches to Software Engineering (FASE)	285	Information and Software Technology (IST)	1026
IEEE/ACM International Conference on Automated Software Engineering (ASE)	624	Automated Software Engineering (ASE journal)	149
ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)	480	Software Maintenance & Evolution - Research & Practice (JSEP)	352
European Conference on Object-Oriented Programming (ECOOP)	275	Software and Systems Modeling (SoSyM)	381
International Symposium on Software Testing and Analysis (ISSTA)	317	Empirical Software Engineering (ESEJ)	371
		Journal of Systems and Software (JSS)	1873
Total	3429	**Total**	4973

TABLE 4.2: Searched data sources

3. Backward and forward snowballing. In this step, we applied backward and forward snowballing in order to take into account also studies that are published outside the conferences and journals considered in the previous step. In particular, this process was carried out by considering the studies selected in the initial search, and subsequently selecting relevant papers among those cited by the initially selected ones. This method is commonly referred to as a *backward snowballing* activity [75].

In addition to the backward snowballing, we also analyzed the research citing the studies selected through the initial search. This process is usually referred to as a *forward snowballing* activity [75]. Specifically, we included this further literature search method in order to consider also newer studies that, at that time, had not been included in official journal volumes or conference proceedings yet.

Regarding the forward snowballing process, the *Google Scholar*[6] bibliographic database was adopted to retrieve the studies citing the ones selected through the initial search phase.

The final decision about the inclusion of the papers was based on the adherence of the full text of the studies to the predefined selection criteria presented in Section 4.1.2.1. This step resulted in a total of 154 potentially relevant studies. The total number of potentially relevant studies increased significantly since in this step we considered papers published in all research venues, which by definition are far more than the top-level ones.

[6] `https://scholar.google.it/`

4. Exclude studies during data extraction activity. While reading in details each potentially relevant study, we agreed that 14 studies were semantically out of the scope of this research, so they were excluded. This final step led us to the final set of 140 primary studies.

4.1.2.1 Selection criteria

Following the guidelines for systematic literature review for software engineering [71], in order to reduce the likelihood of biases, we defined a set of inclusion and exclusion criteria beforehand. In the following, we detail the set of inclusion and exclusion criteria that guided the selection of the potentially relevant studies. A potentially relevant study was included if it satisfied *all* the inclusion criterion stated below; whereas, it was discarded if it satisfied *at least one* of the exclusion criteria reported below. In order to reduce possible biases, three researchers performed the studies selection independently.

Inclusion criteria

I1 Studies proposing or using a static analysis method or technique for mobile apps.

I2 Studies in which the static analysis method or technique takes as input one or more mobile applications in the form of binary files or source code.

I3 Studies providing some kind of evaluation of the proposed method or technique (*e.g.,* via formal analysis, controlled experiment, exploitation in industry, application to a simple example).

Exclusion criteria

E1 Studies not describing any implementation of the proposed method or technique.

E2 Studies in which the static analysis method or technique takes as input only store metadata (*e.g.,* user reviews, ratings) or other app artifacts (*e.g.,* manifest files).

E3 Secondary or tertiary studies (*e.g.,* systematic literature reviews, surveys).

E4 Studies in the form of editorials, tutorial, short, and poster papers, because they do not provide enough information.

E5 Studies not published in English language.

E6 Studies not peer reviewed.

4.1.3 Data extraction

This phase concerns (*i*) the creation of a classification framework for the primary studies, and (*ii*) the collection of data from each primary study.

In order to carry out a rigorous data extraction process, as well as to ease the control and the subsequent analysis of the extracted data, a predefined data extraction was designed prior the data extraction process. The data extraction form is composed of the various categories of the classification framework. For each primary study, the principal researchers collected a record with the extracted information in the data extraction form for subsequent analysis.

TABLE 4.3: Overview of the classification framework

Research trends (RQ1)	
• Year of publication	• Publication venue
• Publication venue type	• Analysis goal
• Macro analysis goal	• Paper goal
Characteristics (RQ2)	
• Platform specificity	• Implementation
• Static/hybrid approach	• Usage of machine learning
• App artifact	• Additional inputs
• Analysis pre-steps	• Analysis technique
Potential for industrial adoption (RQ3)	
• Target stakeholder	• Tool availability
• Technology readiness level	• Execution time
• Number of analysed apps	• Apps provenance
• Evaluation soundness	• Industry involvement

As suggested in [12], the principal researchers piloted the data extraction form independently. In order to validate our data extraction strategy, we performed a sensitivity analysis to check whether the results were consistent, independently from the researcher performing the analysis. More specifically, the principal researchers considered a random sample of 5 primary studies and analyzed them independently by filling the data extraction form for each of them. Then, each disagreement was discussed and resolved with the intervention of the research methodologist.

The classification framework is composed of three distinct parts, one for each research question of our study[7]. The overview of each part of the classification framework is reported in Table 4.3, whereas the definition of each specific parameter is given in Sections 4.2, 4.3, and 4.4.

[7]For the sake of simplicity, we do not report standard publication information (e.g., study ID, title, search strategy), they are available in the replication package.

4.1.4 Data synthesis

The data synthesis activity involves collating and summarising the data extracted from the primary studies [77, § 6.5] with the main goal of understanding, analysing, and classifying current research on static analysis of mobile apps.

Our data synthesis was split into two main phases: vertical analysis and horizontal analysis. When performing **vertical analysis**, we analyzed the extracted data to find trends and collect information about each parameter of each category of our classification framework. When performing **horizontal analysis**, we analysed the extracted data to explore possible relations across different parameters of our classification framework. We used contingency tables for evaluating the actual existence of those relations[8].

In both phases, we performed a combination of content analysis (mainly for categorizing and coding the studies under broad thematic categories) and narrative synthesis (mainly for explaining in details and interpreting the findings coming from the content analysis). During the horizontal analysis, we used contingency tables for evaluating the actual existence of inter-parameter relations.

4.2 Results - research trends (RQ1)

4.2.1 Year of publication

An overview of the year of publication of the primary studies is depicted in Figure 4.2. Overall, the publication rate results to be constantly increasing through time. The only year that is not in line with this trend is 2013, but the decrease of the publication rate of this year can be seen as negligible (-4 publications with respect to previous year). A steep increase of publication rate can be noticed between the years 2011-2012 and 2015-2016, with a difference of 13 publications. We can conjecture that the first steep increase (years 2011-2012) is due to the popularity gained in those years by the operating system Android 4.0. The appearance of lightweight static analysis approaches for mobile application, *e.g.,* Flowdroid [79], could instead be one of the root causes of the increase of publications between the years 2013 and 2014. No publication was found before the year 2011. We conjecture that the topic considered (static analysis methods for mobile) was not a popular research topic before 2011. Additionally the paper selection procedure (manual scan of venues integrated with snowballing) might have influenced this results. Regarding the year 2017, the low number can be attributed to the period in which the paper selection was carried out, i.e., mid 2016 till January 2017.

[8]For our horizontal analysis we applied the same process as the one in [78, § 4.4].

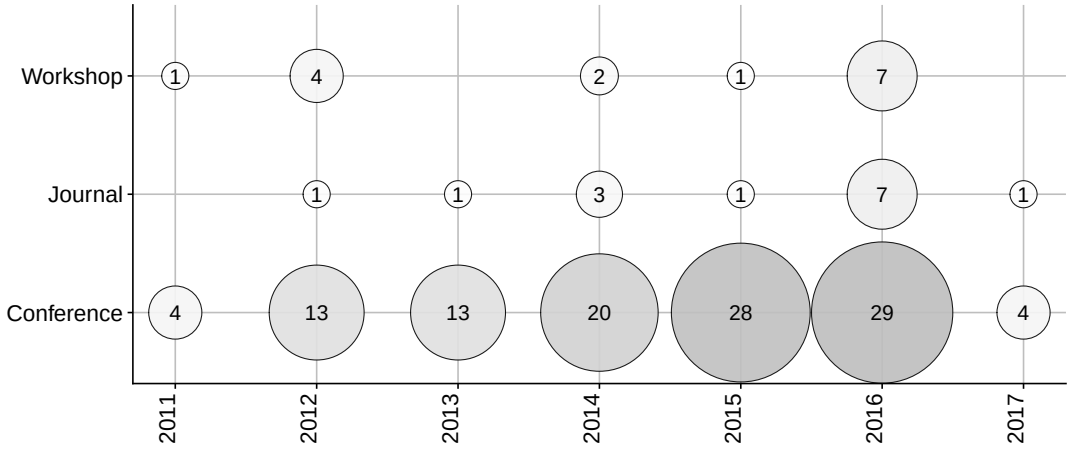


FIGURE 4.2: Bubble plot of primary studies by year and venue type

4.2.2 Publication venue

Studies on static analysis of mobile apps have been published to a certain extent in all the most prominent top-level conferences and journals in software engineering. An overview of the most targeted venues and the papers there published is reported in Table 4.4. The ASE conference results to be the venue in which most studies on this topic were published (21/140), followed by ICSE (12/140). Overall, a high heterogeneity can be found in the publication venues, which led to a total number of 71 different venues. Only a small number of venues results to be focused on mobile related topics. The vast majority results to be focused on general topics, *e.g.,* software engineering, security, testing and program analysis.

4.2.3 Publication venue type

As shown in Table 4.5, most of the papers were published in conferences (111/140), followed by journals (14/140) and workshops (15/140). The higher number of conference papers might be due to the high pace of technological advances in the topic. Specifically, researchers may have focused more on timely publications in conferences, rather than targeting journals, which have a (usually) slower publication timeline. Interestingly, as shown in Figure 4.2, 7 out of 14 journal papers were published in 2016, meaning that the the application of static analysis techniques to mobile apps is maturing as a scientific topic.

TABLE 4.4: Most targeted publication venues

Publication venue (acronym)	**#Studies**	**Studies**
Automated Software Engineering (ASE)	21	P22, P23, P24, P25, P26, P33, P39, P44, P46, P52, P53, P54, P55, P56, P57, P89, P111, P113, P116, P120, P135
International Conference on Software Engineering (ICSE)	12	P20, P40, P43, P49, P50, P71, P72, P73, P88, P97, P98, P115
Conference on Computer and Communications Security (CCS)	8	P12, P19, P93, P95, P109, P119, P134, P137
Network and Distributed System Security Symposium (NDSS)	7	P3, P64, P68, P84, P86, P132, P140
Symposium on Applied Computing (SAC)	6	P32, P83, P85, P106, P112, P124
International Symposium on Software Testing and Analysis (ISSTA)	5	P28, P60, P61, P66, P117
Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)	5	P67, P92, P108, P118, P121
Other	76	P1 P2 P4 P5 P6 P7 P8 P9 P10 P11 P13 P14 P15 P16 P17 P18 P21 P27 P29 P30 P31 P34 P35 P36 P37 P38 P41 P42 P45 P47 P48 P51 P58 P59 P62 P63 P65 P69 P70 P74 P75 P76 P77 P78 P79 P80 P81 P82 P87 P90 P91 P94 P96 P99 P100 P101 P102 P103 P104 P105 P107 P110 P114 P122 P123 P125 P126 P127 P128 P129 P130 P131 P133 P136 P138 P139

TABLE 4.5: Publication venue type

Publication venue type	**#Studies**	**Studies**
Conference	111	P1, P2, P3, P4, P6, P10, P11, P13, P16, P17, P19, P20, P21, P22, P23, P24, P25, P26, P27, P28, P32, P33, P34, P37, P38, P39, P40, P41, P42, P43, P44, P45, P46, P49, P50, P51, P52, P53, P54, P55, P56, P57, P58, P59, P60, P61, P62, P64, P65, P66, P67, P68, P69, P70, P71, P72, P73, P75, P76, P77, P78, P79, P81, P83, P84, P85, P86, P87, P88, P89, P91, P92, P93, P94, P95, P96, P97, P98, P99, P100, P103, P104, P105, P106, P108, P109, P110, P111, P112, P113, P114, P115, P116, P117, P119, P120, P121, P122, P124, P125, P126, P129, P131, P132, P133, P134, P135, P136, P137, P138, P139
Workshop	15	P7, P8, P12, P14, P18, P47, P48, P74, P82, P102, P118, P123, P128, P130, P140
Journal	14	P5, P9, P15, P29, P30, P31, P35, P36, P63, P80, P90, P101, P107, P127

4.2.4 Analysis goal

The *analysis goal* represents the principal purposes for which the static analyses approaches were conceived. By carefully analyzing the primary studies, twelve main analysis goal categories were identified through the keywording process. In Table 4.6, the comprehensive mapping of primary studies to analysis goals is reported. The most recurrent goals result to be *privacy* (50/140), *malware* (31/140), *inter-component communication* (20/140) and *energy* (18/140).

From an inspection of the more recurrent goals, we can observe that most of the studies focus either on analysing crucial aspects of the mobile ecosystem (*e.g., privacy* and *malware*) or on improving existing analysis methods (*e.g., inter/intra-component communication*). We can conjecture that this trend may be due to the fast pace of development that usually characterises mobile applications, where new app releases must be quickly developed and tested in order to be published in the app stores. This may lead to a lack of interest to analyse less critical software aspects of the app, such as refactoring the code of the app itself or identifying specific code anti-patterns.

TABLE 4.6: Analysis goal
(Categories not mutually exclusive)

Analysis goal	**#Studies**	**Studies**
Privacy leaks identification	50	P2, P10, P11, P12, P13, P14, P15, P16, P26, P38, P50, P61, P62, P65, P66, P67, P68, P73, P75, P76, P78, P81, P82, P83, P84, P89, P92, P93, P94, P101, P103, P104, P106, P109, P110, P111, P119, P120, P121, P122, P123, P124, P125, P131, P132, P134, P135, P137, P138, P140
Malware detection	31	P1, P3, P5, P9, P17, P29, P30, P36, P41, P49, P57, P58, P60, P61, P63, P70, P85, P86, P88, P95, P96, P98, P99, P105, P108, P112, P114, P118, P126, P130, P133
Inter-comp. communication	20	P10, P19, P35, P54, P56, P57, P64, P70, P71, P72, P79, P81, P82, P83, P91, P103, P109, P111, P113, P125
Energy assessment	18	P20, P21, P25, P34, P42, P43, P44, P47, P48, P74, P90, P97, P100, P115, P116, P117, P128, P129
Inter-app communication	14	P12, P14, P35, P37, P39, P54, P55, P56, P82, P96, P104, P111, P125, P139
Testing	12	P7, P8, P18, P22, P26, P27, P33, P44, P80, P107, P127, P136
Performance	8	P6, P25, P33, P51, P52, P87, P97, P102
Resource usage	6	P44, P45, P46, P59, P97, P116
Framework	6	P23, P31, P32, P40, P77, P91
Refactoring	5	P34, P52, P59, P84, P87
Reflection	5	P4, P24, P28, P57, P69
Anti-patterns	1	P53

4.2.5 Macro analysis goal

The *macro analysis goal* refers to the generic goal considered by the static analyses. This parameter can have three distinct values, namely: (*i*) *external quality*, if the approach evaluates some external quality attribute, *e.g.,* performance; (*ii*) *internal quality*, if the

approach evaluates some internal quality attribute, *e.g.,* maintainability; (*iii*) *improving of methodology*, if the approach is conceived to improve a static analysis technique.

TABLE 4.7: Macro analysis goal
(Categories not mutually exclusive)

Macro analysis goal	**#Studies**	**Studies**
External quality	104	P1, P2, P3, P5, P6, P9, P10, P11, P12, P13, P14, P15, P16, P17, P20, P21, P25, P26, P29, P30, P33, P34, P36, P38, P41, P42, P43, P44, P47, P48, P49, P50, P51, P52, P57, P58, P60, P61, P62, P63, P65, P66, P67, P68, P70, P73, P74, P75, P76, P78, P81, P82, P83, P84, P85, P86, P87, P88, P89, P90, P92, P93, P94, P95, P96, P97, P98, P99, P100, P101, P102, P103, P104, P105, P106, P108, P109, P110, P111, P112, P114, P115, P116, P117, P118, P119, P120, P121, P122, P123, P124, P125, P126, P128, P129, P130, P131, P132, P133, P134, P135, P137, P138, P140
Methodology improvement	38	P4, P10, P12, P14, P19, P23, P24, P28, P31, P32, P35, P37, P39, P40, P54, P55, P56, P57, P64, P69, P70, P71, P72, P77, P79, P81, P82, P83, P91, P96, P97, P103, P104, P109, P111, P113, P125, P139
Internal quality	21	P7, P8, P18, P22, P26, P27, P33, P34, P44, P45, P46, P52, P53, P59, P80, P84, P87, P107, P116, P127, P136

The macro analysis goals considered by the primary studies are reported in Table 4.7. The majority of the primary studies focus on external quality (104/140). A smaller amount of studies focuses on the improvement of static analysis methodologies (38/140) and only a small portion addresses internal quality (21/140)[9]. From this data, we conjecture that the high pace of the mobile technological advances and the strong role of end users in the mobile ecosystem are leading researchers to give more importance to external qualities. Research aimed to refine static analysis approaches results to be higher than the ones focusing on internal quality, making us conjecture that the ones considering internal quality are either at an early stage of development or have been less explored than the ones improving the existing methods. In addition, the distribution of macro analysis goals throughout the years is depicted in Figure 4.3. Here we observe that, although studies focusing on methodology improvement have been the majority in each of the considered years, a steady increase in number can be observed for studies that focus on either external or internal qualities, starting from the year 2013.

4.2.6 Paper goal

This parameter can be of two types, namely: (*i*) *Quality attribute assessment*, if the research reported in the primary study focuses on assessing a quality attribute of mobile apps (*e.g.,* security); (*ii*) *Improvement of methodology*, if the research reported in the primary study focuses on improving existing static analyses for mobile apps.

[9]It is important to note that these categories are not mutually exclusive, i.e., a paper could be mapped to more than one category if it addresses more than one type of goal

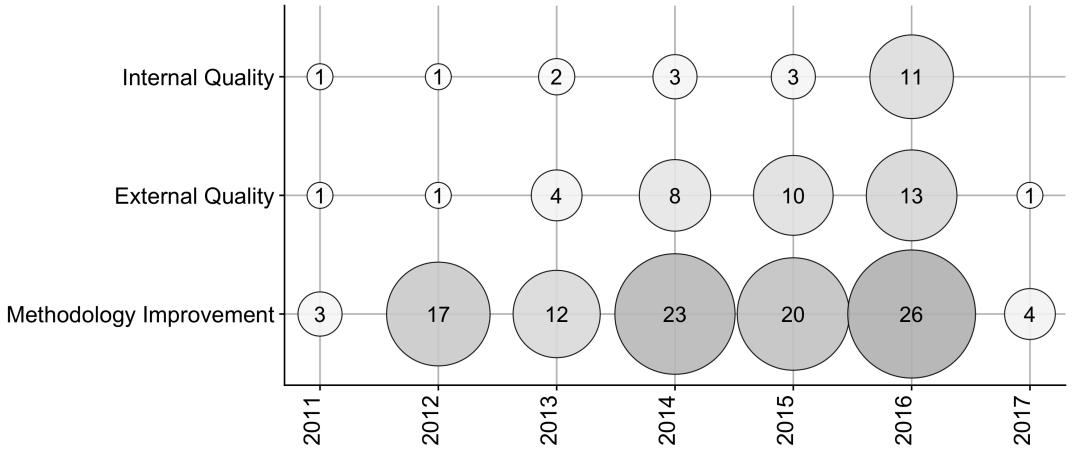


FIGURE 4.3: Macro analysis goal by year

The goals taken into account by the primary studies is documented in Table 4.8. The majority of the primary studies (110/140) focusses on the assessment of some quality attribute(s) of mobile apps. Only a lower number instead (30/140) considers the improvement of static analysis techniques. We can conjecture that this trend can be associated to the more "immediate impact", e.g., ease of adoption and real-life utilization by practitioners. From this, we can conjecture that, while a certain maturity with respect to assessment of apps quality attributes has been achieved (and hence a high presence of such approaches is observable), techniques improving the existing methods are still not yet explored.

TABLE 4.8: Paper goal

Paper goal	**#Studies**	**Studies**
Quality attribute assessment	110	P1, P2, P3, P5, P6, P9, P10, P11, P12, P13, P14, P15, P16, P17, P18, P20, P21, P25, P29, P30, P34, P36, P38, P40, P41, P42, P43, P44, P45, P46, P47, P48, P49, P50, P51, P52, P53, P54, P57, P58, P59, P61, P62, P63, P65, P66, P67, P68, P70, P73, P74, P75, P76, P77, P78, P81, P82, P83, P84, P85, P86, P87, P88, P89, P90, P92, P93, P94, P95, P96, P97, P98, P99, P100, P101, P102, P103, P104, P105, P106, P107, P108, P109, P110, P111, P112, P114, P115, P116, P117, P118, P119, P120, P121, P122, P123, P124, P125, P126, P128, P129, P130, P131, P132, P133, P134, P135, P137, P138, P140
Improvement of methodology	30	P4, P7, P8, P19, P22, P23, P24, P26, P27, P28, P31, P32, P33, P35, P37, P39, P55, P56, P60, P64, P69, P71, P72, P79, P80, P91, P113, P127, P136, P139

Main findings on research trends:

- The intensity of research on static analysis for mobile apps is growing year by year, specially after the introduction of Android 4.0 and app-specific techniques

have been devised (*e.g.,* Flowdroid).

- Researchers are targeting primarily conferences (*e.g.,* ASE and ICSE), even if workshop and journal publications have been much more targeted in 2016 (the last full year considered in this study).
- Many of the approaches are focussing on security-related concerns, such as privacy leaks identification and malware detection.
- Approaches for enhancing the modeling and analysis of both inter-component communication (*e.g.,* intent raising across Android activities) and inter-app communication are receiving quite an intensive scientific attention.
- The vast majority of primary studies is targeting externally visible quality attributes (*e.g.,* security, energy) with respect to internally visible ones (*e.g.,* maintainability, resource usage, code anti-patterns).
- Reasonable research effort is being devoted to improvement of the methodology, such as devising more sound static analyses, support for more events in the mobile components lifecycles (*e.g.,* Android intents sharing).

4.3 Results - characteristics of approaches (RQ2)

4.3.1 Platform specificity

This parameter identifies whether the proposed approach is specifically designed for a specific platform (*e.g., Android* or *iOS*) or if it is *generic* and can in principle be applied to any platform. As shown in Table 4.9, the vast majority of the approaches (119/140) presents an analysis approach specific for Android; only one study (1/140) presents an approach specific for iOS. A smaller amount of studies (20/140) presents an approach that is *generic*. Possible reasons for this imbalance may be due to the popularity and the open-source nature of the Android platform, which eases the effort required by researchers during the design of new analyses. Furthermore, Android app binaries can be straightforwardly disassembled with off-the-shelf software libraries (*e.g.,* apktool[10], dex2jar[11]), and their internal structure and contained static resources are easily analyzable in an automatic way.

[10] `http://ibotpeaches.github.io/Apktool`
[11] `http://github.com/pxb1988/dex2jar`

TABLE 4.9: Platform specificity

Platforms	#Studies	Studies
Android	119	P1, P3, P4, P5, P6, P8, P9, P10, P11, P12, P13, P14, P15, P16, P17, P18, P19, P21, P22, P23, P24, P25, P26, P27, P28, P29, P30, P32, P33, P34, P35, P36, P37, P38, P39, P40, P41, P42, P44, P45, P46, P47, P48, P49, P50, P51, P52, P53, P54, P55, P56, P58, P60, P61, P62, P64, P65, P66, P67, P68, P69, P70, P71, P73, P74, P75, P77, P78, P79, P81, P82, P83, P84, P85, P86, P87, P89, P90, P91, P92, P94, P96, P97, P98, P99, P100, P102, P103, P104, P105, P106, P107, P108, P109, P110, P111, P112, P114, P116, P118, P121, P122, P123, P124, P125, P126, P127, P128, P129, P130, P131, P132, P133, P134, P135, P136, P137, P138, P139
Generic	20	P2, P7, P20, P31, P43, P57, P59, P63, P72, P76, P80, P88, P93, P95, P101, P113, P115, P117, P119, P120
iOS	1	P140

4.3.2 Implementation

Values for the *implementation* parameter, summarized in Table 4.10, were extracted from the primary studies according to whether the implementation used for evaluation purposes is implemented for a specific platform, *e.g.,* Android or iOS, or it is Generic, applicable to apps developed for any platform. Almost all the studies (136/140) implement the proposed approach exclusively for the Android platform. Two studies present approaches (2/140) having a generic implementation, applicable to any platform. Only one study (1/140) presents an approach that is implemented specifically for the iOS platform. Other less popular platforms are almost completely absent, with only one study (1/140) implementing the proposed analysis on TouchDevelop scripts [80]. We speculate that the reason for this disproportion, in addition to the ones already evidenced in the discussion of the *platform specificity* parameter, stem from the fact that some of the most popular static analysis frameworks (*e.g.,* Soot [81] and WALA [82]) are adapted to support analysis of Android apps. The same cannot be said for the other platforms and, hence, researchers interested in performing static analysis on apps designed for those platforms experience a higher barrier to entry as they must develop their own tools, often from scratch.

4.3.3 Static/hybrid approach

The *static/hybrid approach* parameter describes whether an approach relies on static analysis only (Static) or utilizes some form of dynamic analysis also (Hybrid). Results for the extraction of this parameter are summarized in Table 4.11. The preponderance of the studies (112/140) present an approach that relies on static analysis only. Nonetheless, a considerable amount of them (28/140) present an approach that complements static analysis with dynamic one. The presence of dynamic analysis in a considerable portion

TABLE 4.10: Implementation

Implem.	**#Studies**	**Studies**
Android	136	P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11, P12, P13, P14, P15, P16, P17, P18, P19, P20, P21, P22, P23, P24, P25, P26, P27, P28, P29, P30, P31, P32, P33, P34, P35, P36, P37, P38, P39, P40, P41, P42, P43, P44, P45, P46, P47, P48, P49, P50, P51, P52, P53, P54, P55, P56, P57, P58, P59, P60, P61, P62, P63, P64, P65, P66, P67, P68, P69, P70, P71, P72, P73, P74, P75, P76, P77, P78, P79, P81, P82, P83, P84, P85, P86, P87, P88, P89, P90, P91, P92, P94, P95, P96, P97, P98, P99, P100, P101, P102, P103, P104, P105, P106, P107, P108, P109, P110, P111, P112, P113, P114, P115, P116, P117, P118, P119, P121, P122, P123, P124, P125, P126, P127, P128, P129, P130, P131, P132, P133, P134, P135, P136, P137, P138, P139
Generic	2	P93, P120
iOS	1	P140
Other	1	P80

of the studies can be explained by considering that, despite all its drawbacks, dynamic analysis still provides an invaluable contribution for a variety of purposes, such as privacy leaks detection, GUI-modeling, energy profiling. A further discussion on the fields where dynamic analysis is most common can be found in Section 4.5.

TABLE 4.11: Static Hybrid approach

Static Hybrid approach	**#Studies**	**Studies**
Static	112	P3, P4, P5, P7, P9, P11, P13, P15, P16, P17, P18, P19, P20, P21, P22, P23, P24, P25, P28, P29, P30, P31, P32, P33, P35, P36, P37, P39, P40, P41, P42, P43, P44, P46, P49, P50, P51, P52, P53, P54, P55, P56, P57, P58, P59, P60, P61, P64, P65, P66, P67, P68, P70, P71, P72, P73, P75, P77, P78, P79, P80, P81, P82, P83, P84, P85, P86, P87, P88, P89, P91, P92, P93, P95, P96, P97, P98, P99, P101, P103, P104, P105, P107, P109, P111, P112, P114, P115, P116, P119, P120, P121, P122, P123, P124, P125, P126, P127, P128, P129, P130, P131, P132, P133, P134, P135, P136, P137, P139, P139, P140
Hybrid	28	P1, P2, P6, P8, P10, P12, P14, P26, P27, P34, P38, P45, P47, P48, P62, P63, P69, P74, P76, P90, P94, P100, P102, P106, P108, P110, P113, P117, P118

4.3.4 Usage of machine learning techniques

Values for this parameter are summarized in Table 4.12. The possible values identify whether the approach under evaluation complements its analysis with machine learning techniques (Yes) or not (No). A vast majority of the studies (119/140) does not make use of machine learning in the proposed approach. The remaining studies (21/140) perform features extraction from the application source code or other intermediate representations (*e.g.,* a method-level call graph), and applies machine learning techniques on the extracted features. Machine learning techniques are widely used for some specific goals

(*e.g.,* malware detection), but their application to others has not been explored yet by researchers.

TABLE 4.12: Machine Learning

Machine Learning	**#Studies**	**Studies**
No	120	P2, P4, P6, P7, P8, P9, P10, P12, P13, P14, P17, P18, P19, P20, P21, P22, P23, P24, P25, P26, P27, P28, P31, P32, P33, P34, P36, P37, P38, P39, P40, P41, P42, P43, P44, P45, P46, P47, P48, P49, P50, P51, P52, P53, P54, P55, P56, P57, P58, P59, P61, P62, P64, P65, P66, P67, P68, P69, P71, P72, P73, P74, P75, P77, P79, P80, P81, P82, P83, P84, P87, P89, P90, P91, P92, P93, P94, P95, P96, P97, P99, P100, P101, P102, P103, P104, P105, P107, P108, P109, P110, P111, P112, P113, P115, P116, P117, P118, P119, P120, P121, P122, P123, P124, P125, P127, P128, P129, P130, P131, P132, P133, P134, P135, P136, P137, P138, P139, P140
Yes	20	P1, P3, P5, P11, P15, P16, P29, P30, P35, P60, P63, P70, P76, P78, P85, P86, P88, P98, P106, P114, P126

4.3.5 App artifact

The values of this parameter describe what formats are accepted as input by the selected studies for the apps to be analyzed. As shown in Table 4.13, the majority of the studies (122/140) accepts as input apps in the form of binary packages (Binary), *i.e.,* APK (Android PacKage) files for the Android platform or IPA (iPhone Application Archive) packages for the iOS platform. This implies that the proposed analysis can be performed by a variety of subjects (app store moderators, researchers, security experts, etc.), and not only by app developers. Nonetheless, a considerable amount of primary studies (23/140) takes as input the app source code (Source Code), hence targeting app developers and researchers. In those cases, developers can potentially integrate them into their development workflow, *e.g.,* as dedicated analyses integrated into the Android Studio IDE or as specific steps in their continuous integration pipeline. Note that both APK and source code are valid inputs for some of the studies.

4.3.6 Additional inputs

The possible values for the *additional inputs* parameter, listed in Table 4.14, identify what other inputs, if any, are required by the primary studies to perform the proposed analysis (in addition to the app itself). As an example, Figure 4.4 presents *eCalc*, the technique proposed by Hao *et al.* [83] (P129) that provides energy consumption estimates for an app. In this case, eCalc takes as input the app itself, together with the test cases exercising its features and a previously built CPU profile containing the energy cost functions for each type of instruction of the CPU. In P129, those additional inputs are needed for

TABLE 4.13: App artifact
(Categories not mutually exclusive)

App artifact	#Studies	Studies
Binary	123	P1, P2, P3, P4, P5, P8, P9, P10, P11, P12, P13, P14, P15, P16, P17, P20, P21, P22, P23, P24, P25, P26, P27, P28, P29, P30, P31, P32, P35, P36, P37, P38, P39, P40, P41, P42, P43, P44, P45, P46, P47, P48, P49, P50, P51, P53, P54, P55, P57, P58, P59, P60, P61, P62, P63, P64, P65, P66, P67, P68, P69, P70, P72, P73, P74, P75, P76, P77, P78, P82, P83, P84, P85, P86, P88, P89, P90, P91, P92, P94, P96, P97, P98, P99, P100, P101, P102, P104, P105, P106, P107, P108, P109, P110, P111, P112, P114, P115, P116, P117, P118, P119, P121, P122, P123, P124, P125, P126, P127, P128, P129, P130, P131, P132, P133, P134, P135, P136, P137, P138, P139, P140
Source code	22	P6, P7, P18, P19, P33, P34, P40, P52, P56, P68, P71, P79, P80, P81, P87, P93, P95, P103, P113, P116, P120, P130

automatically running and profiling the app under analysis multiple times in order to take into account the well-known phenomenon of energy consumption fluctuations at run-time.

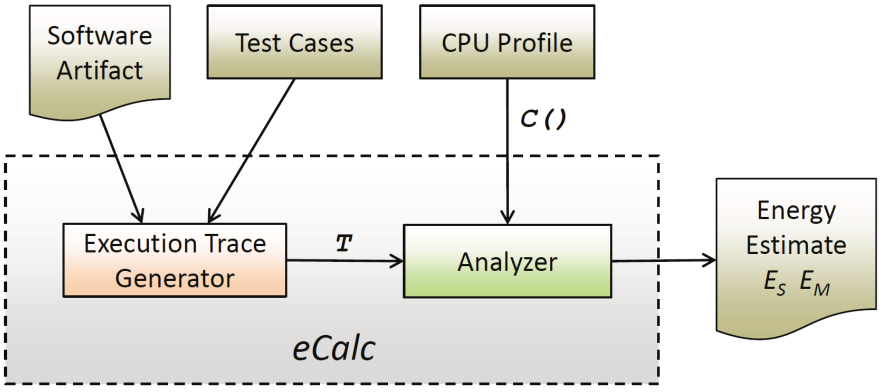


FIGURE 4.4: Example of an analysis technique requiring additional inputs

Overall, the majority of primary studies (109/140) is able to perform the analysis without any additional input, whereas 31/140 studies require some additional inputs. We consider this to be a positive trend, as it simplifies adoption of the proposed techniques by industry and other researchers, additionally enabling batch analysis of a large quantity of apps more easily. Nevertheless, as for P129, in some cases relying on additional inputs is a necessity, *e.g.,* when the app needs to be executed in a controlled, non-random, non-trivial manner.

When focusing on the studies requiring additional inputs, we can observe that mappings from the source code of the app to other auxiliary information is the most commonly required additional input (8/140). It is followed by techniques that verify whether given policies, rules or constraints are violated (5/140), and techniques that focus on a list of one or more methods of interest in the app source code (5/140). A few studies (3/140)

TABLE 4.14: Additional input
(Categories not mutually exclusive)

Additional input	**#Studies**	**Studies**
Source code mappings	8	P22, P57, P96, P111, P112, P118, P131, P137
Methods of interest	5	P1, P6, P67, P81, P91
Policies, rules & constraints	5	P7, P16, P92, P95, P125
Platform description	3	P15, P67, P90
App Store Descriptions	3	P16, P53, P88
System profile	1	P128
Bug information	1	P45
Target permission to revoke	1	P59
User defined Analysis	1	P77
Test cases	1	P128
Workload Description	1	P115

take as input app descriptions coming from the app stores, and leverage the information there contained in order to perform their analyses. For example, CHABADA [25] aims at automatically identifying malicious apps by evaluating how their implementation differs from their description in the app store. Some proposed techniques take as input the platform (3/140) or system (1/140) profiles for application execution. Only one study (1/140) takes as input test cases. This is particularly noteworthy as test cases are artifacts commonly produced during the software development cycle, and how information can be extracted from them has widely been investigated in the software engineering literature [84, 85]. Similarly, another study (1/140) leverages information extracted from bug reports to perform the analysis. A single study (1/140) focuses on the problem of removing unwanted system permissions and, consequently, takes as input an identifier of the permission to remove. Finally, a description of the workload to be executed (1/140) and a user defined analysis (1/140) are required by only one study respectively. It is important to notice that the vast majority of these additional inputs require the knowledge of a developer or a domain expert in order to be reproduced and only a handful can be reproduced by end-users. This makes it harder to reproduce the results and might hinder large-scale adoption.

4.3.7 Analysis pre-steps

The *analysis pre-steps* parameter identifies whether the studies under evaluation require steps that must be executed manually before the analysis can be performed. Results are listed in Table 4.15.

The majority of the approaches (111/140) does not require any analysis pre-step. A still considerable amount (29/140) requires some analysis pre-step to be performed manually. Examples of possible pre-steps include, but are not limited to, building models of the platform APIs or libraries used by the application under analysis, collecting execution

TABLE 4.15: Analysis pre steps

Analysis pre steps	**#Studies**	**Studies**
No presteps required	111	P2, P3, P4, P5, P6, P7, P8, P9, P10, P11, P13, P14, P18, P19, P20, P21, P22, P23, P24, P25, P26, P27, P28, P29, P30, P31, P32, P34, P36, P37, P38, P39, P40, P41, P42, P43, P44, P46, P47, P48, P49, P50, P51, P52, P54, P55, P56, P58, P59, P60, P61, P64, P65, P66, P67, P68, P69, P70, P71, P73, P74, P75, P78, P79, P80, P82, P83, P84, P86, P88, P89, P91, P92, P93, P94, P96, P97, P99, P101, P102, P103, P104, P105, P107, P108, P109, P110, P112, P113, P114, P116, P118, P119, P120, P121, P122, P123, P125, P126, P127, P129, P130, P132, P133, P134, P135, P136, P137, P138, P139, P140
Presteps required	29	P1, P12, P15, P16, P17, P33, P35, P45, P53, P57, P62, P63, P72, P76, P77, P81, P85, P87, P90, P95, P98, P100, P106, P111, P115, P117, P124, P128, P131

traces, collecting runtime power consumption measures, creating rule sets or security policies. Similarly to the previous parameter, having to perform manual steps before or during the application of a static analysis approach may hinder its reproducibility and large-scale adoption.

4.3.8 Analysis technique

This parameter identifies the family of static analysis techniques performed by the approaches proposed in the primary studies. Results are summarized in Table 4.16. A wide variety of static analysis techniques is used in the primary studies, the most common being Flow (82/140) and Taint Analysis (26/140). A considerable amount of primary studies limit their analysis to data mining (24/140) to extract relevant information from the application bytecode or source code. Machine learning classification, slicing and model-based analysis are also other relevantly used techniques, each being used in eighteen (18/140), eleven (11/140), and ten (10/140) studies, respectively. Other less frequently used techniques are string analysis (6/140), abstract interpretation (4/140), type inference (3/140), code instrumentation (3/140), symbolic execution (3/140), points-to analysis (2/140), constant propagation (2/140), termination analysis (1/140), typestate analysis (1/140), statistical analysis (1/140), responsiveness analysis (1/140), pattern-based contextual analysis (1/140), nullness analysis (1/140), and class analysis (1/140). We speculate that the popularity of Flow and Taint analysis is due to the fact that many of the issues researchers want to detect in mobile apps can be modeled under those analysis paradigms and, as further discussed in Section 4.5, it appears that researchers identify the technique to be used in a goal-driven fashion. We also believe that, again, researchers are limited by the available frameworks and tools, and choose to focus more on those techniques for which mature tools exist (*e.g.,* Soot).

TABLE 4.16: Analysis technique
(Categories not mutually exclusive)

Analysis technique	**#Studies**	**Studies**
Flow Analysis	82	P1, P4, P5, P6, P7, P8, P10, P11, P13, P14, P15, P17, P18, P19, P20, P21, P24, P25, P26, P27, P28, P30, P31, P34, P35, P36, P39, P45, P46, P47, P49, P50, P51, P52, P54, P55, P56, P57, P58, P62, P63, P64, P65, P67, P68, P71, P74, P77, P79, P83, P87, P89, P92, P93, P94, P95, P96, P98, P100, P104, P106, P107, P108, P111, P112, P115, P116, P118, P120, P123, P124, P125, P128, P129, P130, P132, P133, P134, P135, P137, P139, P140
Taint Analysis	26	P14, P23, P37, P38, P39, P40, P41, P44, P61, P66, P70, P72, P73, P75, P76, P81, P82, P84, P91, P96, P99, P103, P107, P122, P125, P131
Data Mining	24	P5, P16, P29, P33, P48, P60, P63, P65, P69, P76, P78, P85, P86, P88, P97, P105, P110, P114, P119, P126, P131, P135, P137, P138
Classification	18	P5, P16, P29, P36, P60, P63, P76, P78, P85, P86, P88, P105, P110, P114, P119, P126, P132, P133
Slicing	11	P2, P22, P33, P37, P59, P65, P84, P106, P112, P113, P140
Model Based Analysis	10	P3, P9, P11, P12, P38, P53, P74, P85, P90, P130
String Analysis	5	P4, P43, P96, P101, P121
Abstract Interpretation	4	P57, P80, P81, P136
Constant Propagation	4	P24, P28, P117, P121
Code Instrumentation	3	P32, P33, P117
Type inference	3	P4, P59, P95
Symbolic Execution	3	P12, P17, P109
Pointer Analysis	2	P4, P68
Nullness Analysis	1	P127
Termination Analysis	1	P127
Statistical Analysis	1	P48
Typestate Analysis	1	P13
Pattern-based Analysis	1	P42
Responsiveness Analysis	1	P102
Class Analysis	1	P127

Main findings on characteristics of approaches:

- Being Open source pays off from a scientific perspective. The vast majority of the studied approaches is specific to the Android platform, both from a conceptual and implementation perspective. Thanks to its open-source nature, Android gives more control and flexibility, and fuels an ecosystem of accompanying tools and libraries useful for static analysis (avoiding to reinvent the wheel). This is also proved by the fact that Android has been chosen as implementation platform also for generic static analysis approaches.
- Static analysis of mobile apps is widely performed in isolation and by considering only the app to be analysed (no additional input like test cases or platform profiles). If on one side this is a confirmation of the fact that static analysis is a very versatile tool for analysing non-trivial properties of mobile apps, on the

other side, researchers may be loosing an opportunity for pushing further by complementing static analysis with other artifacts and/or additional analysis techniques (*e.g.,* like done in the eCalc approach in P129).

- Machine learning techniques seem to be promising and are applied in conjunction with static analysis techniques. However, machine learning techniques are widely used for some goals (mainly for security), but they are yet fully explored in other areas, such as app store analysis [86] and software repository mining.
- Many are the static analysis techniques used by researchers when considering mobile apps, ranging from flow analysis, to taint analysis, to type inference and abstract interpretation. The clear winner is flow analysis. We conjecture that this success is mainly due to a combination of factors: (*i*) as of today the programming model of mobile platforms is inherently based on a flow of (often asynchronous) messages exchanged between a set of components (*e.g.,* Android activities, iOS views) reacting to events (*e.g.,* a tap of the user, a callback from a sensor request), (*ii*) flow analysis nicely lends itself to identify and predicate on both intra- and inter-app interactions (a cornerstone capability for security and reliability analyses), and (*iii*) the availability of open-source tools like Soot that developers can use as building blocks for their own approaches.

4.4 Results - potential for industrial adoption (RQ3)

4.4.1 Target stakeholder

As shown in Table 4.17, app *developers* are the most recurrent stakeholders of static analysis approaches (76/140). *Platform vendors* (59/140) like Apple and Google distribute apps via their own dedicated mobile application markets. They can benefit from the use of static analysis approaches for systematically assessing the level of quality of their distributed apps, possibly identifying those apps with an unacceptable level of quality (*e.g.,* apps with well-known security flaws, apps asking for suspicious permissions, apps with strong energy inefficiencies). Interestingly, some approaches directly target app *users*, who might use static analyses to better understand how their installed apps behave and for examining and granting explicit information flows within an application. Also, users may be interested in implicit information flows across multiple applications, such as permissions for reading the phone number and sending it over the network. As an example, one of the 12 studies targeting users focuses on debugging energy efficiency of apps in

their real context of use. Specifically, in P42 the user can launch an automatically instrumented app to precisely record and report observed energy-related failures in order to assists the developer by automatically localizing the reported defects and suggesting patch locations. Last but not least, 7 primary studies explicitly mention *researchers* as target stakeholders, who can extend and/or apply the proposed techniques (and their results) to their own studies on mobile applications.

TABLE 4.17: Target stakeholder
(Categories not mutually exclusive)

Target stakeholder	**#Studies**	**Studies**
App developer	76	P1, P4, P6, P7, P8, P10, P13, P14, P15, P16, P18, P19, P20, P21, P22, P24, P25, P26, P27, P28, P31, P33, P34, P39, P42, P43, P44, P45, P46, P47, P48, P49, P50, P51, P52, P53, P54, P55, P56, P57, P59, P61, P62, P64, P67, P74, P79, P80, P81, P82, P83, P84, P85, P87, P90, P91, P93, P97, P100, P101, P102, P104, P106, P107, P113, P115, P116, P117, P121, P125, P127, P128, P129, P134, P136, P139
Platform vendor	59	P2, P3, P5, P9, P12, P13, P16, P17, P18, P29, P30, P35, P36, P37, P40, P41, P54, P57, P58, P60, P63, P65, P66, P69, P70, P73, P75, P76, P77, P82, P85, P88, P89, P92, P94, P95, P96, P98, P99, P103, P105, P108, P109, P112, P114, P118, P122, P123, P126, P130, P131, P132, P133, P134, P135, P137, P138, P139, P140
User	12	P11, P38, P42, P68, P78, P86, P92, P110, P111, P120, P124, P139
Researcher	7	P19, P23, P32, P71, P72, P91, P119

4.4.2 Tool availability

All the primary studies contribute with a tool implementing the proposed approach. Nonetheless, our results also show that only 28 studies over 140 (see Table 4.18) released the tool, making it publicly available for download and adoption. When possible, the availability of a tool supporting the proposed approach is desirable as it surely helps in making the obtained results more credible, reproducible, and replicable by the community.

4.4.3 Technology readiness level

Defined by the systematic measurement system for assessing the maturity of a particular technology [87], the technology readiness level (TRL) is an integer n where $1 \leq n \leq 9$. This measure has been proposed by the Horizon 2020 European Commission for the 2014/2015 work program[12]. In the context of this study, we assess the technology

[12] `http://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf`

TABLE 4.18: Tool availability

Tool availability	**#Studies**	**Studies**
No	112	P1, P2, P5, P6, P7, P8, P9, P10, P11, P12, P15, P16, P17, P19, P21, P23, P24, P25, P27, P29, P30, P31, P32, P34, P35, P37, P38, P39, P40, P41, P42, P45, P46, P47, P48, P49, P51, P52, P53, P54, P55, P56, P57, P58, P59, P61, P62, P63, P64, P65, P66, P67, P68, P70, P71, P72, P73, P74, P75, P76, P77, P78, P79, P80, P81, P84, P85, P86, P88, P90, P92, P93, P94, P96, P98, P99, P100, P101, P102, P103, P105, P106, P107, P108, P109, P110, P113, P114, P115, P116, P117, P118, P119, P120, P121, P122, P123, P124, P125, P126, P128, P129, P130, P131, P132, P133, P134, P135, P137, P138, P139, P140
Yes	28	P3, P4, P13, P14, P18, P20, P22, P26, P28, P33, P36, P43, P44, P50, P60, P69, P82, P83, P87, P89, P91, P95, P97, P104, P111, P112, P127, P136

readiness level of an approach by using three values (see Table 4.19): *high* if the approach is evaluated or adopted in an industrial environment ($n \geq 7$), *medium* if the approach has been applied on real apps, e.g., those mined from the Google Play store ($5 \leq n \leq 6$), *low* if the approach is evaluated via ad-hoc, synthetic apps ($n \leq 4$). Our analysis reveals that the majority of primary studies have a *medium* TRL (117/140), followed by *low* (22/140), and *high* (1/140). The only primary study classified with an high TRL is P96 since the proposed approach is adopted in an industrial environment, in addition to having been evaluated by using an large dataset of real applications.

TABLE 4.19: Technology Readiness Level (TRL)

TRL	**#Studies**	**Studies**
High	1	P96
Medium	117	P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P12, P14, P16, P17, P18, P19, P20, P21, P22, P24, P25, P27, P28, P29, P30, P31, P32, P33, P34, P35, P36, P37, P38, P39, P41, P42, P43, P44, P46, P47, P48, P49, P50, P52, P53, P54, P55, P56, P58, P59, P62, P63, P64, P65, P66, P67, P68, P69, P70, P71, P72, P73, P74, P75, P76, P77, P78, P79, P81, P83, P85, P86, P87, P88, P89, P90, P91, P93, P94, P97, P98, P99, P100, P101, P102, P103, P104, P105, P106, P107, P109, P110, P111, P112, P114, P115, P116, P117, P119, P120, P121, P122, P123, P125, P126, P129, P130, P131, P132, P133, P134, P135, P136, P137, P138, P139, P140
Low	22	P11, P13, P15, P23, P26, P40, P45, P51, P57, P60, P61, P80, P82, P84, P92, P95, P108, P113, P118, P124, P127, P128

4.4.4 Execution time

The classical argument against static analysis is its long execution time, which may impact its applicability in the everyday workflow of developers. With this parameter we want to investigate on this and report the average execution time of each static analysis approach, as reported in its corresponding primary study. Specifically, we categorize

execution times according to the following sets: *low* if the analysis execution time is less than 1 minute, *medium* if the analysis execution time is between 1 and 10 minutes, *high* if the analysis execution time is higher than 10 minutes, and *NA* if the execution time is not reported in the primary study. Unfortunately, as shown in Table 4.20, in many primary studies the researchers are not reporting the execution time of their proposed approach (*NA* values in the table). When the execution time is reported, in more than half of primary studies the execution time is *medium* (49/90), followed by *low* (34/90), and *High* (7/90). This result shows that static analysis approaches can be executed in acceptable time, depending on the used approach.

TABLE 4.20: Execution time

Execution time	**#Studies**	**Studies**
High	7	P1, P19, P20, P31, P88, P89, P100
Medium	49	P3, P5, P6, P8, P15, P16, P21, P25, P28, P29, P30, P33, P36, P39, P40, P41, P42, P43, P45, P47, P49, P50, P55, P56, P60, P69, P74, P75, P77, P84, P85, P90, P94, P97, P98, P99, P104, P107, P108, P109, P110, P111, P113, P115, P117, P122, P123, P127, P136
Low	34	P7, P10, P11, P14, P17, P18, P22, P24, P35, P44, P48, P61, P62, P64, P65, P66, P67, P76, P79, P80, P83, P86, P87, P91, P93, P103, P112, P116, P128, P129, P131, P132, P133, P134
NA	50	P2, P4, P9, P12, P13, P23, P26, P27, P32, P34, P37, P38, P46, P51, P52, P53, P54, P57, P58, P59, P63, P68, P70, P71, P72, P73, P78, P81, P82, P92, P95, P96, P101, P102, P105, P106, P114, P118, P119, P120, P121, P124, P125, P126, P130, P135, P137, P138, P139, P140

4.4.5 Number of analysed apps

The authors of the analyzed primary studies evaluate and validate their findings by using an input set of applications. The evaluation of this parameter builds on the assumption that approaches evaluated on a larger set of apps are more adoptable in industry since it is less likely that they exhibit unexpected behaviors (specially for corner cases). Here we categorized the primary studies according to the number of apps used for evaluating them. As shown in Table 4.21, in the majority of studies (58/140) the number of applications used for evaluating the proposed approach is greater than 1,000, followed by those studies which evaluated their approach by using less than 100 apps (50/140), and those studies (32/140) which took into account a medium set of apps (between 100 and 1,000). This result is promising in that a relatively good number of approaches was evaluated on a high number of applications, making the scientific community and practitioners reasonably confident about their applicability in industrial contexts. Nevertheless, it is important to note that evaluating an approach on a low number of apps should not be seen as a strongly negative point because it may have been a necessity from an empirical perspective. For

example, the number of analyzed apps could depend on the execution time of the analysis tool; if the analysis tool requires a large amount of time for each app (*e.g.,* including user thinking time), then the input set of applications is inevitably small in order to keep the experiment duration acceptable from a pragmatic perspective.

TABLE 4.21: Number of analyzed applications

# Apps evaluated	**#Studies**	**Studies**
High (more than 1,000)	58	P1, P4, P6, P7, P8, P9, P12, P16, P17, P19, P20, P22, P24, P25, P30, P32, P33, P35, P36, P37, P38, P42, P44, P46, P50, P55, P64, P65, P66, P67, P73, P75, P76, P77, P78, P81, P85, P86, P88, P89, P91, P93, P94, P99, P101, P103, P104, P105, P106, P110, P112, P114, P122, P131, P132, P133, P134, P140
Medium (between 100 and 1,000)	32	P11, P14, P27, P29, P41, P43, P48, P52, P53, P54, P58, P59, P68, P70, P71, P72, P80, P83, P96, P98, P109, P111, P116, P119, P120, P121, P125, P126, P135, P137, P138, P139
Low (less than 100)	50	P2, P3, P5, P10, P13, P15, P18, P21, P23, P26, P28, P31, P34, P39, P40, P45, P47, P49, P51, P56, P57, P60, P61, P62, P63, P69, P74, P79, P82, P84, P87, P90, P92, P95, P97, P100, P102, P107, P108, P113, P115, P117, P118, P123, P124, P127, P128, P129, P130, P136

4.4.6 Apps provenance

From the analysis of the primary studies, it emerged that the majority of the studies, during the evaluation phase, use exclusively unmodified applications (see Table 4.22) mined from an app market (118/140). In other cases, the applications to be analysed were created for the purpose of the evaluation, or they were customized versions of real apps (34/140). In some cases (*e.g.,* P12, P14, P16, P17) a combination of real and custom applications is used; in those cases, custom apps support the evaluation of the proposed approach to exercise specific aspects of the proposed static analysis approach (*e.g.,* corner cases when building a control flow graph of the app under analysis), which are not fully covered by the mined original apps.

Overall, the obtained results are promising since approaches evaluated on (a potentially large number of) real apps in principle undergo a more realistic investigation with respect to those evaluated on synthetically-built apps. This realism comes also from the fact that apps mined from app stores are developed in real industrial contexts involving practitioners working under real business and organizational constraints (*e.g.,* release deadlines, specific development workflows). Moreover, apps mined from app stores can be totally different from synthetic apps because the former are distributed to and downloaded by real users; it is well known that users play a central role in the success (and indirectly in the development process) of the apps, *e.g.,* by providing publicly accessible app ratings and reviews [86], deciding to uninstall disappointing apps, etc.

TABLE 4.22: Apps provenance
(Categories not mutually exclusive)

Apps provenance	**#Studies**	**Studies**
Unmodified App	118	P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P12, P14, P16, P17, P18, P19, P20, P21, P22, P24, P25, P27, P28, P29, P30, P31, P32, P33, P34, P35, P36, P37, P38, P39, P41, P42, P43, P44, P46, P47, P48, P49, P50, P52, P53, P54, P55, P56, P58, P59, P62, P63, P64, P65, P66, P67, P68, P69, P70, P71, P72, P73, P74, P75, P76, P77, P78, P79, P81, P83, P85, P86, P87, P88, P89, P90, P91, P93, P94, P96, P97, P98, P99, P100, P101, P102, P103, P104, P105, P106, P107, P109, P110, P111, P112, P114, P115, P116, P117, P119, P120, P121, P122, P123, P125, P126, P129, P130, P131, P132, P133, P134, P135, P136, P137, P138, P139, P140
Customized App	34	P11, P12, P13, P14, P15, P16, P17, P23, P26, P38, P40, P45, P48, P51, P57, P60, P61, P80, P81, P82, P83, P84, P85, P86, P92, P95, P106, P108, P113, P116, P118, P124, P127, P128

4.4.7 Evaluation soundness

When proposing a scientific contribution is of paramount importance to rigorously evaluate it by following methodologies which are well-known and accepted by the community. According to this, we assessed the evaluation phase of each primary study by considering the checklist for empirical guidelines proposed by Wohlin *et al.* [12, § A.2]. The proposed checklist is composed of 23 items, each of them dealing with a specific aspect to consider when reading an experiment article. Examples of items include: *Is the sample used representative of the population? Are the hypotheses clearly formulated? Is the type of design clearly stated?*

TABLE 4.23: Evaluation soundness

Evaluation soundness	**#Studies**	**Studies**
High	49	P2, P3, P4, P5, P9, P13, P16, P17, P20, P21, P25, P27, P29, P30, P31, P32, P33, P34, P35, P36, P37, P43, P49, P63, P64, P66, P68, P69, P70, P82, P89, P91, P93, P94, P95, P97, P100, P101, P106, P109, P110, P112, P114, P117, P121, P127, P129, P133, P140
Medium	58	P1, P6, P8, P10, P11, P12, P14, P15, P23, P40, P42, P48, P50, P52, P53, P54, P56, P59, P60, P61, P62, P65, P67, P71, P72, P73, P74, P75, P76, P77, P78, P83, P84, P85, P86, P87, P90, P96, P98, P103, P104, P105, P107, P108, P111, P113, P115, P120, P122, P123, P125, P126, P128, P131, P132, P134, P137, P139
Low	33	P7, P18, P19, P22, P24, P26, P28, P38, P39, P41, P44, P45, P46, P47, P51, P55, P57, P58, P79, P80, P81, P88, P92, P99, P102, P116, P118, P119, P124, P130, P135, P136, P138

We ranked the evaluation soundness of a primary study as *high* if it covers more than 15 items of the empirical checklist, *medium* if it covers between 5 and fourteen items, and *low* if it covers less than 5 items of the checklist.

As shown in Table 4.23, a large number of the studies partially adhere to the checklist for empirical studies (*medium*, 58/140). To follow, 49 primary studies ranked as *high* in terms of evaluation soundness, whereas 33 primary studies have a *low* evaluation soundness. Overall, these results show that more than half of the considered primary studies perform a medium to high evaluation in terms of empirical evaluation, leading therefore to a reasonably higher level of confidence about the results of their evaluation.

4.4.8 Industry involvement

Each primary study was classified as (*i*) *Academia*, if the authors are affiliated exclusively to an academic organization, *e.g.,* university or research center; (*ii*) *Industry* if the authors are affiliated exclusively to an industrial organization, *e.g.,* a company, startup, or software house; (*iii*) *Academia and Industry* if some of the authors are affiliated to an academic organization and some others to an industrial one. As depicted in Figure 4.5, the vast majority of the authors of our primary studies is academic (115/140), followed by a combination of researchers and industrial practitioners (24/140), and finally 1 contribution involves industrial authors only. The emerged result is quite disappointing, as in almost all of the studies there is no involvement of industrial researchers or practitioners.

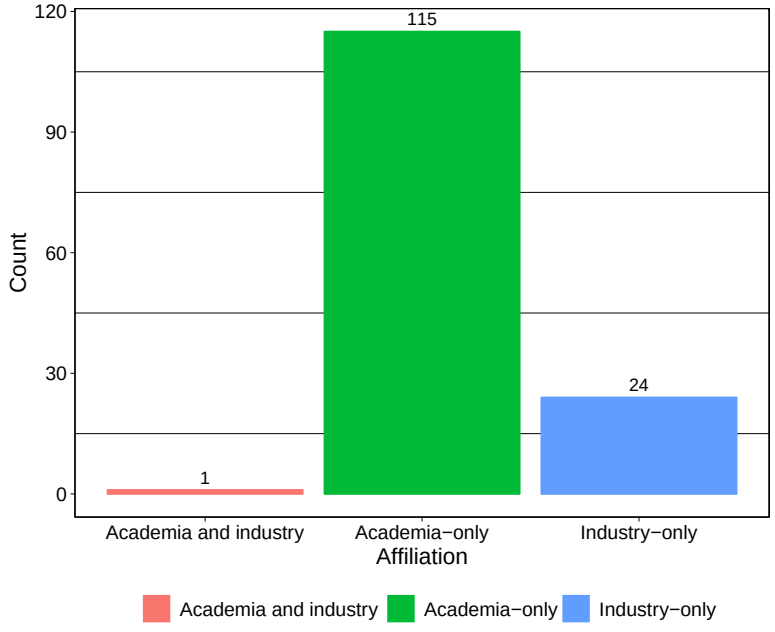


FIGURE 4.5: Distribution of industry involvement

If we consider the single industry-only primary study (*i.e.,* P96), an industrially-relevant problem related to mobile apps is about the fact that "*the danger of Android application collusion is often overlooked, as single-app exploitation is still a profitable means for adversarial activity; however, we* [the authors of P96] *believe that the perceived difficulty of using multiple applications is much higher than the actual cost*". Therefore, the authors present a collection of tools that provide static information flow analysis across sets of applications, showing a holistic view of all the applications running on a particular device. The techniques proposed in P96 include: (1) static binary single-app analysis, (2) security lint tool to mitigate the limits of static binary analysis, (3) multi-app information flow analysis, and, (4) evaluation engine to detect information flows that violate specified security policies. We believe that P96 is a good example of a research study tackling an industrially-relevant problem and proposing an industry-driven solution. Academic researchers could compare with or be inspired by the work in P96 for designing and evaluating the approaches for static analysis of mobile apps of the future.

Main findings on potential for industrial adoption:

- It comes without a surprise that app developers and platform vendors are the most targeted stakeholders. Still, a potentially unexplored venue is related to static analysis targeting the end users of mobile apps, who may have different requirements and needs with respect to the apps currently installed in their devices.
- In the vast majority of primary studies, researchers are not providing any tool implementing their proposed approaches. This result is strongly negative, as it impacts study replications and comparative evaluations, which are at the basis of the scientific method. We suggest researchers to always provide publicly available implementations of their approaches (when possible); this will help researchers and practitioners in improving the overall quality of research in static analysis of mobile apps.
- Execution times of current static analysis approaches for mobile apps are relatively low; this provides evidence that practitioners can adopt static analysis to improve the quality of their mobile apps, without having a strong impact on their development workflows (*e.g.,* by including the analysis in their continuous integration workflow).
- The evaluation of the proposed approaches is generally performed on unmodified apps (*i.e.,* experimentation in the wild). The number of apps considered in the evaluation phase is either high (more than 1,000) or low (less than 100).

The empirical soundness of the performed evaluations are generally between medium and high.

- As a community, we should encourage new connections between academia and industry in order to potentially improve the knowledge exchanged between them, where (*i*) research is performed on industrially relevant problems and (*ii*) new methods, technologies and tools are transferred from academia to industry.

4.5 Orthogonal findings

In the following, we report the results of our horizontal analysis. We recall that in this phase of the study we built contingency tables for pairs of parameters coming from our vertical analysis, we analyzed each of them, and identified perspectives of interest.

Analysis goal - Platform specificity. In general, *privacy* results to be the most recurrent analysis goal for all platforms, and specifically results to be vastly studied for the Android operating system. The only iOS approach found in the literature is also focusing on *privacy*. *Malware* results to be the second most studied subject in both Android and generic approaches. Overall, very little studies result to be platform-independent, and none appear for the categories *performance*, *inter-app communication*, and *antipatterns*.

We conjecture that the popularity of *privacy* and *malware* analysis goals can be associated to the ubiquity and handling of sensitive data that nowadays characterizes mobile apps. As a consequence, new methods and techniques to address the associated challenges is receiving a growing attention. In addition, many of the research focusing on *privacy* rely on a technique, namely, the inspection of the `AndroidManifest.xml`, that is quite simple to implement. This consideration further explains the high occurrences of such studies. Regarding the *performance*, *inter-app communication* and *antipatterns* goals, we hypothesize that such goals can be studied exclusively from a platform-specific point of view due to their tight relationship with the platform on which the app is running.

Analysis goal - Static\Hybrid approach. In general, all of the goal categories are mostly studied through hybrid approaches, indicating that approaches combining static and dynamic analyses are far more popular than pure static or dynamic ones. Overall, *privacy* results to be the most studied subject in both static and dynamic approaches

(40 static approaches and 11 hybrid ones). *Energy consumption* (11 static approaches and 7 hybrid ones) is the second most recurrent goal of hybrid analyses. *Frameworks* and *antipatterns* result to be supported exclusively by static analysis.

We believe that the rationale behind the popularity of hybrid approaches resides in the ability to circumvent weaknesses that arise when using only one kind of analysis, hence making it possible to gather more comprehensive, yet precise, results. As presented in the previous section, the popularity of the *privacy* goal can be justified by the interest of final users, developers and app store vendors to protect sensitive data from unauthorised access. The high number of hybrid approaches targeted at the *energy* goal evidences the reliance of such approaches on dynamic methodologies, utilised to exercise the applications under analysis, and gather empirical energy consumption measurements. On the other hand we conjecture that the lack of usage of dynamic analysis by approaches aimed at the *frameworks* and *antipatterns* goals is due to the nature of these goals, which are more tightly related to source code metrics rather than runtime ones, thus making static analysis techniques more suitable for them.

Analysis goal - App artifact. In general, the vast majority of the approaches require the *APK* package of the mobile application. This has to be attributed to the skewed data gathered for this research, from which most of the approaches result to focus on Android applications. In contrast, the most studied goal requiring source code as input is focusing on *inter-component communication*. The only goals that require more often source code than *APK*s are the ones focusing on *refactoring* and *performance*. Additionally, two goals that do not require access to the source code of the application were identified, namely *reflection* and *antipatterns*.

Regarding the goals for which analyses are mostly performed on source code, we believe that the underlying reason for this skewness is that these types of analysis require the exact source code of the app under analysis to be carried out properly. Even though Android decompilers and disassemblers do exist, at the time of writing, their precision is not high enough to perform these kind of analysis on packaged applications [88]. On the other hand, when focusing on the analysis goals requiring an *APK* as input, we can notice that for *testing*, *privacy* and *energy consumption* researchers have been focusing on black-box approaches, while neglecting white-box ones (at least partially). For these goals, approaches of the latter kind could be of assistance during development of mobile apps, either notifying developers when they unknowingly insert known antipatterns in their code (*e.g.,* an energy hotspot in the case of *energy consumption* or a privacy leak in the case of *privacy*) or in helping them in performing more efficient testing (in the case of *testing*).

Analysis technique - Analysis pre-steps. Eight out of 20 analysis techniques do not require pre-steps. Such analyses, such as *nullness*, *points-to* and *termination* analysis, are in fact carried out by inspecting the source code repository of the application, and hence do not require additional tooling or configuration. The remaining 12 analysis techniques require pre-steps of different nature. As expected, most of the analysis techniques needing pre-steps require the manipulation of source code, such as *code instrumentation* and *abstract interpretation* (for which two out of three papers required analysis pre-steps). In general, only three of the 20 identified analysis techniques resulted to require in the majority of the cases analysis pre-steps. This indicates that the vast majority of analysis techniques is executable "as is", *i.e.,* without requiring any additional process before the analysis can be actually carried out.

Target stakeholder - Analysis goal. Approaches targeting *app stores vendors* result to be mostly interested in *privacy* (27 studies) and *malware* (25), followed by *inter-component* and *inter-app communication* (7 studies each). Approaches targeting *developers* also result to be mostly interested in *privacy*-related analyses (21 studies), but also consider more low-level goals, such as *energy consumption* (18 studies), *inter-component communication* (13 studies), and *testing* (12 studies).

Approaches targeting *researchers* result to be mostly related to the improvement of the state of the art analysis techniques, hence often considering goals related to *inter-component communication* (4 studies), and *frameworks* (3 studies). As expected, approaches targeting *end users* result to be mostly interested in *privacy* (9 studies), and approaches targeting *app store vendors* are more interested in *malware* than *developers* (27 against 5 studies). In contrast, approaches targeting *developers* result to be more interested than those targeting app store vendors in analyses related to *testing* (12 against 1 study), *resources* (5 against 0), *refactoring* (5 against 0), *performance* (8 against 0), and *energy* (18 against 0). Again, this indicates that approaches targeting developers are more interested in the quality of the applications than those targeting app store vendors; the latter are mainly focused on ensuring the security of the *end user* by identifying potential *malware* and *privacy* leaks.

TRL - Analysis goal. For all goals there is a noticeable lack of advanced field-tested techniques ready for industrial adoption, with the majority of the studies positioned at the medium level of TRL. Studies that achieved a high TRL are only present for the goals *malware* (1 out of 31), and *inter-app communication* (1 out of 14). It is worth to note that, even if the majority of the studies focuses on *privacy*, none of the approaches addressing *privacy* were used in practice. When considering studies that achieved a middle-level technological readiness, the most can be found for the goals *privacy* (42 out of 51), *malware* (24 out of 30), *energy* (17 out of 18), and *inter-component communication*

(17 out of 20). *Framework* and *testing* are the goals that are lagging behind the most, with 2 out of 6 and 3 out of 12 studies positioned at *low*, respectively.

From these results, we can notice that static analysis techniques for mobile apps are not yet ready for industrial adoption, although popularly targeted by academia. We advocate that, to favor adoption in industry, researchers should not only further refine and improve existing techniques, but also rethink how evaluation is performed, aiming at performing it in a real industrial setting, when possible.

Evaluation soundness - Analysis goal. Adherence to evaluation guidelines varies among different goals. Studies targeting *malware* and *privacy* are the ones that follow them more strictly, with 14 and 16 studies having a *high* evaluation soundness. Studies that partially adhere to evaluation guidelines (*i.e., medium* evaluation soundness) are most commonly targeting the *refactoring* (4 out of 5), *privacy* (27 out of 51), *inter-component communication* (10 out of 20), *inter-app communication* (9 out 14), and *antipatterns* (1 out of 1) goals. Studies with *testing* (7/12) and *resource* (4/6) goals are those that at least follow the evaluation guidelines, with 7/12 and 4/6 studies having a *low* evaluation soundness, respectively.

We conjecture that the unbalance in evaluation soundness among goals is due to the fact that the empirical research approach is traditionally followed more strictly in some fields, *e.g.,* security; whereas, it is less frequent in others (*e.g.,* resource analysis). We advocate the adoption of empirical research techniques to researchers working in those fields in which evaluation soundness is lagging behind in order to foster the rigor, objectivity, and replicability of the proposed research results.

Execution time - Analysis technique. Almost all of the studied approaches require either a *low* or *medium* computation time in order to perform their analysis. *High* values for analysis time can be found only for *class analysis* (1), *classification* (1), *data mining* (1), and *flow analysis* (6).

Focusing only on the studies that require either a *medium* or *low* execution time, we can notice that *medium* execution times are more common across all analysis types. The only exceptions are *statistical analysis* (1 *low* and no *medium*), *symbolic execution* (1 *low* and 1 *medium*), *slicing* (3 *low* and 3 *medium*) and *abstract interpretation* (1 *low* and 1 *medium*). Nevertheless, this result is likely due to the low number of studies that make use of those kinds of analyses.

Execution time - Analysis goal. Almost all of the studies under evaluation report an execution time that is either *medium* or *low*, with only 7 studies that require a *high* execution time spread among different analysis goals without a noticeable trend. We can

therefore conclude that no analysis goal requires an amount of time of a greater order of magnitude with respect to the others in order to complete the analysis.

Focusing on the studies that require either a *low* or *medium* execution time, we can notice that those that fall into the *malware* and *energy* goals have an average execution time slightly skewed more towards *medium* since: the former has 12 studies that require *medium* and 5 *low*; the latter has 10 studies that require *medium* and 5 *low*. A similar trend can be noticed for *performance* (a single *low* and 4 *medium*), albeit with a smaller sample size. Hence, we can conjecture that analyses focusing on such subjects are generally more expensive w.r.t. execution time.

The opposite trend can be noticed only for the *inter-component communication* goal, which has 7 studies characterized by a *low* execution time and only 4 characterized by a *medium* one. We believe that this can be justified by the higher level of abstraction at which these analyses are usually carried out, as only component interfaces must be considered.

Usage of machine learning - Analysis goal. Usage of machine learning techniques is not evenly distributed among all goals. In particular, machine learning techniques are mostly employed for the goal of malware detection: out of 23 studies leveraging machine learning techniques in their analyses, 15 fall into the *malware* goal, the remainder is split among *privacy* (5), *inter-component communication* (2) and *inter-app communication* (1). This trend is traceable to the common techniques utilized to identify malware applications, which most often rely on training a classifier on a collected dataset of both benign and malicious applications. It is meaningful to notice that the same machine learning techniques can potentially be applied when targeting other goals, such as *performance* or *energy consumption*; surprisingly, none of the studies that fall into those goals make use of machine learning. We believe that this is due to the greater effort required for the collection of large datasets when considering these goals.

Industry involvement - Analysis goal. As expected, all analysis goals are considered by academic researchers. *Privacy* (38/140), *malware* (26/140), *energy* (18/140), and *inter-component communication* (16/140) are the most targeted goals for academic researchers. In some cases, when the analysis goal concerns *privacy* (12/140), *malware* (4/140), *inter-component communication* (4/140), *inter-app communication* (4/140), *framework* (2/140), *testing* (1/140), *resource* (1/140), and *refactoring* (1/140), academic researchers are supported by industrial professionals.

By analyzing these results, we can conjecture that, although industrial organizations are interested in addressing the problematics related to these goals, there is still a lack of

industrial involvement when targeting other research goals, such as *energy* and *performance*, that would improve the overall user experience of mobile apps, and thus also translate into benefits for industry practitioners. We argue that researchers should try more actively to involve industry practitioners when working on such goals.

Target stakeholder - Analysis technique. Approaches to be utilized by app stores vendors rely mostly on techniques such as *data mining* (17/140), *taint analysis* (15/140), and *classification* (14/140). This is in line with the most prominent goal of such stakeholder, *i.e.,* identifying malicious applications in order to remove them from their stores. On the contrary, approaches to be utilized by *developers*, that are more interested in the inner workings of the applications, result to be characterized by a higher usage of techniques based on *flow analysis* (51/140). An explanation for this trend is the difference in performance among different static analysis techniques: approaches targeted at *app stores* must be highly scalable, as they have to be executed daily on thousands of apps, while approaches targeted at *developers* have less stringent requirements. This evidences that improving the performance of some techniques is a relevant open problem, as they are currently a limiting factor for the kind of analyses that can be performed on *app stores*.

Tool availability - Analysis goal. When dealing with static analysis, automation is a crucial requirement for an approach to be effectively adopted in practice. Although for the majority of the identified analysis goals many different approaches have been proposed, most of them do not have a (released) tool ready for adoption by practitioners. On the one hand we can argue that addressing goals such as *privacy* and *malware*, may require the realization of a mature supporting tool requiring a development effort that cannot be always afforded. On the other hand, addressing some goals represent more a theoretical interest, with potentially marginal practical impact, such as the study of an analysis *framework* itself. Nonetheless, we encourage researchers to undergo the extra effort required for making their analysis tool available to the research community: not only it makes easier to replicate their results but also analysis types for which a mature tool has been made available have been far more explored by the scientific community (as in the case of *Flowdroid* [79] for *flow analysis*).

4.6 Discussion and future challenges

The results presented in the previous sections give a data-driven, objective overview of the current state of the art on static analysis for mobile apps. In this section, we provide our own interpretation of the main points we deem as important challenges for future researchers in this area.

Is there life after Android? When considering the targeted platforms, it is evident that Android is the clear winner, with more than 85% of approaches targeting it. If on the one hand, we could have expected this result (as of today, Android is the most popular mobile operating system with more than 90% market share [3] and a relatively large number of open-source tools for apps analysis), on the other hand, it makes us wonder what will be the fate of this Android-specific large body of knowledge and tools we researchers are producing in the future. If we look back in time, it is widely recognized that the mobile ecosystem is extremely dynamic, with platforms unpredictably raising and failing in terms of sales of devices, company acquisitions, users flowing to/from other platforms, etc. For example, 8 years ago, Apple iOS and Symbian were having 38% and 16% of the market share, whereas today they account for less than 14% together[13].

It is encouraging to see that 20 approaches out of 140 are generic (even though the implementation of the majority of them is again Android-specific). We believe that in the future researchers should reason at a higher level of abstraction, and focus more on approaches which are technology-independent, generic, and applicable to different platforms with reasonable effort. It is only in this way that our research results will pass the test of time and will (hopefully) remain relevant also in the future, despite the inevitable technological waves we will be facing. It is important to note that we are not suggesting to totally neglect platform-specific aspects, rather we are proposing to design our own research products to be platform-independent and robust with respect to (future) technologies; among many, researchers might take advantage of the well-known principles of extensibility and separation of concerns, of layered or plugin-based architectures for making their research products applicable in the context of new technologies without disrupting their general principles and base mechanisms. This will also speed up research by helping researchers in avoiding to reinvent the wheel whenever a (potentially applicable) research product will be applied to a new mobile platform.

Analysis goals shall be expanded substantially. The results of our study tell that privacy and malware are the most targeted analysis goals, far more than the others (*e.g.,* performance, energy, resources usage). This is a clear gap that we, as researchers in the area of mobile apps analysis, should be filling in the future.

Given its strong importance for mobile apps, it seems that *performance* is extremely under-explored. Indeed, performance is a fundamental aspect of mobile apps development, which are often providing computationally-intensive features and being used for mission-critical tasks. Last but not least, a pleasant user experience is often a key factor to determine the success of an app [89]. Moreover, *anti-patterns identification and*

[13] https://www.statista.com/statistics/263453/global-market-share-held-by-smartphone-operating-systems/

refactoring are among the least explored analysis goals so far, despite the fact that bug fixing and code re-organization are among the most recurrent activities of mobile apps developers [90]. In this context, P52 can be considered as a reference study about how to propose, design, and evaluate a refactoring method for mobile apps. Specifically, P52 presents a preliminary large-scale formative study about how developers approach asynchronous programming in Android apps. Then, based on the obtained results (*e.g.,* that developers are using the Android AsyncTask construct also for long running operations, potentially leading to memory leaks, lost results, and wasted energy), a tool-based method is proposed for (*i*) statically identifying usages of the AsyncTask construct which can be automatically improved, and (*ii*) refactoring those parts of the app via a safe code rewriting algorithm. Finally, an empirical evaluation provides objective and reproducible evidence about the applicability and saved effort of the proposed method.

Users are being left out of the equation. From the results of RQ3 it emerged that only 12 studies consider end users as stakeholders, revealing that researchers are mostly focusing on techniques aimed at assisting developers, store moderators and researchers instead. Although this unbalance is not unexpected, when also considering that the majority of studies focused on privacy as their goal, we can notice a lack of users-first privacy approaches. Indeed, privacy is a subjective property, as different users may have different requirements to consider an application trustable, current solutions fail to address this subjective aspect of privacy, considering all users as equals. Hence, in light of these considerations, we can identify one research area currently open and overlooked: the design of more user-centric approaches to privacy, where users are provided with the necessary tools to specify and validate the "personal" requirements to which an application must comply [91].

Developers are being left out of the equation too! Even though when answering RQ3 it emerged that practitioners were involved in 25 studies, it also emerged that almost all approaches have not been evaluated or adopted in an industrial environment (*i.e.,* high TRL). We consider this finding as an indication that practitioners are involved in the technical phases of the study (*e.g.,* elicitation of the requirements for the approaches, analysis steps definition, experiments results evaluation), but not as subjects of the evaluation of the proposed approaches. This situation is in strong contrast with the fact that the most recurrent stakeholders of the proposed approaches are the practitioners themselves. For the future, we strongly advise to close the loop by including practitioners in all the phases of the studies, specially while (*i*) defining the assumptions, requirements, and usage scenarios of the proposed static analysis approaches, as well as (*ii*) evaluating the proposed approaches in terms of their usefulness, applicability, and usability. At best, the latter can be performed by applying the case study methodology [12]. This is

already happening in other research areas within the software engineering domain, such as software energy efficiency [92], technical debt [93] and software testing [94].

Tools and datasets shall be released and publicly available. An underlying problem which hinders static analysis of mobile apps research lies in tool availability. In fact, from the results of our research, we evince that only a small portion of tools utilized or developed in the primary studies are available online. This constitutes a serious problem for researchers interested in extending or adapting tools which have been already developed. Additionally, the data utilized in the primary studies (*e.g.,* accurate versioning history of apps used for experimentation) is only seldom available. This potentially slows down investigations, as datasets still have to be created on an *ad-hoc* basis for research, as the number of already available ones is scarce. In recent times, this trend has been opposed by the constitution of some conference tracks explicitly aimed to make datasets publicly available. Among the most prominent ones are the "Artifact" track of the International Conference on Software Maintenance and Evolution (ICSME), and the "Data Showcase" track of the Mining Software Repositories (MSR) conference. Research belonging to these tracks range from general purpose datasets, *e.g.,* large versioning datasets focusing on Android applications [95], to context-specific datasets, *e.g.,* to support dynamic analyses of Android applications [96]. Finally, from the findings of our study, we detect a shortcoming shared by many studies of static analysis of mobile apps, namely the impossibility to replicate the reported results. In fact, the absence of structured replication packages, in form of tools and dataset utilized, precludes the possibility to replicate the results reported in the primary studies. This constitutes a major problem affecting not only researchers interested in the field of mobile static analysis, but also the soundness of the studies itself.

4.7 Threats to validity

In order to ensure the high quality of the data gathered for this study a well-defined research protocol was established before carrying out the data collection. The research activities were designed by following a set of well-accepted and revised guidelines for systematic mapping studies [77]. From the formalization of such guidelines we established the research protocol that was strictly followed throughout the study, as documented in Section 4.1. In addition, in order to further ensure the adherence to the protocol and the envisioned quality standards, all the steps of the research (*e.g.,* study design, search and selection, data extraction, data analysis, etc.) were carried out as a team. This activity was deemed necessary also to lower potential sources of bias by discussing crucial considerations in the team. Even by adopting a methodic literature review approach,

threats to validity are still unavoidable. The following reports on the main threats to validity to our study and how we mitigated them.

External validity refers to conditions that hinder the ability to generalize the results of our research [12]. The major threat of this category is represented by the fact that our primary studies are not representative of the state of the art research on static analysis of mobile applications. In order to mitigate this threat, we adopted a search strategy consisting of a manual search encompassing all the top-level software engineering conferences[14] and international journals[15] according to well known sources in the field. Such process was further extended by executing a backward and forward snowballing process on the selected literature. In order to ensure the quality of the selected research, we exclusively considered peer-reviewed papers and excluded the so-called grey literature, such as white papers, editorials, etc. We disregard such decision as a significant source of bias, as peer-review processes are a standard requirement for high quality publications. Finally, we adopted a set of well-defined inclusion and exclusion criteria, which rigorously guided our selection of the literature.

Internal Validity refers to the influences that can affect the design of the study, without the researcher's knowledge [12]. In this regard, we defined *a priori* a rigorous research protocol for the study. The classification framework adopted was established iteratively by strictly following the keywording process. Regarding the synthesis of the collected data, such process was carried out by adopting simple and well-assessed descriptive statistics. Subsequently, during the orthogonal analysis, we performed sanity tests on the extracted data by cross-analyzing different parameters of the established classification framework.

Construct validity refers to the extent to which the primary studies selected are suited to answer our research questions [12]. In order to mitigate such threat, we manually inspected thoroughly the literature published in the top-level software engineering conferences and journals. This procedure was performed by adhering to a rigorous predefined protocol. In addition, the results of such process were expanded by integrating the results gathered through a backward and forward snowballing process. Subsequently, we methodologically selected the identified studies by applying a set of well-documented inclusion and exclusion criteria. This latter process was carried out by three researchers independently. As recommended by Wholin *et al.* [12] a random sample of eight studies were selected and analyzed by all three researchers in order to ensure that the analyses were aligned.

[14] `http://goo.gl/auU7su`
[15] `http://www.webofknowledge.com`

Conclusion validity refers to issues that might hinder the ability to draw the correct conclusion from the data gathered [12]. In order to minimize the presence of such threat, we carefully carried out the data extraction and analysis by strictly adhering to an *a priori* defined protocol. Such protocol was specifically conceived to collect the data necessary to answer our research questions. This enabled us to reduce potential sources of bias resulting from the data extraction and analyses processes. In addition, such methodology guaranteed us that the extracted data was fitted to answer our research questions. In order to further mitigate potential threats to conclusion validity, we adhered to the best practices reported in several well known guidelines for systematic literature reviews [12, 13, 71]. Such guidelines were strictly followed throughout each phase of our research, and were comprehensively documented in order to make our research approach transparent and replicable.

4.8 Conclusions

In this chapter we reported on the design, execution and results of a systematic mapping study aimed at answering the first of the research questions addressed by this dissertation (discussed in Chapter 1.3). That is:

RQ1 - *What is the state on the art on static analysis for mobile applications?*

The systematic mapping study reported in this study permitted us to precisely characterize the most relevant methods and techniques for statically analyzing mobile apps. Starting from over 8,000 potentially relevant studies, we applied a rigorous selection procedure resulting in 140 primary studies along 71 scientific venues and a time span of 7 years.

We rigorously defined a classification framework with the target of identifying, evaluating and classifying the characteristics of existing approaches to the static analysis of mobile apps, while understanding trends and potential of industrial adoption.

The main findings of this study have been synthesized by performing (*i*) a combination of content analysis and narrative synthesis (vertical analysis), and (*ii*) a correspondence analysis via contingency tables (horizontal analysis). Of particular interest, among other results, is the fact that only a small minority of studies consider end users as stakeholders despite the fact that the majority of studies focused on privacy as their goal. Hence, **we can notice a lack of users-first privacy approaches**. Indeed, as previously discussed in Chapter 1, privacy is a subjective property, as different users may have different requirements to consider an application trustable, with current solutions failing to address this subjective aspect, considering all users as equals.

Our study will help researchers and practitioners in identifying the purpose and the limitations of existing research on static analysis of mobile apps. Also, we assessed the potential of research on static analysis of mobile apps, discussing how to foster industrial adoption and technological transfer. The knowledge of the potential of existing methods and techniques constitutes a reference framework in support of researchers and practitioners, such as app developers, who are interested in selecting/choosing existing static analysis approaches, and want to critically understand what they offer and how. In this sense, we can argue that this work constitutes a valuable asset to the academic and industrial world in the wide spectrum of static analysis.

Chapter 5

An investigation into Android permissions from an end users' perspective

As mentioned in Chapter 3, researchers have studied usability and effectiveness of the Android permission system, evidencing the existence of several issues with it: only a minority of users are aware of the implications of their privacy decisions and warning dialogs are not easily understood [35, 36, 97]. Towards addressing these problems, the permission system has been revamped and, starting with Android 6, access to privacy- and security-relevant parts of the platform is enforced by a new run-time permission system.

We investigated *how end users perceive the new run-time permission system of Android*, with the ultimate goal of identifying possible points of improvement, despite the recent changes. For this purpose, we conducted a large-scale empirical study, collecting and inspecting over 4.3 million user reviews about 5,572 apps published in the Google Play Store that adopt the run-time permission system. Using a combination of an established keyword-based approach [98] and machine learning techniques, we identified among them permission-related reviews regarding the new Android permission system and categorized the main concerns expressed by end users into a taxonomy.

5.1 Study design

This section describes the study design and how the different steps were executed. In order to perform an objective and replicable study, we followed the guidelines on empirical software engineering in [12] and [99].

5.1.1 Goal and research questions

Goal of the study is to evaluate the Android run-time permission system for the purpose of characterizing the way end users perceive its issues and benefits in the **context** of 15,124 free Android apps published in the Google Play Store. We refined this goal into the following research questions:

RQ1 - How *accurate* is an automated approach in classifying user reviews via different combinations of machine learning techniques?

RQ2 - To what *extent* do app reviews express concerns about the Android run-time permission system?

RQ3 - What are the *main concerns* about the Android Run-time permissions system in app reviews?

RQ1 is a meta research question. By answering it, we aim at objectively assessing the accuracy of different combinations of machine learning techniques, *e.g.,* Naive Bayes classifier and support vector machines. Indeed, since the proposed software pipeline can include different components, it is expected that different combinations will result in different levels of accuracy. Obviously, we use the results of the most accurate software configuration when answering RQ2 and RQ3.

RQ2 aims at assessing how end users consider issues and benefits related to the Android run-time permission system, and if they vary across app categories.

The rationale behind RQ3 is to identify the main concerns of end users about the Android run-time permission system, what issues are still unresolved, but also positive reactions about it (*e.g.,* praises).

5.1.2 Subject selection

Hereafter, we describe how we built the dataset used as a basis for our study. The collection process is summarized in Figure 5.1.

Apps selection – As a starting point for our selection, we considered the top 500 most popular free apps from each of the 35 categories of the Google Play Store, as ranked by the App-Annie service for app ranking analysis[1] as of October 11, 2016. The total amount of entries we extracted is 15,517. At the time, some new categories (such as

[1] www.appannie.com/apps/google-play/top-chart/united-states

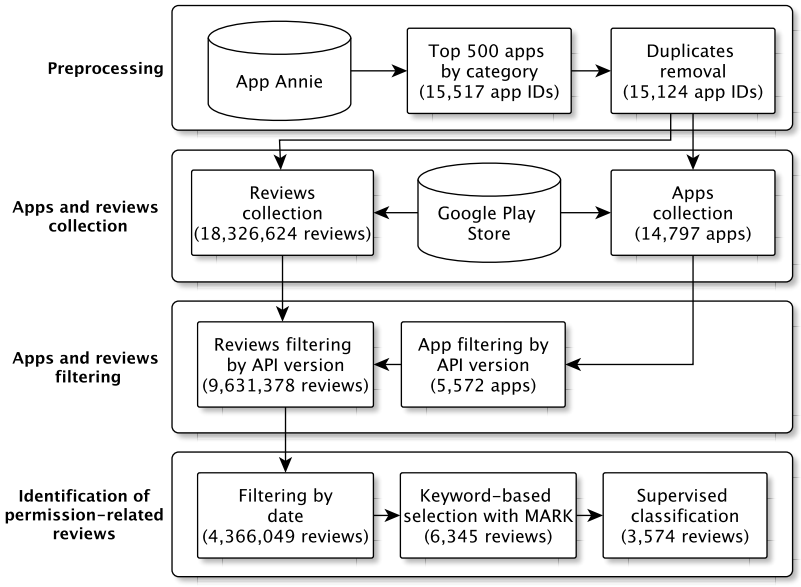


FIGURE 5.1: Summary of the data collection process

Dating and *Parenting*) had recently been introduced in the store, and they contained less than 500 entries. We removed those duplicate apps that appeared in more than one category, achieving a set of 15,124 unique app IDs. Selection was limited to free apps, as binaries are necessary to identify apps that adopt the new permission system. In fact, free apps represent 93.99% of all Google Play Store apps and they are downloaded more often [100].

Apps and reviews collection – We downloaded from a third party service[2] the binary files (*i.e.,* the APKs) of the apps identified in the previous step. Some of the apps did not exist anymore in the Google Play store, and have therefore been excluded. This can happen if the developers decide to remove the app from the store or if Google decides to remove the app for violation of some publishing policies. This led to the discarding of 327 apps, resulting in a total of 14,797 apps. Using an open-source web scraper[3], we collected all the user reviews published in the Google Play store for all surviving apps. For each review, we collected the full *review text*, the *publication date*, and the *review rating*, *i.e.,* a grade, on a scale from one to five, assigned by the reviewer to the app. As the Google Play store exhibits only a limited amount of reviews for each app at a given time [101], we repeated the collection process multiple times to collect a more extensive dataset. Afterwards, we merged the results of each collection iteration, discarding duplicates, leading to an acquired total of 18,326,624 individual reviews. The whole process lasted 8 weeks. The most recent review in our dataset is dated 28 February 2017.

[2]https://apkpure.com
[3]https://github.com/facundoolano/google-play-scraper

Apps and reviews filtering – We developed a tool for automatically disassembling the APK of an Android app, analyzing its manifest file, and identifying the Android API Level targeted by the application (*i.e.,* the value of the `androidtargetSdkVersion` attribute of the `uses-sdk` tag[4]). With such a tool, we identified the apps using an Android API version equal or greater than 23, which is the Android version in which the run-time permission system was first introduced. A total of 5,572 apps fell into this last category. Since we are only interested in reviews that discuss the new run-time permission system, we excluded all reviews belonging to apps that still use an API version earlier than 23. This filtering step resulted in a set of 9,631,378 reviews. From this set, we further filtered out all the reviews predating 5 October 2015, the release date of Android API 23, hence achieving a total of 4,366,049 reviews.

Keyword-based selection – In order to identify potential permission-related reviews, we performed an additional keyword-based selection, following the intuition that users often use semantically similar or related keywords to describe an issue or a feature, as suggested in [98]. To select our keywords we relied on the MARK tool by Vu *et al.* [98, 102]. We chose MARK over other tools (*e.g.,* CALAPPA [103] or AR-Miner [104]) for its ability to grasp semantic similarity among words and provide suggestions of keywords similar to the ones given as input. We provided the keyword *permission* as input and, by following the tool suggestions, we added to the set of keywords the words *privacy* and *consent*. We choose not to expand our keywords set further and limit ourselves to a concise set of neutral, pertinent keywords. Indeed, expanding it further would allow us to identify a larger set of potentially permission-related reviews. However, it would come at the expense of precision, while potentially introducing biases into our subsequent analysis stemming from positive/negative connotations associated with some words or expressions. The result of this filtering step is a set of 6,345 reviews. The high discard-rate in this step is not surprising, rather it is in accordance with existing research confirming that only a (relatively) small fraction of app reviews mention application permissions [105].

Supervised classification – The final step of our data collection process is a supervised classification procedure. First, we manually classified a sample of 1,000 randomly extracted potential permission-related reviews to build a ground truth for a subsequent automatic classification procedure. This resulted in a set of 780 classified permission-related reviews (others were discarded as, despite containing selected keywords, clearly do not deal with Android permissions, *e.g.,* "*Simple enough for beginners. Includes features like backup, paper wallet and privacy settings.*"). Complete details about the manual classification procedure are provided in Section 5.2.1. Secondly, we extended the classification to the remaining 5,345 potential permission-related reviews leveraging an

[4] `http://developer.android.com/guide/topics/manifest/uses-sdk-element.html`

automatic classification pipeline built on top of established machine learning and natural language process techniques, described in Section 5.2.2. The result of this step is a set of 3,574 permission related reviews, which form the objects of our study.

5.1.3 Variables

The main focus of RQ1 is on the configuration of the classification pipeline. The independent variables are three, each of them mapping to a specific phase of the pipeline: preprocessing, features extraction, and classification. The specific levels for these variables are described in detail in Section 5.2. For what concerns the dependent variables, we measure the *accuracy* of our classification pipeline by relying on two key metrics commonly used in the field of automatic classification: *precision* and *recall* [106]. $Precision_c$ is the fraction of reviews that are classified correctly to belong to class c. $Recall_c$ is the fraction of reviews of class c which are classified correctly. We used the following formulas to calculate them:

$$Precision_c = \frac{TP_c}{(TP_c + FP_c)} \qquad Recall_c = \frac{TP_c}{(TP_c + FN_c)} \tag{5.1}$$

TP_c stands for *True Positives*, the number of reviews classified as of class c that actually belong to class c. FP_c stands for *False Positives*, the number of reviews *not* belonging to class c mistakenly classified as belonging to class c. FN_c stands for *False Negatives*, the number of reviews mistakenly classified as *not* belonging to class c even though they actually belong to class c. Considering precision and recall, we calculated one further metric, the F_1-$Score$, defined as the harmonic mean of the two, and provides a single measure of *accuracy* [106].

When addressing RQ2, for each app a in our dataset, we considered the following variables: the number of its permission-related reviews (R_a^p), the total number of reviews it received in the Google Play Store (R_a), and its category in the Google Play Store (cat_a).

Concerning RQ3, we considered the different categories of reviews concerning the runtime permission system, as extracted by our classification pipeline (see Section 5.2).

5.1.4 Execution

Concerning RQ1, we implemented our classification pipeline in Python, by relying on the Scikit-learn [107] and Nltk [108] Python modules version 0.19.01 and 3.2.2 respectively, with default parameters. We executed all the meaningful (possible) configurations for

our classification pipeline, investigating the impact that each single component of the pipeline had on its overall accuracy. Then, considering the limited availability of training data, we decided to evaluate each configuration by adopting the *k-fold cross-validation method* [109], with $k = 10$. According to this method, we split the available training data into k equal sized partitions. A single partition was retained as the validation data for testing; the remaining $k - 1$ partitions were used as training data. The training-validation process was repeated k times, with each of the k partitions being used exactly once, as per the validation data. At each iteration, we computed both the precision metric and the recall metric. In order to further account for the possible variability of the results that might occur due to the selection of the random partitions, we *repeated the whole process 100 times for each configuration of our classification pipeline.* Then, at the end of all iterations, we calculated the average to produce a single estimation. The whole experiment was performed on a Ubuntu Linux virtual machine equipped with 8GB RAM and an Intel Xeon CPU E5630 Processor.

We extracted the values of the variables for answering RQ2 and RQ3 by relying on the data we mined when building the dataset and on the semi-automated process for identifying permission-related reviews. There, R_a^p is the count of permission-related reviews for each specific app category (as defined in the Google Play Store). We approached automatic classification of user reviews with a dual goal in mind: (i) we wanted to automate a process that would be infeasible to perform manually; (ii) we still wanted to retain a satisfactory level of accuracy. R_a and cat_a are extracted when building the dataset (see previous section).

5.1.5 Study replicability

In order to allow for complete replicability of the study, the source code of the classification pipeline, the source code developed for the experiments, and the raw data with the results of both manual and automatic processes are publicly available in the on-line replication package[5].

5.2 The classification pipeline

Given the number of items to classify (*i.e.,* 6,435 potential permission-related user reviews), it would be infeasible to perform a complete manual analysis. Therefore, we resorted to a two-step semi-automated classification pipeline. First, we manually analyzed a sample of the data and built a taxonomy of end users' comments, grouping together

[5]http://cs.gssi.it/MobileSoft2018ReplicationPackage

permission-related reviews into relevant groups with descriptive labels (see Section 5.2.1). Then, we trained a machine-learning classifier with the manually-built sample, and we used the trained classifier on all remaining reviews (see Section 5.2.2).

5.2.1 Manual analysis

To perform the initial manual analysis, we randomly selected a sample of 1,000 reviews from the permission-related set of our dataset. Such a sample allowed us to achieve a confidence level higher than 95% and a 5% confidence interval. We further divided the sample into two equal parts and assigned each to a different researcher. To reduce bias, each researcher independently surveyed the reviews in his sample, grouping together reviews containing similar concerns, and determined an informative label for each group found. After completing the analysis, the two researchers discussed together the identified groups, aligned the labels with each other, and solved all the cases in which there was a disagreement. During this revision step, a total of 48 reviews were reclassified: one category identified by one researcher was divided into more finer-grade ones detected by the other; another category was discarded (it grouped user complaints about apps performing actions without their knowledge but not related to the Android permission system, e.g., *"Sends requests to Facebook friends without permission."*). The resulting categories are discussed in Section 5.3.3, whereas, the manually-classified reviews are available in the replication package of this study.

The manual analysis of the 1,000 reviews led to the definition of a taxonomy composed of 10 categories of recurring user concerns. In addition, we identified two macro-categories: positive opinions (in the following marked with a + sign), and negative ones (marked with a − sign). In the following we describe each category.

\+ **Permission Praise (PP)** – The reviewer expresses some praise for the app handling of permissions, but it does not delve into details. A clarifying example is the following: "*Cool game, clean code and permission friendly app.*"

\+ **Minimal Permissions (MP)** – The reviewer feels that the app asks only for permissions strictly needed in order to carry on with its advertised functionalities. An example of this kind of review is: "*I've tried others, this is the simplest, easiest to use. Does not ask for permissions!!*"

\− **Permissions Complaint (PC)** – The reviewer expresses some complaints about the app handling of permissions, but does not provide details about it. An example: "*Used to love it. Don't like the new permissions.*"

- **Too Many Permissions (TMP)** – The reviewer complains about the excessive amount of permissions requested by the app. An illustrative case: "*This app demands access to contacts, the camera, and SMS. Too many permissions for a glorified chat app. A real shame.*"
- **Unclear Permissions (UP)** – The app does not explain why some of the requested permissions are needed or the provided explanation is not convincing. An example is the following: "*Why on earth does it need that permission?*"
- **Permission-related Bug (PB)** – The app contains some bug related to permission requests that prevents its correct functioning. An instance: "*Whenever I tried to measure my bp, it gave error that doesn't have permission to access camera but actually it has access to camera.*"
- **Repeated Permission Requests (RPR)** – When some permissions are denied, the app repeatedly keeps requesting them, rendering usage of the app itself impossible or burdensome. An example of this kind of review is: "*App sounds like a great idea. However you cannot go without swiping one app with location denied before it asks for a permission again. Im not searching for jobs in my area I don't want the location on. [...]"*
- **Settings Permissions (SP)** – Despite adopting an Android API version greater or equal to 23, the app does not perform run-time permission requests properly. Thus, in order to properly use the app functionalities, the review author was forced to manually grant permissions to it through the device system settings. One example is the following: "*Nexus 5, had to manually give the app permissions to access GPS. Seems to be working now.*"
- **Bad Request Timing (BRT)** – According to the review author, run-time permission requests are performed at a wrong time during app execution. One example of such reviews is the following: "*Strange that you ask for all the permissions up front now, instead of the Marshmallow approach of asking for permissions when people actually invoke the corresponding functionality [...]*"
- **Functionality Unavailable (FU)** – Without granting one or more permissions, usage of some key app functionality is impossible. A clarifying example is the following: "*The app requests permissions per API 23, but if you refuse to grant them, the app shuts down and refuses to run [...]*"

Notice that a review can potentially be assigned to more than one category, as not-all categories are mutually exclusive (*e.g., Settings permissions* and *Permission-related bug* are non-exclusive, as the user might be forced to assign permissions from the device

settings to circumvent a bug). A breakdown of the amount of reviews classified in each category is provided in Table 5.6.

5.2.2 Automatic classification

The structure of our classification pipeline is outlined in Figure 5.2. A detailed description follows.

The main input of our classification pipeline is composed of the raw text of end users' reviews. Also, since the quality of the results may potentially be improved by feeding the classification pipeline with additional data, we included also user ratings as part of the input.

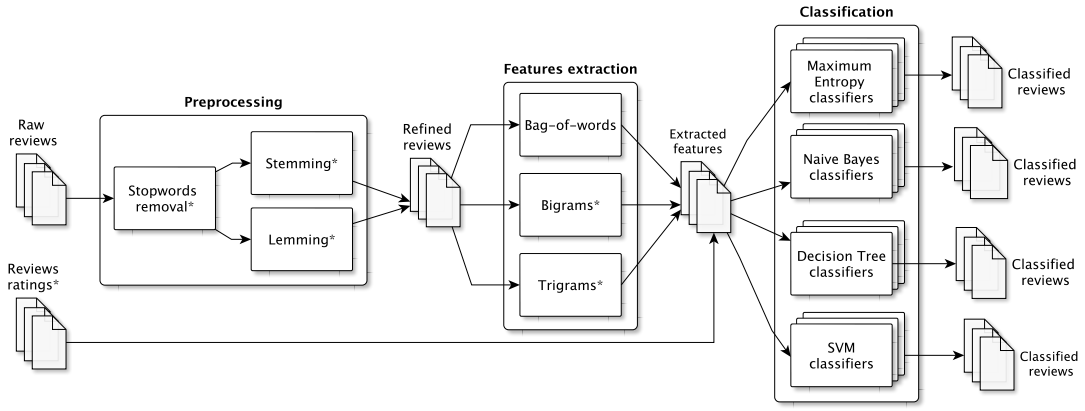


FIGURE 5.2: Overview of the classification pipeline (steps marked with an * are optional)

In the following, we illustrate the design of our classification pipeline and the main components that constitute it.

Preprocessing – The quality of the results can be affected by the preprocessing steps performed on the inputs. Therefore, we experimented with different preprocessing steps commonly performed in the field of Natural Language Processing, whose purpose is to refine the data fed to the subsequent classification step. We experimented with the following techniques:

- *Removal of stopwords* – words commonly used in the English language, such as "as", "can", "it", "so", which do not greatly affect the semantics of a sentence. Removing stopwords from reviews potentially removes noise from input data, thus allowing classifiers to focus on more relevant words. We experimented with the default list used by Scikit-learn[6].

[6] `https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/feature_extraction/stop_words.py`

- *Stemming* – the process of reducing inflected or derived words to their root form. For instance, the words "connections", "connective" and "connected" are all reduced to the same base word "connect".
- *Lemmatization* – a process that reduces a word to its canonical form named *lemma*. Unlike stemming, lemmatization is performed with the aid of dictionaries and takes the linguistic context of the term into consideration. For instance, lemmatization correctly identifies "good" as the lemma of the word "better".

Reviews representation – We selected the popular *bag-of-words* model [110] to represent reviews in our pipeline. Bag-of-words is a simplifying representation in which a document is represented as the multiset of its words, disregarding grammar and word order, but keeping multiplicity. From this representation classification algorithms can learn the review type based on the terms existence and frequency. However, common words like "the", "a", "to" are almost always the terms with highest frequency in documents. To address this problem we employed *tf-idf* (term frequency-inverse document frequency) normalization [110], that weighs with diminishing importance terms that occur in the majority of documents.

One of the known disadvantages of bag-of-words representation is that spatial information about words sequences is not preserved. Hence, we also experimented with *n-grams* [110], another commonly used document representation model. An n-gram is a sequence of n contiguous words extracted from document text. We consider n-grams of length 2 and 3, namely *bi-grams* and *tri-grams* respectively.

Classification – Since, potentially, a review can contain multiple users' comments, it can correctly be classified as belonging to multiple categories. Therefore, our case falls into the problem of *multi-label classification*. This poses the question of whether to train a single *multi-label classifier* or multiple *binary classifiers*, one for each category. A binary classifier is a classifier trained for the task of deciding whether an input instance belongs to a given class or not. A multi-label classifier instead is trained to assign a given instance to one of multiple classes, greater than two, with no restriction on how many of the classes the instance can be assigned to. We decided to rely on the former for two main reasons: (i) previous research provided evidence that multiple binary classifiers *perform better* than multi-label ones for user reviews [111]; (ii) training multiple binary classifiers gives us more *flexibility*, potentially allowing us to choose a different classifier for each taxonomy category.

We included four commonly used binary classification techniques known to perform well on textual inputs in our experimentation [112]:

- *Naive Bayes* [113], a popular algorithm for binary classifiers based on applying Bayes' theorem with strong independence assumptions between features. It is computationally efficient and achieves good predictive performance in many real-world applications.

- *Decision tree learning* [114], which iteratively constructs a decision tree to be used as a classifier. A decision tree is a tree-shaped graph in which each non-leaf node denotes a test on a specific feature, each branch represents the outcome of a test, and each leaf holds a class label. Decision trees are simple to interpret and mirror human decision making more closely than other approaches. For our experimentation we adopted a *CART* tree [115].

- *Maximum entropy* [116] (also known as MaxEnt or multinomial logistic regression), a probabilistic classifier based on the principle of maximum entropy. It does not assume conditional independence of features.

- *Support vector machines* (SVM) [117], a non-probabilistic binary classifier. SVMs plot each data item as a point in n-dimensional space and finds the hyper-plane that maximizes the gap among the two different classes.

5.3 Results

5.3.1 RQ1 - How accurate is an automated approach in classifying user reviews via different combinations of machine learning techniques?

As discussed in Section 5.1.4, we executed a large number of runs of our classification pipeline, exploring different combinations of review representations, preprocessing steps, and classification algorithms. Complete results of all the performed runs are available in the replication package, together with a complete implementation of our classification pipeline.

Baseline: No preprocessing – bag-of-words representation. Table 5.1 summarizes the results of the first combination we explored. We provided as inputs to the classification algorithms the raw text of the reviews in the bag-of-words representation and without any preprocessing. This is the basic configuration that allowed us to assess the baseline performance of the different algorithms for our problem.

Decision Tree performed the worst, achieving an F_1-Score of 0.635, well below the 0.75 threshold reached by the other algorithms. Still, despite achieving a similar F_1-Score,

	PP	**PC**	**MP**	**TMP**	**UP**	**PB**	**FU**	**RPR**	**SP**	**BRT**	**Avg**
Precision											
Naive Bayes	0.786	0.664	0.790	0.671	0.676	0.753	0.608	0.635	0.640	0.615	0.684
Decision Tree	0.693	0.612	0.716	0.641	0.731	0.679	0.597	0.620	0.587	0.529	0.640
Max Entropy	**0.788**	**0.686**	**0.833**	**0.733**	**0.811**	0.788	0.675	0.696	0.700	0.638	**0.735**
SVM	0.766	0.681	0.821	0.712	0.791	**0.792**	**0.680**	**0.702**	**0.747**	**0.653**	**0.735**
Recall											
Naive Bayes	0.732	**0.777**	**0.897**	**0.873**	**0.903**	**0.915**	**0.895**	**0.887**	**0.910**	**0.799**	**0.859**
Decision Tree	0.714	0.595	0.735	0.649	0.743	0.670	0.582	0.577	0.554	0.488	0.631
Max Entropy	0.751	0.723	0.865	0.747	0.775	0.849	0.779	0.812	0.868	0.713	0.788
SVM	**0.754**	0.714	0.841	0.737	0.773	0.833	0.768	0.794	0.890	0.732	0.784
F-Score											
Naive Bayes	0.758	**0.716**	0.840	**0.759**	0.773	**0.826**	**0.724**	0.740	0.751	**0.695**	0.758
Decision Tree	0.703	0.603	0.725	0.644	0.737	0.675	0.589	0.597	0.570	0.508	0.635
Max Entropy	**0.769**	0.704	**0.848**	0.740	**0.793**	0.818	0.723	**0.750**	0.775	0.673	**0.759**
SVM	0.760	0.697	0.831	0.725	0.782	0.812	0.721	0.745	**0.812**	0.690	0.757

TABLE 5.1: Baseline precision, recall and F_1-Score for all algorithms

we can notice a difference in the performance of the remaining algorithms: Naive Bayes attained an high recall (the highest for 9 categories out of 10, with an average of 0.859) at the cost of a lower precision, while the opposite is true for Maximum Entropy and SVM, as both attained an higher precision (0.735 on average for both) at the cost of a lower recall.

	PP	**PC**	**MP**	**TMP**	**UP**	**PB**	**FU**	**RPR**	**SP**	**BRT**	**Avg**
Bag-of-words											
Naive Bayes	0.758	**0.716**	0.840	**0.759**	0.773	**0.826**	**0.724**	0.740	0.751	**0.695**	0.758
Decision Tree	0.703	0.603	0.725	0.644	0.737	0.675	0.589	0.597	0.570	0.508	0.635
Max Entropy	**0.769**	0.704	**0.848**	0.740	**0.793**	0.818	0.723	**0.750**	0.775	0.673	**0.759**
SVM	0.760	0.697	0.831	0.725	0.782	0.812	0.721	0.745	**0.812**	0.690	0.757
Bigrams											
Naive Bayes	**0.709**	**0.675**	**0.813**	**0.730**	0.771	**0.774**	**0.700**	0.738	0.711	0.655	**0.729**
Decision Tree	0.598	0.570	0.618	0.663	0.683	0.635	0.464	0.569	0.611	0.484	0.594
Max Entropy	0.685	0.665	0.803	0.726	**0.777**	0.765	0.683	0.725	0.702	0.641	0.718
SVM	0.689	0.663	0.809	0.725	**0.777**	0.771	0.696	**0.743**	**0.737**	**0.661**	0.728
Trigrams											
Naive Bayes	0.586	**0.572**	**0.722**	**0.677**	**0.718**	**0.685**	**0.638**	0.663	**0.555**	**0.625**	**0.648**
Decision Tree	**0.678**	0.457	0.365	0.564	0.646	0.292	0.143	0.262	0.262	0.110	0.426
Max Entropy	0.597	0.570	0.711	0.668	0.715	0.669	0.627	0.639	0.544	0.601	0.637
SVM	0.645	**0.572**	0.700	0.667	0.715	0.671	0.628	**0.664**	0.532	0.617	0.642
Bag-of-words + Bigrams											
Naive Bayes	0.759	**0.724**	0.853	**0.762**	0.782	0.825	0.723	0.750	0.755	0.689	0.766
Decision Tree	0.699	0.594	0.702	0.642	0.722	0.671	0.577	0.616	0.596	0.498	0.632
Max Entropy	0.761	0.716	0.856	0.757	0.806	0.817	0.723	0.750	0.753	0.688	0.765
SVM	**0.764**	0.714	**0.859**	0.754	**0.808**	**0.827**	**0.729**	**0.771**	**0.788**	**0.697**	**0,773**
Bag-of-words + Bigrams + Trigrams											
Naive Bayes	0.755	**0.725**	0.857	0.762	0.784	0.822	0.718	0.742	0.742	0.691	0.763
Decision Tree	0.687	0.588	0.682	0.639	0.721	0.666	0.590	0.616	0.606	0.514	0.632
Max Entropy	0.758	0.717	0.857	**0.763**	**0.807**	0.818	0.720	0.744	0.732	0.682	0.762
SVM	**0.759**	0.719	**0.863**	0.761	0.805	**0.825**	**0.722**	**0.767**	**0.778**	**0.693**	**0.771**

TABLE 5.2: Algorithms F_1-Score for different reviews representation

No preprocessing – different review representations. Going further, for our second set of runs, we aimed at evaluating which review representation, or combination of, leads to a better classification accuracy. Hence, we performed several runs changing the review representation while still not performing any preprocessing or introducing the score metadata. Table 5.2 summarizes results of this second set of runs.

Results highlight that adopting bigrams or trigrams does not improve the accuracy for any of the algorithms. We conjecture that this lack of improvement over the bag-of-words representation is due to the short length of reviews (in the sample we used for the manual analysis, the average length is 204 characters, with a standard deviation of 128) from which only a relatively low number of features can be extracted when choosing n-grams over bag-of-words. However the quality of the results improves when combining bigrams with bag-of-words. With this combined representation, Naive Bayes is outperformed by both Max Entropy and SVM, the latter achieving the best F_1-Score of 0.773. We conjecture that this improvement is due to the ability of this combined representation to extract a sufficient number of features, while preserving spatial information that helps when having to discern different usages of frequent words, e.g., the bigrams "excessive permissions" and "no permissions" have a very different meaning despite both containing the term "permission". Additionally including trigrams in this combined representation does not seem to lead to a further improvement.

With preprocessing – combined review representations. For our next set of runs, we introduced different preprocessing steps in the classification pipeline, while still maintaining the combined representation of bigrams and bag-of-words, which resulted in the best quality of results in the previous case. Results of this set of runs are provided in Table 5.3.

The results of this new set of runs show evidence that stemming, stopwords removal, and the combination of the two slightly improve the accuracy of the Naive Bayes and Decision Tree algorithms, with the former achieving an F_1-Score of 0.778 (the highest so far). SVM and Maximum Entropy are instead negatively affected by this preprocessing method. Usage of lemmatization, both alone and combined with stopwords removal, instead improves average accuracy *for all algorithms*. With lemmatization, SVM also achieves an F_1-Score of 0.778, tying for the best performance so far.

With preprocessing – with user ratings. In this case, we consider also user ratings as input to the classification pipeline. We performed both stopwords removal and lemmatization since they gave the best results in the previous set of runs. Table 5.4 shows the results we obtained in this set of runs.

	PP	PC	MP	TMP	UP	PB	FU	RPR	SP	BRT	Avg
Stemming											
Naive Bayes	0.770	**0.729**	**0.852**	**0.768**	0.789	**0.835**	**0.721**	**0.752**	0.765	**0.711**	**0.773**
Decision Tree	0.705	0.631	0.752	0.673	0.753	0.712	0.601	0.699	0.648	0.540	0.672
Max Entropy	**0.783**	0.689	0.827	0.738	**0.791**	0.821	0.695	0.714	0.784	0.657	0.751
SVM	0.755	0.664	0.815	0.735	0.784	0.796	0.671	0.709	**0.796**	0.601	0.734
Stopwords											
Naive Bayes	0.769	**0.729**	**0.854**	**0.770**	0.787	**0.834**	**0.722**	**0.758**	0.793	**0.718**	**0.778**
Decision Tree	0.703	0.629	0.756	0.676	0.753	0.714	0.615	0.724	0.702	0.548	0.683
Max Entropy	**0.781**	0.688	0.827	0.736	**0.792**	0.821	0.692	0.728	0.803	0.669	0.755
SVM	0.755	0.663	0.815	0.731	0.784	0.796	0.677	0.722	**0.815**	0.626	0.739
Lemming											
Naive Bayes	0.758	**0.727**	0.862	**0.765**	0.788	0.830	0.716	0.746	0.759	**0.711**	0.771
Decision Tree	0.702	0.597	0.724	0.653	0.721	0.703	0.593	0.648	0.663	0.517	0.653
Max Entropy	0.761	0.719	**0.864**	0.757	**0.822**	0.834	0.718	0.759	0.775	0.694	0.773
SVM	**0.765**	0.719	**0.864**	0.749	0.815	**0.836**	**0.723**	**0.777**	**0.803**	0.707	**0.778**
Stemming + Stopwords											
Naive Bayes	0.771	**0.728**	**0.853**	**0.768**	0.790	**0.833**	**0.721**	**0.753**	**0.770**	**0.702**	**0.773**
Decision Tree	0.707	0.630	0.754	0.673	0.754	0.712	0.603	0.718	0.675	0.561	0.680
Max Entropy	**0.780**	0.688	0.826	0.736	**0.791**	0.820	0.685	0.730	0.784	0.645	0.749
SVM	0.755	0.664	0.814	0.736	0.785	0.795	0.680	0.698	0.795	0.601	0.733
Lemming + Stopwords											
Naive Bayes	**0.781**	0.719	0.809	**0.775**	0.778	0.839	0.733	0.761	0.785	**0.722**	0.774
Decision Tree	0.718	0.628	0.681	0.661	0.686	0.717	0.641	0.710	0.609	0.484	0.654
Max Entropy	0.776	**0.720**	**0.810**	0.769	**0.786**	**0.853**	0.729	0.770	0.785	0.703	0.772
SVM	0.772	**0.720**	0.804	0.763	0.777	0.852	**0.745**	**0.776**	**0.807**	0.709	**0.775**

TABLE 5.3: Algorithms F_1-Score for different preprocessing techniques

	PP	PC	MP	TMP	UP	PB	FU	RPR	SP	BRT	Avg
Precision											
Naive Bayes	0.604	**0.911**	0.578	**0.868**	**0.892**	**0.820**	**0.833**	**0.784**	0.611	**0.667**	**0.757**
Decision Tree	**0.762**	0.691	0.791	0.664	0.739	0.699	0.641	0.661	0.648	0.542	0.684
Max Entropy	0.740	0.663	0.737	0.676	0.686	0.731	0.680	0.624	0.627	0.648	0.681
SVM	0.756	0.718	**0.799**	0.733	0.761	0.790	0.686	0.667	**0.681**	0.642	0.723
Recall											
Naive Bayes	**0.997**	0.278	**0.999**	0.451	0.615	0.702	0.240	0.432	0.745	0.317	0.578
Decision Tree	0.764	0.654	0.797	0.652	0.758	0.710	0.601	0.644	0.711	0.487	0.678
Max Entropy	0.923	**0.779**	0.914	0.820	0.824	0.816	0.868	0.747	0.828	0.745	0.826
SVM	0.885	0.774	0.916	**0.834**	**0.843**	**0.861**	**0.902**	**0.819**	**0.916**	**0.807**	**0.856**
F-Score											
Naive Bayes	0.752	0.426	0.732	0.593	0.728	0.757	0.373	0.557	0.672	0.429	0.655
Decision Tree	0.763	0.672	0.794	0.658	0.748	0.705	0.620	0.652	0.678	0.513	0.681
Max Entropy	**0.822**	0.717	0.816	0.741	0.749	0.771	0.763	0.680	0.713	0.693	0.747
SVM	0.815	**0.745**	**0.854**	**0.780**	**0.800**	**0.824**	**0.780**	**0.735**	**0.781**	**0.715**	**0.787**

TABLE 5.4: Final precision, recall and F_1-Score for all algorithms

In this new configuration, the Naive Bayes algorithm achieved the highest average precision, with a value of 0.757, but it came at the cost of recall, for which it performed worst with a value of 0.578. Differently, Max Entropy and SVM have an increased F_1-Score, while still retaining a reasonable balance among precision and recall. SVM, with an average F_1-Score of 0.787, is the configuration that attained the best performance. We statistically tested this result by applying the Wilcoxon rank-sum test [118]

for pairwise data comparison under the alternative hypothesis that the samples do not have equal medians. The performed test confirmed that this last configuration performs statistically better than the others (higher p-value = $2.2e^{-16}$). We believe that this improvement comes from the synergistic effect of all the techniques that we selected during the exploration. Based on this result, we adopted the **lemmatization+bigrams+bag-of-words+SVM** configuration to answer RQ1 and RQ2.

5.3.2 RQ2 - To what extent do app reviews express concerns about the Android run-time permission system?

We identified a total of 3,574 reviews that discuss the Android run-time permission system. Even if they amount for a very small portion of the 18,326,624 reviews we started from, these belong to a total of 1,278 unique apps, equal to the 23% of the collected apps that employ the run-time permission system, and 8.6% of the total amount of all apps of our dataset.

Looking at the frequencies of permission-related reviews among app categories (summarized in Table 5.5), we can observe that permission-related reviews occur in almost all categories, with the exception of *Libraries and demo* and *Events* (notice that both categories were recently introduced in the Google Play Store and thus contain a smaller amount of apps). This observation may be an indication that permission-related reviews are somehow orthogonal across apps, independent of the specific context and permissions requirements. We can also notice that, even if still limited in numbers, categories *Productivity* and *Tools* contain more permission-related reviews than others. The opposite is true for other categories, like *Games* and *Photography*, which have a lower amount, despite a high number of reviews.

5.3.3 RQ3 - What are the main concerns about the Android run-time permissions system in app reviews?

Results of both the manual and automatic classification are summarized in Table 5.6. Using our classification pipeline, we have been able to assign a total of 3,251 labels distributed among all categories of the manually extracted taxonomy. The total amount of assigned labels from the combination of manual and automatic classification is 4,156 (they are distributed over 3,574 unique reviews). The majority of the classified reviews belong to categories *Unclear permissions* (914), *Too many permissions* (745), *Permissions praise* (700), *Permission-related bug* (543) and *Minimal permissions* (521). A smaller amount has been identified for categories *Repeated permission request* (95), *Functionality unavailable* (86), *Setting permissions* (42) and *Bad request timing* (21), presumably since,

Category (cat_a)	**Apps**	**Reviews (R^a)**	**Permission-related reviews (R^a_p)**
Productivity	483	982,607	453 (0.0046‰)
Tools	490	1,343,676	372 (0.0028‰)
Communication	471	953,134	287 (0.0030‰)
Health and fitness	480	678,978	276 (0.0041‰)
Entertainment	498	1,039,598	234 (0.0023‰)
Shopping	484	631,367	216 (0.0034‰)
Business	495	482,456	209 (0.0043‰)
Lifestyle	494	713,815	201 (0.0028‰)
Social	491	747,139	176 (0.0023‰)
Android Wear	500	851,086	176 (0.0201‰)
News and magazine	491	419,504	149 (0.0035‰)
Travel and local	481	435,270	138 (0.0032‰)
Media and video	486	729,509	131 (0.0018‰)
Transportation	492	249,239	119 (0.0048‰)
Finance	493	523,841	102 (0.0020‰)
Music and audio	485	830,703	101 (0.0012‰)
Weather	484	237,229	99 (0,0041‰)
Personalization	497	713,124	99 (0.0014‰)
Education	500	675,976	81 (0.0012‰)
Photography	496	1,002,740	78 (0.0008‰)
Sports	488	331,062	78 (0.0023‰)
Games	444	1,654,846	76 (0.0004‰)
Books and reference	495	551,705	67 (0.0012‰)
Family	418	848,046	58 (0.0007‰)
Medical	497	195,632	47 (0,0024‰)
Food and drink	498	115,387	35 (0.0030‰)
Auto and vehicles	493	54,428	31 (0.0057‰)
House and home	292	57,990	23 (0.0040‰)
Comics	500	116,881	19 (0.0016‰)
Dating	247	61,489	19 (0.0031‰)
Beauty	95	6,760	11 (0.00162‰)
Art and design	492	44,323	4 (0.0090‰)
Parenting	185	26,667	2 (0.0001‰)
Libraries and demo	137	17,615	0 (0‰)
Events	52	2,748	0 (0‰)
Sum total	15,124	18,326,624	3,574 (0.0019‰)

TABLE 5.5: Distribution of apps, reviews, and permission-related reviews across categories

for these categories the amount of reviews identified during the manual analysis was insufficient for proper training of the classifier.

The heatmap in Figure 5.3 provides an overview of how the frequency of these permission-related reviews is distributed among apps, grouping the latter according to the number of requested permissions. Here, we focused only on dangerous permissions [119], the only ones that must be granted at run-time after the changes introduces by Android 6. We can notice that, for apps that only request one or two dangerous permissions the majority of permission-related reviews belongs to categories *Minimal Permissions* (MP) and *Permission Praise* (PP) thus suggesting that privacy-aware users notice, and appreciate, the low amount of privileges required by these apps. Unsurprisingly instead, for apps that request all 9 dangerous permissions, the majority of permission-related reviews belong to category *Too Many Permissions* (TMP). Furthermore, focusing on apps that request between 3 and 6 dangerous permissions, we notice that the majority of permission-related reviews for these apps belong to categories *Too Many Permissions*

Category	Manual Classification	Automatic Classification	Total
Permission praise (+)	128	572	700
Minimal permissions (+)	121	400	521
Permission complaint (-)	100	386	486
Too many permissions (-)	151	594	745
Unclear permissions (-)	175	739	914
Permission-related bug (-)	120	423	543
Functionality unavailable (-)	41	45	86
Repeated permission requests (-)	28	67	95
Settings permission (-)	20	22	42
Bad request timing (-)	21	3	24
Sum total	905	3,251	4,156

TABLE 5.6: Breakdown of classification results

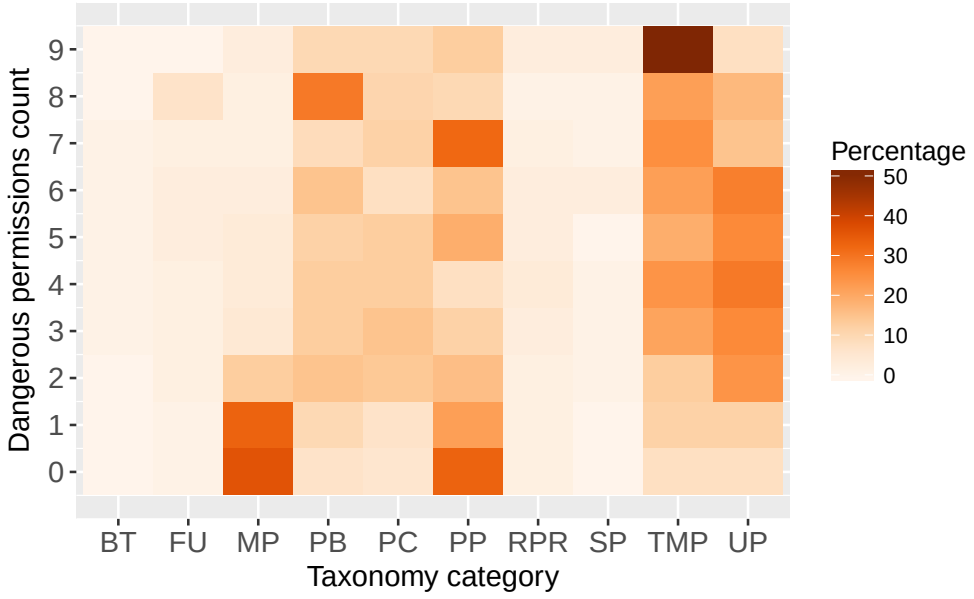


FIGURE 5.3: Permission-related reviews by number of requested permissions

(TMP) and *Unclear Permissions* (UP) but does not significantly increase, as one would expect, as the number of requested permissions increases. We may conjecture that users are keen to form a binary opinion about the number of permissions requested by an app: they appreciate it when a low amount of permissions is requested and they are disappointed when it is higher than they expect.

Figure 5.4 show the distribution of categories of permission-related reviews across the types of requested permission. Praises and complaints about permissions expressed by users in app reviews are distributed evenly, independently of the type of requested permission, with the sole exception of the *Sensors* permission, for which users seem to be concerned more when too many permissions are requested.

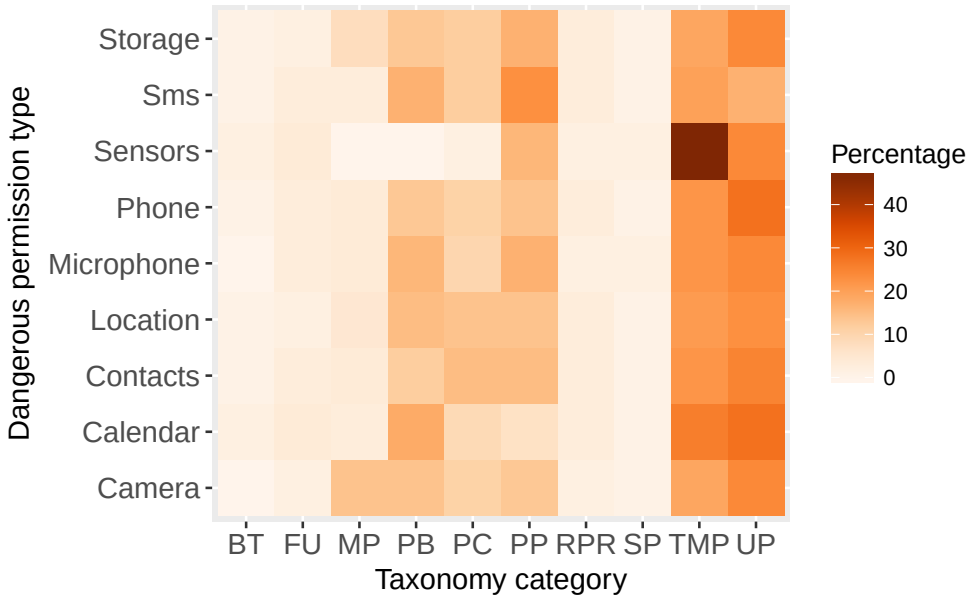


FIGURE 5.4: Permission-related reviews by requested permission

5.4 Discussion

Overall we identified 3,574 reviews discussing the run-time permission system. These reviews belong to a total of 1,278 unique apps, which are 23% of the collected apps that employ the Android run-time permission system, and indeed the 8.6% of the total amount of apps collected from the store. Considering these numbers, we can infer that:

> The number of users having concerns w.r.t. the Android run-time permission system is quite limited, even though problems pertaining to permissions are widespread among apps (As shown in Section 5.3.3 23% of apps in our study employing the Android 6 permission system are affected).

Going one step further, when focusing on the macro-categories of positive and negative opinions, we can notice that the majority of classified reviews belong to the latter and amount to 70.6% of the total. Negatives opinions appear in a total amount of 2,459 reviews, as opposed to 1,185 reviews containing positive ones (note that only 70 reviews contain both positive and negative opinions).

The distribution of ratings, shown in Figure 5.5, is skewed towards the maximum score (*i.e.,* 5 stars) for positive categories; instead the distribution of ratings of negative categories is skewed towards the lower end. We verified for statistical significance of these differences in ratings by performing the two-tailed Mann-Whitney U-test among

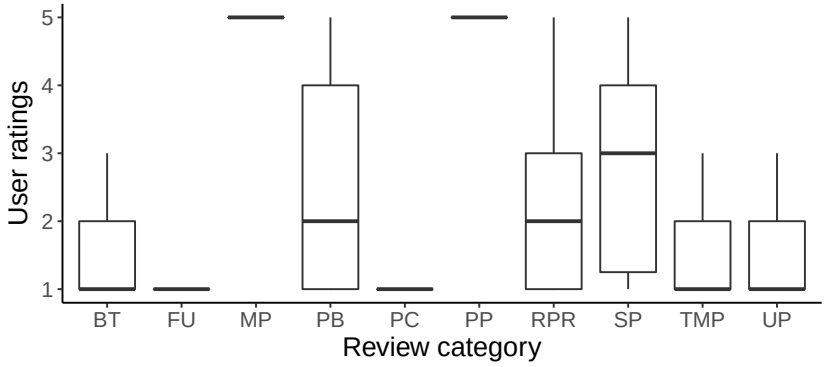


FIGURE 5.5: User ratings across reviews categories (outliers not shown)

each possible pair positive-negative categories, obtaining always a p-value below $2.2e^{-16}$. Henceforth, we can deduce that:

> For privacy-aware users, negative concerns on how permissions are handled are correlated with negative concerns about the whole app, thus confirming the importance of users' judgments about permissions.

We identified a total of 972 apps whose reviews contain negative opinions. Analyzing the number of identified reviews, displayed in Table 5.6, negative categories with the highest count are *Unclear permissions* (914), *Too many permissions* (745) and *Permission-related bug* (543). The high cardinality of the first category evidences one of the problems of the run-time permission system:

> Users still do not understand why they are being asked for a permission. More work is required to fully point out the reasons, which could derive from poor design of the run-time permission system itself or from developers that do not utilize it properly, often requesting permissions without providing an explanation.

App `com.chopracenter.meditationexperience` is a clear example of the latter, one of the apps with the highest amount of this kind of reviews. One of them points out: "*Why does the app now need to use my permission to use my phone and my contacts? Seems unnecessary.*". Furthermore, the high number of reviews belonging to the category *Too many permissions* leads us to believe that:

> The problem of *permission creep* [23, 120], i.e., apps routinely requesting more permissions than needed to carry on with advertised functionalities, is still present under the new run-time permission system.

This issue is clear when looking at the reviews of `com.lge.app1`, an app having numerous reviews of this kind: "*Update app to use less permissions and I'll reconsider. No way it needs to make phone calls or see my calendar to function with the TV.*". The high count of reviews classified under *Permission-related bug* evidences that:

> The introduction of the new run-time permission system also introduced a new class of bugs that didn't exist before, in the old install-time permission system. These bugs derive from the fact that developers do not always correctly perform permission requests.

We believe that this misusage of permission requests by developers is partially due to a lack of understanding of the inner workings of the run-time permission system, and partially because it is hard for developers to foresee all points where a permission must be requested at run time. Indeed, Android apps often have a complex event-driven control-flow with multiple entry points [121]. For instance, one of the apps with a high amount of this kind of reviews is `com.getittechnologies.getit`, where users highlight that the developer never requested the required permission: "*It can't access my GPS location but I don't think it ever requested permission to do so*".

Even if not among the most numerous, the categories *Repeated permissions requests* and *Functionality unavailable* can be an indication of the fact that:

> Some developers do not adhere to the guidelines provided by Google [122, 123]. Rather, they attempt at coercing users to provide some permissions, by constantly repeating permission requests or completely blocking access to specific features when permissions are not granted.

One example of the former case is `com.htc.Weather`: "*Annoying that it keeps asking permission of location [...]*". An instance of the latter case is `com.dteenergy.mydte`: "*I can't effectively use it because I denied accessing my contacts and files. There is no reason to allow that [...]* ".

From a more technical standpoint, our analysis shows that:

> Well-established, off-the-shelf machine learning techniques, combined with basic NLP preprocessing, can be profitably used to derive interesting insights from app store reviews, even on specific topics when sufficient amounts of data are considered.

5.5 Threats to validity

In the following, we discuss the threats to validity of our study according to the Cook and Campbell categorization [124].

Internal validity refers to the causality relationship between treatment and outcome [12]. In our study we classified user reviews dealing with Android run-time permissions relying on machine learning techniques. This kind of analysis is non-deterministic as different runs of the analysis may lead to different results. To mitigate this potential threat, we repeated the experiments multiple times to correctly assess the accuracy of the different combinations of applied techniques. Moreover, we manually created training data for the machine learning models, following the procedure described in Section 5.2.1. To ensure that the final model used for classification does not show abnormal behavior, we manually analyzed 100 classification results to exclude the presence of glaring anomalies.

External validity deals with the generalizability of obtained results [12]. To ensure that our subjects are representative of the population of Android apps, we downloaded the top 15,124 apps in the United States across all categories of the Google Play Store, as ranked by App-Annie. Since the apps are the top ranking apps of all categories, we can expect that they have a high number of users because they are ranked using a combination of number of downloads and aggregate user ratings. Also, by considering the most popular free apps per category, we increase the chance to include apps with a more active user base in terms of quality and number of reviews. Free apps represent 75% of all Google Play Store apps and they are downloaded more often [125].

Construct validity deals with the relation between theory and observation [12]. The goal of our study is to analyse user reviews dealing with the Android run-time permission system, with the ultimate goal of identifying its recurrent issues from the end user perspective. Even when focusing on issues potentially noticeable by end users, only a minority of them show sufficient awareness. Hence, the proposed approach might not discover the more subtle issues. We mitigate this threat by considering an initially large set of reviews. Moreover, as we employed keyword-based filtering, we might have missed an amount of reviews that reference permissions using keywords not in our list. We mitigated this threat by adopting a list of highly pertinent neutral keywords, thus, albeit in lower number, the selected reviews are highly on target for our purposes.

Conclusion validity deals with the statistical correctness and significance [12]. In this study we assumed that user reviews are a reliable source for inferring user concerns about the Android run-time permission system. However, there may be other factors that potentially may affect users judgment. To mitigate this threat, while performing the initial manual analysis we also ensured that many permissions-related reviews bring

meaningful insights on the subject. Numerous examples of purposeful reviews have been reported in the paper to highlight the usefulness of classified concerns. The full set of classified user reviews is publicly available in the replication package of this study.

5.6 Open challenges

From the observations of Section 5.4, three open challenges for future research can be identified. We list and discuss each in the following.

Link permissions to functionalities - In Section 5.4, it was evidenced that: (*i*) users still tend to not understand why they are being asked for permissions, and (*ii*) developers routinely request more permissions than needed. We believe that one possible way to address these problems is to design new permission systems in which permissions are granted to individual app functionalities, rather than to the full app as a whole. With such systems, users may better understand why a given permission is requested. The first stepping stone in this direction is to devise an effective way to *logically link permissions to functionalities*.

Better support for developers - A numerous category of user perceptions is about *Permission-related bugs*. Indeed, the shift to run-time permissions burdened developers with one additional task, as now they have to preemptively ask for allowance of permissions before accessing restricted parts of the platform. Failure to properly do so renders functionalities or, in worst cases, the whole app unreachable and/or unusable. In order to properly deal with this additional task, developers should not only be further educated, but also empowered with tools that assist them in correctly handling permissions and suitably positioning the corresponding requests.

Impact of run-time permission requests on user experience - Developers are faced with one additional challenge when dealing with run-time permissions: on the one hand, they must promptly and precisely inform users about needed sensitive data; on the other hand, they must carefully plan the usage of notification dialogs, so as to avoid disrupting the user experience (recall the phenomenon of warning fatigue [33]). When using run-time permissions, there can be cases in which introducing new functionalities that require additional permissions may even be detrimental to the app success. Hence, a further open research area is understanding and quantifying the effects of run-time permission requests on the overall user experience and how performing them at different phases of program execution impacts user experience.

5.7 Conclusions and future work

In this chapter we reported on the design, execution and results of an empirical study aimed at investigating answers to the second of the research questions addressed by this dissertation (discussed in Chapter 1.3). That is:

RQ2 - *Are there any existing issues in current Android security- and privacy-preserving mechanisms that negatively affect the users? trust on the whole platform?*

To investigate possible answers to this question we conducted a large-scale empirical study to investigate users' perceptions about the run-time permission system offered by Android. We inspected over 4.3 million user reviews out of 5,572 apps within the Google Play Store. The reviews were classified and analyzed by employing a classification pipeline based on machine learning and natural language processing techniques. Its accuracy has been empirically evaluated with promising results and, starting from recurring concerns expressed by users, we identified several points of improvement for the run-time permission system.

Our investigation confirms the presence of issues in current security- and privacy-preserving mechanisms as **we have found permission-related issues to be widespread among apps** (23% of Android 6 compliant apps are affected) and, for privacy-aware users, **they have a significant impact on their judgment about the whole app**. Moreover, multiple problems that existed in the previous iteration of the Android permission system are still present today: **users still do not comprehend why they're being asked for a permission** while developers routinely ask for more permissions than needed and fail to follow best practices. In Chapter 7, we present an approach that addresses the first two of these challenges, while providing users with increased control over granted permissions.

Chapter 6

Permission Issues in Open-source Android Apps:An Exploratory Study

Mobile apps nowadays empower users with the ability to quickly and efficiently perform a wide range of tasks, from stock trading to recording of vital health information. Although these apps provide immense amounts of power, they also present an unparalleled opportunity for security and privacy threats. Due to the magnitude of these threats, it is imperative that mobile developers create apps capable of sufficiently protecting our privacy and security [126].

The sensitive data and functionality used by an app is protected through permissions. Android apps use a permission-based system where an app requires specific permissions to carry out specific operations [119]. It is crucial for developers to make proper permission-related decisions since improperly used permissions (under and over-permissions) carry a wide range of ramifications. These include increased app susceptibility to malware and unwanted data leakage to ad libraries [35, 127, 128]. Additionally, not adhering to permissions best practices may have a wide range of implications. These may range from hurting the user experience, to creating functional defects and privacy and security-related issues [43, 129, 130].

Unfortunately, developers do not always correctly use permissions for numerous reasons, including a lack of permissions-related knowledge [131] and even confusion over the permission's name [35]. There is substantial work examining the detrimental effects of permissions misuse [19, 63, 97] and tools to assist in the identification of a variety of *permission-related issues* (PRIs) [35, 132]. However, none of the existing work examines when, why and who is making permissions-related mistakes when developing apps.

In this study, we aim to provide a better understanding of how developers are creating and fixing permissions-related issues and the types of mistakes developers were making.

To this aim, we analyzed the GitHub repositories of 574 apps collected from the F-Droid repository [133]. Using custom-built software along with the existing permission analysis tools M-Perm [134] and P-Lint [130], we identified a variety of PRIs ranging from not properly adhering to permissions best practices to apps requesting too many permissions. This empirical information provides us with a history of the app's development life cycle including (*i*) When permissions and their related issues were introduced and fixed, (*ii*) who is making these decisions, (*iii*) file-change history that we could examine using permissions analysis tools, and (*iv*) all other commit information such as commit messages.

Our obtained results reveal that: (*i*) *PRIs are a frequent phenomenon* in the context of Android apps (about 50% of examined apps exhibit at least one PRI, with over-permissions being the most recurrent one); (ii) the majority of issues are fixed in a timespan of a few weeks after their introduction; (iii) in many cases, permission-related issues can linger inside an app for an extended period of time, that can be as high as several years, before being fixed; (iv) developers tend to introduce (and fix) different types of PRIs independently of their experience within the project, but more experienced developers tend to introduce and fix more under-permission issues.

To summarize, the main contributions of this study are:

- characterization of the *frequency* of PRIs and their *decay time* in the context of 574 open-source Android apps;
- an objective assessment if PRI introduction and repair correlates with the *experience of the developer* performing it;
- the *replication package* of the study containing its results, raw data, and mining- and data analysis scripts [135].

6.1 Goal and research questions

The primary **goal of this study** is to provide a better understanding of permission-related issues introduced and fixed by developers in Android apps. To achieve this goal, we first collect 2,002 Android repositories from F-Droid and then analyze these repositories using three existing open-source analysis tools: M-Perm [134], P-Lint [130], and oSARA [136].

The following are the research questions that we set to answer:

RQ1 - What are the most common types of permission-related issues in Android apps?

RQ2 - How long do permission-related issues tend to remain in Android apps across their lifetime?

RQ3 - To what extent does developer experience correlate with the introduction of permission-related issues?

RQ4 - To what extent does developer experience correlate with fixes of permission-related issues?

By answering RQ1, we aim to determine the most prevalent permission-related issues in Android apps, thus developers can be made cognizant of these issues and devote appropriate efforts to avoid them in their apps. Answering RQ1 will also help researchers in gaining better insights into the prevalence of permission-related issues in Android apps. Previous work has examined permissions-issues on the older install-time model [35, 132, 137, 138], yet, to our knowledge, this is the first study that examines permission-related issues on a large scale on the current Android run-time model.

Answering RQ2 will allow us to understand how long permission-related issues typically exist in the code of Android apps and can provide insight on how long introduced issues can be expected to impact the app. Indirectly, answering RQ2 provides an objective indication about how Android developers are considering a priority to locate and address permission-related issues.

Answering RQ3 allows us to determine if a developer's experience within a project significantly correlates with the introduction of PRIs and provides insight on who should be making permission-based decisions and modifications in Android apps. Also, it can provide additional insight on if developers with low or high experience in the project are creating a disproportionately high amount of permissions-related issues. This can create the foundation for improving the assignment of code reviews. For example, security-oriented reviews may be performed on code authored by developers whose experience is more correlated with the introduction of PRIs.

Answering RQ4 provides insight on if developers with higher amounts of project experience are devoting a correlating amount of effort to fix PRIs. Additionally, this provides insight on if developers with more experience are more adept at fixing permission-related issues. The knowledge related to RQ4 can guide the assignment of security-oriented refactoring sessions or code reviews directly to developers with more than one year of experience, as they tend to fix more PRIs.

6.2 Data collection and analysis

The data collection and analysis process consists of three phases: (*i*) *Repository Collection*; (*ii*) *Detection of PRIs*; and (*iii*) *Data Analysis*. In the Repository Collection phase, we mine the F-Droid catalog to obtain a list of open-source Android apps and perform a set of filtering steps on collected apps. The PRI Detection phase involves the execution of the P-Lint and M-Perm tools for statically analyzing apps source code and project files. In the last phase (Data Analysis), the results of the static analysis tools are statistically analyzed.

6.2.1 Repository collection

F-Droid is a catalog of free open source apps for the Android platform. F-Droid contains links to Android app Github repositories. These projects range from small infrequently updated apps, to large popular apps. We chose F-Droid as our primary source for open-source Android projects due to the diversity of apps in its catalog and for its use in prior research work [139–141]. To retrieve the project repositories of the cataloged apps, we first cloned the F-Droid repository and then parsed the text files associated with each app to extract the apps' metadata. Extracted metadata includes name, description, category and repository URL of each app. We then cloned the GitHub repository of all the apps. In order to avoid duplicates, we excluded apps from our dataset that were duplicated/forked by ensuring that all source URLs and commit log SHA's are unique. After cloning the repositories, we extract the following data from each of them:

- *Commit Log Details*: via the `commit log` git tool, we retrieve additional data associated to each commit, such as the author and committer of the commit, their respective timestamp, the commit message.
- *Affected Files*: for each commit of all apps, we examine the list of affected files and extract the revision of all the "`*.java`" and "`AndroidManifest.xml`" files.

As shown in Figure 6.1, we mined a total of 2,002 GitHub repositories. Since we used GitHub repositories, we ran the risk of including inactive or unmaintained repositories in our study [142]. To help mitigate this risk, we consider only those repositories that (i) have a lifetime span[1] of at least 8 weeks, (ii) contain at least 10 commits, (iii) with at least one commit since January 2017, and (iv) also published on the Google Play store. The 10-commits threshold comes from the fact that 90% of all considered repositories

[1]Lifetime span: the range between the first and last commits of a repository.

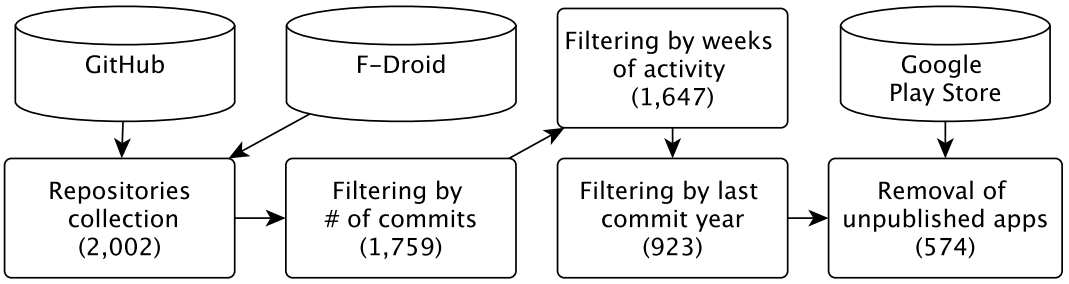


FIGURE 6.1: Repositories collection and filtering process

have more than 10 commits before this filtering step. The 8-weeks threshold comes from the fact that 8 weeks is the average development time for an Android app [143] and it has been used in previous study on mining GitHub repositories of Android apps [144]. The 2017 rule has been adopted in order to filter out unmaintained apps, without removing apps that are seldom updated. We excluded from our dataset apps not published on the Google Play store in order to filter out unfinished or proof-of-concept apps. This filtering results in a final dataset of 574 active GitHub repositories, containing a total of 502,907 commits performed by 7,945 unique developers.

Table 6.1 provides a summary of demographics for apps included in the study. As demonstrated, apps in our dataset have a median rating on the Google Play store of 4.294 (out of a maximum of 5), while the median number of installs[2] is 10k. At the same time, median number of commits for apps in our study is 260 and median number of committers per app is 7. Based on these numbers, we are relatively confident that the apps considered in our study are of good quality and representative enough of real-world projects.

TABLE 6.1: Demographics of apps included in the study (SD = standard deviation, IQR = inter-quartile range)

Metric	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**IQR**
Rating	0	5	4.294	4.179	0.6681	0.4767
Installs	1	100m	10k	926.1k	7,594k	99k
Commits	11	34,380	260	876.10	2246.97	707.5
Committers	1	486	7	16.67	32.71	13

[2] Google Play does not provide the precise number of installs, but instead provides a range (*i.e.,* 100-1000). We conservatively adopted the bottom of the range for our calculations. Hence, all statistics on installs should be considered as a lower bound.

6.2.2 Detection of PRIs

To detect permission-related issues in Android apps, we run the M-Perm [134] and P-Lint [130] tools. Although both have been used in foundational studies [130, 134], we decided to further evaluate them before including them in our own research. Other permission analysis tools, such as Stowaway [35] and PScout [132], have been used in existing literature to conduct permission analysis. However, a direct comparison with these tools was unfeasible, as both are several Android versions out of date, and neither is compatible with the current run-time permission model. Hence, we decided to conduct our own evaluation of *precision* and *recall* [145] of M-Perm and P-Lint. This was accomplished by creating a small set of 5 oracle Android apps in our study. These were a simple calendar, camera, SMS messaging, contact storage and location recording app. We then created a second version of these apps with each having: one unique over and under permissions and one permission-smell. We then ran each tool against these 10 apps and obtained a value of 1.00 for both precision and recall. Although largely elementary, these positive results provided confidence in the benefits of incorporating these tools into our study. These oracle apps are available on the project website [135].

After the successful analysis of these tools, we used them to analyze all 502,907 commits belonging to the 574 apps in our dataset. These tools enable us to identify a variety of permissions-based issues, ranging from not correctly adhering to the permission standards proposed by Google [129], to more severe issues such as over-permissions. Table 6.2 presents the PRIs considered in this study. **M-Perm** is able to detect occurrences of over and under-permission issues (*i.e., O* and *U* PRIs). An app is *over-privileged* if it requests too many permissions. Likewise, if it asks for too few permissions then it is *under-privileged* [35]. Apps that misuse permissions have an increased attack surface, making them more susceptible to a variety of security and privacy related issues [35, 134]. M-Perm analyzes Android $\geq$ 23 apps and identifies instances of being over and under-privileged.

Similar to code smells, *permission smells* are symptoms of issues, but are not a definitive indication that a problem exists [130]. **P-Lint** analyzes Android $\geq$ 23 apps for proper permissions usage from a standards perspective. In this study we focused on the missing check (*i.e., MC*) and multiple requests in proximity (*i.e., MRP*) PRIs since (i) they were prevalent, occurring in a large number of apps and (ii) they were well-defined and had a clear negative impact. We focused on these four types of PRIs because they are (I) Impactful (II) Well-defined (III) Have been extensively analyzed in existing work (permission gap) [35, 63, 138]. We focused on Android because we were able to easily collect and reverse engineer a large set of Android apps. We are unaware of such an available dataset for iOS apps, or the necessary tools to analyze them.

TABLE 6.2: Permission-related issues detected in this study

ID	Permission Issue	Primary Issue		Tool
		Quality	Security	
O	*Over-permission:* too many permissions, violating the principle of least privilege.		✓	M-Perm
U	*Under-permission:* not enough permissions requested.	✓		M-Perm
MC	*Missing Check:* app does not call `checkSelfPermission()` when making permission requests.	✓		P-Lint
MRP	*Multiple Requests in Proximity:* Multiple permission requests made in close proximity, possibly overwhelming user with requests.	✓		P-Lint

After the detection of the PRIs, we next determine the commits that *introduced* and *fixed* each of them. This is a non-trivial task because identifying these issues involves much more analysis than merely examining each committed version with the static analysis tools.

The following statuses were used to define each PRI event:

- **New.** When a PRI is found, we check if it exists in the app at the time of the previous commit. If it does not, starting from the version containing the issue we examine each version of the app in a commit-by-commit fashion to determine the commit that introduced the PRI. Identifying this commit allows us to determine the committer responsible for introducing the PRI.

- **Exist.** If the detected PRI is also found to exist in the previous and subsequent versions of the app, then we record it as 'Exist' since the commit does not modify the state of the issue. These are expectedly observed quite frequently as developers often make a variety of changes to apps that are not permission-related.

- **Fix.** For every detected PRI, we check if the PRI exists in the subsequent committed version of the app. If it does not exist, then our task is to determine the commit that fixed the issue. We accomplish this by starting with the immediately subsequent commit after the version of the app exhibiting the detected PRI and examining its source code using the static analysis tools. If the issue is not found, then we mark the current commit as the commit that fixed the issue. If the issue persists, then we perform the same process on each subsequent commit until we find the commit that fixed the issue. This enables us to identify the committer responsible for fixing the permission issue. If we reach the last commit of the repository and no PRI fixing commit is found, then the PRI is marked as *unresolved*.

Demographics information about the detected PRIs and their related commits are provided in Section 6.3.1.

oSARA Tool and Replication Package. We developed the open Source Android Repository Analyzer (oSARA) tool to perform the necessary data collection and analysis for our study. oSARA performs the following tasks: (I) Collects all relevant Android repository information from F-Droid (II) Extracts all relevant permission information and versions from these repositories (III) Analyzes each extracted version for PRIs using M-Perm and P-Lint (IV) When PRIs are discovered, oSARA analyzes previous and subsequently committed files to determine the commit that either added or removed the PRI. Using this commit information, we are able to discern information about the developer performing the commit.

Our project website [135] contains all source code, along with installation and usage instructions for oSARA. The project website also contains the complete raw dataset (> 6 GB) and schema details of our collected data, along with the oracle Apps used to verify P-Lint and M-perm. The objective is to make these tools and datasets not only available for other researchers to validate and recreate our study, but to also build off our work as well.s

6.2.3 Data analysis

In the following we list the techniques and tools adopted to answer each research question:

RQ1. We account for all occurrences of each type of PRI and provide an indication about their distribution by means of summary statistics. We employ the Fisher?s exact test [146] to assess independence of observations among occurrences of the four PRIs types. We adopt the Fisher?s test over other alternatives (e.g., χ^2-test) due to its robustness when dealing with sparse, unbalanced data. We employ the same test to perform post-hoc analysis, performing all tests for all pairs of populations and adjusting resulting p-values for inflation due to multiple comparisons via the Holm correction procedure. The omnibus Friedman test [118] is then used to statistically determine if the four types of PRIs exhibit a significant difference. The Friedman test is a non-parametric test for one-way repeated measures analysis of variance by ranks. We use the Friedman test because (i) RQ1 is designed as a 1 factor – 4 treatments experiment, (ii) the collected data is not adhering to the assumptions of the ANOVA statistical test, and (iii) the Friedman test is a non-parametric alternative to ANOVA that does not assume independence of observations. We apply the Conover's all-pairs comparison test as post-hoc analysis for performing pairwise comparisons among each pair of PRI types [147]. Since we are applying multiple statistical tests, we correct the obtained p-values via the Holm correction

procedure [148]. In addition, we compute the effect-size of the differences among PRIs distributions using the Cliff's delta (d) non-parametric effect size measure [149], which measures how often values in a distribution are larger than the values in a second distribution. Cliff's d ranges in the interval [-1, 1] and is considered negligible for $d < 0.147$, small for $0.148 \leq d < 0.33$, medium for $0.33 \geq d < 0.474$, and large for $d \geq 0.474$.

RQ2. In this phase of the study, we collect the *decay time* of each occurrence of PRI. The decay time of a PRI represents the number of days in which a PRI is present in the source code of an app. We compute the decay time of a PRI as the difference (in days) between the timestamp of the commit in which the PRI has been fixed and the timestamp of the commit in which it has been introduced in the GitHub repository of the app. In this phase of the study we consider exclusively the PRIs which have been fixed along the lifetime of the app, so that their decay time is meaningful (*i.e.,* the last commit of a PRI includes the actual fix of the PRI and it does not correspond to the last commit within the whole repository).

Summary statistics are used for providing an indication about how decay times vary across the four types of PRIs. The same statistical tests as in RQ1 (*i.e.,* Friedman, Conover, Holm correction and Cliff's delta) are used for statistically assessing the differences of decay times across PRIs.

RQ3, RQ4 - Both research questions RQ3 and RQ4 are based on the concept of developer' experience. In existing literature, a number of repository-based metrics for proxying developer's experience have been proposed, such as (*i*) the *Developer's Commit Ratio* (DCR), defined as the number of contributions made by a given developer for a repository divided by the number of all commits done by all repository contributors [150], (*ii*) *maintainers* and *contributors* defined as those contributors with more than 30% and less than 10% of all repository's commits, respectively [151], and (*iii*) project *newcomers* defined as those contributors with less than 3 commits in a repository [152]. However, even if those metrics are relatively simple to implement, they are not applicable in our study because of the following issues: (*i*) they heavily rely on the number of commits a developer does, which may greatly vary due to different developer's habits or organizational guidelines, (*ii*) they do not take into account the exact time in which a developer is performing a commit, *e.g.,* a developer can be a simple contributor at the beginning of the project and later become a maintainer, and (*iii*) some of them strongly depend on the total number of commits in the repository, making them semantically different and not comparable across projects, *e.g.,* a developer with 30 commits is categorized as a contributor in a project with 1,000 commits, whereas he/she is categorized as a maintainer in a project with only 100 commits. Based on these reflections, in this study

we represent the experience of a developer d at a given commit c in a repository r as:

$$exp(d, c, r) = authDate(c^d) - authDate(fCommit(d, r)) + 1 \tag{6.1}$$

where c is the specific commit in r for which we want to calculate d's experience, $authDate(c^d)$ is the day in which a commit c has been authored by a developer d and $fCommit(d, r)$ is the day of the first commit authored by d in r. Intuitively, $exp(d, c, r)$ represents the number of days in which the developer is active in a specific repository at the time of commit c. We opted for the $exp(d, c, r)$ metric since (i) it is independent from the commit frequency of the developer (which may very depending on her development style, personality, project guidelines), (ii) it takes into account the time in which a commit is performed, and (iii) it is independent from the size of the repository r. Intuitively, an *exp* value of 1 indicates that commit c is the first contribution of the developer to the project, while an *exp* value of 365 indicates that a year has passed since the developer first contribution to the project.

In order to avoid the well-known *aliasing problem*, *i.e.,* the same developer having multiple identities in GitHub repositories [153], in this study we apply the Naive heuristics proposed by Kouters *et al.* for resolving developers using multiple identities when committing to the same repository [154]. The heuristics merges committers with the same email prefix, *i.e.,* the part before the `@` symbol. Despite its apparent simplicity, there is empirical evidence that the Naive heuristics proposed by Kouters *et al.* provides a good enough trade-off between performance and simplicity of implementation when considering long time frames as in our study [155]. We refer the reader to [155] for a detailed evaluation of various heuristics for solving the aliasing problem.

For answering RQ3 and RQ4, we produce a combination of violin plots and descriptive statistics showing how developer experience is distributed among commits introducing or fixing each type of PRI. Next, we apply the omnibus Friedman test and the Conover test as post-hoc analysis for statistically testing how the medians of developers' experience vary across the four types of PRIs. We then correct the obtained p-values using the Holm correction procedure. We use the Cliff's delta to estimate the magnitude of the difference between PRI types

6.3 Results and discussion

Discussed below are the results obtained for our research questions.

6.3.1 RQ1 – What are the most common types of permission- related issues in Android apps?

Results. Table 6.3 provides descriptive statistics for occurrences of PRIs, as well as counts of unique issues and affected apps. A total of 3,900 unique permission issues were identified. They are distributed across 402 distinct apps, with a median of 1 PRI per app. For all types of PRI, we can observe that the mean amount of occurrences is higher than the median, meaning that the average is influenced by apps in the upper part of the data that exhibit an especially high amount of PRIs. Furthermore, we can observe that over and under-permissions are the two most common issues, with 2,635 and 939 occurrences respectively. We can also note that in our dataset apps have on average more than four over-permission issues and more than one under-permission issue. Diffusion of issue types *MC* and *MRP* appears to be on a lower scale, with 205 and 91 instances affecting 60 and 9 apps, respectively.

TABLE 6.3: Descriptive statistics for occurrences of PRIs
(SD = standard deviation, IQR = inter-quartile range)

PRI	#	**Affected apps #**	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**IQR**
O	2,635	387	0	269	1	4.59	16.98	3
U	969	82	0	251	0	1.69	12.31	0
MC	205	60	0	32	0	0.36	1.76	0
MRP	91	9	0	67	0	0.16	2.84	0
Aggr.	3,900	402	0	377	1	6.79	26.28	4

As a preliminary step to further analysis, we test for independence of observations among the four PRIs types. We statistically test this hypothesis by applying the Fisher's exact test. The results of the test (p - value < 0.01) allow us to reject the null hypothesis of independence among occurrences of PRI types. Likewise, the null hypothesis is always rejected (p - value < 0.01) for all post-hoc pairwise comparisons.

Differences in mean and standard deviation across the four types of PRIs suggest that the distribution of occurrences differs according to PRI type. We statistically test this hypothesis by applying the Friedman omnibus test. The results of the test ($p - value < 0.01$) allow us to reject the null hypothesis, thus informing us that difference of medians across different PRI types is statistically significant. Results of pairwise comparisons, performed via the Conover's test, reveal that the distribution of occurrences of each PRI type is statistically different from the others. Estimations of magnitude of differences, via pairwise applications of Cliff's d, reveal a large effect size for all pairs involving PRIs of type O, while it is negligible for all other pairs.

Main outcomes. Permission-related issues are a frequent phenomenon in Android apps. The vast majority of the analyzed apps suffer from the presence of at least one PRI. Over and under-permissions are more prevalent than MC and MRP PRIs. The distribution of occurrences significantly differs for each PRI type. By examining the number of issues identified for each PRI type, we can easily observe that the majority of issues is of types *O* and *U*. The mean amount of occurrences per app differs among the two, with a value $\mu = 4.59$ for the former and $\mu = 1.69$ for the latter. These results provide an initial notion of the prevalence of over- and under-permission phenomena in Android apps, as partially also confirmed by Felt *et al.* [35]. Moreover, by examining the counts of identified issues for all PRI types, we notice that issues of types MC and MRP amount to a comparatively small minority of the total. Although further research is required to fully determine the reason behind this imbalance, we believe that a primary factor is that MC and MRP issues are harder to introduce. In fact, in order to introduce MC or MRP, specific conditions must be met in the application code. However, types O or U may only require a mistake in the Android Manifest file. The dependence among occurrences of PRI types hints that whenever one type of PRI is found in the development history of an app, then also other kinds of PRIs are likely to be present. Indeed, this is not surprising, as we expect developers that are not knowledgeable about or not attentive to permissions to introduce multiple types of PRIs in their apps.

Actionable Insights. The most common types of PRIs occurring in Android apps are of types O and U. This result evidences that, even if these issues and their consequences are well known and have been studied in-depth by the academic community [35, 132, 137], they are still a common occurrence, even in apps developed for newer versions of Android. As a consequence, not only we advise developers to pay more attention to these PRI types but we advocate for the adoption of permission analysis tools (such as M-Perm [134] and P-Lint [130]) during app development. Investigating how to encourage adoption of such tools and assessing possible benefits that developers could reap from them is an open area of research.

6.3.2 RQ2 – How long do permission-related issues tend to remain in Android apps across their lifetime?

Results. Descriptive statistics of decay time for each type of PRI are summarized in Table 6.4. We can observe that for all PRI types the minimum decay time is equal to 1 day, while the maximum is close to 7, 3, 2 and 1.5 years for issues of type *O*, *U*, *MC*, and *MRP*, respectively. Median decay is quite similar for *O* and *U* issues, with a value of approximately one week, but significantly differs for issues *MC* and *MRP*, with a value of about 12 weeks for the former and 1 day for the latter. As expected, results

of the application of the Friedman omnibus test ($p - value < 0.01$) allows us to reject the *null* hypothesis that the medians of decay times across the four types of PRIs do not significantly differ. Post-hoc analysis, performed via the Conover's test, reveals that the distribution of decay times for each PRI type is significantly different from the others, with the sole exception of the *U-MC* pair, for which we cannot reject the null hypothesis.

TABLE 6.4: Descriptive statistics for decay time of PRIs
(SD = standard deviation, IQR = inter-quartile range)

PRI	Min.	Max.	Median	Mean	SD	IQR
O	1	2,784	6	187.6	419.11	105
U	1	1,066	5	45.25	102.35	28
MC	1	760	82.5	166.8	200.96	303.75
MRP	1	544	1	43.82	124.29	1.5
Aggr.	1	2,784	6	150.3	360.51	84

Mean of decay time is much higher than the median for all PRI types. This suggests that the average is greatly influenced by a subset of the data on the higher part of the scale. This observation is even more notable for *O* and *MC* PRIs, that exhibit a comparatively much higher mean (187.6 and 166.8 days, against 45.25 and 43.82 days for issues *U* and *MRP*) and standard deviation (419.11 and 200.96 days, opposed to 102.35 and 124.29 days). We additionally observe that *MC* issues exhibit a relatively higher inter-quartile range (303.75 days), implying that decay time for *MC* issues is much more dispersed than for other PRIs. Results of applications of Cliff's *d* for effect size estimations reveal a small effect between *O* and *MC* and negligible for all other pairs.

Main outcomes. Results indicate that *the majority of PRIs are fixed in a timespan of a few weeks after their introduction.* Nonetheless, in many cases *PRIs can linger inside an app for an extended period of time*, that can be as high as several years, before being fixed.

Discussion. The PRIs considered in this study can impact the end-users opinion of the app [39, 43], and can result in security problems [35]. Therefore, understanding characteristics and reasons for the persistence of these longer-living PRIs represents a relevant research question that demands further investigation.

Of particular interest are the higher median values of issues *MC*. As previously mentioned in RQ1, specific conditions must be met inside an app's source code to introduce one of these issues, meaning that the issue cannot solely exist in the AndroidManifest file. We speculate that this greater specificity of necessary conditions is also the reason behind the greater median decay time, *i.e.,* once introduced, more non-trivial changes in the

source code must be carried out to fix such issues. In other words, *MC issues are harder to introduce but also harder to fix once introduced.*

Actionable Insights. Given the fact that PRIs of all kinds can linger inside an app for an extended period of time we encourage developers and organizations to pay increased attention to code that has been written during early project life, during quality assurance activities (*i.e.,* code review sessions). Moreover, since *MC* issues tend to persist a long time once introduced, extra attention should be paid by developers and organizations to both ensure that they are not introduced, but to also regularly check their apps for these types of issues. Further work is needed to understand precisely why *MC*s tend to last longer compared with other PRIs.

6.3.3 RQ3 – To what extent does developer experience correlate with the introduction of permission-related issues?

Results. Figure 6.2 and Table 6.5 report the distributions and the descriptive statistics of developers' experience when introducing each type of PRI, respectively. We observe that PRIs are introduced by developers with varying levels of experience within the project, with extreme cases ranging from 1 day to 3,309 days of experience (˜9 years). Moreover, the median experience of developers introducing PRIs is 314.5 days (almost 1 year) and ranges from 244 days (˜8 months) when introducing *MRP* issues to 843 days (˜2 years and 4 months) when introducing *U* issues.

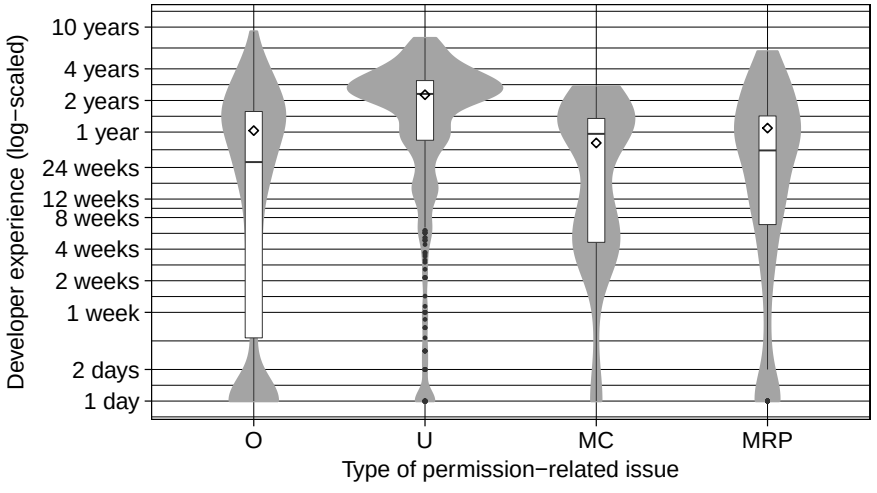


FIGURE 6.2: Distributions of developers' experience when introducing PRIs

The Friedman's omnibus test reveals that the medians of the experience of developers when introducing a PRI are different among PRI types ($p-value < 0.01$). An additional pairwise comparison between all types of PRI, via the Conover's test, reveals that the

distribution in terms of developers' experience significantly differs for all pairs. Estimations of effect size reveal a small magnitude for pairs *O-MC* and *O-MRP* and negligible for all other pairs.

TABLE 6.5: Descriptive statistics for developers' experience (in days) when introducing PRIs
(SD = standard deviation, IQR = inter-quartile range)

PRI	Min.	Max.	Median	Mean	SD	IQR
O	1	3,309	376.6	189.0	478.7	569.0
U	1	2,865	843.0	843.0	636.0	824.0
MC	1	991	351.0	287.5	253.2	459.5
MRP	1	2,140	244.0	398.3	473.3	472.0
Aggr.	1	3,309	314.5	487.9	553.9	756.0

Main outcomes. Developers with more than 2 years of experience tend to introduce under-permission issues in Android apps. Developers' experience at PRI introduction time is quite homogeneous across all the other types of PRIs. We do not have an evidence-based explanation about why developers with more than 2 years of experience are injecting more under-permission issues. However, we can conjecture that this phenomenon is happening because more experienced developers are the ones who are in charge of performing changes in the Android manifest file, which is one of the most important components in an Android app and frequently contains permission-related information. We suggest organizations and project maintainers to be cognizant of under-permission issues, even when experienced developers are involved. As future work, we will perform a more in-depth analysis of this phenomenon.

Interestingly, IQRs are relatively high (between ~459.5 and ~824.0 days). High IQRs may be an indication that *both experienced and inexperienced developers actually risk to introduce PRIs when working on their Android apps.* We suggest organizations and project maintainers to take special care of PRIs during the first days of activity of project newcomers, *e.g.,* by planning dedicated code review sessions.

The number of PRI introducing commits across different types of PRIs tend to increase for developers with more than 1 year of experience. However, the same tendency can be observed when looking at *all* commits within our dataset ($median = 314.5$, $mean = 487.9$, $IQR = 756$ days). Therefore we can conjecture that this phenomenon merely depends on the fact that in our dataset developers with more than 1 year of experience tended to commit more in their GitHub repositories.

Actionable insights. Since developers with more than two years experience are more likely to introduce issues, then this demonstrates that even experienced developers need

to be cognizant of PRIs. This fact strengthens the case for the adoption of permission analysis tools during app development, as already discussed in Section 6.3.1. In addition, we suggest organizations and project maintainers to be cognizant of under-permission issues during activities that might require changes to app permissions, even when experienced developers are involved.

6.3.4 RQ4 – To what extent does developer experience correlate with fixes of permission-related issues?

Results. We answer this research question by following the same procedure of RQ3; the only differences are that (i) now we are focusing on the commits in which PRIs have been fixed (as opposed to when they are firstly introduced) and (ii) we are considering exclusively the PRIs which have been fixed along the lifetime of the app, so that their PRI fixings commit is meaningful. Figure 6.3 and Table 6.6 report the distributions and the descriptive statistics of developers' experience when fixing each type of PRI, respectively.

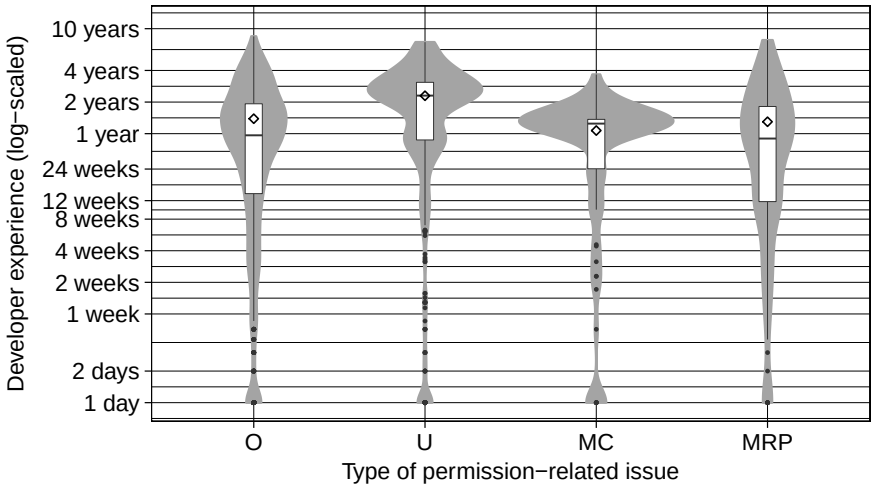


FIGURE 6.3: Distributions of developers' experience when fixing PRIs

The data exhibits a high skewness towards developers with more than 1 year of experience in the project. The Friedman's omnibus test results in a $p - value < 0.01$. Similarly to what happened also for RQ3, the post-hoc analysis with the Conover's test reveals that the distribution in terms of developers' experience significantly differs for all the pairs involving PRIs of type *U*. Applications of Cliff's d reveal an effect size of small magnitude for all pairwise comparisons involving *O*, while it is negligible for all other pairs.

Main outcomes. The distribution of developers' experience when fixing PRIs tends to follow the same trends as the ones related to the introduction of PRIs (see Section 6.3.3), but with one main difference: *less experienced developers tend to fix fewer PRIs.* We can

TABLE 6.6: Descriptive statistics for developers' experience (in days) when fixing PRIs (SD = standard deviation, IQR = inter-quartile range)

PRI	Min.	Max.	Median	Mean	SD	IQR
O	1	3,086	352.0	506.7	522.9	605.0
U	1	2,717	841.0	838.2	625.7	807.5
MC	1	1,337	455.0	390.9	249.2	330.5
MRP	1	2,838	328.0	473.9	546.6	581.0
Aggr.	1	3,086	420.0	578.8	563.3	732.0

expect this observation since we can speculate that PRI are non-trivial issues and are managed (and fixed) by developers who are more familiar with the internals of the app being developed. The obtained results confirm the reasonable intuition that, being PRIs non-trivial issues in an Android app, they tend to be fixed by developers who are not newcomers in the project. At the time of writing we do not have an objective explanation for explaining why issues of type *U* are fixed by developers with longer experience in the project, this is left for a future in-depth analysis.

Actionable insights. The distributions of developers? experience when fixing PRIs tend to follow the same trends as the ones related to the introduction of PRIs (see Section 6.3.3), but with one main difference: less experienced developers tend to fix fewer PRIs. We can expect this observation since we can speculate that PRI are non-trivial issues and are managed (and fixed) by developers who are more familiar with the internals of the app being developed. The obtained results confirm the reasonable intuition that, being PRIs non-trivial issues in an Android app, they tend to be fixed by developers who are not newcomers in the project. At the time of writing we do not have an objective explanation for explaining why issues of type *U* are fixed by developers with longer experience in the project, this is left for a future in-depth analysis.

6.4 Threats to validity

Although our research led to several interesting results, there are several threats to validity.

Internal Validity. We heavily relied upon M-Perm and P-Lint, and while these tools have been published in peer-reviewed venues, they are both still reasonably new. Like with all static analysis tools, they are not perfect, and tool imperfections have the capability to skew research results. While we analyzed a large number of open-source Android applications, we only examined a small subset of the millions of available Android apps. However, we feel that our analysis provides a reasonable representative subset of apps.

We examined 'commit ownership' and not 'code ownership' in our study. While 'code ownership' is a general term used to describe whether one person is primarily responsible for a software component [156], commit ownership is merely the author who made the commit to the repository. Due to our empirical examination of existing repositories, it would have been impossible to examine code ownership in our study. Since we only knew the committing author, we were unable to account for other developers who may have contributed to the commit, for example in the case of pair programming. We, therefore, considered 'code ownership' out of scope for this study and focused on 'commit ownership'. However, future work could also include code ownership to provide a possible alternative view on the results.

We utilized Git user names to identify developers. An inherit limitation of using this process is that developers could use different user names throughout the project, and the researcher would only be able to assume that these are two different developers. An additional limitation of many empirical studies is if developers are following a pair programming process, then the committer of the code will be assumed to be the sole developer. The study would not be able to account for the efforts of the non-committing developer.

During the analysis of permission smells in the commits, we always assume that the same smell occurs in the methods with the same name. If the developer changes the name of the method (or the file name), then we treat it as two different permission-smell occurrences. Additionally, we only consider the authors of the commits in the master branch. This may be a threat in the event a project fully adheres to the open-source development model. In this scenario, external contributors fork the repository, conduct their changes, and perform pull requests. In our analysis, we do not see their commits directly, but we just consider the commit in which the pull request is merged.

In some cases, due to licensing reasons, open-source app repositories might not contain parts of the app code that is added at a later stage, before publication in app stores (*e.g.,* ad libraries). In these cases, the app manifest file might include some permissions currently not used in its code but added in anticipation of additions. Our analysis of over-permissions might have been influenced by these instances.

For our study, we empirically analyzed the version control repositories of open-source apps. Other permission analysis tools such as PScout [132] could also have been included to examine apps that rely on the install-time permission system, in use until Android API versions 5.1. In our study, we did not include other tools as we focused on the current Android permission model and for consistency reasons. M-Perm uses a call graph to determine the reachability of the app's source code. However, during our analysis we did not evaluate M-Perm's ability to reach dynamically loaded code. We may, therefore,

consider this a potential limitation to our study. Our work is empirical in nature, enabling us to analyze a large number of apps. Future work could conduct a laboratory study and include developer interviews to further understand developer permissions-decisions and mistakes.

6.5 Conclusion and future work

In this chapter we reported on the design, execution and results of an empirical study aimed at further investigating the second of the research questions addressed by this dissertation (discussed in Chapter 1.3). That is:

RQ2 - *Are there any existing issues in current Android security- and privacy-preserving mechanisms that negatively affect the users? trust on the whole platform?*

For this purpose we conducted an empirical study on permission-related issues in Android apps. By mining a set of 574 GitHub repositories containing Android apps, we detected four types of permission-related issues by executing two static analysis tools (M-Perm and P-Lint) on a commit-by-commit fashion. The execution of the analysis tools required the development of custom software (oSARA), which is available in the replication package of the study [135].

The results of the study provide evidence-based insights for better understanding and managing permission-related issues in Android apps. Specifically: (i) **permission-related issues are a frequent phenomenon in Android apps**, with a strong prevalence of over-and under- permissions; (ii) the majority of permission-related issues are fixed in a timespan of a few weeks, even though in many cases some issues can plague the app for an extended period of time (*i.e.,* years) before being fixed; (iii) under-permission issues are significantly more prone to be introduced and fixed by experienced developers; (iv) less experienced developers tend to fix fewer PRIs, possibly indicating that permission-related issues are non-trivial and are managed (and fixed) by developers who are more familiar with the internals of the app being developed.

Future work includes the investigation on whether PRIs accumulate-diminish over the lifetime of an Android app, potentially revealing interesting patterns about their evolution. We are also planning to perform a more in-depth study in order to understand what developers do when they introduce or fix PRIs. Our work provides a direct benefit to both developers and researchers to better understand permission-related issues. For researchers, this study creates the foundation for future work in the area of permissions-related issues. For developers, this provides insight on how teams can better plan their development activities.

Chapter 7

Enhancing trustability of Android applications via user-centric flexible permissions

In this chapter we describe the design, development and evaluation of Android Flexible Permissions (AFP), a *user-centric approach to flexible permissions management* aimed at empowering end users to play an active role with respect to Android permissions. AFP embraces the European vision of next generation internet, more human-centric and concerned with privacy protection by giving control back to users [157]. End users are allowed to specify and customize fine-grained permission levels on private or sensitive resources, according to their own subjective privacy and security concerns. AFP leverages a novel permission model through which fine-grained app permissions are specified on a per-feature basis. Differently from the current Android permission model, AFP empowers end users to selectively grant finer-grained permissions by specifying (i) the desired *permission levels* (*e.g.,* access to the contacts list can be granted to all contacts that do not belong to specific circles of people like relatives or close friends), and (ii) the *features* of the app in which the specified permission levels are granted (*e.g.,* access to the relatives circle in the contacts list can be granted only during a video call in a messaging app). AFP offers a dedicated external mobile app for managing flexible permissions.

From the developer's point of view, AFP enables apps to *dynamically adapt to user-defined permission levels* with very limited additional effort. Developers can work on their mobile apps as usual, without using any additional library or tool. In order to comply with AFP, a developer is provided with automatic support by the AFP Web application that (by means of a guided workflow) allows to (*i*) define the features offered by the mobile app; (*ii*) map each feature to the components that implements it, *i.e.,* Android activities, services, broadcast receivers, or content providers. Given

the feature-component mappings, AFP leverages static control-flow analysis for *automatically* retrofitting the app so that it is able to *dynamically* handle fine-grained and feature-based permission levels at runtime.

We evaluated AFP by designing, conducting, and reporting four independent experiments aimed at empirically investigating on key aspects of AFP. Specifically, we assessed the performance of the AFP instrumenter via 1,277 real-world apps, the performance at runtime of 7 AFP-enabled real-world apps, the usability and acceptance of AFP for both end users and developers (involving 47 and 11 subjects, respectively).

7.1 Design philosophy

Design of our proposed approach has been done with the main principles of the Next Generation Internet initiative in mind, according to whom the end user should have the power to decide how and by whom her data are used [8, 9]. In addition, each of the studies presented in previous chapters provides valuable insights that have an influence on the design. In particular:

- Results of the study of Chapter 4 evidence that **there is lack of users-first privacy approaches**. Indeed, as discussed in Chapter 1, privacy is a subjective property, as different users may have different requirements to consider an application trustable. This fact guides the design of our approach towards a customizable solution, to better fit to individual privacy and security preferences without considering all users as equals. In addition, as the **execution times of current static analysis approaches for mobile apps are relatively low**, strengthened the choice of adopting static analysis for our approach (specifically, for the instrumentation of Android apps as will be shown in Section 7.2).

- One of the major takeaways of the study of Chapter 5 is that in current solutions **users do not comprehend why they are being asked for a permission**. As such, the design of the proposed approach has to be carried out with the goal of making explicit the link between permissions and app functionalities.

- The study of Chapter 6 evidences that **permisson-related mistakes by developers are a frequent occurrence** in Android apps, with **over-permissioning being the most frequent one**, even in the case of experienced developers. Our solution takes this fact into account and addresses this issue by enabling end users with the ability of specifying fine-grained permission levels on private or sensitive resources.

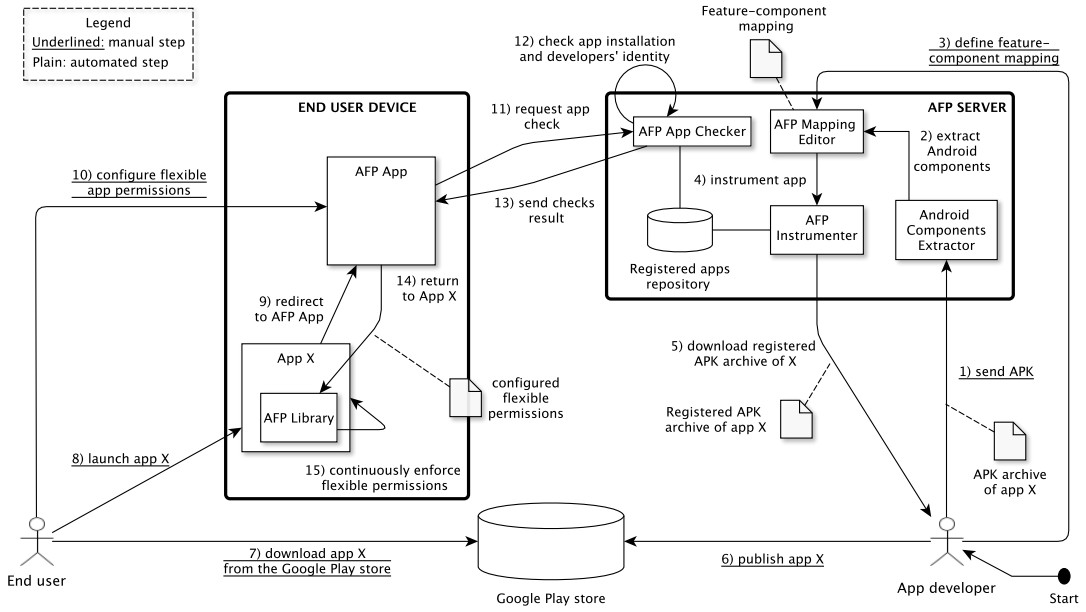


FIGURE 7.1: Overview of the AFP approach
(underlined labels represent manual steps, whereas all the others are fully automatic)

7.2 The AFP approach

Android Flexible Permissions (AFP) grants permissions on a per-feature basis by (*i*) keeping track of user security and privacy preferences, and (*ii*) automatically enacting and enforcing them at runtime. AFP is composed of the following main components:

- *AFP App*, an app from which users can manage their own flexible permissions;
- *AFP Library*, a library to enforce permissions at runtime;
- *AFP Server*, a web app that allows developers to automatically retrofit an existing app so as to comply with AFP. It also offers mechanisms for signing and verifying AFP-enabled apps.

With reference to Figure 7.1, in the following we will describe the workflow of AFP. We divide its explanation into two subsections. From the developer point of view, Section 7.2.1 describes the steps to be followed by developers before publication, in order to make their apps compatible with AFP. From the end-user point of view, Section 7.2.2 describes the steps to be followed upon the first app execution, in order to specify the desired permissions.

7.2.1 App developer perspective

Our design goal for the developer workflow is to provide them with an automatic approach, as to minimize the effort needed to create apps compatible with the flexible permission system.

Developers create their mobile apps as usual, without using any additional library or tool. When an app X is ready to be published (right-hand side of Figure 7.1), the developer can send the APK archive of X to the *AFP Server* so to enable AFP (1). As detailed in Section 7.5, the *Android Components Extractor* extracts all the Android components of X, *i.e.,* its constituent activities, services, broadcast receivers, and content providers (2). Then, the developer uses a *web-based editor* for (i) defining the features of X in terms of their name and description (later used by end users), and (ii) mapping each one of them to (a subset of) the extracted Android components implementing it (3). Step 3 is the only additional effort we request to developers, and it is greatly facilitated by the web-based editor together with the automatic extraction of Android components. The output of this phase is the *feature-component mapping model*, specifying the mapping between app features and Android components.

The *AFPInstrumenter* statically analyzes and automatically retrofits app X to enable AFP on it (4). The instrumenter performs the following operations (that are totally transparent to the developer): (i) automatically includes our *AFP Library* in the app; (ii) instruments X so that all calls to sensitive Android APIs are proxified and redirected to the *AFP Library*; (iii) injects the code in the main activity of X for allowing the end user to switch to the AFP App when launching X for the first time; (iv) assigns a unique secret key to the app X, which will be used at runtime by the *AFP App Checker* (12); (v) creates a new record into the repository of registered apps; (vi) rebuilds and re-signs X as a new APK archive. Finally, the instrumented APK of X is made available to the developer (5), who can then proceed with the publication of the APK in the Google Play Store (6).

7.2.2 End user perspective

The user workflow is designed to minimize the end user effort to specify flexible permissions.

Users can download and install (7) apps that adopt the AFP system directly from the Google Play Store since no modifications to the Android OS are required. Upon the first launch of the newly installed app (8) they are redirected to the *AFP App* (9), which in turn invites them to configure the flexible permissions. Should the user be unwilling to

do so, she can immediately abort the process, and the app usage will continue with the standard permission system provided by the Android platform.

Once inside the *AFP App*, for each feature of the app the user can specify her own permission preferences (10). This aspect of AFP allows to address a well-known problem in the current Android permission model, *i.e.,* the fact that users tend to not understand why they are being asked for certain permissions, often complaining about this aspect in their reviews in the Google Play store (as seen in Section 5). Indeed, by explicitly asking end users to define the permission levels on a per-feature basis allows users to (i) read the description of each feature provided by the developer and (ii) better understand why certain permissions are being requested in the context of each specific feature of the app, rather than within the app as a whole. By using AFP, end users have a more transparent view of the features provided by their apps, and a better knowledge about the context in which (sensitive) permissions are requested by the app.

While the end user is setting her desired permissions, in the background, the *AFP App* also interacts with the *AFP Server* (11). The server uses an internally generated secret key (4) to check the app installation and verify the developer's identity, hence certifying that nobody tampered with the *AFP Library*. Moreover, it verifies that the APK downloaded from the Google Play Store is exactly the one produced by our approach (12). When the results of the checks are ready (13), and the configuration phase finishes, the user will be redirected to the newly installed app, together with the configured flexible permissions configuration (14). The permissions configuration is then associated with the AFP-enabled app and the user can continue with app usage, in a completely transparent way, *i.e.,* no further user interaction or dialogs are required.

The access to private or sensitive resources will be granted by the *AFP Library* according to the specified permissions configuration (15). The *AFP Library* proxifies each call of the app to sensitive Android APIs (*e.g.,* call to the Android geolocation manager), hence wrapping the access to sensitive resources.

The *AFP App* allows to specify default levels for the permissions (*e.g.,* geolocation is allowed only at the city-level, independently of the app requesting it), that will be used as a basis during the configuration of the flexible permissions for any newly installed AFP-compliant app. This characteristic permits to speed up the configuration of the permissions for each newly installed AFP-enabled app.

7.3 Flexible permission data model

This section presents the AFP data model to which permission configurations conform (Figure 7.2). It is based on the following core concepts:

- *Resource* represents a sensitive part of the Android platform whose access can be controlled by the AFP library. In AFP, resources are both physical parts of the device, such as the device camera and microphone, and logical ones, such as the user's contacts book.
- *Feature* represents a user-level functionality of the app. Every *Feature uses* one or more *Resources*. In addition, every *Feature* is directly connected through the *realizedBy* relation with one or more *AndroidComponent* instances, each of them representing one or more source code files inside the app.
- A *PolicyItem* regulates access to the *Resources* used by the *Feature*. It represents a single access restriction rule that can be imposed upon one or more *Resources*. For instance, a *PolicyItem* could specify that the access to the device camera has to be forbidden, or that only the user's city should be shared when the device is queried for the user position.
- *AccessPolicy* is a conjunction of one or more *PolicyItems* and it is linked to a *Feature*.

The remaining classes in the data model define the restrictions that can be enforced:

- *BooleanPolicyItem* permits either full access to the resource or no access at all.
- *RestrictedPolicyItem* restricts the access to a *restrictionSet*, whose type can be either *BLACKLIST* or *WHITELIST*. For instance, it can be used to restrict access to the contacts list to only contacts that do not belong to specific circles of people like relatives or close friends.
- *ReadWritePolicyItem* and *LocationPolicyItem* extend the *LayeredPolicyItem* class and allow for granting access to a *Resource* at incremental levels. Higher levels are used for less restrained access to the *Resource*. *ReadWritePolicyItem* has four possible levels. In particular, *ADD_NEW_ACCESS* and *MODIFY_EXISTING_ACCESS* allow for adding new records to the *Resource* and editing existing ones, respectively. For example, it is possible to grant access to the sms messages but prevent creating and sending new ones. *LocationPolicyItem* instead provides four levels of different precisions for access to the user position.

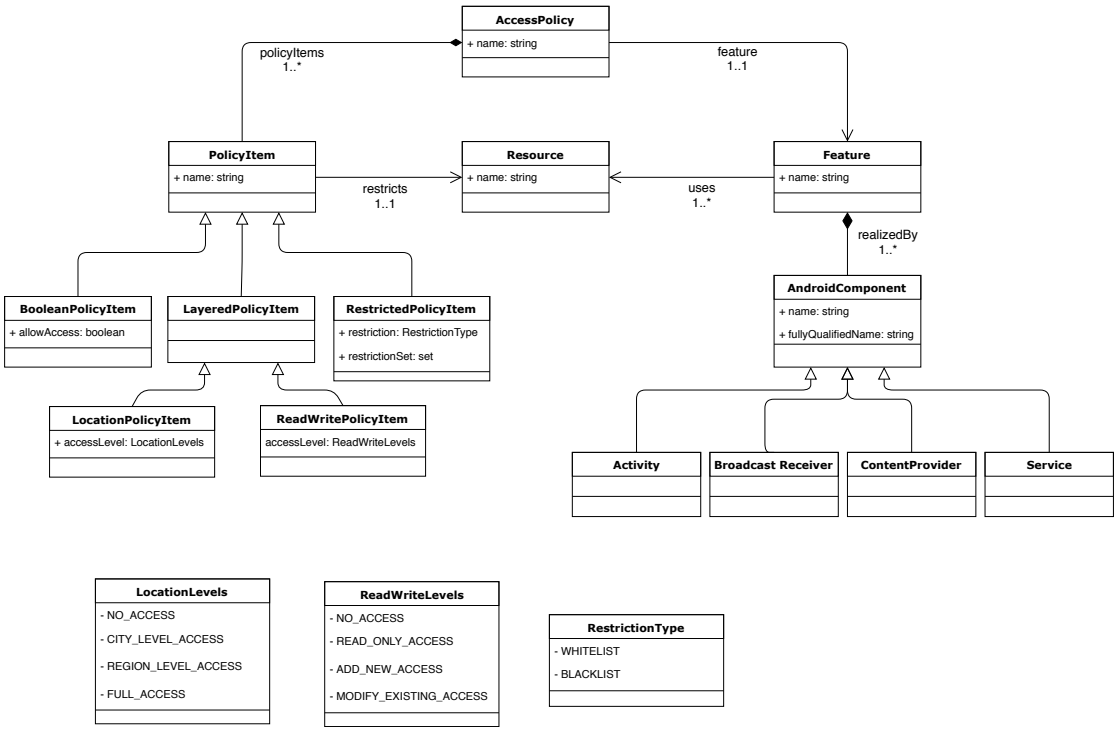


FIGURE 7.2: Flexible permissions data model

7.4 Features specification

The AFP approach involves (i) the automatic extraction of Android components composing the mobile app (step 2 in Figure 7.1) and (ii) the definition of a mapping between features and the Android components implementing them in the app (step 3).

For what concerns step 2, all relevant information is extracted from both the XML file of the Android manifest and the bytecode of the Java classes of the app. The output of this step is a fragment of configuration conforming to the flexible permission data model described in Section 7.3 containing only instances of the `AndroidComponent` class and its subclasses.

In order to specify the mappings (step 3), the developer uses a form (Figure 7.3) where she can declare the main features of the app and, by means of check boxes, associate them to the automatically extracted Android components. The final result of this step is the complete configuration, which also includes the required instances of the `Feature` class.

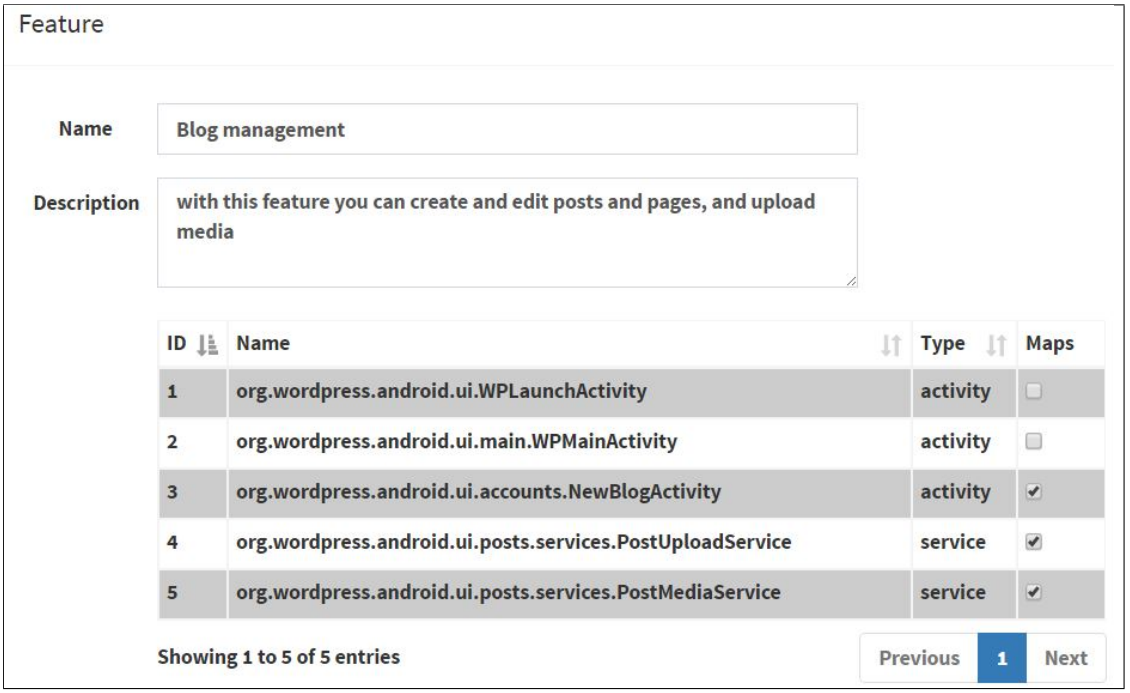


FIGURE 7.3: Feature to Android components mapping specification form

7.5 App instrumentation

The AFP approach involves also the automatic instrumentation of the app (step 4 in Figure 7.1). Such step is carried on by the *AFP Instrumenter* that performs a set of operations that can be grouped into three main phases: (*i*) decomposing the input APK, (*ii*) rewriting the app bytecode, and (*iii*) repackaging the rewritten app.

The goal of the first phase is to extract, from the compiled binary files of the app under analysis, the app Java bytecode. Although the bytecode is a low-level representation, it is suitable to perform analysis and instrumentation. In turn, the third phase performs the reverse operation, transforming the instrumented bytecode back into a compiled binary file. The logic behind the first and the third phases is straightforward, and they are both carried out using freely available tools arranged in a pipe-and-filter pattern. The adopted tools are described in Section 7.7.

During the second phase, rewriting of the app bytecode is performed by an ad-hoc analyzer, following Algorithm 1. Given as input an Android app and its feature-component mappings, the algorithm returns an AFP-compliant version of the app. In order to do so, the algorithm iterates over all Android components that constitute the app (line 2 in Algorithm 1) and extracts from each of them the set of sensitive API calls performed inside its logic (line 3). Each call is analyzed for the purpose of identifying the affected resource (lines 10-16), and creating and binding instances of the `Resource` class in the AFP data model. Each sensitive call is then replaced with a corresponding call to the

AFP library (line 17), which contains a proxy class for each of the Android APIs that enable access to restricted parts of the platform (further details in Section 7.6). While performing the rewriting, the algorithm also checks that all classes containing calls to restricted parts of the Android API belong to a feature (lines 4-6). This ensures that the developer does not leave some Android components out of the mapping. Additional code (via Android *Intents*) is added in the main activity (*i.e.,* the app entry point) to set up communication between the app under rewriting and the *AFP App*. It enables the configuration of the flexible permissions on the startup of the app (lines 7-9).

```
input  : App, an Android app
         C, set of Android components ∈ App
         F, set of features ∈ App
         M, mapping of elements of F to C
output: App′, AFP-compliant version of App

1  begin
2      foreach c ∈ C do
3          scan C to extract SC, set of sensitive calls ∈ C
4          if SC ≠ ∅ and c ∉ M then
               /* found sensitive call not mapped to any feature, raise error
                  and terminate */
5              break
6          end
7          if c is App main activity then
8              add intent trigger towards AFPapp to allow for permissions configuration
9          end
10         foreach sc ∈ SC do
11             identify r, resource affected by sc
12             foreach f ∈ F do
13                 if c is mapped to f then
14                     add r to f_r, set of resources used by f
15                 end
16             end
17             replace sc with sc′, call to AFPlibrary wrapper
18         end
19     end
20 end
```

Algorithm 1: Rewriting algorithm of AFPinstrumenter

The illustrative example, given in Figure 7.4, shows bytecode rewriting of an API call for reading the last registered user's location. In the original version (Listing A), a `LocationManager` object is loaded from a local variable and pushed onto the stack (instruction 1). Then, the constant string "network", used as a parameter in the upcoming method invocation, is also pushed onto the stack (instruction 2). Both are consumed from the stack with an invocation to the virtual method `getLastKnownLocation` (instruction 9). Finally, the `Location` object resulting from the method call is read from

the stack and stored in a static field (instruction 10). In the rewritten version (Listing B), the invocation to the virtual method `getLastKnowLocation` has been replaced with a static call to its proxified method in the *AFPLibrary* (instruction 9). Since this method also requires, as an array parameter, the identifiers of features to which it belongs, this information is also pushed onto the stack (instructions 4-8).

```
1   aload_0
2   ldc network
3
4
5
6
7
8
9   invokevirtual android/location/
     LocationManager.getLastKnownLocation
     :(Ljava/lang/String;)Landroid/
     location/Location;
10  putstatic location:Landroid/location/
     Location;
11
```

Listing A

```
1   aload_0
2   ldc network
3   iconst_1
4   anewarray java/lang/String
5   dup
6   iconst_0
7   ldc WeatherForecast
8   aastore
9   invokestatic gssi/aq/it/afplibrary/
     AFPLocationManager.
     getLastKnownLocation:(Landroid/
     location/LocationManager;Ljava/lang/
     String;[Ljava/lang/String;)Landroid/
     location/Location;
10  putstatic location:Landroid/location/
     Location;
11
```

Listing B

FIGURE 7.4: Comparison between an original byte code file (Listing A) and the rewritten version produced by the *AFPInstrumenter* (Listing B).

7.6 Permissions enactment and enforcement

As described in Section 7.2 (see Figure 7.1), a user can download and install AFP-enabled apps from the Google Play Store, as she would normally do for all apps. Upon starting a newly installed app for the first time, she is redirected to the *AFPApp*, which enables permission enactment by allowing her to configure the flexible permissions associated to it. Two screenshots of the app are presented in Figure 7.5. During the configuration, she is presented the list of features offered by the AFP-enabled app. For each feature, the accessed sensitive resources are listed and, for each one of them, she can set her preferences, hence regulating the access to the resources for that single feature. As an example, consider a user interested in the Facebook app. After downloading and installing it on her device, on the first run she is presented with the list of app features, *i.e.*,, *Wall*, *Messaging*, *Events*, etc. Assuming that she does not want her friends to know her exact location every time she posts on her Wall, she can restrict the precision of the Location resource for the Wall feature, while leaving it unchanged for Events in order to still discover ongoing events nearby.

While the configuration is ongoing, the *AFP App* establishes communication with the *AFP Server* and checks the validity of the app secret key, promptly raising a warning should it be different from the one stored on the server.

The configuration procedure can be terminated at any time and, upon termination, the configured permissions configuration is transmitted back to the calling app, and stored by the *AFP Library*.

Permission enforcement is in the hands of the *AFP Library*, which contains a proxy class for each of the Android APIs that enable access to restricted parts of the platform. The methods contained in the proxy classes perform a check against the configured permission model and allow access to the restricted parts of the platform only if admitted by the model. If access to a resource has to be allowed only at certain level, then partial adjustments on returned data are performed.

7.7 Implementation and used technologies

AFP makes use of a number of different techniques and technologies. Static analysis techniques are used to verify APK packages uploaded by developers on the AFP Server. Specifically, static analysis is utilized to verify that developers accessed sensitive resources only through the methods provided by the AFP Library (otherwise we cannot guarantee that the preferences set by users in permissions configurations will be fulfilled). In order to do so, AFP utilizes an intra-procedure analysis to detect Android API invocations within each method of the app's code. The complete list of sensitive Android API calls is obtained with SuSi [158], a machine learning based tool provided by Artz et al. For

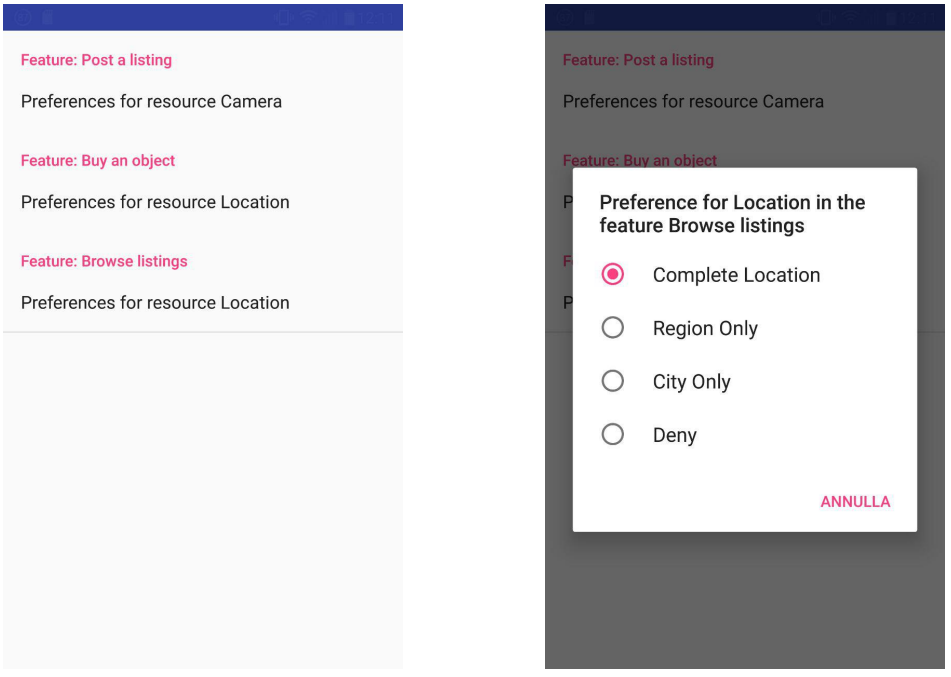


FIGURE 7.5: Screenshots of the permissions configuration procedure enabled by the AFPApp

the implementation of this analysis we rely on the static analysis framework Soot [81], coupled with Dexpler [159] to disassemble APK packages and transform Android's Dalvik bytecode into a format suitable for analysis.

In order to avoid the possibility that malicious developers could tamper with the AFP Library implementation, thus circumventing the need of obtaining authorization against the permissions configuration, integrity of the aforementioned library is also checked by the AFP Server through a checksum-based integrity verification mechanism.

Communication between the AFP App and any AFP-compliant app is enabled by Android's *Intents* [160]. An *Intent* is a special kind of object used to enable inter-app communication. In AFP, *explicit Intents* are leveraged to both redirect the end user to the AFP App and return the configured fine-grained permissions configuration once it has been personalized.

Changes to the permission system introduce the risk of app instability, as apps may not expect to have their permission requests denied [62]. When denying permissions leads to crashes, users are likely to become more permissive to improve app stability, thus counteracting the whole reasoning behind feature- and level-based permissions. With this concern in mind, in our implementation, we make use of "mocking"[1]: in the event of a denied permission our system supplies apps with well-formed but non-sensitive data. For example, if the end user allows only city-level geolocation, when the app calls the Android location manager, the AFP Library intercepts that call and returns the geographical center of the city where the user is, instead of her precise location. This enables apps to continue functioning usefully unless access to the protected resource is critical for its correct behavior.

To perform the automatic extraction of the Android components composing the mobile app (step 2 in Figure 7.1) we firstly decode the input APK using `apktool`[2]. The Android manifest file is then analyzed via a simple XML parser we developed in Java. The analyzer of the Java bytecode is implemented by using the `Apache Commons Byte Code Engineering Library`[3].

The tool that allows developers to specify feature-components mappings (step 3) has been implemented as a web-based tool, built upon the `Flask`[4] web framework.

The AFP Instrumenter is implemented by using several tools. The tool `apktool` is used for decomposing the APK and producing a `Classes.dex` file containing the app bytecode.

[1] This aspect is inspired by the Mockdroid approach by Beresford et al. [56])
[2] `http://ibotpeaches.github.io/Apktool/`
[3] `http://commons.apache.org/proper/commons-bcel/`
[4] `http://flask.pocoo.org`

Then, the `dex2jar`[5] tool is used to obtain a conventional `jar` file that, subsequently unpacked via the `zip` shell command, permits to obtain the `.class` files constituting the app. Rewriting of the `.class` files is done by our Java implementation of Algorithm 1, leveraging the Apache Commons Byte Code Engineering Library. Instrumented `.class` files are then repackaged back to a `.dex` archive via the Android SDK `dx`[6] tool, and the APK archive is reassembled using again `apktool`. At the end, the resulting package is signed using `jarsigner`[7]. The whole end-to-end process is tied together by a Python script.

7.8 Evaluation

In this section, we report the four independent experiments we performed to evaluate the AFP approach. For the purposes the experiments 2, 3 and 4, we focused on the three Android APIs that are among the ones considered the most sensible by end users [161] while at the same time widely used by apps on the Google Play market [162]: `Camera`, `LocationManager`, and `MediaRecorder`. To allow easy replication and verification of the experiments, we provide a complete **replication package**[8] including: the source code of all the components of the AFP approach, the source code of the measuring tools we implemented for carrying on the experiments, the raw data we obtained from the experiments, and all the scripts for analysing the experiments' results.

7.8.1 Experiment 1: performance of the AFP instrumenter

Design – The *goal* of this experiment is to assess the performance of the AFP Instrumenter, the module of the AFP server that performs the app static analysis and instrumentation. We chose the AFP Instrumenter as the *object* of our experiment since (i) it is the most complex component in our AFP Server, and (ii) its malfunctioning or low performance in terms of execution time may negatively impact the adoption of the whole approach by developers, who will not be willing to spend a (relatively) long time for the result of the app instrumentation phase. This experiment is designed as a multi-test within object study [12], because it is conducted on a single *object* (*i.e.,* the current implementation of the AFP Instrumenter) across a set of *subjects* (*i.e.,* the APKs archives). More specifically, we randomly selected 1,277 APK archives from a dataset consisting of 11,917 free apps from the Google Play Store; the dataset was created in

[5] `http://github.com/pxb1988/dex2jar`
[6] `http://wing-linux.sourceforge.net/guide/developing/tools`
[7] `http://docs.oracle.com/javase/tutorial/deployment/jar/signing.html`
[8] https://github.com/gianlucascoccia/androidflexiblepermissions

the context of a previous research in which we mined the top 500 most popular free apps for each category of the Google Play store [163]. We executed the experiment by (i) automatically generating a feature-component mapping containing a feature for each Java class of the app (this can be considered a worst case scenario for our instrumenter), (ii) isolating the AFP Instrumenter component so that it could be programmatically executed in isolation, and (iii) sequentially executing AFP Instrumenter for all the 1,277 APKs. For each execution of the AFP Instrumenter, we measured the time for performing each single step of its internal pipeline (see Section 7.7). Measurements were taken via a Macbook Pro-Retina running Mac OSX 10.11.5 with a 2.6 GHz Intel core i5 processor and 8 Gb of memory.

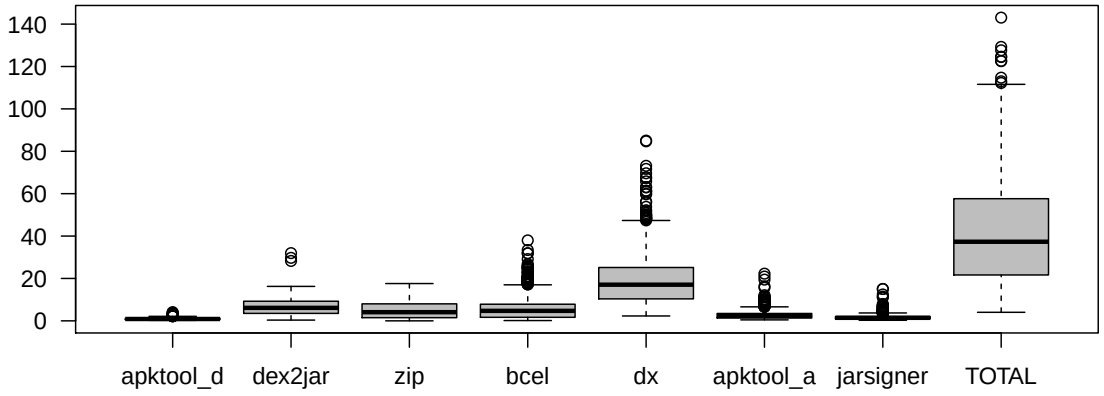


FIGURE 7.6: Execution times of the AFPInstrumenter pipeline (in seconds)

Results – Figure 7.6 shows the execution times of each step of the AFP Instrumenter pipeline. Each step takes an average of less than 10 seconds, with the only exception of the `dx` tool (18.78 seconds in average), mainly because of its heavy I/O operations and performed optimizations[9]. When considering the total execution time of the whole pipeline (see the last box plot in the figure), we can observe that our AFP Instrumenter takes an average of 41.19 seconds to complete, with a minimum of 4.03 seconds and a maximum of 143.07 seconds.

Discussion – We consider the results as satisfactory. On average, developers have to wait less than a minute for obtaining the AFP-enabled app from the AFP Server, and less than 3 minutes in the worst case of our experiment. Since the AFP Instrumenter is executed only once for each app, we consider the waiting time for developers due to app instrumentation reasonable. Hence:

[9] `https://android.googlesource.com/platform/dalvik/+/a9ac3a9d1f8de71bcdc39d1f4827c04a952a0c29/dx/src/com/android/dx/command/dexer/Main.java`

The results of Experiment 1 provide evidence that the performance of the AFP Instrumenter are satisfactory, requiring on average less than a minute.

Threats to validity – A possible threat to validity of our experiment is represented by the selection of only free apps as subjects. Although this choice was driven by budgetary constraints, free apps are representative as they represent 75% of all Google Play Store apps and are downloaded more often [164] than paid apps.

A second threat is represented by the fact that, in this experiment we consider only one kind of *hardware* machine. This choice is mainly guided by budget constraints related to both time and available resources. Notwithstanding, the used hardware is consumer-grade, hence we believe that collected measures are representative of performance that can be obtained on ordinary hardware. Moreover, it is important to note that the AFP Instrumenter is deployed in the AFP Server, whose hardware and software performance can be far higher than the machine we used for this experiment and can be easily scaled up if deployed in a virtualized/containerized environment.

7.8.2 Experiment 2: Performance of AFP-enabled apps

Design – The *goal* of this experiment is to assess the performance of the AFP-enabled apps at run-time (*i.e.,* app X in Figure 7.1). The main rationale behind this experiment is to assess how the application of our AFP approach may actually impact the performance of instrumented apps, thus potentially impacting the overall user experience.

For this experiment, we selected 7 mobile apps from a publicly available dataset composed of 2,443 open-source Android apps that are freely distributed in the Google Play store and whose source code is hosted on GitHub [89]; these two conditions permitted us to have a dataset with apps designed and developed as real projects with real users, and to easily check that instrumented apps behave as the original ones (we did this by a combination of source code inspection and the addition of logging instructions in key parts of the app). Among the 2,443 apps, we randomly selected the 7 apps (see Table 7.1) among those requesting at least one of the permissions considered for our experiments (*i.e.,* geolocation, camera and microphone access).

We executed the experiment by performing the following steps for each app: (i) we defined one or two (depending on the complexity of the app, see Table 7.1) common usage scenarios that start from the main activity and end with the complete stop of the app; (ii) we executed each usage scenario, while measuring the CPU load and memory consumption of the process of the app; (iii) we created feature-component mappings using

TABLE 7.1: Selected apps for study 2

ID	Name	Version	Type	Scenarios
a_1	WordPress	5.6.1	Blog manager	2
a_2	Ottawa Bus Follower	2.0.11	Bus-related utility	2
a_3	Streetlight Seattle Reporter	1.2.0	Citizen participation	2
a_4	Local Weather	1.0.0.7	Weather	1
a_5	Run Helper	1.3	Fitness tracker	1
a_6	Selfie HD	1.1	Photo camera	1
a_7	Flickr Uploader	2.3.2	Photo upload	1

the AFP Web application; (iv) we instrumented the app via the AFP Instrumenter by using the mapping defined in the previous step; (v) we executed and measured again each usage scenario on the instrumented version of the app.

From a tooling perspective, we used (i) `MonkeyRecorder`[10] for recording all the actions we manually performed for each app during a pilot manual execution of all the basic usage scenarios, (ii) a shell script using Android `monkeyrunner`[11] for replaying the previously recorded scenarios, and (iii) another shell script periodically executing the Android `top` or `dumpsys` tools via the Android Debug Bridge (ADB)[12] for collecting the CPU load and memory consumption of the app at runtime. All the shell scripts and tools were executed from the same laptop of Experiment 1, whereas the apps have been executed on an LG G3 855 (quad-core CPU at 2.5 GHz and 3 Gb of memory) running Android 5.0.

Results – Collected measurements are presented in Figure 7.7. For each app, we performed a comparison of both the CPU load and memory consumption for its two versions (*i.e.,* original or instrumented) by using the Mann-Whitney test [12] with $\alpha = 0.05$, one-tailed. In all cases, we obtained a p-value much larger than α, thus allowing us to confirm that there is low difference in the medians of CPU and memory consumption, with and without app instrumentation.

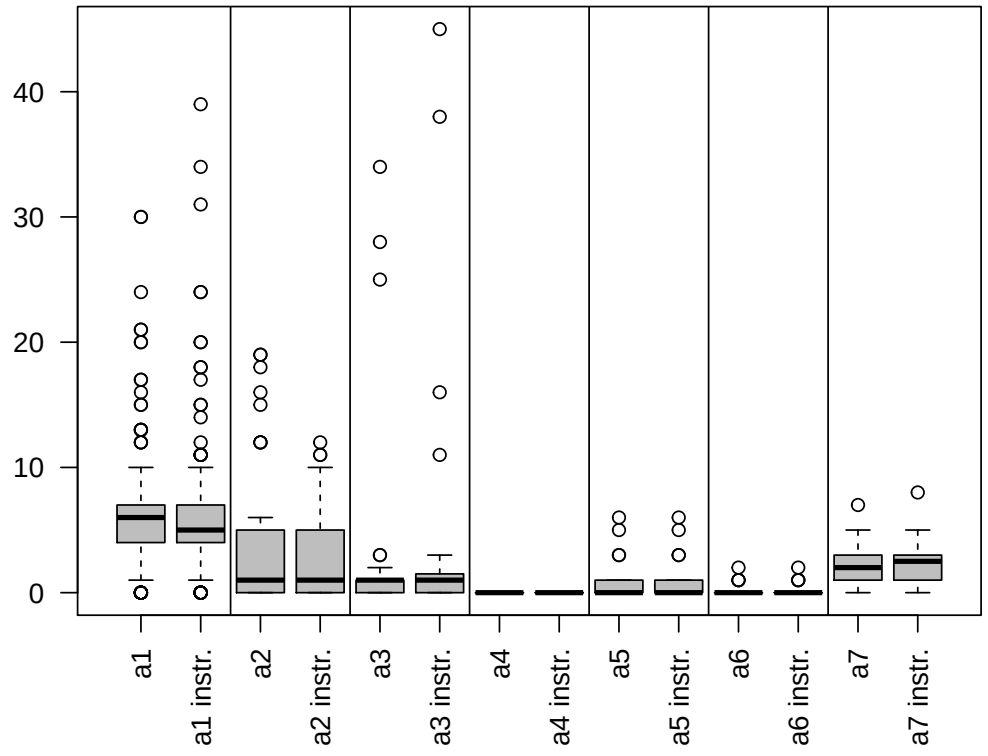
Discussion – From Figure 7.7 it is evident that both CPU load and memory consumption of the original and instrumented versions of each app are comparable, as confirmed by statistical tests. The results of this experiment give a positive indication about the performance of AFP-based apps.

It is important to note that the focus is not on the formal systematic assessment of the *precision* of the app instrumentation (*i.e.,* we do not have a formal proof that instrumented apps do not crash in some corner cases); nevertheless, we performed a manual

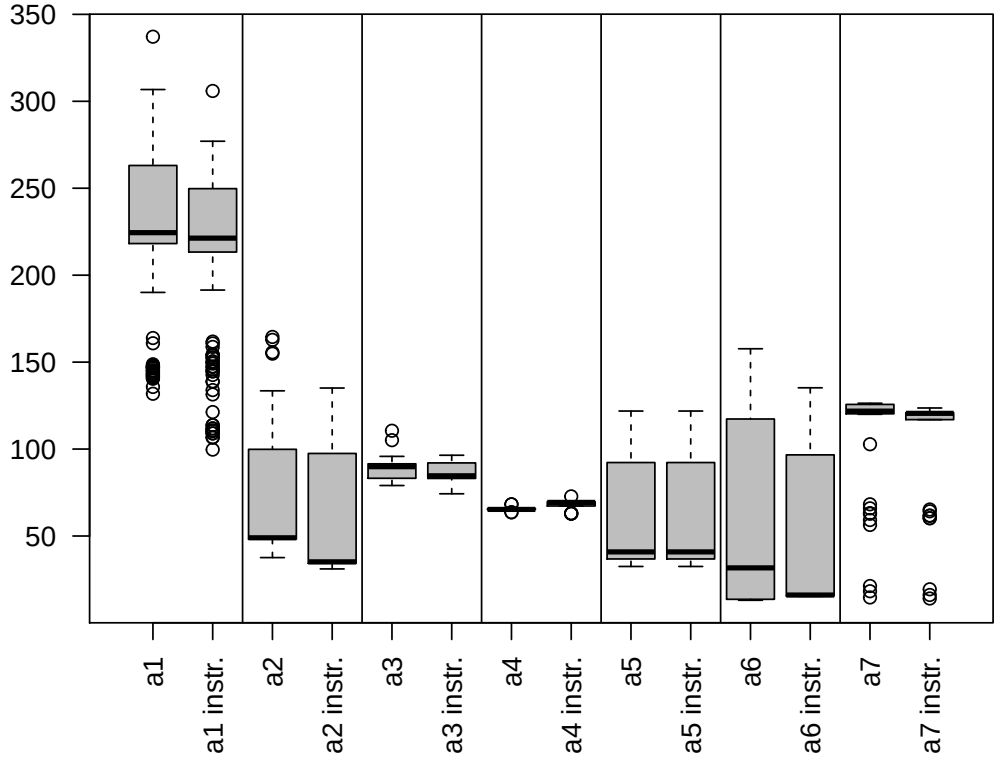
[10] `https://android.googlesource.com/platform/sdk/+/6db5720/monkeyrunner/src/com/android/monkeyrunner/recorder/MonkeyRecorder.java`

[11] `http://developer.android.com/studio/test/monkeyrunner`

[12] `http://developer.android.com/studio/command-line/adb.html`



(a) CPU load of selected apps (percentage)



(b) Memory consumption of selected apps (in Mb)

FIGURE 7.7: Performance of selected apps (both original and instrumented)

assessment of stability of the 7 apps in experiment 2 by performing a set of evaluation runs, observing that the instrumented app conformed to the expected behavior. To recap:

> The results of Experiment 2 evidence that the performance of AFP-enabled apps are comparable to those of regular apps.

Threats to validity – One common risk to validity of the experiment is the threat that adopted feature-component mappings and execution scenarios may not be representative of real app usage. To mitigate this threat, as a preliminary step, selected apps were examined by (i) analysing apps description in the Google Play store, (ii) manually inspecting their source code, and (iii) performing a set of preliminary runs while observing app behavior. Two different researchers were involved in the definition of both artifacts: first they were proposed by one and then verified to be representative by a second one.

A second threat is represented by the limited amount of apps involved in the experiment (7), a very small minority of all apps available to mobile app users. Hence, results might not generalize to other apps. Nonetheless, selected apps have reasonably varied purposes (see Table 7.1), thus partially mitigating the threat.

Finally, as for experiment 1, in this experiment we consider only one kind of *hardware* machine for the server and mobile device. As the hardware we used is likewise consumer-grade, we believe that collected measures are representative of those that can be obtained on ordinary hardware.

7.8.3 Experiment 3: Usability and acceptance of AFP by developers

Goal of the experiment – The *goal* of this experiment is to evaluate the usability and acceptance of AFP by developers. For this purpose, we conducted an on-line study, involving Android developers, in which we asked them to build the feature-component mappings for one or more apps they developed. We focused on this aspect as it is the main effort required to developers to make their app AFP-compliant (all the other steps are automated).

Research questions – we formalize the experiment goal as the following two research questions:

RQ1 – What is the *acceptance* of AFP from Android developers?

RQ2 – How do Android developers perceive the *usability* of AFP?

Answering RQ1 will provide an indication of how welcoming are Android developers of the AFP approach and of their willingness to adopt it in real world Android apps.

Answering RQ2 will provide an objective assessment of the usability of the AFP web editor. Assessing the perceived usability of this component is of particular interest, given the fact that this is the only part of the approach that requires manual effort from developers (as seen in Section 7.2). Hence, perceived usability of the web editor has a major impact on the willingness to adopt AFP.

Subjects selection – We posted an announcement on pertinent on-line discussion groups (*i.e.,* Android developer forums, mailing lists) to enlist developers willing to take part in the study. Each developer was asked to provide (at least) one link to an app published on the Google Play store, on which (s)he worked (either alone or in a team). No compensation was given in exchange for participation in the study.

Independent and dependent variables – The independent variable in our experiment is the app for which a feature-component mapping is being created (apps used in the experiment are listed Table 7.2). Dependent variables are the developer *acceptance* of the AFP approach, its perceived *usability* and the *mapping time*, *i.e.,* the time required time required by each participant for creating the feature-component mappings.

To measure both *acceptance* and *usability* we rely on an evaluation questionnaire, whose structure is shown in Table 7.3. The acceptance part is composed of three closed questions (q_1 - q_3) with possible answers ranging on a five-point scale. For the usability part, we adopted the System Usability Scale (SUS), a simple and widely-adopted scale for assessing the usability of products and services [165, 166]. The SUS consists of a questionnaire composed of 10 items (s_1 - s_{10}) and each item can be assessed by respondents along a 5-level Likert scale ranging from *Strongly agree* (4 points) to *Strongly disagree* (0 points). SUS is proved to be a valuable, robust, and reliable evaluation tool and it correlates well with other subjectives measures of usability [165, 167]. The *mapping time* is instead measured in seconds and is recorded during experiment execution in a manner transparent to the subjects.

Experiment execution – Participants in the experiment were invited to access a web-based app containing: (i) the definitions of feature- and level-based permissions, and (ii) the AFP web editor for the feature-component mappings. The participants were instructed to create the feature-component mappings for one of their previously-linked apps, and we collected the mappings defined by developers. After completing the mapping, participants were asked to complete the on-line evaluation questionnaire about AFP described in Table 7.3. Demographic information was also collected in this phase.

TABLE 7.2: Apps developed by participants of Experiment 3

Developer ID	App ID	Package Name	Version	Type	Resources	Features	Mapping time (s)
D_1	A_1	com.rubikssolutions .daily5sec	3.2	Video sharing	Camera, Micro-phone	2	21
D_2	A_2	com.ambiensvr.mobile	0.9	Augmented reality	Camera	4	924
D_3	A_3	fr.inria.es.electrosmart	1.6	Electromagnetic waves meter	Location	8	337
D_3	A_4	ums.lovely.university	9.8.4	University management system	Camera, Location	4	1809
D_4	A_5	com.digitech.foodel	1.9	Food delivery	Camera, Location	22	1758
D_5	A_6	com.mobile .wabi_tech.gadfly	2.2.9	Citizen participation	Location	3	49
D_6	A_7	com.Jitendra93266 .jitu.knowmovies	1.0	Movies database	-	3	54
D_7	A_8	de.jw.mymensa	0.9.0	Menu viewer	-	2	47
D_8	A_9	com.peaklens.ar	1.0.11	Augmented reality	Camera, Location	7	455
D_9	A_{10}	com.yopapp.yop	1.9.4	Online marketplace	Camera, Location	4	192
D_{10}	A_{11}	com.myoxygen .press.weaf	3.0	Aerospace news	Location	4	61
D_{11}	A_{12}	ro.notnull.Identical FilesFinder	3.4.0	System management	-	4	80

TABLE 7.3: Structure of the evaluation questionnaire used for developers

Evaluation goal	Question ID	Question text
Acceptance	q_1	*Is the definition of level-based and feature-based permission clear to you?*
	q_2	*Do you rate the definition of level-based and feature-based Android permissions as useful?*
	q_3	*Are you willing to use feature- and level-based permissions in your apps?*
Usability	s_1-s_{10}	As defined in the System Usability Scale [165].

A total of 11 developers completed both the mapping definition and the evaluation questionnaire, providing us with twelve mappings in total (one developer performed the mapping construction for 2 apps). Developers participating in the experiment are also quite heterogeneous, both in terms of experience, number of developed apps, and size of organization. Specifically, participants have an average of 3.45 years of Android development experience (standard deviation = 2.66) and their majority (5) developed between 2 and 5 Android apps during their career, followed by 2 developers who developed more than ten. For what concerns organizations, the majority of developers work in small organizations (*i.e.,* with 2 to 10 members), but we have also participants working in organizations with a number of members *between 2 and 10* and *between 10 and 50.*

Finally, six developers declared to be *Satisfied* with current Android 6 permissions, three declared to be *Unsatisfied* and two are *Unsure*.

Data analysis – A breakdown of the apps submitted by the participants is provided in Table 7.2. Minimum and maximum amount of features defined by developers is 2 and 22, respectively. For what concerns the time for creating the feature-component mappings, developers took an average of 482.25 seconds, *i.e.,* 8.03 minutes (median = 136s, min = 21s, max = 1809s, SD = 660s). Even in the worst case, the time required by the participants to create the feature-activities mapping is close to half an hour. We consider such amount of time acceptable, considering that the definition of such mapping is conducted only once for an app.

Figure 7.8 summarizes the distribution of answers for questions q_1, q_2 and q_3. When developers were asked about whether they understood the concepts behind feature- and level-based permissions (q_1), only one developer answered *No*, three answered *Absolutely Yes* and the remainder *Yes*. On a similar note, when asked whether they consider feature- and level-based permissions as useful (q_2), answers were two *Absolutely Yes*, five *Yes*, one *Don't know*, two *No* and one *Absolutely No*. Concerning whether they would be willing to use the AFP permissions in their apps (q_3) answers were two *Absolutely Yes*, four *Yes*, two *Don't know*, two *No* and one *Absolutely No*. For all three questions median value of answers is *Yes*.

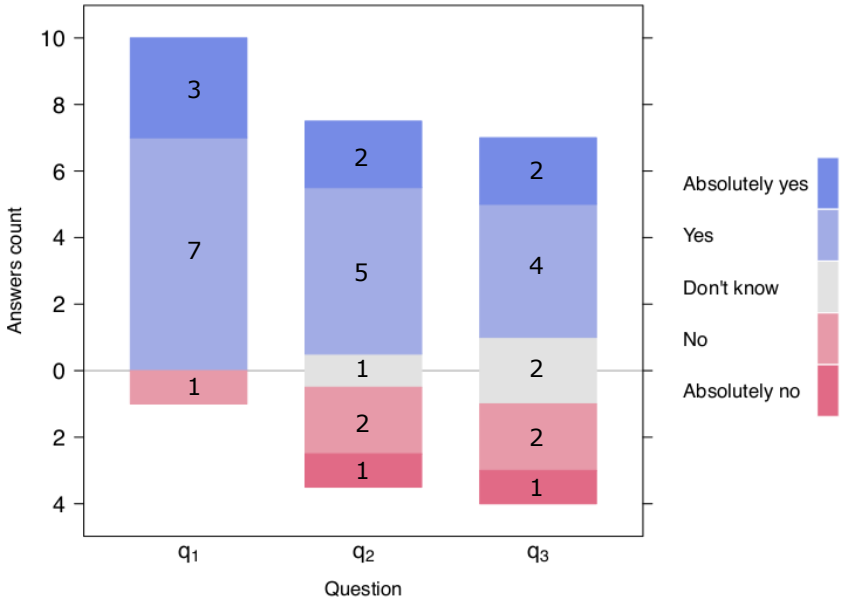


FIGURE 7.8: Results about the acceptance of AFP by developers

Results of the usability part of the evaluation questionnaire are presented in Figure 7.9, where each column of the heatmap represents the distribution of answers for each of the

ten statements that comprise SUS. The procedure described in the SUS guidelines [165] was applied to normalize answers to each statement. For most statements, answers provided by respondents are mostly agglomerated towards the middle of the scale, with the exception of s_3 and s_4, skewed towards the upper and lower end of the scale respectively. A mean SUS score of 49.77 was obtained across all participants.

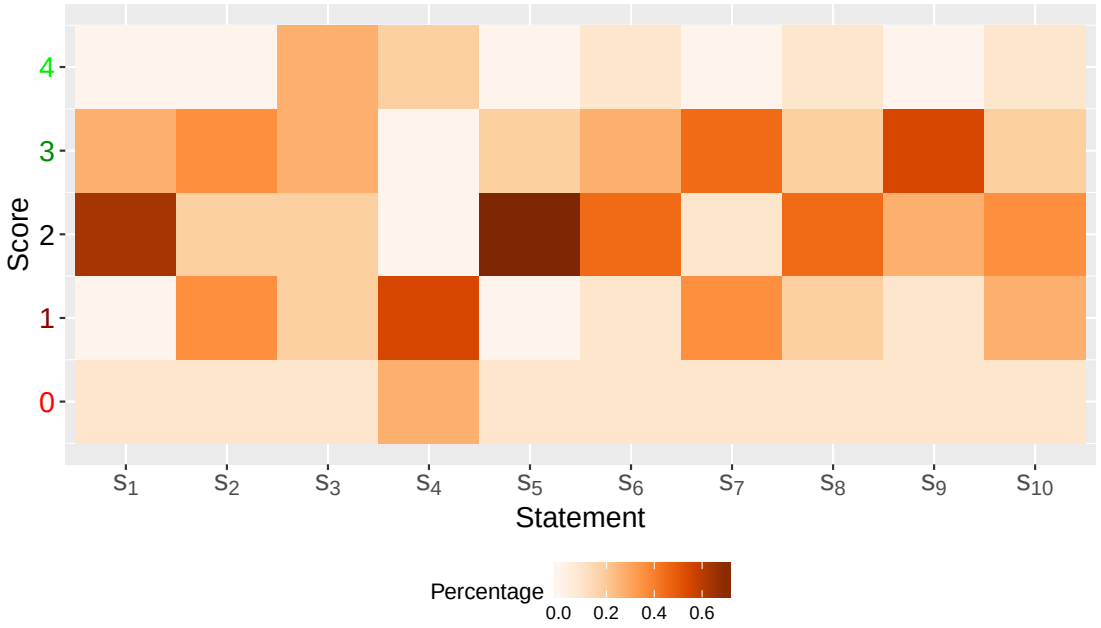


FIGURE 7.9: Frequency distribution of answers to SUS statements by developers

Discussion – To answer RQ1 we consider the answers to the *acceptance* part of the evaluation questionnaire (presented in Figure 7.8). We can see that almost all developers understood the concepts required to be able to use feature- and level-based permissions (q_1). On a similar note, the majority of participants considered feature- and level-based permissions as useful (q_2). Summarizing, a first answer to RQ1 is the following:

> The results of the *acceptance* evaluation provide evidence that feature- and level-based permissions are both understood and deemed useful by developers.

Concerning their willingness to adopt the proposed permissions approach in their apps (q_3 in Figure 7.8), the majority of developers declared that they would be willing to adopt it in their apps. Although the number of participants is relatively limited, such results are encouraging. Further expanding our answer to RQ1, we can state that:

> The majority of participants in the *acceptance* evaluation are willing to adopt feature- and level-based permissions in their apps.

To answer RQ2, we focus on answers to the *usability* part of the evaluation questionnaire (shown in Figure 7.9). We can see that answers are clustered towards the middle of the scale for the majority of statements, revealing an acceptable usability. Further confirmation is found in the obtained mean SUS score (49.77) that, according to prior research [166], is to be evaluated as *Ok*. Exceptions are statements s_3 and s_4. Most participants are in agreement with the former, that reads *"I thought the system was easy to use"*, confirming that the process was not overly difficult. At the same time developers felt in agreement with the latter: *"I think that I would need the support of a technical person to be able to use this system"*. Such results, and the fact that developers are not always familiar with permission-related technical aspects [168], encourages us to investigate in the future on techniques to assist in (or even to automate) the definition of feature-component mappings. Summarizing, we provide the following answer to RQ2:

> During the *usability* evaluation, developers judged positively the usability of the AFP approach. Improvements can be made on making the definition of the feature-component mappings more straightforward from a technical perspective.

Threats to validity – Possible threats to the validity and points of improvement for experiment 3 are as follows. The number of participants in the experiment (11) represents a very small minority of mobile app developers in the real world, hence results might not generalize. This threat is mitigated by the fact that developers who participated in the experiment have varied years of experience, nationalities and work in organizations of different size.

A second threat is represented by the limited number of apps for which developers created the mappings. Thus, the results of our study might potentially not generalize to other apps. This threat is mitigated by the fact that submitted apps have very different purposes, provide different features, and have been developed by different developers.

Finally, as the study has been conducted on-line, we had no way to ascertain that participants fully understood the task they were asked to complete. We mitigated this potential threat by directly asking in the final questionnaire whether developers had additional comments or doubts to clarify with respect to AFP.

7.8.4 Experiment 4: Usability and acceptance of AFP by end users

Goal of the study – The *goal* of this study is to evaluate usability and acceptance of AFP from the end-user perspective. To this end, we conducted an in-person study involving 47 participants.

Research question – We formalize the experiment goal in the following three research questions:

RQ1 – How does AFP *trustability* compare to the one of the current Android permission system?

RQ2 – What is the *acceptance* of AFP from end-users?

RQ3 – How do end-users perceive the *usability* of AFP?

Answering RQ1 will provide an indication of whether end-users feel more in control of their own data, having a greater degree of control on how and by whom her data are used, when using AFP as opposed to the current Android permission model.

Answering RQ2 will provide an indication of how welcoming are Android users of the AFP approach and of their willingness to use it in everyday activities.

Answering RQ3 will provide an objective assessment of the usability from the end-user perspective. Users' perceived usability of the approach is of paramount importance as it has a major impact on the their willingness to adopt AFP.

Subjects selection – We recruited participants for the experiment contacting in-person potential subjects in the University of L'Aquila and other public facilities. A total of 47 participants volunteered to take part in the experiment. Professions and backgrounds of participants are varied and include students, shop assistants, mechanical engineers, lawyers, etc.

Independent and dependent variables – during the experiment each participant was asked to try out in succession three Android apps, employing in succession either AFP or the current Android 6 permission system. As such, the independent variable in the experiment is the *permission system*. Dependent variables are instead the user perceived *trustability*, *acceptance* and *usability*.

To measure the three dependent variables we relied on an evaluation questionnaire, whose structure is shown in Table 7.4. It can be divided into three main parts, according to the different goal of each one: the first part comparatively evaluates AFP and Android 6 *trustability*, the second one evaluates the *acceptance* of AFP and the last one assesses *usability* of AFP. The first part of the questionnaire (*Trustability*) consists of two questions (Q_1, Q_2). It was filled by participants twice, respectively after trying out both permission systems. Second part of the questionnaire (*Acceptance*) contains four questions (Q_3 - Q_6). All answers to questions Q_1 – Q_6 range on a five-point Likert scale. Third part of the questionnaire concerned *Usability*. For this part, we relied again on the System Usability Scale (S_1 - S_{10}). The second and third parts of the questionnaire were filled by

participants at the end of the experiment, when we also collected open comments from the participants.

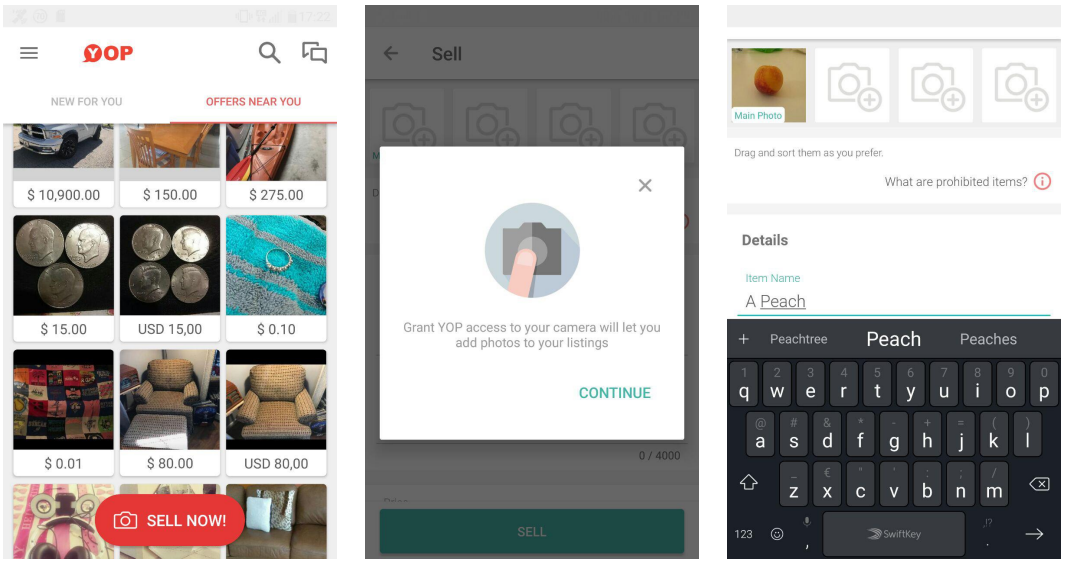
TABLE 7.4: Structure of the evaluation questionnaire for end users

Evaluation goal	Evaluated for	Question ID	Question text
Trustability	AFP vs Android 6	Q_1	*How do you judge the way the app asked for permissions?*
		Q_2	*How likely are you to use the app on your device, considering the permission you were asked for?*
Acceptance	AFP	Q_3	*Is the definition of feature-based permission clear to you?*
		Q_4	*Is the definition of level-based permission clear to you?*
		Q_5	*Do you rate feature-based permission as useful?*
		Q_6	*Do you rate level-based permission as useful?*
Usability	AFP	S_1-S_{10}	As defined in the System Usability Scale [165].

Experiment execution – Experiment execution is composed of three main phases:

1. **Pre-study:** each participant was given a short description about the goal of the experiment, together with the definition of feature- and level-based permissions. Demographic information was also collected.
2. **First trial:** each participant was asked to try out three Android apps, employing either AFP or the current Android 6 permission system. First part of the evaluation questionnaire was given at the end of the trial.
3. **Second trial:** the participant was asked to repeat the trial, this time employing the permission system that was not used during the previous phase. Remainder of the evaluation questionnaire was given at the end of the trial.

During each trial, participants were allowed to freely explore the given apps, while being monitored by one researcher that provided assistance, when needed. The researcher ensured that a minimal set of steps, namely an *execution scenario*, was executed for each app, to guarantee that participants were properly familiar with the apps and the underlying permissions systems before filling out the questionnaires. Each execution scenario was defined a priori and focused on one of the app main functionalities. An example of execution scenario for app A_{10} is given in Figure 7.10: in order to sell an object on the marketplace, users have to (a) tap on the "sell now" button, (b) take a picture of the object with the device camera (granting the required permissions if



(a) (b) (c)

FIGURE 7.10: Example of execution scenario for the `com.yopapp.yop` app: to sell an object the user (a) taps on the "sell now" button, (b) takes a picture and (c) fills out listing details

needed) and (c) fill out remaining listing data before submitting. Users were asked to complete the same execution scenarios between the two trials although inevitably some middle steps differed, *i.e.,* users had to grant permissions at run-time under Android 6 as opposed to app startup with AFP. All trials were performed on a device specifically made available, namely a LG G3 running Android 6.0.

In order to keep the experiment as representative as possible, we decided to reuse three of the apps for which real developers provided a mapping in Experiment 3. The app selection was performed with the goal of having at least one app making use of each of the device resources currently adopted by the current implementation of AFP (*i.e.,* Camera, Microphone, and Location). Unfortunately, we could not successfully instrument app A_1, the only one in our dataset that uses the microphone, because it relies on Java reflection (a language construct traditionally hard to deal with by approaches relying on static analysis [169]). Hence, we discarded it and selected an alternative app, thus ending with apps A_3, A_{10}, A_{11} in our final app selection.

Participants were mostly male (68%) and the mode of their age is *Between 21 and 30 years old.* The mean self-assessed knowledge of Android is 3.43 on a 1-5 scale (1.17 standard deviation). Roughly half of the participants (24/47) tried out the apps with the Android 6 permission system as first, and with AFP as second. The opposite order was adopted for the others.

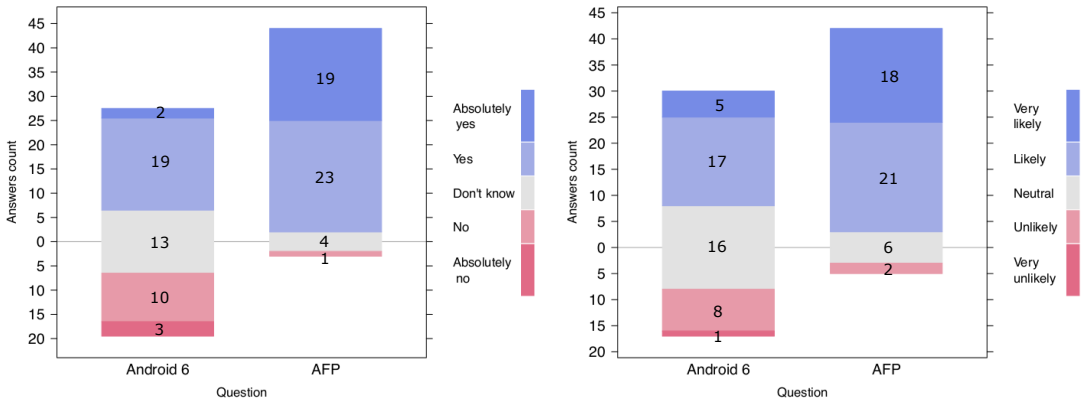
Data analysis – Table 7.5 provides the breakdown of the permission preferences configured by participants during the experiment. Overall, preferences were varied, with each access level being selected by at least one participant for all features. On average, participants required 17.27 seconds to configure their permission preferences for A_3, 36.45 seconds for A_{10} and 13.63 seconds for A_{11} (with a standard deviation of 13.21, 25.08 and 12.64 seconds respectively).

TABLE 7.5: Breakdown of end users privacy preferences

App	μ Configuration time (σ)	Feature	Resource	Access level (%)	
A_3	17.27s (13.21s)	F_1	Location	Full access: Region only: City only: Deny:	19 (40%) 6 (13%) 17 (36%) 5 (11%)
A_{10}	36.45s (25.8s)	F_2	Camera	Allow: Deny:	38 (81%) 9 (19%)
		F_3	Location	Full access: Region only: City only: Deny:	4 (9%) 17 (36%) 23 (49%) 3 (6%)
		F_4	Location	Full access: Region only: City only: Deny:	5 (11%) 20 (43%) 18 (38%) 4 (9%)
A_{11}	13.63s (12.64s)	F_5	Location	Full access: Region only: City only: Deny:	9 (19%) 15 (32%) 10 (21%) 13 (28%)

Figure 7.11 summarizes the distribution of answers for questions Q_1 and Q_2 for both AFP and the Android 6 permission system (recall that participants were asked these questions twice). For both questions, users provided more favourable answers for AFP, with a median value of *Trustable* for Q_1 and *Likely* for Q_2, as opposed to Android 6 which achieved a median of *Neutral* for both questions. Obviously, differences in answers are statistically significant, which we confirmed by performing the two-tailed Mann-Whitney U-test [118]. We obtained a p-value of $7.623e^{-08}$ for Q_1 and $4.802e^{-05}$ for Q_2, thus rejecting the null hypothesis that the distributions of the answers about Android 6 and AFP are equal.

Answers for the Acceptance part of the questionnaire (*i.e.,* Q_3, Q_4, Q_5 and Q_6) are summarized in Figure 7.12. The answers are skewed towards the positive part of the scale and the median value is *Absolutely yes* for most of them, with the only exception of Q_5 for which the median value is *Yes*.



(a) Q_1 (b) Q_2

FIGURE 7.11: Perceived trustability of Android 6 and AFPpermission systems w.r.t. the way the app asked permissions (Q_1) and how likely the participant is likely to use the app (Q_2)

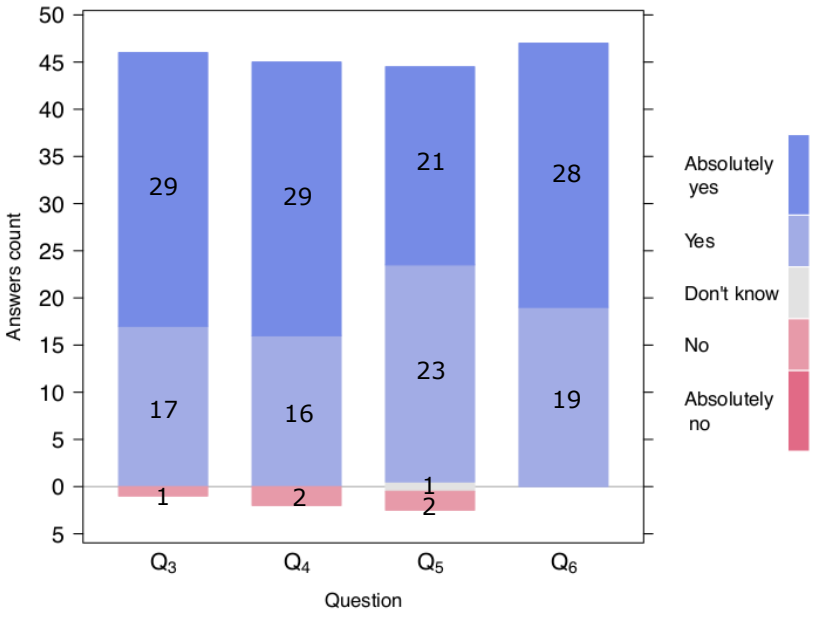


FIGURE 7.12: Acceptance of AFPby end users in terms of: clarity of the definitions (Q_3 and Q_4) and usefulness (Q_5 and Q_6) of feature-based and level-based permissions).

Results of the usability evaluation are shown in Figure 7.13. Each column of the heatmap presents the frequency distribution of answers for one of the ten SUS statements. The procedure described in the SUS guidelines [165] was applied to normalize answers to each statement. For all ten statements, participants provided mostly positive answers with the total amount of negative ones being less than 15% for all statements. A mean SUS score of 78.19 was obtained across all participants.

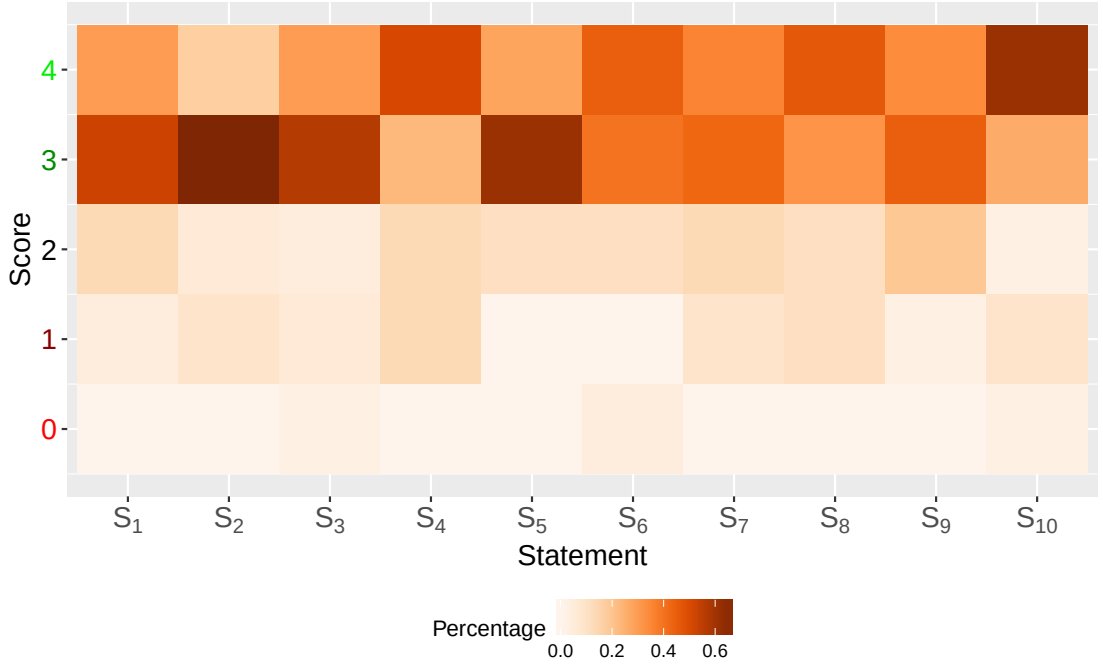


FIGURE 7.13: Frequency distribution of answers to SUS statements by users

Discussion – From the collected users privacy preferences (see Table 7.5), we can notice that choices are varied and, for all features, each level was selected by at least a minority of users. This variability in choices reveals that participants indeed took advantage of the added control provided by feature- and level-based permissions. Such data substantiates the intuition that users have diverse privacy and security concerns that cannot really be satisfied by one-size-fits-all approaches, like the current Android permission model. Hence, we can infer that:

> Collected users privacy preferences indicate that feature- and level-based permissions allow for an experience more tailored to individual privacy inclinations.

Still concerning users choices, it is worth noting that, when possible, users favoured more restricted access levels and seldom provided full location access. An exception is represented by feature F_1, for which full access is the consensus choice. During the execution of the experiment, we observed that users believed that full access was strictly

necessary for performing this feature, thus explaining such difference. Consequently, we can state that:

> Collected privacy preferences provide an indication that, when possible, most users choose levels that disclose a restricted amount of data.

To answer RQ1 we focus on answers to questions Q_1 and Q_2 (see Figure 7.11). We can point out that users generally felt more secure when employing AFP and praised the added control over shared personal data provided by it. Such considerations are also supported by some of the comments collected during the evaluation questionnaire: one participant stated that *"(Between the two systems) I prefer AFP as I have more control on permissions and I feel more secure"*, while another participant pointed out that *"as a user, with AFP I am more aware on how an app uses my phone"*. This leads us to the following reflection point, as an answer to RQ1:

> Answers to Q_1 and Q_2 provide evidence that users feel more secure and are more willing to use apps on their smartphone when using feature- and level-based permissions.

To answer RQ2 we focus on answers to the acceptance part of the evaluation questionnaire (questions Q_3 through Q_6, summarized in Figure 7.12). The definitions of feature- and level-based permissions were both well understood and deemed useful by users, although the former achieved worse answers regarding its perceived usefulness (Q_5). We conjecture this difference is due to the fact that the perceived benefit of level-based permissions is more immediate to users. Further confirmation of the perceived usefulness can be found in the open comments: one participant stated *"I appreciated the greater choice of options provided by AFP"*; a second one noted that *"(Android 6) permission pop-ups are misleading and enforce a binary choice. I appreciated AFP's level-based permissions"*. In conclusion, to answer RQ2:

> Results of the *acceptance* evaluation show that feature- and level- based permissions are both well understood and deemed useful by end users, although the usefulness of the latter is more immediate to them.

To answer RQ3 we focus on the results of the usability evaluation. According to prior research [166], the achieved mean SUS score (78.19) is to be interpreted as a *Good* level of usability (see Figure 7.13). Focusing on the distribution of answers for each statement,

positive answers are the majority for all statements although some points of improvement can be identified. Statements S_1, S_2, S_3 and S_5 achieved a comparatively lower amount of maximum score answers, revealing that some users did experience some difficulties while using the system. Investigating the comments left by participants, we noticed that some users would prefer to grant feature- and level-based permissions at run-time. In particular, one user noted *"I would like to grant permissions when needed. Configuring permissions preemptively could take too long for some apps"*. To address this issue, in the future we plan to investigate alternative ways to elicit user's privacy preferences (see Section 7.9). On the positive side, for S_{10}, mostly maximum score answers were collected, hence highlighting that participants did not consider the amount of new notions that they had to learn as excessive. Summarizing, to answer RQ3 we can state that:

During evaluation, end-users judged AFP usability as *Good*.

Threats to validity – There are several threats and points of improvement for Experiment 4. Although users were instructed to act as they would do with their own smartphone, they were still performing the trials in a controlled environment, potentially different from the normal. This means that participant activities and answers may differ from what can be observed in the real world. As future work, we will mitigate this potential threat to validity by performing the experiment via an app that users can install on their own smartphone in order to monitor the AFP-enabled apps during their usage.

Despite our efforts to have a balanced and unbiased set of participants, we ended up with a group of relatively young people (age *between 21 and 30 years old*) and with a majority of male participants (32 males as opposed to 15 females). Moreover, we are aware that the sample size of this experiment (47 participants) is limited with respect to all Android smartphone users today. We mitigated this potential threat by contacting participants with different backgrounds and professions, different experience about the Android platform, and by letting them interact with the real apps instead of Android emulators or simulated environments.

Another possible threat is represented by the limited amount of apps used to perform trials in the study. Hence, results of our study might potentially not generalize to other apps. We mitigated this threat by selecting three apps with very different purposes and features, thus collecting data on users behavior in varied scenarios. Additionally, participants were not familiar with selected apps in advance. Hence, their behavior during the trial might properly resemble the one of a user that runs a freshly installed app for the first time.

7.9 Conclusions and future work

In this chapter, we have presented an approach aimed at overcoming some limitations of the current Android permission model evidenced in previous chapters. The proposed approach and its evaluation answers the third of the research questions addressed by this dissertation (discussed in Chapter 1.3). That is:

RQ3 - *How can we rethink Android security- and privacy-preserving mechanisms in order to make them more user-centric and in line with the NGI vision?*

The proposed approach empowers end users to selectively grant permission by specifying (*i*) **the desired level of permissions granularity** and (*ii*) **the specific features of the app in which the chosen permission levels are granted**. The approach is supported by an infrastructure comprising three main components: a library internal to the apps enacting the approach at runtime; a stand-alone mobile app that allows end users to configure and negotiate at any time the permissions for each app on their devices; a web-based server for allowing developers to analyze their own mobile apps, and enhance them with the new flexible permission system with very low effort. **Four experiments have been designed, conducted, and reported for evaluating performance, usability, and acceptance from both the end users and developers perspective, confirming confidence on the approach**.

As anticipated in Section 7.8.4, a future work concerns the definition of a procedure for (semi) automatically extracting the features provided by an Android app from its binary or source code. A second line of future work concerns assisting end users in the configuration of flexible-permissions. Since runtime permissions expose context, which can help users in making their decision, the extension would require to understand how the context can be suitably extracted and presented to users during the apps configuration phase, or even leveraged to automatically configure permissions without user intervention.

Chapter 8

Conclusions

This concluding chapter briefly recaps on the contributions of this dissertation and presents possible future work.

8.1 Contributions

The goal of this work is to investigate how mobile apps can be made more in line with the next-generation internet vision, shifting to a more human-centric approach to privacy protection by giving control back to the user.

The results achieved so far by this research work are summarized in the following. Each result is defined in the context of the corresponding research question, as defined in Section 1.3.

RQ1 *What is the state on the art on static analysis for mobile applications?*

The systematic literature review, reported in Chapter 4, was aimed at identifying, evaluating and classifying characteristics, trends and potential for industrial adoption of existing research in static analysis of mobile apps. Reviewing the results of our study, we identified that the main corpus of research on the topic has focused on the Android platform, and that privacy and malware are the most targeted analysis goals. Nonetheless, researchers are mostly focusing on techniques aimed at assisting developers, store moderators and researchers, while end users are mostly being left out of the equation. We further noticed that all approaches have not been evaluated or adopted in an industrial environment.

RQ2 *Are there any existing issues in current Android security- and privacy-preserving mechanisms that negatively affect the users' trust on the whole platform?*

Chapter 5 presented the results of an empirical study in which we investigated the issues perceived by end users in the current Android run-time permission system. Even under the new system, apps still request an amount of permissions that is perceived as excessive from end-users, often without providing an explanation on why these permissions are needed. Moreover, developers do not always perform permission requests correctly, thus introducing quality issues into their apps that contribute negatively to the user experience. Some developers deny the usage of key app functionalities when permissions are denied.

In Chapter 6, we report on a second study that analyzed the occurrences of four types of permission-related issues across the development lifetime of 574 open-source apps. The study represents a first step in understanding *why* and *when* such issues are introduced into apps, by providing evidence-based insights for better understanding and management of them. Our findings revealed that permission-related issues are a frequent phenomenon in Android apps and that, despite the fact that the majority of issues are fixed in a time span of few weeks, in many cases they can linger inside an app for longer, up to several years.

RQ3 *How can we rethink Android security- and privacy-preserving mechanisms in order to make them more user-centric and in line with the NGI vision?*

In Chapter 7, we presented an approach aimed at overcoming some limitations of the current Android permission model. The proposed approach uses a new flexible permission model thanks to which end users can grant and negotiate the level of each single permission, on a per feature-basis. Developers can easily produce apps compatible with the proposed approach, as it does not require changes to their traditional workflow. Although much still needs to be done, confidence on the approach is confirmed by four experiments aimed at evaluating performance, usability, and acceptance from both the end users and developers perspective.

8.2 Limitations

In the following, we list the major known limitations for each of the chapters of this dissertation:

8.2.1 Limitations of Chapter 4

As with all systematic mapping studies, even though a well-defined research protocol was established before carrying out the data collection and followed during all the steps of the research, some limitations still exist. In particular we adopted a search strategy consisting

of a manual search encompassing all the top-level software engineering conferences and international journals according to well known sources in the field. Publications from other venues might have potentially been missed despite the fact that the search process was further extended by executing a backward and forward snowballing process on the selected literature.

8.2.2 Limitations of Chapter 5

Despite the extensive review collection process performed at the beginning of the study, only a limited share of reviews that refer to permissions was identified. As such, to confirm generalizability of the results, it would be opportune to replicate the study, extending the initial review dataset or integrating it with other sources of data (*e.g.,*, direct interviews with Android users).

Furthermore, identification of permission-related reviews was performed starting out from a keyword-based selection. Precision of this selection process is another limitation of the study, that can potentially be improved adopting more precise approaches or resorting to manual analysis.

8.2.3 Limitations of Chapter 6

The empirical study of Chapter 6 provides information about a variety of aspects of permission-related issues, such as: (i) when permissions and their related issues were introduced and fixed, (ii) who is making these decisions, (iii) file-change history that we could examine using permissions analysis tools, and (iv) all other commit information such as commit messages. Still, other aspects still remain to be explored in future works, such as whether PRIs accumulate-diminish over the lifetime of an Android app, and an investigation of the reasons behind the introduction of permission related issues.

8.2.4 Limitations of Chapter 7

Our current implementation of AFP is fully contained in the application layer of the Android stack. This might represent a vulnerability that can be exploited by malicious software such as malware. Our prototype implementation has been developed with the main goal of assessing users' perception of feature- and level-based permissions and, in order to guarantee the complete safety of users' data, a more tightly-coupled integration with the lower levels of the Android stack has to be introduced. This limitation can

be addressed adopting AFP in conjunction with other existing works on kernel-space security [45, 170, 171].

As most of the approaches in the literature using static analysis, the use of reflection, self-decrypting code, or obfuscation techniques in general challenge our approach. We are currently investigating a hybrid approach combining our static analysis with dynamic flow analysis. We have preliminary evidence that such a hybrid approach may result in a valid and viable compromise towards mitigating this challenge.

Another limitation of our approach resides in the policy configuration module. In it's current implementation, the list of sensitive Android APIs has to be kept constantly updated and wrappers for each of them have to be manually created and maintained. In the future we plan on investigating possible categorizations of Android APIs in order to develop more general configuration policies, each able to encompass a specific category of APIs.

8.3 Future research directions

Multiple are the possible future extensions of this work. We can broadly classify them in two main research directions: *(i)* deepening our understanding of user's and developer's behavior towards application permissions; *(ii)* developing new technical solutions to challenges that are still open.

Some possible future directions towards deepening our understanding of user's and developer's behavior are the following:

- **Characterizing the impact of run-time permission requests on user experience -** When employing run-time permissions, developers are faced with one additional challenge: they must carefully plan the usage of notification dialogs, as to avoid excessive disruptions in the user experience. There can be cases in which introducing new functionalities that require additional permissions may even be detrimental to the app success. All this considered, currently we only have a limited understanding of the impact of run-time dialogs on user experience. Hence, an open research area is understanding and quantifying the effects of run-time permission requests on the overall user experience.
- **Expanding our knowledge on permission-related issues in Android apps -** The work presented in Chapter 6 represents a first step towards understanding why and when permission-related issues are introduced in Android apps. Nonetheless, our knowledge on the topic is far from complete and a possible future work is to

investigate whether permission-related issues accumulate/diminish over the lifetime of Android apps, potentially revealing interesting patterns about their evolution. In addition, a more in-depth study would be required to understand what developers do when they introduce or fix permission-related issues.

- **Understanding the factors impacting user's decision making -** As seen in Section 7.8.4, end users have varied privacy preferences, resulting in different choices when they are confronted with permission requests. Understanding what are the factors that have an impact on their decision making is an open question, the answer to which may have a significant impact on ongoing efforts to create automatic permission granting mechanisms (see Section 3.3.3). In fact, a deeper understanding of users behavior can lead to automatic granting mechanism that are more precise and more in line with each user preferences.

Regarding the development of new technical solutions, the following are some open challenges:

- **Enriching permission requests with contextual information -** permission requests fall in the category of transactions with asymmetrical information described by Akerlof [30]. When answering a permission request end users lack some information that is instead available to developers, *e.g.,* what will the permission be used for, what functionalities will not be available without granting it, etc. It is known that this lack of information has a negative impact on users judgment. Hence, one open research challenge is inferring the missing information and presenting it to the user in a suitable fashion.

- **Design better support tools for developers -** As evidenced in Chapter 5, many users complained in app reviews about permission-related bugs. Indeed, developers now have to preemptively ask for allowance before accessing restricted parts of the platform. Failure to properly do so renders functionalities or, in worst cases, the whole app unreachable and/or unusable. In order to solve this issue, developers should not only be further educated, but also empowered with tools that assist them in correctly handling permissions and suitably positioning the corresponding requests.

- **Devising techniques for automated feature detection -** While designing AFP, particular care was employed in limiting the additional effort asked of developers to make their apps AFP-compliant. Nonetheless, as evidenced in section 7.8.3, some of them still experienced a noticeable amount of friction during the mapping of features to components. Hence, to ease this task, one possible point

of improvement is the definition of techniques for automated feature discovery and location. Benefits would not only be limited to our approach but could expand to tasks of interest for both developers and researchers. The former would gain a benefit from such techniques while managing large code-bases and in order to automate certain tasks during app development (*i.e.,* writing app descriptions). The latter could employ such techniques to perform more in depth empirical studies.

- **Enhancing the AFP data-model** The current AFP data model only permits the specification of *PolicyItems* that reference a single resource. The data model could be expanded to allow reasoning on multiple resources (*e.g.,* do not allow access to contacts book if the app also requests Internet access), and the specification of dynamic constraints (*e.g.,* do not allow access to the contacts book during nighttime).

Appendix A

Appendix: mapping primary studies

Table 1 reports the full list of the 140 primary studies included in the mapping study of Chapter 4.

ID	Title	Authors	Year
P1	NeSeDroid–Android Malware Detection Based on Network Traffic and Sensitive Resource Accessing [172]	N.T. Cam, N.C.H. Phuoc	2017
P2	Analyzing Remote Server Locations for Personal Data Transfers in Mobile Apps [173]	M. Eskandari, B. Kessler, M. Ahmad, A. Santana de Oliveira, B. Crispo	2017
P3	MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models [174]	E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, G. Stringhini	2017
P4	Ripple: Reflection Analysis for Android Apps in Incomplete Information Environments [175]	Y Zhang, T Tan, Y Li, J Xue	2017
P5	AndroDialysis: Analysis of Android Intent Effectiveness in Malware Detection [176]	A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez, S. Furnell	2017
P6	Profiling the responsiveness of Android applications via automated resource amplification [177]	Y. Wang, A. Rountev	2016
P7	Detecting Invalid Layer Combinations Using Control-Flow Analysis for Android [178]	N. Suzuki, T. Kamina, K. Maruyama	2016
P8	Graph-aided directed testing of Android applications for checking runtime privacy behaviours [179]	J.C.J. Keng, L. Jiang, T.K. Wee, R.K. Balan	2016
P9	Dexteroid: Detecting malicious behaviors in Android apps using reverse-engineered life cycle models [180]	M. Junaid, D. Liu, D. Kung	2016
P10	IacDroid: Preventing Inter-App Communication capability leaks in Android [181]	D. Zhang, R. Wang, Z. Lin, D. Guo, X. Cao	2016
P11	Practical, formal synthesis and automatic enforcement of security policies for android [182]	H. Bagheri, A. Sadeghi, R. Jabbarvand, S. Malek	2016
P12	CapaDroid: Detecting Capability Leak for Android Applications [183]	T. Wu, Y. Yang	2016
P13	Asynchrony-aware static analysis of Android applications [184]	A. Mishra, A. Kanade, Y.N. Srikant	2016
P14	Identifying Android inter app communication vulnerabilities using static and dynamic analysis [185]	B.F. Demissie, D. Ghio, M. Ceccato, A. Avancini	2016
P15	Towards Automatically Generating Privacy Policy for Android Apps [186]	L. Yu, T. Zhang, X. Luo, L. Xue, H. Chang	2016
P16	Revisiting the Description-to-Behavior Fidelity in Android Applications [187]	L. Yu, X. Luo, C. Qian, S. Wang	2016
P17	Triggerscope: Towards detecting logic bombs in android applications [188]	Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, G. Vigna	2016
P18	Automated test generation for detection of leaks in Android applications [189]	H. Zhang, H. Wu, A. Rountev	2016
P19	Automatic Construction of Callback Model for Android Application [190]	C. Guo, Q. Ye, N. Dong, G. Bai, J.S. Dong, J. Xu	2016
P20	Automated energy optimization of HTTP requests for mobile applications [22]	D. Li, Y. Lyu, J. Gui, W.G.J. Halfond	2016
P21	Understanding and detecting wake lock misuses for Android applications [191]	Y. Liu, C. Xu, S.C. Cheung, V. Terragni	2016
P22	Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps [192]	L. Wei, Y. Liu, S.C. Cheung	2016

P23	HybriDroid: static analysis framework for Android hybrid applications [193]	S. Lee, J. Dolby, S. Ryu	2016
P24	Reflection-aware static analysis of Android apps [194]	L. Li, T.F. Bissyandé, D. Octeau, J. Klein	2016
P25	Relda2: an effective static analysis tool for resource leak detection in Android apps [195]	T. Wu, J. Liu, X. Deng, J. Yan, J. Zhang	2016
P26	Automated testing and notification of mobile app privacy leak-cause behaviours [196]	J.C.J. Keng	2016
P27	Finding resume and restart errors in Android applications [197]	Z. Shan, T. Azim, I. Neamtiu	2016
P28	DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps [198]	L. Li, T.F. Bissyandé, D. Octeau, J. Klein	2016
P29	Empirical assessment of machine learning-based malware detectors for Android [199]	K. Allix, T.F. Bissyandé, Q. Jérome, J. Klein, Y. Le Traon	2016
P30	Effective detection of android malware based on the usage of data flow APIs and machine learning [200]	S. Wu, P. Wang, X. Li, Y. Zhang	2016
P31	On the Static Analysis of Hybrid Mobile Apps [201]	A.D. Brucker, M. Herzberg	2016
P32	Towards a Generic Framework for Automating Extensive Analysis of Android Applications [202]	L. Li, D. Li, A. Bartel, T.F. Bissyandé, J. Klein, Y. Le Traon	2016
P33	Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps [192]	L. Wei, Y. Liu, S.C. Cheung	2016
P34	Static Program Analysis for Identifying Energy Bugs in Graphics-Intensive Mobile Apps [203]	C.H.P. Kim, D. Kroening, M. Kwiatkowska	2016
P35	Combining static analysis with probabilistic models to enable market-scale android inter-component analysis [204]	D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, Y. Le Traon	2016
P36	DroidNative: automating and optimizing detection of android native code malware variants [205]	S. Alam, Z. Qu, R. Riley, Y. Chen, V. Rastogi	2016
P37	Enabling Automatic Protocol Behavior Analysis for Android Applications [206]	J. Kim, H. Choi, H. Namkung, W. Choi, B. Choi, H. Hong, D. Han	2016
P38	PERUIM: Understanding Mobile Application Privacy with permission-UI Mapping [207]	Y. Li, Y. Guo, X. Chen	2016
P39	HybriDroid: Static analysis framework for Android hybrid applications [193]	S. Lee, J. Dolby, S. Ryu	2016
P40	StubDroid: automatic inference of precise data-flow summaries for the android framework [208]	S. Arzt, E. Bodden	2016
P41	FlowMine: Android app analysis via data flow [209]	L. Sinha, S. Bhandari, P. Faruki, M.S. Gaur, V. Laxmi, M. Conti	2016
P42	Debugging energy-efficiency related field failures in mobile apps [210]	A. Banerjee, H.F. Guo, A. Roychoudhury	2016
P43	Automated energy optimization of HTTP requests for mobile applications [22]	D. Li, Y. Lyu, J. Gui, W.G.J. Halfond	2016
P44	State-Taint Analysis for Detecting Resource Bugs [211]	Z. Xu, D. Fan, S. Qin	2016
P45	Fixing Resource Leaks in Android Apps with Light-Weight Static Analysis and Low-Overhead Instrumentation [212]	J. Liu, T. Wu, J. Yan, J. Zhang	2016
P46	Relda2: an effective static analysis tool for resource leak detection in Android apps[195]	T. Wu, J. Liu, X. Deng, J. Yan, J. Zhang	2016
P47	Detecting energy leaks in android app with poem [213]	A. Ferrari, D. Gallucci, D. Puccinelli, S. Giordano	2016
P48	Lightweight measurement and estimation of mobile ad energy consumption [214]	J. Gui, D. Li, M. Wan, W.G.J. Halfond	2016
P49	AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context [215]	W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, W. Enck	2015
P50	Mining Apps for Abnormal Usage of Sensitive Data [23]	V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, E. Bodden	2015
P51	CLAPP: characterizing loops in Android applications [216]	Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, G. Vigna	2015
P52	Study and Refactoring of Android Asynchronous Programming [217]	Y. Lin, S. Okur, D. Dig	2015
P53	Tracking the Software Quality of Android Applications Along Their Evolution [218]	G. Hecht, O. Benomar, R. Rouvoy, N. Moha, L. Duchien	2015
P54	Covert Communication in Mobile Applications [219]	J. Rubin, M.I. Gordon, N. Nguyen, M.C. Rinard	2015
P55	Static Window Transition Graphs for Android [220]	S. Yang, H. Zhang, H. Wu, Y. Wang, A. Rountev	2015
P56	Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents [221]	P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. D'Amorim, M.D. Ernst	2015
P57	String Analysis of Android Applications [222]	J. Del Vecchio, F. Shen, K.M. Yee, B. Wang, S.Y. Ko, L. Ziarek	2015
P58	Interactively verifying absence of explicit information flows in Android apps [223]	O. Bastani, S. Anand, A. Aiken	2015
P59	ShamDroid: gracefully degrading functionality in the presence of limited resource access [60]	L. Brutschy, P. Ferrara, O. Tripp, M. Pistoia	2015
P60	WuKong: a scalable and accurate two-phase approach to Android app clone detection [224]	H. Wang, Y. Guo, Z. Ma, X. Chen	2015
P61	Scalable and precise taint analysis for Android [24]	W. Huang, Y. Dong, A. Milanova, J. Dolby	2015
P62	Reevaluating Android Permission Gaps with Static and Dynamic Analysis [225]	H. Wang, Y. Guo, Z. Tang, G. Bai, X. Chen	2015

P63	Andro-autopsy: Anti-malware system based on similarity matching of malware and malware creator-centric information [226]	J. Jang, H. Kang, J. Woo, A. Mohaisen, H.K. Kim	2015
P64	EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework [227]	Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, Y. Chen	2015
P65	What the app is that? deception and countermeasures in the android user interface [228]	A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, G. Vigna	2015
P66	Scalable and Precise Taint Analysis for Android [24]	W. Huang, Y. Dong, A. Milanova, J. Dolby	2015
P67	AutoPPG: Towards Automatic Generation of Privacy Policy for Android Applications [229]	L. Yu, T. Zhang, X. Luo, L. Xue	2015
P68	Information-Flow Analysis of Android Applications in DroidSafe [230]	M.I. Gordon, D.Kim, J.H. Perkins, L.Gilham, N.Nguyen, M.C. Rinard	2015
P69	StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications [231]	Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, F. Massacci	2015
P70	Potential Component Leaks in Android Apps: An Investigation into a New Feature Set for Malware Detection [232]	L. Li, K. Allix, D. Li, A. Bartel, T.F. Bissyandé, J. Klein	2015
P71	Static Control-Flow Analysis of User-Driven Callbacks in Android Applications [121]	S.Yang, D.Yan, H.Wu, Y.Wang, A.Rountev	2015
P72	Composite Constant Propagation: Application to Android Inter-Component Communication Analysis [233]	D. Octeau, D. Luchaup, M. Dering, S. Jha, P.D. McDaniel	2015
P73	IccTA: Detecting Inter-Component Privacy Leaks in Android Apps [234]	L. Li, A. Bartel, T.F. Bissyandé, J.Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P.D. McDaniel	2015
P74	EcoDroid: An Approach for Energy-based Ranking of Android Apps [235]	R.J. Behrouz, A. Sadeghi, J. Garcia, S. Malek, P. Ammann	2015
P75	Supor: Precise and scalable sensitive user input detection for android apps [236]	J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, G. Jiang	2015
P76	Uipicker: User-input privacy identification in mobile applications [237]	Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, X. Wang	2015
P77	Andro Lyze: A Distributed Framework for Efficient Android App Analysis [238]	L. Baumgärtner, P. Graubner, N. Schmidt, B. Freisleben	2015
P78	Using text mining to infer the purpose of permission use in mobile apps [239]	H. Wang, J. Hong, Y. Guo	2015
P79	Static reference analysis for GUI objects in Android software [240]	A. Rountev, D. Yan	2014
P80	Static analysis for independent app developers [241]	L. Brutschy, P. Ferrara, P. Müller	2014
P81	Cochecker: Detecting capability and sensitive data leaks from component chains in android [242]	X. Cui, D. Yu, P.P.F. Chan, L.C.K. Hui, S.M. Yiu, S. Qing	2014
P82	Android Taint Flow Analysis for App Sets [243]	W. Klieber, L. Flynn, A. Bhosale, L. Jia, L. Bauer	2014
P83	Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps [244]	F. Wei, S. Roy, X. Ou, Robby	2014
P84	AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications [245]	M. Zhang, H. Yin	2014
P85	Semantics-aware android malware classification using weighted contextual api dependency graphs [246]	M. Zhang, Y. Duan, H. Yin, Z. Zhao	2014
P86	DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket [247]	D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck	2014
P87	Retrofitting concurrency for android applications through refactoring [248]	Y. Lin, C. Radoi, D. Dig	2014
P88	Checking app behavior against app descriptions [25]	A. Gorla, I. Tavecchia, F. Gross, A. Zeller	2014
P89	Information Flows As a Permission Mechanism [52]	F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E.J. Lehner, S.Y. Ko, L. Ziarek	2014
P90	Greendroid: Automated diagnosis of energy inefficiency for smartphone applications [249]	Y. Liu, C. Xu, S.C. Cheung, J. Lu	2014
P91	FlowDroid: Precise context-,flow-,field-,object-sensitive and lifecycle-aware taint analysis for android apps [79]	S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P.D. McDaniel	2014
P92	Cassandra: Towards a Certifying App Store for Android [250]	S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, A. Weber	2014
P93	Code Injection Attacks on HTML5-based Mobile Apps:Characterization, Detection and Mitigation [251]	X. Jin, X. Hu, K. Ying, W. Du, H. Yin, G. Nagesh Peri	2014
P94	Efficient, context-aware privacy leakage confinement for android applications without firmware modding [252]	M. Zhang, H. Yin	2014
P95	Collaborative Verification of Information Flow for a High-Assurance App Store [253]	M.D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, E.X. Wu	2014
P96	Multi-App Security Analysis with FUSE: Statically Detecting Android App Collusion [254]	T. Ravitch, E.R. Creswick, A. Tomb, A. Foltzer, T. Elliott, L. Casburn	2014
P97	Characterizing and detecting performance bugs for smartphone applications [255]	Y. Liu, C. Xu, S.C. Cheung	2014
P98	AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction [256]	J. Huang, X. Zhang, L. Tan, P. Wang, B. Liang	2014
P99	Apposcopy: semantics-based detection of Android malware through static analysis [257]	Y. Feng, S. Anand, I. Dillig, A. Aiken	2014

P100	Detecting energy bugs and hotspots in mobile apps [258]	A. Banerjee, L.K. Chong, S. Chattopadhyay, A. Roychoudhury	2014
P101	Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges and Solutions for Analyzing Android [259]	A. Bartel, J. Klein, M. Monperrus, Y. Le Traon	2014
P102	Responsiveness analysis tool for android application [260]	T. Ongkosit, S. Takada	2014
P103	Automatically exploiting potential component leaks in android applications [261]	L. Li, A. Bartel, J. Klein, Y. Le Traon	2014
P104	Effective inter-component communication mapping in android: An essential step towards holistic security analysis [262]	D. Octeau, P.D. McDaniel, S. Jha, A. Bartel, E.ÂăBodden, J. Klein, Y. Le Traon	2013
P105	DroidAPIMiner: Mining API-level features for robust malware detection in android [263]	Y. Aafer, W. Du, H. Yin	2013
P106	An empirical study of cryptographic misuse in android applications [264]	M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel	2013
P107	Targeted and depth-first exploration for systematic testing of android apps [26]	T. Azim, I. Neamtiu	2013
P108	Sound and precise malware analysis for android via pushdown reachability and entry-point saturation [265]	S. Liang, A.W. Keep, M. Might, S. Lyde, T. Gilray, Liang, S., Keep, A. W., Might, M., Lyde, S., Gilray, T., P. Aldous, D. Van Horn	2013
P109	AppIntent: analyzing sensitive data transmission in android for privacy leakage detection [266]	Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, X.S. Wang	2013
P110	AppProfiler: a flexible method of exposing privacy-related behavior in android applications to end users [267]	S. Rosen, Z. Qian, Z.M. Mao	2013
P111	Flow permissions for android [53]	S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S.Y. Ko, L. Ziarek	2013
P112	Slicing Droids: Program Slicing for Smali Code [268]	J. Hoffmann, M. Ussath, T. Holz, M. Spreitzenbarth	2013
P113	A grey-box approach for automated GUI-model generation of mobile applications [269]	W. Yang, M.R. Prasad, T. Xie	2013
P114	Structural detection of android malware using embedded call graphs [270]	H. Gascon, F. Yamaguchi, D. Arp, K. Rieck	2013
P115	Estimating mobile application energy consumption using program analysis [271]	S. Hao, D. Li, W.G.J. Halfond, R. Govindan	2013
P116	Characterizing and detecting resource leaks in Android applications [272]	C. Guo, J. Zhang, J. Yan, Z. Zhang, Y. Zhang	2013
P117	Calculating source line level energy information for Android applications [273]	D. Li, S. Hao, W.G.J. Halfond, R. Govindan	2013
P118	Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications [274]	C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, W. Zou	2012
P119	Why Eve and Mallory love Android: An analysis of Android SSL (in) security [275]	S. Fahl, M. Harbach, T. Muders, M.ÂăSmith, L. Baumgärtner, B. Freisleben	2012
P120	User-aware privacy control via extended static-information-flow analysis [276]	X. Xiao, N. Tillmann, M. Fähndrich, J. De Halleux, M. Moskal	2012
P121	Dr. Android and Mr. Hide: fine-grained permissions in android applications [51]	J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, T. D. Millstein	2012
P122	LeakMiner: Detect Information Leakage on Android with Static Taint Analysis [277]	Z. Yang, M. Yang	2012
P123	SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications [278]	J. Kim, Y. Yoon, K. Yi, J. Shin, S. Center	2012
P124	A framework for static detection of privacy leaks in android applications [279]	C. Mann, A. Starostin	2012
P125	DroidChecker: analyzing android applications for capability leak [280]	P.P.F. Chan, L. C. K. Hui, S. M. Yiu	2012
P126	DroidMat: Android Malware Detection through Manifest and API Calls Tracing [281]	D.J. Wu, C.H. Mao, T.E. Wei, H.M. Lee, K.P. Wu	2012
P127	Static analysis of Android programs [282]	E. Payet, F. Spoto	2012
P128	Estimating Android applications' CPU energy usage via bytecode profiling [83]	H. Hao, D. Li, W. G. J. Halfond, R. Govindan	2012
P129	What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps [283]	A. Pathak, A. Jindal, Y. Charlie Hu, S. P. Midkiff	2012
P130	User-centric dependence analysis for identifying malicious mobile apps [284]	K.O. Elish, D. Yao, B.G. Ryder	2012
P131	AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale [285]	C. Gibler, J. Crussell, J. Erickson, H. Chen	2012
P132	Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. [286]	Y. Zhou, Z. Wang, W. Zhou, X. Jiang	2012
P133	RiskRanker: Scalable and Accurate Zero-day Android Malware Detection [287]	M.C. Grace, Y.Zhou, Q. Zhang, S. Zou, X. Jiang	2012
P134	Chex: statically vetting android apps for component hijacking vulnerabilities [288]	L. Lu, Z. Li, Z. Wu, W. Lee, G. Jiang	2012
P135	Automatically securing permission-based software by reducing the attack surface: An application to android [289]	A. Bartel, J. Klein, Y. Le Traon, M. Monperrus	2012
P136	Static analysis of Android programs [282]	E. Payet, F. Spoto	2011

P137	Android permissions demystified [35]	A. Porter Felt, E. Chin, S. Hanna, D. Song, D. A. Wagner:	2011
P138	Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications [290]	L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A. D. Schmidt, S. Albayrak	2011
P139	Analyzing Inter-application Communication in Android [137]	E. Chin, A. Porter Felt, K. Greenwood, D. A. Wagner	2011
P140	PiOS: Detecting Privacy Leaks in iOS Applications [291]	M. Egele, C. Kruegel, E. Kirda, G. Vigna	2011

TABLE 1: Primary Studies

Bibliography

[1] Patrick Cerwall et al. Ericsson mobility report - june 2018. *Ericsson, Stockholm*, 2018.

[2] Jelle Kooistra et al. Newzoo global mobile market report 2018. 2018.

[3] Adam Lella, Andrew Lipsman. The U.S. Mobile App Report, 2017. comsCore white paper.

[4] University of Alabama at Birmingham Online Masters in Management Information Systems. The Future of Mobile Application, 2014. `http://businessdegrees.uab.edu/resources/infographic/the-future-of-mobile-application/`.

[5] Mary Madden, Lee Rainie, Kathryn Zickuhr, Maeve Duggan, and Aaron Smith. Public perceptions of privacy and security in the post-snowden era. *Pew Research Center*, 12, 2014.

[6] Jan Lauren Boyles, Aaron Smith, and Mary Madden. Privacy and data management on mobile devices. *Pew Internet & American Life Project*, 4, 2012.

[7] Monique Calisti. The data economy and the next generation internet: highlights from the digital assembly 2018. Technical report, 07 2018.

[8] Internet of humans - how we would like the internet of the future to be. `https://ec.europa.eu/digital-single-market/en/news/internet-humans-how-we-would-internet-future-be`. Accessed: 2018-11-21.

[9] Steve Taylor and Michael Boniface. Next generation internet the emerging research challenges. 2017.

[10] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

[11] Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: Msr for app stores. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 108–111. IEEE Press, 2012.

[12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[13] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.

[14] Adam Lella and Andrew Lipsman. The us mobile app report. *Tech. Rep.*, 8, 2014.

[15] Michael Mandel and Elliott Long. The app economy in europe: Leading countries and cities, 2017. *Practically pragmatic*, 2017.

[16] Robin Nunkesser. Beyond web/native/hybrid: a new taxonomy for mobile app development. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 214–218. ACM, 2018.

[17] Gartner Inc. Gartner says worldwide sales of smartphones grew 7 percent in the fourth quarter of 2016. `http://www.gartner.com/newsroom/id/3609817`, 2017 (accessed September 27, 2017).

[18] M. Zinoune. Why is android built on linux kernel?, 2012. `https://www.unixmen.com/why-is-android-built-on-linux-kernel/`.

[19] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 7–7, 2011.

[20] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[21] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[22] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. Automated energy optimization of http requests for mobile applications. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 249–260. IEEE, 2016.

[23] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of

sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[24] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 106–117, 2015.

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

[26] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM, 2013.

[27] Gary S Becker. *The economic approach to human behavior*. University of Chicago press, 2013.

[28] Herbert A Simon. Models of man; social and rational. 1957.

[29] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian Schaub, Manya Sleeper, et al. Nudges for privacy and security: understanding and assisting users? choices online. *ACM Computing Surveys (CSUR)*, 50(3):44, 2017.

[30] George A Akerlof. The market for ?lemons?: Quality uncertainty and the market mechanism. In *Uncertainty in Economics*, pages 235–251. Elsevier, 1978.

[31] Avi Charkham. 5 design tricks facebook uses to affect your privacy decisions. techcrunch.(aug. 2012), 5.

[32] Rainer Böhme and Jens Grossklags. The security cost of cheap user interaction. In *Proceedings of the 2011 New Security Paradigms Workshop*, pages 67–82. ACM, 2011.

[33] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do windows users follow the principle of least privilege?: investigating user account control practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, page 1. ACM, 2010.

[34] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *USENIX security symposium*, volume 13, 2013.

[35] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[36] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: installing applications on an android smartphone. In *International Conference on Financial Cryptography and Data Security*, pages 68–79. Springer, 2012.

[37] Alexios Mylonas, Anastasia Kastania, and Dimitris Gritzalis. Delegate the smartphone user? security awareness in smartphone platforms. *Computers & Security*, 34:47–66, 2013.

[38] Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3393–3402. ACM, 2013.

[39] Serge Egelman, Adrienne Porter Felt, and David Wagner. Choice architecture and smartphone privacy: There?sa price for that. In *The economics of information security and privacy*, pages 211–236. Springer, 2013.

[40] Panagiotis Andriotis, Martina Angela Sasse, and Gianluca Stringhini. Permissions snapshots: Assessing users' adaptation to the android runtime permission model. In *Information Forensics and Security (WIFS), 2016 IEEE International Workshop on*, pages 1–6. IEEE, 2016.

[41] Panagiotis Andriotis, Shancang Li, Theodoros Spyridopoulos, and Gianluca Stringhini. A comparative study of android users? privacy preferences under the runtime permission model. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 604–622. Springer, 2017.

[42] Panagiotis Andriotis, Gianluca Stringhini, and Martina Angela Sasse. Studying users? adaptation to android?s run-time fine-grained access control system. *Journal of information security and applications*, 40:31–43, 2018.

[43] Anthony Peruma, Jeffrey Palmerino, and Daniel E Krutz. Investigating user perception and comprehension of android permission models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 56–66. ACM, 2018.

[44] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM symposium on information, computer and communications security*, pages 328–332. ACM, 2010.

[45] Rubin Xu, Hassen Saïdi, and Ross J Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*, volume 2012, 2012.

[46] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *International Conference on Information Security*, pages 331–345. Springer, 2010.

[47] Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. Crêpe: A system for enforcing fine-grained context-related policies on android. *IEEE Transactions on Information Forensics and Security*, 7(5):1426–1438, 2012.

[48] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard–enforcing user requirements on android apps. In *International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems*, pages 543–548. Springer, 2013.

[49] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard–fine-grained policy enforcement for untrusted android applications. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 213–231. Springer, 2014.

[50] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Nikhilesh Reddy, Yixin Zhu, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: Fine-grained security policies on unmodified android. Technical report, 2011.

[51] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[52] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y Ko, and Lukasz Ziarek. Information flows as a permission mechanism. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 515–526. ACM, 2014.

[53] Shashank Holavanalli, Don Manuel, Vishwas Nanjundaswamy, Brian Rosenberg, Feng Shen, Steven Y Ko, and Lukasz Ziarek. Flow permissions for android. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 652–657. IEEE Press, 2013.

[54] Giovanni Russello, Bruno Crispo, Earlence Fernandes, and Yuri Zhauniarovich. Yaase: Yet another android security extension. In *Privacy, Security, Risk and*

Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on, pages 1033–1040. IEEE, 2011.

[55] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

[56] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*, pages 49–54. ACM, 2011.

[57] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. Taming information-stealing smartphone applications (on android). In *International conference on Trust and trustworthy computing*, pages 93–107. Springer, 2011.

[58] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[59] Kassem Fawaz and Kang G Shin. Location privacy protection for smartphone users. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 239–250. ACM, 2014.

[60] Lucas Brutschy, Pietro Ferrara, Omer Tripp, and Marco Pistoia. Shamdroid: gracefully degrading functionality in the presence of limited resource access. In *ACM SIGPLAN Notices*, volume 50, pages 316–331. ACM, 2015.

[61] Primal Wijesekera, Joel Reardon, Irwin Reyes, Lynn Tsai, Jung-Wei Chen, Nathan Good, David Wagner, Konstantin Beznosov, and Serge Egelman. Contextualizing privacy decisions for better prediction (and protection). In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 268. ACM, 2018.

[62] Zheran Fang, Weili Han, Dong Li, Zeqing Guo, Danhao Guo, Xiaoyang Sean Wang, Zhiyun Qian, and Hao Chen. revdroid: Code analysis of the side effects after dynamic permission revocation of android apps. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 747–758. ACM, 2016.

[63] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security Symposium*, pages 499–514, 2015.

[64] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[65] Katarzyna Olejnik, Italo Dacosta, Joana Soares Machado, Kévin Huguenin, Mohammad Emtiyaz Khan, and Jean-Pierre Hubaux. Smarper: Context-aware and automatic runtime-permissions for mobile devices. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 1058–1076. IEEE, 2017.

[66] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[67] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 1077–1093. IEEE, 2017.

[68] Daniel Votipka, Kristopher Micinski, Seth M Rabin, Thomas Gilray, Michelle M Mazurek, and Jeffrey S Foster. User comfort with android background resource accesses in different contexts. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. USENIX Association, 2018.

[69] Marco Autili, Paolo Di Benedetto, and Paola Inverardi. A hybrid approach for resource-based comparison of adaptable java applications. *Science of Computer Programming*, 78(8):987–1009, 2013.

[70] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[71] Barbara Kitchenham and Pearl Brereton. A systematic review of systematic review process research in software engineering. *Information and software technology*, 55 (12):2049–2075, 2013.

[72] Victor R Basili-Gianluigi Caldiera and H Dieter Rombach. Goal question metric paradigm. *Encyclopedia of software engineering*, 1:528–532, 1994.

[73] Marco Autili, Ivano Malavolta, Alexander Perucci, and Gian Luca Scoccia. Perspectives on static analysis of mobile apps (invited talk). In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 29–30. ACM, 2015.

[74] Samireh Jalali and Claes Wohlin. Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 29–38. ACM, 2012.

[75] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 38. ACM, 2014.

[76] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *EASE*, volume 8, pages 68–77, 2008.

[77] Barbara A Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, Keele University and University of Durham, 2007.

[78] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering*, 2017.

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

[80] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 49–60. ACM, 2011.

[81] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[82] Stephen Fink and Julian Dolby. Wala–the tj watson libraries for analysis, 2012.

[83] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 1–7. IEEE Press, 2012.

[84] OtáVio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology*, 53(4):294–306, 2011.

[85] Hira Agrawal, James L Alberi, Joseph R Horgan, J Jenny Li, Saul London, W Eric Wong, Sudipto Ghosh, and Norman Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, 1998.

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

[87] John C Mankins. Technology readiness levels. *White Paper, April*, 6, 1995.

[88] Godfrey Nolan. *Decompiling android*. Apress, 2012.

[89] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. A quantitative and qualitative investigation of performance-related commits in android apps. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 443–447. IEEE, 2016.

[90] Luca Pascarella, Franz-Xaver Geiger, Fabio Palomba, Dario Di Nucci, Ivano Malavolta, and Alberto Bacchelli. Self-reported activities of android developers. In *5th IEEE/ACM International Conference on Mobile Software Engineering and Systems, New York, NY*, 2018.

[91] Gian Luca Scoccia, Ivano Malavolta, Marco Autili, Amleto Di Salle, and Paola Inverardi. User-centric android flexible permissions. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 365–367. IEEE, 2017.

[92] Roberto Verdecchia, Giuseppe Procaccianti, Ivano Malavolta, Patricia Lago, and Joost Koedijk. Estimating energy impact of software releases and deployment strategies: The KPMG case study. In *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*, pages 257–266. IEEE, 2017.

[93] Antonio Martini and Jan Bosch. The danger of architectural technical debt: Contagious debt and vicious circles. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 1–10. IEEE, 2015.

[94] Zahra Sahaf, Vahid Garousi, Dietmar Pfahl, Rob Irving, and Yasaman Amannejad. When to automate software testing? decision support based on system dynamics: an industrial case study. In *Proceedings of the 2014 International Conference on Software and System Process*, pages 149–158. ACM, 2014.

[95] Franz-Xaver Geiger, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli. A graph-based dataset of commit history of real-world android apps. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR. ACM, New York, NY*, 2018.

[96] Haipeng Cai and Barbara G Ryder. Artifacts for dynamic analysis of android apps. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 659–659. IEEE, 2017.

[97] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, page 3. ACM, 2012.

[98] Phong Minh Vu, Tam The Nguyen, Hung Viet Pham, and Tung Thanh Nguyen. Mining user opinions in mobile app reviews: A keyword-based approach (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 749–759. IEEE, 2015.

[99] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007.

[100] Distribution of free and paid android apps in the google play store from 3rd quarter 2017 to 4th quarter 2017. `https://www.statista.com/statistics/266211/distribution-of-free-and-paid-android-apps/`. Accessed: 2018-03-09.

[101] William Martin, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. The app sampling problem for app store mining. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 123–133. IEEE Press, 2015.

[102] Phong Minh Vu, Hung Viet Pham, Tam The Nguyen, and Tung Thanh Nguyen. Tool support for analyzing mobile app reviews. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 789–794. IEEE, 2015.

[103] Vitalii Avdiienko, Konstantin Kuznetsov, Paolo Calciati, Juan Carlos Caiza Román, Alessandra Gorla, and Andreas Zeller. Calappa: a toolchain for mining android applications. In *Proceedings of the International Workshop on App Market Analytics*, pages 22–25. ACM, 2016.

[104] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. Ar-miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, pages 767–778. ACM, 2014.

[105] Elizabeth Ha and David Wagner. Do android users write about electric sheep? examining consumer reviews in google play. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, pages 149–157. IEEE, 2013.

[106] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[107] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[108] Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.

[109] Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society. Series B (Methodological)*, pages 111–147, 1974.

[110] Dan Jurafsky and James H Martin. *Speech and language processing*, volume 3. Pearson London, 2014.

[111] Walid Maalej and Hadeer Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *2015 IEEE 23rd international requirements engineering conference (RE)*, pages 116–125. IEEE, 2015.

[112] Charu C Aggarwal and ChengXiang Zhai. A survey of text classification algorithms. In *Mining text data*, pages 163–222. Springer, 2012.

[113] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[114] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[115] Leo Breiman. *Classification and regression trees.* Routledge, 2017.

[116] Luis Torgo. *Data mining with R: learning with case studies.* Chapman and Hall/CRC, 2016.

[117] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning*, pages 137–142. Springer, 1998.

[118] Gregory W Corder and Dale I Foreman. *Nonparametric statistics: A step-by-step approach.* John Wiley & Sons, 2014.

[119] Normal and dangerous permissions. `https://developer.android.com/guide/topics/permissions/requesting.html#normal-dangerous`. Online; accessed 12-January-2017.

[120] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing android permission creep. In *Proceedings of the Web*, volume 2, pages 91–96, 2011.

[121] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 89–99. IEEE Press, 2015.

[122] Permissions usage notes. `https://developer.android.com/training/permissions/usage-notes.html`, . Online; accessed 24-October-2017.

[123] Requesting permissions at run time. `https://developer.android.com/training/permissions/requesting.html`, . Online; accessed 24-October-2017.

[124] Thomas D Cook and Donald T Campbell. Quasi-experimentation: Design and analysis for field setting. *MA: Houghton Mifflin*, 1979.

[125] Christy Pettey and Rob van der Meulen. Gartner says free apps will account for nearly 90 percent of total mobile app store downloads in 2012. *Gartner),[Online] September*, 11, 2012.

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

[127] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[128] Xing Gao, Dachuan Liu, Haining Wang, and Kun Sun. Pmdroid: Permission supervision for android advertising. In *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*, pages 120–129. IEEE, 2015.

[129] Permissions best practices. https://developer.android.com/training/permissions/best-practices.html. URL `https://developer.android.com/training/permissions/best-practices.html`.

[130] Colton Dennis, Daniel E Krutz, and Mohamed Wiem Mkaouer. P-lint: a permission smell detector for android applications. In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pages 219–220. IEEE, 2017.

[131] Ryan Stevens, Jonathan Ganz, Vladimir Filkov, Premkumar Devanbu, and Hao Chen. Asking for (and about) permissions used by android apps. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 31–40. IEEE Press, 2013.

[132] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[133] F-droid - free and open source android app repository. https://f-droid.org. URL `https://f-droid.org`.

[134] Piper Chester, Chris Jones, Mohamed Wiem Mkaouer, and Daniel E Krutz. M-perm: a lightweight detector for android permission gaps. In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pages 217–218. IEEE, 2017.

[135] Investigating permission issues in open-source android apps. https://mobilepermissions.github.io/.

[136] Osara: Open source android repository analyzer. https://www.hiddenToKeepAnonymous.com. URL `https://www.hiddenToKeepAnonymous.com`.

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

[138] Junwei Tang, Ruixuan Li, Hongmu Han, Heng Zhang, and Xiwu Gu. Detecting permission over-claim of android applications with static and semantic analysis approach. In *Trustcom/BigDataSE/ICESS, 2017 IEEE*, pages 706–713. IEEE, 2017.

[139] Lingfeng Bao, David Lo, Xin Xia, Xinyu Wang, and Cong Tian. How android app developers manage power consumption?: An empirical study by mining power management commits. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 37–48. ACM, 2016.

[140] Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. Code smells in ios apps: How do they compare to android? In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pages 110–121. IEEE, 2017.

[141] Daniel E Krutz, Mehdi Mirakhorli, Samuel A Malachowsky, Andres Ruiz, Jacob Peterson, Andrew Filipski, and Jared Smith. A dataset of open-source android applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 522–525. IEEE Press, 2015.

[142] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[143] How long does it take to build a mobile app? `http://www.kinvey.com/how-long-to-build-an-app-infographic`, 2017. URL `http://www.kinvey.com/how-long-to-build-an-app-infographic/`.

[144] Ivano Malavolta, Roberto Verdecchia, Bojan Filipovic, Magiel Bruntink, and Patricia Lago. How maintainability issues of android apps evolve. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 24-29, 2018*, 2018.

[145] Ricardo Baeza-Yates, Berthier de Araújo Neto Ribeiro, et al. *Modern information retrieval*. New York: ACM Press; Harlow, England: Addison-Wesley,, 2011.

[146] Alan Agresti and Maria Kateri. *Categorical data analysis*. Springer, 2011.

[147] David J Sheskin. Parametric and nonparametric statistical procedures. *Chapman & Hall/CRC: Boca Raton, FL*, 2000.

[148] Jarrett Rosenberg. Statistical methods and measurement. In *Guide to Advanced Empirical Software Engineering*, pages 155–184. Springer, 2008.

[149] Robert J Grissom and John J Kim. Effect sizes for research. *A broad practical approach. Mah*, 2005.

[150] Daniel E Krutz, Nuthan Munaiah, Anthony Peruma, and Mohamed Wiem Mkaouer. Who added that permission to my app? an analysis of developer permission changes in open source android apps. In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pages 165–169. IEEE, 2017.

[151] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering*, pages 511–522. ACM, 2018.

[152] Vladimir Kovalenko and Alberto Bacchelli. Code review for newcomers: is it different? In *2018 IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 29–32. IEEE, 2018.

[153] Mathieu Goeminne and Tom Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986, 2013.

[154] Erik Kouters, Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ van den Brand. Who's who in gnome: Using lsa to merge software repository identities. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 592–595. IEEE, 2012.

[155] Igor Scaliante Wiese, José Teodoro da Silva, Igor Steinmacher, Christoph Treude, and Marco Aurélio Gerosa. Who is who in the mailing list? comparing six disambiguation heuristics to identify multiple addresses of a participant. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 345–355. IEEE, 2016.

[156] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[157] Horizon 2020. ICT Leit Work Programme 2018-2020, draft February 2017, 2017. URL `https://ec.europa.eu/programmes/horizon2020/en/area/ict-research-innovation`.

[158] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[159] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[160] Intents and intent filters. Intents and intent filters. `https://developer.android.com/guide/components/intents-filters`.

[161] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 33–44. ACM, 2012.

[162] Pern Hui Chia, Yusuke Yamamoto, and N Asokan. Is this app safe?: a large scale study on application permissions and risk signals. In *Proceedings of the 21st international conference on World Wide Web*, pages 311–320. ACM, 2012.

[163] Ivano Malavolta, Stefano Ruberto, Tommaso Soru, and Valerio Terragni. Hybrid mobile apps in the google play store: An exploratory investigation. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, pages 56–59. IEEE Press, 2015.

[164] Inc Gartner. Gartner says free apps will account for nearly 90 percent of total mobile app store downloads in 2012, 2012. `http://www.gartner.com/newsroom/id/2153215`.

[165] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[166] Aaron Bangor, Philip T Kortum, and James T Miller. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6): 574–594, 2008.

[167] Jurek Kirakowski and Mary Corbett. Sumi: The software usability measurement inventory. *British journal of educational technology*, 24(3):210–212, 1993.

[168] Rebecca Balebako, Abigail Marsh, Jialiu Lin, Jason I Hong, and Lorrie Faith Cranor. The privacy and security behaviors of smartphone app developers. 2014.

[169] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[170] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in android. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 25–36. ACM, 2014.

[171] Abdallah Dawoud and Sven Bugiel. Droidcap: Os support for capability-based permissions in android. 2019.

[172] Nguyen Tan Cam and Nguyen Cam Hong Phuoc. Nesedroid?android malware detection based on network traffic and sensitive resource accessing. In *Proceedings of the International Conference on Data Engineering and Communication Technology*, pages 19–30. Springer, 2017.

[173] Mojtaba Eskandari, Bruno Kessler, Maqsood Ahmad, Anderson Santana de Oliveira, and Bruno Crispo. Analyzing remote server locations for personal data transfers in mobile apps. *Proceedings on Privacy Enhancing Technologies*, 2017(1):118–131, 2017.

[174] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. 2017.

[175] Yifei Zhang, Tian Tan, Yue Li, and Jingling Xue. Ripple: Reflection analysis for android apps in incomplete information environments. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 281–288. ACM, 2017.

[176] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell. Androdialysis: Analysis of android intent effectiveness in malware detection. *computers & security*, 65:121–134, 2017.

[177] Yan Wang and Atanas Rountev. Profiling the responsiveness of android applications via automated resource amplification. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 48–58. ACM, 2016.

[178] Noriyuki Suzuki, Tetsuo Kamina, and Katsuhisa Maruyama. Detecting invalid layer combinations using control-flow analysis for android. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, pages 27–32. ACM, 2016.

[179] Joseph Chan Joo Keng, Lingxiao Jiang, Tan Kiat Wee, and Rajesh Krishna Balan. Graph-aided directed testing of android applications for checking runtime privacy behaviours. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 57–63. ACM, 2016.

[180] Mohsin Junaid, Donggang Liu, and David Kung. Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models. *computers & security*, 59:92–117, 2016.

[181] Daojuan Zhang, Rui Wang, Zimin Lin, Dianjie Guo, and Xiaochun Cao. Iacdroid: Preventing inter-app communication capability leaks in android. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*, pages 443–449. IEEE, 2016.

[182] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 514–525. IEEE, 2016.

[183] Tianjun Wu and Yuexiang Yang. Capadroid: Detecting capability leak for android applications. In *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*, pages 95–104. Springer, 2016.

[184] Ashish Mishra, Aditya Kanade, and YN Srikant. Asynchrony-aware static analysis of android applications. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 163–172. IEEE, 2016.

[185] Biniam Fisseha Demissie, Davide Ghio, Mariano Ceccato, and Andrea Avancini. Identifying android inter app communication vulnerabilities using static and dynamic analysis. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 255–266. ACM, 2016.

[186] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. Toward automatically generating privacy policy for android apps. *IEEE Transactions on Information Forensics and Security*, 12(4):865–880, 2017.

[187] Le Yu, Xiapu Luo, Chenxiong Qian, and Shuai Wang. Revisiting the description-to-behavior fidelity in android applications. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 415–426. IEEE, 2016.

[188] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 377–396. IEEE, 2016.

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

[190] Chenkai Guo, Quanqi Ye, Naipeng Dong, Guangdong Bai, Jin Song Dong, and Jing Xu. Automatic construction of callback model for android application. In *Engineering of Complex Computer Systems (ICECCS), 2016 21st International Conference on*, pages 231–234. IEEE, 2016.

[191] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the*

2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 396–409. ACM, 2016.

[192] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237. ACM, 2016.

[193] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: static analysis framework for android hybrid applications. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 250–261. IEEE, 2016.

[194] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 756–761. IEEE, 2016.

[195] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. Relda2: an effective static analysis tool for resource leak detection in android apps. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 762–767. IEEE, 2016.

[196] Joseph Chan Joo Keng. Automated testing and notification of mobile app privacy leak-cause behaviours. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 880–883. IEEE, 2016.

[197] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. Finding resume and restart errors in android applications. In *ACM SIGPLAN Notices*, volume 51, pages 864–880. ACM, 2016.

[198] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329. ACM, 2016.

[199] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Empirical Softw. Engg.*, pages 183–211, 2016.

[200] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow apis and machine learning. *Information and Software Technology*, 75:17–25, 2016.

[201] Achim D Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps. In *International Symposium on Engineering Secure Software and Systems*, pages 72–88. Springer, 2016.

[202] Li Li, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Towards a generic framework for automating extensive analysis of android applications. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1460–1465. ACM, 2016.

[203] Chang Hwan Peter Kim, Daniel Kroening, and Marta Kwiatkowska. Static program analysis for identifying energy bugs in graphics-intensive mobile apps. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pages 115–124. IEEE, 2016.

[204] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *ACM SIGPLAN Notices*, volume 51, pages 469–484. ACM, 2016.

[205] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *computers & security*, 65:230–246, 2017.

[206] Jeongmin Kim, Hyunwoo Choi, Hun Namkung, Woohyun Choi, Byungkwon Choi, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han. Enabling automatic protocol behavior analysis for android applications. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 281–295. ACM, 2016.

[207] Yuanchun Li, Yao Guo, and Xiangqun Chen. Peruim: Understanding mobile application privacy with permission-ui mapping. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 682–693. ACM, 2016.

[208] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering*, pages 725–735. ACM, 2016.

[209] Lovely Sinha, Shweta Bhandari, Parvez Faruki, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. Flowmine: Android app analysis via data flow. In *Consumer Communications & Networking Conference (CCNC), 2016 13th IEEE Annual*, pages 435–441. IEEE, 2016.

[210] Abhijeet Banerjee, Hai-Feng Guo, and Abhik Roychoudhury. Debugging energy-efficiency related field failures in mobile apps. In *Proceedings of the International*

Conference on Mobile Software Engineering and Systems, pages 127–138. ACM, 2016.

[211] Zhiwu Xu, Dongxiao Fan, and Shengchao Qin. State-taint analysis for detecting resource bugs. In *Theoretical Aspects of Software Engineering (TASE), 2016 10th International Symposium on*, pages 168–175. IEEE, 2016.

[212] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. Fixing resource leaks in android apps with light-weight static analysis and low-overhead instrumentation. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 342–352. IEEE, 2016.

[213] Alan Ferrari, Dario Gallucci, Daniele Puccinelli, and Silvia Giordano. Detecting energy leaks in android app with poem. In *Pervasive Computing and Communication Workshops (PerCom Workshops), 2015 IEEE International Conference on*, pages 421–426. IEEE, 2015.

[214] Jiaping Gui, Ding Li, Mian Wan, and William GJ Halfond. Lightweight measurement and estimation of mobile ad energy consumption. In *Green and Sustainable Software (GREENS), 2016 IEEE/ACM 5th International Workshop on*, pages 1–7. IEEE, 2016.

[215] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 303–313. IEEE Press, 2015.

[216] Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Clapp: Characterizing loops in android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 687–697. ACM, 2015.

[217] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 224–235. IEEE, 2015.

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

[219] Julia Rubin, Michael I Gordon, Nguyen Nguyen, and Martin Rinard. Covert communication in mobile applications (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 647–657. IEEE, 2015.

[220] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static window transition graphs for android. *Automated Software Engineering*, 25(4):833–873, 2018.

[221] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 669–679. IEEE, 2015.

[222] Justin Del Vecchio, Feng Shen, Kenny M Yee, Boyu Wang, Steven Y Ko, and Lukasz Ziarek. String analysis of android applications (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 680–685. IEEE, 2015.

[223] Osbert Bastani, Saswat Anand, and Alex Aiken. Interactively verifying absence of explicit information flows in android apps. In *ACM SIGPLAN Notices*, volume 50, pages 299–315. ACM, 2015.

[224] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 71–82. ACM, 2015.

[225] Haoyu Wang, Yao Guo, Zihao Tang, Guangdong Bai, and Xiangqun Chen. Reevaluating android permission gaps with static and dynamic analysis. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6. IEEE, 2015.

[226] Jae-wook Jang, Hyunjae Kang, Jiyoung Woo, Aziz Mohaisen, and Huy Kang Kim. Andro-autopsy: Anti-malware system based on similarity matching of malware and malware creator-centric information. *Digital Investigation*, 14:17–35, 2015.

[227] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[228] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 931–948. IEEE, 2015.

[229] Le Yu, Tao Zhang, Xiapu Luo, and Lei Xue. Autoppg: Towards automatic generation of privacy policy for android applications. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 39–50. ACM, 2015.

[230] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[231] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 37–48. ACM, 2015.

[232] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential component leaks in android apps: An investigation into a new feature set for malware detection. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 195–200. IEEE, 2015.

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

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

[235] Reyhaneh Jabbarvand Behrouz, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. Ecodroid: An approach for energy-based ranking of android apps. In *Green and Sustainable Software (GREENS), 2015 IEEE/ACM 4th International Workshop on*, pages 8–14. IEEE, 2015.

[236] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *USENIX Security Symposium*, pages 977–992, 2015.

[237] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *USENIX Security Symposium*, pages 993–1008, 2015.

[238] Lars Baumgärtner, Pablo Graubner, Nils Schmidt, and Bernd Freisleben. Andro lyze: A distributed framework for efficient android app analysis. In *Mobile Services (MS), 2015 IEEE International Conference on*, pages 73–80. IEEE, 2015.

[239] Haoyu Wang, Jason Hong, and Yao Guo. Using text mining to infer the purpose of permission use in mobile apps. In *Proceedings of the 2015 ACM International*

Joint Conference on Pervasive and Ubiquitous Computing, pages 1107–1118. ACM, 2015.

[240] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 143. ACM, 2014.

[241] Lucas Brutschy, Pietro Ferrara, and Peter Müller. Static analysis for independent app developers. In *ACM SIGPLAN Notices*, volume 49, pages 847–860. ACM, 2014.

[242] Xingmin Cui, Da Yu, Patrick Chan, Lucas CK Hui, Siu-Ming Yiu, and Sihan Qing. Cochecker: Detecting capability and sensitive data leaks from component chains in android. In *Australasian Conference on Information Security and Privacy*, pages 446–453. Springer, 2014.

[243] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[244] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

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

[246] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[247] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[248] Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 341–352. ACM, 2014.

[249] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Jian Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, (1):1–1, 2014.

[250] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 93–104. ACM, 2014.

[251] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[252] Mu Zhang and Heng Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 259–270. ACM, 2014.

[253] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1104. ACM, 2014.

[254] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2014.

[255] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[256] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.

[257] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[258] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 588–598. ACM, 2014.

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

[260] Thanaporn Ongkosit and Shingo Takada. Responsiveness analysis tool for android application. In *Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile*, pages 1–4. ACM, 2014.

[261] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 388–397. IEEE, 2014.

[262] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis*, 2013.

[263] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[264] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[265] Shuying Liang, Andrew W Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 21–32. ACM, 2013.

[266] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage

detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[267] Sanae Rosen, Zhiyun Qian, and Z Morely Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 221–232. ACM, 2013.

[268] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851. ACM, 2013.

[269] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 250–265. Springer, 2013.

[270] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[271] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 92–101. IEEE Press, 2013.

[272] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in android applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 389–398. IEEE, 2013.

[273] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89. ACM, 2013.

[274] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

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

[276] Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan De Halleux, Michal Moskal, and Tao Xie. User-aware privacy control via extended static-information-flow analysis. *Automated Software Engineering*, 22(3):333–366, 2015.

[277] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*, pages 101–104. IEEE, 2012.

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

[279] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 1457–1462. ACM, 2012.

[280] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.

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

[282] Étienne Payet and Fausto Spoto. Static analysis of android programs. *Information and Software Technology*, 54(11):1192–1201, 2012.

[283] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 267–280. ACM, 2012.

[284] Karim O Elish, Danfeng Yao, and Barbara G Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Workshop on Mobile Security Technologies*, 2012.

[285] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.

[286] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

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

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

[289] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277. ACM, 2012.

[290] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72. IEEE, 2011.

[291] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, pages 177–183, 2011.