
# Security and Privacy in App Ecosystems

Vincent F. Taylor

Magdalen College
University of Oxford

A thesis submitted for the degree of
*Doctor of Philosophy*

Trinity 2017

To my family, my friends, and the other people I love.

# Abstract

Smartphones are highly-capable mobile computing devices that have dramatically changed how people do business, interact with online services, and receive entertainment. Smartphone functionality is enhanced by an ecosystem of apps seemingly covering the entire gamut of functionality. While smartphone apps have undoubtedly provided immeasurable benefit to users, they also contribute their fair share of drawbacks, such as increases in security risks and the erosion of user privacy. In this thesis, I focus on the Android smartphone operating system, and pave the way for improving the security and privacy of its app ecosystem.

Chapter 3 starts by doing a comprehensive study on how Android apps have evolved over a three-year period, both in terms of their dangerous permission usage and the vulnerabilities they contain. It uncovers a trend whereby apps are using increasing numbers of dangerous permissions over time and at the same time becoming increasingly vulnerable to attack by adversaries.

By analysing the Google Play Store, Android's official app marketplace, Chapter 4 shows that many general-purpose apps can be replaced with functionally-similar alternatives to the benefit of the user. This confirms that users still wield power to improve their own security and privacy. Chapter 5 combines this insight with real-world data from approximately 30,000 smartphones to understand the actual risk that the average user faces as a result of their use of apps, and takes an important first step in measuring the improvements that can be made.

Users, however, are not always aware of the risks they face and thus Chapter 6 demonstrates the feasibility of a classification system that can transparently and unobtrusively identify and alert users to the presence of apps of concern on their devices. This classification system identifies apps from features in the network traffic they generate, without itself analysing the payload of their traffic, thus maintaining a high threshold of privacy.

While the work presented in this thesis has uncovered undesirable trends in app evolution, and shows that a large fraction of users are exposed to non-trivial risk from the apps they use, in many cases there is sufficient diversity in the offerings of general-purpose apps in the Google Play Store to empower users to mitigate the risks coming from the apps they use. This work takes us a step further in keeping users safe as they navigate and enjoy app ecosystems.

# Acknowledgements

My heartfelt gratitude goes out to my brilliant and supportive supervisor, Ivan Martinovic. He was insightful, resourceful, and encouraging. Ivan was a fantastic collaborator and was never short of great ideas and useful feedback.

This work was made possible by The Rhodes Trust and the UK EPSRC (grant number EP/P00881X/1). I am also grateful to the Department of Computer Science, Magdalen College, and my supervisor for their funding of conference-related activities.

Thanks to my thesis examiners Andrew Martin and N. Asokan for their useful and extensive feedback. My thanks also go to the intermediate assessors I had throughout the course of my research: Ivan Flechais and Andrew Simpson.

Many thanks to the anonymous reviewers of my published work. Your critical feedback shaped my thoughts and helped to make this thesis into what it is today.

Thank you to Alastair Beresford, Riccardo Spolaor, and Mauro Conti. I co-authored papers with you and our many discussions were quite useful. Thanks also go to the developers and maintainers of the Device Analyzer project. Easy access to such a large dataset allowed me to include valuable real-world measurements in my research.

Thanks to the many researchers I met at conferences, workshops and symposia. The questions you asked and the discussions we had certainly helped to broaden my perspective.

Many thanks to the Jamaicans I met and became close to in Oxford: 'Andy', 'Comie', David, 'Den Den', Ian, 'Ibby', 'Jeff', 'Junior', Kevin, 'Madoo', 'Q', Tariq, Timar, and 'Tocos'. You helped me to feel at home while living 4,670 miles from home.

Many thanks to the other friends I made in Oxford and especially to Sanj. Sanj was an amazing listening ear throughout my D.Phil.

Many thanks to my parents and siblings who have been a constant source of support and love.

Many thanks to my close friends back in Jamaica: Anthony, Garth, Jason, Jodiann, Mario, Stefan H., and Stefan W. You provided much encouragement and made my vacations at home that much more enjoyable.

# Contents

# List of Figures

# List of Tables

# CHAPTER 1

# Introduction

*If there's a book that you want to read, but it hasn't been written yet, then you must write it.*

– Toni Morrison

One of the earliest general-purpose computers was the ENIAC, which saw use in the 1940s/50s. The ENIAC used approximately 18,000 vacuum tubes, and required more than 167 square metres of floor space to house its hardware and cooling systems [16]. Computer technology has improved dramatically since then. Computers have seen orders of magnitude of increases in processing power and orders of magnitude of decreases in size. This miniaturisation of processing power has given rise to ultra-portable computing devices such as smartphones.

In 2011, only 35% of adults in the U.S. owned a smartphone, compared to 77% in 2016; a more than two-fold increase in only five years [137]. Smartphone usage also increased markedly in that period. Nielson reports that in the final quarter of 2014, users spent over an hour a day using their smartphones [29], more

than double that of the same period in 2011 [28]. This appetite for smartphone usage is fuelled in part by myriad apps available to users, providing out-of-the-box functionality at the touch of a button. Easy access to apps has seen users interacting with approximately 27 apps per month [29]. There seemingly is an app for everything, with apps covering the entire spectrum of functionality; from gaming to entertainment to banking to healthcare.

Apps are mainly distributed from convenient central repositories known as *app stores* (or *app marketplaces*). App stores allow third-party app developers to publish apps for users to download. Usually, app developers invest their time into making apps with the expectation of making a return on their investment. The symbiosis of users, smartphones, apps, app stores, and app developers is known as an *app ecosystem* [123] (a specific type of software ecosystem [107]).

With the portable storage that smartphones provide, users typically use them as a sort of personal digital assistant; storing contacts, messages, pictures, videos, and calendar appointments. Smartphones are also equipped with sensors, such as gyroscopes, cameras, and GPS receivers, making them suitable for navigation and photography. The various services provided by a smartphone are further enhanced by an internet connection, usually provided by a cellular of Wi-Fi interface. Internet access allows the smartphone to backup files, send and receive messages, browse the internet, provide dynamic content, and the like.

The convenience provided by smartphones, however, comes at the cost of the security and privacy of the user. The smartphone itself contains a trove of sensitive user data. It has sensors that can pinpoint a user's location or spatial orientation. It allows code from third-parties, in the form of apps, to be executed. It provides wireless interfaces through which apps can talk to the outside world. If devices continue to get smaller, continue to have increasing numbers of sensors, continue to be more inter-connected, and continue to run potentially untrustworthy apps, it is important to understand the security and privacy of app ecosystems.

## 1.1 Problem Background

At the time of writing, the two major smartphone operating systems (OS) are Google's Android and Apple's iOS [27]. The official app store for Android is the Google Play Store, while Apple's App Store is the official app store for iOS. Apps undergo preliminary screening for maliciousness, using systems such as Bouncer [5], before they are admitted to the Google Play Store. Apple also screens apps before they are admitted to the Apple App Store [11].

For these reasons, users have some assurance that the apps they download have undergone at least preliminary checking for maliciousness. Wang et al. [158], however, demonstrated ways to circumvent the protections offered by the 'walled-garden' approach used in app stores and so vetting is not a perfect solution. Sophisticated malware has found its way into the Google Play Store, further confirming this fact [81][124][59].

Aside from malware, there is the concern of "phalibs" (*p*otentially *ha*rmful third-party *lib*rarie*s*) that permeate apps, and though not being overtly malicious themselves, may surreptitiously cause malfeasance [54]. So-called 'piggybacked' apps also pose a threat, since they feature malicious payloads inserted into legitimate apps [170].

In the third quarter of 2016, the Android operating system dominated the global smartphone market with over 86% market share [27] and an official app marketplace with over two million apps. A distant second was iOS, with 12.5% market share but a similar number of apps in its official marketplace. Given the popularity of the Android OS, this thesis focuses exclusively on the Android app ecosystem. All app metadata and app binaries used for analysis in this thesis was collected from the Google Play Store.

In addition to downloading apps from an app store, users may also *sideload* apps. Sideloading is a manual process where users download the `apk` file for an app, transfer it to the target device (if it was downloaded using a separate de-

vice), and then manually executing the file. In this thesis, only apps downloaded and installed through the Google Play Store are considered.

Screening provided by the app store is merely one level of the defence-in-depth approach employed by the Android security architecture [138]. As an additional layer of protection for users, from both malicious and benign apps alike, Android runs each app as a separate[1] Linux user (different UID), i.e., in its own sandbox and with variable access to the device [65]. This access control is enforced using system permissions[2], which guard access to sensitive user data [38]. Apps seeking sensitive access to a device must have their permissions granted in full at install-time (on Android 5.1 or lower *or* if the app has a `targetSDK` of 22 or lower) or selectively granted at run-time (on Android 6.0 or higher *if* the app has a `targetSDK` of 23 or higher) [25].

## 1.2 The Permissions Problem

The ability to grant or deny permissions gives users additional power over the apps on their smartphone, but also has its own disadvantages. In the first instance, many users fail to understand what particular permissions mean and what access they are granting to apps on their devices [78][76]. Additionally, many users blindly accept (or have no option but to accept) all permissions when presented with install-time permission prompts [152]. It was hoped that run-time permissions would alleviate this problem by forcing users to accept permissions individually when an app first required them to perform a function, but Eling et al. [68] showed that 40.4% of users still blindly accept individual run-time permissions. Thus, run-time permissions are not the panacea they were hoped to be.

[1]Note that multiple apps may be run under the same UID provided that they are signed with the same cryptographic key [7].

[2]This thesis exclusively focuses on *dangerous permissions*, the 24 system permissions on Android that guard access to sensitive user data [30]. Thus, for brevity, *dangerous permissions* as simply referred to as *permissions* for the remainder of this thesis.

Although run-time permissions provide an improvement in principle, users need to have compatible devices *and* the app needs to be built to support run-time permissions. While devices continue to be upgraded and more and more devices will become compatible with run-time permissions, it remains unclear whether app developers will update their apps to support the feature. Indeed, there may not be much incentive for developers to choose a `targetSDK` of 23 or higher, while doing so comes with the associated cost of updating app codebases, as well as a risk to developer revenue[3] if users are allowed to reject unnecessary permissions. Thus, there exists an undesirable scenario where app developers have both disincentives to support run-time permissions and 'veto' power over run-time permissions and can nullify the feature altogether [31].

## 1.3 Support for Run-time Permissions

Developers not updating their apps can be a challenge to run-time permissions becoming ubiquitous. To highlight the problem of the lack of support for run-time permissions, I now use tools and techniques later described in Chapter 3 to determine the extent to which popular apps in the Google Play Store support run-time permissions. One million downloads was arbitrarily chosen as the threshold for an app to be considered popular. Fig. 1.1 shows the `targetSDK` of popular apps. Approximately 40% of popular apps currently have a `targetSDK` of 23 or higher. That is, at the time of writing only 40% of popular apps have the capability to trigger run-time permissions, but will only do so if they are installed on compatible devices.

I further used Google Play Store app metadata to understand the update history of popular apps not currently supporting run-time permissions. Approximately 64% of popular apps not supporting run-time permissions have been

[3]Free apps are typically monetised using advertisements. Advertisements may generate more revenue if the advertisement delivery system is able to leverage more permissions for better user profiling.

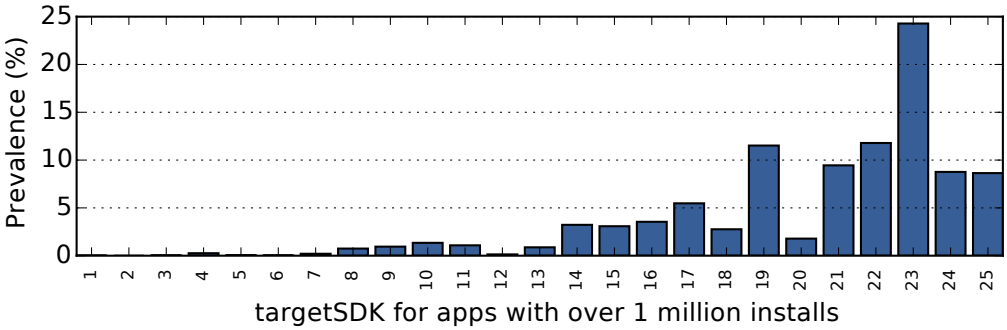



Figure 1.1: Breakdown of the `targetSDK` of apps with more than one million downloads.

updated since the release of Android 6.0 on October 5, 2015 when run-time permissions were officially introduced [14]. On average, these apps received their latest update one year (353 days to be precise) after the introduction of run-time permissions. Thus, there is evidence to suggest that the developers of these apps have limited interest in supporting run-time permissions at the current time. The 36% of apps that have not been updated since the introduction of run-time permissions were last updated 378 days, on average, before the release of Android 6.0. Thus, a non-trivial portion of popular apps are not even receiving updates that would be needed to allow run-time permissions to work as intended.

## 1.4 Motivation

The walled-garden approach to providing apps to users has worked to keep malware off devices for the most part, but a new sort of threat has emerged, whereby users face risks from greyware apps. These apps are particularly insidious, since they actually do provide functionality, but may surreptitiously profit from sensitive user data or device resources in ways that may not be immediately clear [36]. Even benign apps can be a source of concern, since they may contain third-party libraries (henceforth called libraries, for brevity) embedded inside. These libraries help app developers to rapidly provide functionality, but also contribute risks since they obtain the privileges granted to the host app [88].

The Android permission system forms one of the main lines of defence between apps and the sensitive data they may access from a device. For this reason, prior work has aimed to understand permission usage in apps and the libraries they embed [139][88][45][169]. These pieces of work have highlighted areas of concern such as overprivilege [75] and the use of undocumented permissions by libraries [139]. However, it remains unclear whether apps are getting more or less risky to use and whether apps and libraries are seeking greater access to devices over time.

The *freemium* model of app delivery has emerged as a dominant business model in app stores, whereby app developers provide basic features to users free of charge, with users often having the option to purchase more advanced functionality as they see fit [101]. Thus, users may enjoy a plethora of apps and out-of-the-box functionality without any direct monetary risk. Whether the pricing strategy is free or freemium, apps are typically monetised by their developers through the use of in-app advertisements (hereafter called ads), provided by advertising libraries (hereafter called *ad libraries*) [150].

App developers are either paid according to the number of impressions (times an ad is shown) or clicks on an ad from within their apps. Targeting ads based on user demographics will typically be more profitable, since users are more likely to click on ads that appeal to their interests. For these reasons, ad libraries will sometimes recommend app developers to request permissions outside of those required for the core functionality of the app, ostensibly so they are better able to obtain rich targeting data. Thus, many apps request permissions beyond what is required for core app functionality [151]. More insidiously, ad libraries are known to gratuitously leverage additional permissions that have been granted to the host app, including permissions that are undocumented in the ad library software development kit (SDK) [139].

If apps use combinations of permissions that are unique for the particular type of app, it may be a useful indicator that the app is permission-hungry, potentially

malicious, or generally undesirable. Several authors have used features such as app category and combination of permissions used to understand whether an app can be considered as an outlier [131][122][85][117]. It remains unclear, however, whether a more fine-grained approach such as understanding app functionality could be used to identify suspicious apps. Additionally, it is unclear the extent to which suspicious apps could be replaced with more favourable functionally-similar alternatives.

A number of authors quantified the impact of individual apps on devices and their threat to user privacy and security [118][168][57], but there has been little work on understanding the cumulative effect of the set of apps installed on a device. Furthermore, while apps may be undesirable from a security and privacy perspective for one or more reasons, scanning devices and making users aware that they are using such apps remains a non-trivial task. Users would need to manually submit a list of the apps they use or install a third-party app that provided such functionality. Neither of these approaches are scalable or satisfactory.

## 1.5 Contributions

My goal in this thesis is to improve security and privacy in the Android app ecosystem. I consider apps downloaded (or downloadable) from the Google Play Store. The risks I focus on come from either the apps themselves or the libraries that are bundled within these apps. I also consider security risks that come from vulnerabilities in the way apps implement their components or program logic. I further consider data privacy risks whereby apps or libraries (ab)use the permissions they have been granted to access sensitive user data. Finally, I explore ways for devices to be transparently scanned for apps of concern so that their users can be notified.

To this end, I have designed, built, and tested several frameworks for taking

measurements of interest. This thesis is structured as four main contribution components, each with a corresponding chapter, enabling an understanding of a core area of my research interest. In what follows, I briefly describe each of these four components in the order that they later appear, and explain how they combine to support my overall argument.

### 1.5.1 LongLook

Chapter 3, LongLook, presents the results of a *long*-term *look* at app evolution in the Google Play Store over a three-year period. I am interested in understanding how apps evolve in terms of three main characteristics: permission usage, vulnerabilities, and 'library empowerment'. As it relates to permission usage, apps seem to be getting more permission-hungry over time. If this is the case, it warrants further investigation since apps and their libraries are getting greater access to sensitive data.

With added access to sensitive data, vulnerabilities within apps magnify the risk to users. Thus, I am also interested in studying how vulnerabilities within apps evolve as apps are updated. Finally, I am interested in library empowerment, a phenomenon that happens when embedded libraries get greater access to user data as a result of increased permission usage by apps. The analysis in this chapter highlights trends in how apps are changing over time and also provides additional motivation for the work in the remaining chapters.

Chapter 3 shows that apps are getting more permission-hungry over time and that they are getting less secure, i.e., containing more vulnerabilities as they are updated. This highlights a worrying trend in that apps are seeking increasing access to sensitive user data, while at the same time providing greater opportunities for attackers to exploit them. Moreover, the study of library empowerment demonstrates how users may unwittingly lose additional sensitive data to third-party libraries as apps are updated to use more permissions.

### 1.5.2 SecuRank

Apps becoming more permission-hungry and more vulnerable over time is not immediately clear to users, who are left powerless to defend themselves. In the case of general-purpose apps, such as *flashlight* apps, there are a wide variety to choose from, but users may not be readily able to identify or choose one that is safer from a security or privacy perspective. Many users will likely default to installing one of the most highly ranked search results when searching for apps, just as they do in traditional web searches [97]. Thus, the general-purpose apps users choose to install on their devices may be less than ideal.

Motivated by the observations from Chapter 3, Chapter 4 presents SecuRank, a tool useful for providing a *secu*rity *rank*ing of general-purpose apps. The goal of SecuRank is to assess the (entire) Google Play Store to understand the availability of functionally-similar apps that can be used to replace apps that are considered to be less than ideal. SecuRank focuses on apps that are general-purpose in nature, i.e., apps that may be feasibly replaced with functionally-similar alternatives.

There are several motivations for developing the SecuRank system. First, it is important to quantify the existence of general-purpose apps in the Google Play Store as a whole. Second, clustering general-purpose apps by functionality is an important first step that enables comparing functionally-similar apps using metrics such as permission usage, library usage, and the like. Chapter 4 shows that although there is a large amount of apps with less-than-ideal security and privacy characteristics, in many cases they can be replaced with functionally-similar alternatives, to the benefit of the user.

### 1.5.3 DeviceAnalyzer

Building on the observations from Chapter 4, Chapter 5 measures the overall risks that real-world users face as a result of their use of apps, and quantifies

improvements that can be made. This study is facilitated by real-world data from approximately 30,000 devices. The real-world data that I focus on is lists of apps installed on smartphones, provided by the Device Analyzer project [154].

Chapter 5 has two main research threads. In the first thread, I combine the aforementioned real-world data with the SecuRank tool I developed in Chapter 4 to understand improvements to privacy that can be made if less permission-hungry apps are chosen by users at install-time. In addition, I contribute a technique for identifying permission-protected APIs (and the required permissions) within apps, and attributing them to either the app itself or one of its embedded libraries. In this way, permission usage by libraries may be better understood.

Motivated by the observation that many libraries obtain significant access to devices through their host apps, the second research thread in this chapter examines the extent to which users can be profiled by these libraries. User profiling by libraries is greatly facilitated on smartphones due to a phenomenon I call *intra-library collusion*. Intra-library collusion is the potential that individual libraries have to aggregate significant user data, by virtue of being embedded within multiple apps, with each app using different sets of permissions.

### 1.5.4 AppScanner

Chapters 3-5 demonstrate that many apps are undesirable for a number of reasons, such as the vulnerabilities they contain, their permission usage compared to functionally-similar apps, and their use and transmission of sensitive user data. However, many users blindly trust the apps they download from app stores, and are thus unaware of the many security or privacy risks that they face.

Chapter 6 demonstrates the feasibility of a system, called AppScanner, which can be used to transparently identify apps of concern on user devices, so that their users can be notified. AppScanner identifies apps installed on smartphones by using machine learning algorithms to understand the network traffic that they

generate. In this way, the list of apps installed on a device may be determined transparently, with no effort required from the user.

AppScanner does not inspect the payloads of unencrypted network traffic, so it is able to identify apps without itself posing any additional risk to privacy. Note that programmatically obtaining the list of apps installed on devices is an operation that is not protected by a permission on Android. Thus, AppScanner does not uncover any data that the Android OS itself actively protects.

AppScanner may be installed at the home (perhaps built into home Wi-Fi routers), corporate, or internet service provider (ISP) level. At the corporate level, for example, AppScanner may be used to enforce a policy relating to the security or privacy characteristics of apps that are permitted on BYOD-devices, if said devices are to access the company network.

## 1.6 Published Work

During my D.Phil., the following pieces of work were peer-reviewed and accepted for publication.

### 1.6.1 Journal Papers

- Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Robust Smartphone App Identification Via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security*, 13(1):63–78, January 2018. [Online]. Available: `https://dx.doi.org/10.1109/TIFS.2017.2737970`

### 1.6.2 Book Chapters

- Vincent F. Taylor and Ivan Martinovic. *Intrusion Detection and Prevention for Mobile Ecosystems*, chapter Attacking Smartphone Security and Privacy, pages 25–64. Taylor & Francis, 2017. ISBN: 978-1-138-03357-3

### 1.6.3 Conference Papers

- Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, EuroS&P '16, pages 439–454, March 2016. [Online]. Available: `https://dx.doi.org/10.1109/EuroSP.2016.40`
- Vincent F. Taylor and Ivan Martinovic. Short Paper: A Longitudinal Study of Financial Apps in the Google Play Store. In *Proceedings of the 21st Financial Cryptography and Data Security*, FC '17, 2017. [Online]. Available: `https://dx.doi.org/10.1007/978-3-319-70972-7`
- Vincent F. Taylor and Ivan Martinovic. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security*, ASIACCS '17, pages 45–57, New York, NY, USA, 2017. ACM. [Online]. Available: `https://dx.doi.org/10.1145/3052973.3052990`
- Vincent F. Taylor, Alastair R. Beresford, and Ivan Martinovic. There Are Many Apps for That: Quantifying the Availability of Privacy-preserving Apps. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '17, pages 247–252, New York, NY, USA, 2017. ACM. [Online]. Available: `https://dx.doi.org/10.1145/3098243.3098266`

### 1.6.4 Workshop Papers

- Vincent F. Taylor and Ivan Martinovic. SecuRank: Starving Permission-Hungry Apps Using Contextual Permission Analysis. In *Proceedings of the 6th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '16, pages 43–52, New York, NY, USA, 2016. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2994459.2994474`

### 1.6.5 Posters & Demos

- Vincent F. Taylor and Ivan Martinovic. DEMO: Starving Permission-Hungry Android Apps Using SecuRank. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS '16, pages 1850–1852, New York, NY, USA, 2016. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2976749.2989032`

## 1.7 Work Done in Collaboration

Parts of the work conducted throughout my D.Phil. was done in collaboration with other researchers. Riccardo Spolaor contributed the idea of using statistical features generated from flows (and approximately 40 lines of code for extracting statistical features from a flow) for the AppScanner system (Chapter 6). Riccardo also did the work on comparing the performance of AppScanner to related work. Mauro Conti, Riccardo's Ph.D. supervisor, provided high-level suggestions for the AppScanner system. Alastair Beresford provided guidance for interfacing with the Device Analyzer project and provided high-level suggestions used in Chapter 5. All other output is the result of my own work (except where other work is cited) with guidance from my supervisor, Ivan Martinovic.

## 1.8 Thesis Roadmap

Chapter 2 gives background on the problem area by surveying directly related work. It highlights existing gaps in the literature and my contributions to addressing those gaps. Chapter 3 starts by presenting the results of a longitudinal study on app evolution, in terms of vulnerabilities, permissions, and library empowerment. Chapter 4 proposes a framework for identifying functionally-similar apps and suggesting more preferable alternatives that can be used to replace them. Chapter 5 examines the risk landscape on real-world devices and performs several measurement studies to understand the cumulative effect of using multiple apps on devices. It also quantifies improvements that can be made if undesirable apps are replaced. Chapter 6 details a system that can transparently identify apps running on user devices, so that they can be notified of undesirable ones. Chapter 7 concludes by discussing my key findings and highlighting future work.

# CHAPTER 2

# Background

*Everything has been thought of before, but the problem is to think of it again.*

– Johann Wolfgang von Goethe

This chapter highlights the most closely related work on understanding and improving the security and privacy of the Android app ecosystem. It starts by exploring prior work on the analysis of apps to detect malware or undesirable behaviour. Since permissions play a major role in the defence-in-depth architecture provided by Android, I continue by assessing work on examining permission usage within apps. Then, since undesirable apps contribute negative effects to users and the app ecosystem as a whole, the chapter explores prior work on measuring the real-world impact of apps. The chapter ends by reviewing related work on network traffic analysis on smartphones. Network traffic analysis has been used before as a means of attacking user privacy, but as I later show, can also be a valuable tool for improving user security and privacy.

Parts of the material in this chapter formed the basis of a book chapter in *Intrusion Detection and Prevention for Mobile Ecosystems*, edited by Georgios Kambourakis, Asaf Shabtai, Constantinos Kolias, and Dimitrios Damopoulos [144].

## 2.1 Analysis of Apps

Enck et al. [70] took an early look at the security characteristics of smartphone apps. The authors developed the `ded` decompiler and used it to analyse 1,100 popular Android apps. The analysis showed that apps regularly (mis)use device identifiers and generally contained advertising and analytics libraries. At the time of the study (2011), the authors did not find evidence of malware or vulnerabilities in apps.

Viennot et al. [153] later performed a very large-scale study of the Google Play Store using a tool they developed, called PlayDrone, to analyse over 1.1 million apps. The authors discovered that many apps were insecurely embedding authentication tokens and that simple app decompilation using readily available tools could compromise these secret tokens. The authors also measured the prevalence of cloned apps and found that 25% of free apps were clones. This snapshot of the app store was a useful first step, but the authors did not identify long-term trends in the app store, or provide analysis of more critical characteristics of apps, such as permission usage or vulnerabilities.

### 2.1.1 By Permission Usage

Permission-based access control abstracts the capabilities of a piece of software into permissions [1]. Access to a permission-protected resource is permitted by the OS only if the software seeking access has been granted the permission that protects the resource. The number of permissions supported by the Android OS varies depending on its version.

Wei et al. [161] characterise the evolution of the permission system on Android. They analyse changes in the permission model since Android's commercial release in 2008. The authors find that permission evolution typically offers developers access to new hardware features as opposed to offering more fine-grained control to users. As it relates to permission usage within apps, Vidas et al. [151] observe that some apps use permissions that are not required for the app to properly function. To mitigate this problem, they propose a tool that can be used by developers to identify such extraneous permission usage. Since permissions are central to the protections provided by the Android operating system, apps using extraneous permissions, or permission-hungry apps in general, are even riskier if they also contain vulnerabilities.

### 2.1.2 By Vulnerabilities Contained

Fahl et al. [72] studied the extent to which Android apps were susceptible to man-in-the-middle (MITM) attacks by virtue of insecure network programming. The authors developed a tool called MalloDroid and used it to analyse 13,500 apps. They found that approximately 8% of apps were potentially susceptible to a MITM attack.

Along similar lines, Egele et al. [67] developed an approach to check for improper usage of cryptographic methods within apps. The authors checked for bad practices such as the use of ECB mode for encryption, using a non-random IV for CBC encryption, using constant salts and keys, and the like. By analysing over 10,000 apps, the authors found that approximately 88% of apps made at least one mistake in their use of cryptography.

Other authors analysed whether apps were properly securing their constituent components. Jiang and Xuxian [93] measured the prevalence of unprotected `ContentProviders`. From an analysis of over 62,000 apps, the authors found that apps were susceptible to private information disclosure and data manipulation in 2.0% and 1.4% of cases respectively. Across app stores with millions

of apps, tens of thousands of apps are potentially vulnerable to these attacks.

Similarly, Li et al. [99] presented IccTA, a static taint analysis system that can detect privacy leaks between components in Android apps. IccTA was able to identify 2,395 leaks across a sample of 15,000 apps downloaded from the Google Play Store. Along similar lines, Lu et al. [103] presented CHEX, a tool for the static analysis of Android apps to detect component hijacking vulnerabilities. Using their tool, the authors were able to find 254 potential vulnerabilities in a sample of 5,486 apps. The authors argue that the low median execution time of their tool means that it can be useful at app-store-scale.

While the aforementioned authors either identify vulnerabilities or provide tools for their detection (or both), they fail to give insight on particular trends in vulnerability evolution across app stores. Thus, it is unclear whether app developers are getting better at writing safer code or if newly deployed/updated apps continue to be vulnerable.

### 2.1.3 By Other Metrics

Carbunar and Potharaju [51] did a six-month longitudinal analysis of apps in the Google Play Store, but focused on developer publishing and pricing behaviour. The authors observe that price increases for paid apps are typical during updates and that, at most, 50% of apps are updated across app store categories. This work captures developer behaviour at a high level, but does not assess the effect of security and privacy related changes when apps are updated.

Individual apps only begin to markedly affect the app ecosystem if they are downloaded and installed on many devices. Thus, unsavoury app developers sometimes use fraudulent techniques to increase the app store search ranking of their apps in an attempt to garner more downloads. Chen et al. [53] propose an approach to detect such ranking manipulation. The authors exploit anomalous ranking changes to identify artificially promoted apps and identify colluding parties. In a similar vein, Xie and Zhu [165] present AppWatcher, a system that

identifies artificially promoted apps. Both sets of authors highlight the problem of collusion that aims to improve the ranking of apps.

From these studies, we know that unsavoury developers artificially increase the popularity of their apps and will sometimes achieve high search rankings and consequently more downloads. Once an app is installed on a device, however, all is not lost since the permission system offers the next line of defence.

## 2.2 Limitations of the Permissions System

Android protects sensitive user data using an access control model that relies on users explicitly granting permissions to apps. As explained in Chapter 1, users are required to grant all permissions at install-time or selectively grant permissions at run-time, depending on the version of the Android operating system they are using and whether the app they are using targets at least API 23, the minimum API level that supports run-time permissions.

Felt et al. [78] studied whether users paid attention to permission warnings at install-time. Only 17% of users paid attention to permissions and a smaller 3% of users were able to correctly answer follow-up comprehension questions about permissions. In a similar study, Felt et al. [76] surveyed smartphone users to understand how they perceived risks from apps. Users demonstrated little concern for sensitive information leaks by apps, such as the sending of their location to remote servers. Similarly, Kelley et al. [94] observe that users are generally unaware of security risks coming from mobile apps.

One solution to the problem of users blindly granting permissions at install-time is requesting permission from users at the point during app usage where the permission is about to be used, i.e., run-time permissions. Run-time permissions serve as a more obvious indication to users that they are about to allow an app to have access to sensitive data. While this gives added power to the user, Eling et al. [68] found that 40.4% of users accepted fine-grained run-time per-

missions (that were both intrusive and unnecessary) for minimal reward. This highlights a broader problem in that users sometimes ignore privacy policies, permissions, and terms of agreement altogether [58]. Indeed, users are known to disclose personal information for short-term benefits [33, 34], and their actual behaviour often diverges from their intentions when it comes to protecting their privacy [32, 115].

To make matters worse, run-time permissions are only invoked if the apps themselves are designed to make use of them. It remains unclear, however, whether app developers are sufficiently motivated to make the switch. As Bogdanas [44] shows, it can require significant developer investment to identify where their apps use permissions and where they should insert permission requests. The challenge is also highlighted by Fang et al. [74] who observe that 70% of apps do not appropriately catch permission-related exceptions. The authors propose a tool, called revDroid, which can assist in identifying unwanted side effects of the introduction of run-time permissions. Interestingly, the authors found that library developers and malware developers were better at handling run-time permissions than ordinary app developers.

Several authors proposed strategies to improve the permissions system and user privacy on Android. Beresford et al. [42] propose MockDroid, a modified Android OS that allows a user to fake the data returned from sensitive API calls. MockDroid allows users to effectively revoke permissions that have already been granted, while minimising negative effects such as app crashes. Similarly, Hornyack et al. [91] propose AppFence, a system that protects sensitive data from permission-hungry apps. Like MockDroid, AppFence also substitutes sensitive data with mocked data, but in addition to that it also blocks network transmissions if they contain data that a user specified should not leave the device. Using a sample of 50 apps, the authors find that permissions granted to an app can be reduced in 66% of cases without any side effects.

Barrera et al. [41] use self-organising maps to analyse permission-based secu-

rity models. Using a sample of 1,100 apps, the authors use their methodology to identify improvements that can be made in the Android permission model. In a similar vein, Conti et al. [62] propose CRePE, a system that allows fine-grained policy enforcement. In contrast to coarse-grained policy enforcement, CRePE can impose security policies depending on location, time, temperature, noise, or light. Subsequently, Zhou et al. [175] developed TISSA, a system that provides a 'privacy mode' on Android. TISSA allows a user to specify fine-grained access controls to apps. The access controls can be dynamically adjusted depending on the current context that the user is in.

The aforementioned pieces of work highlight shortcomings in the Android permission system and strategies that have been employed to provide improvements. In order to better protect users, however, it is important to also understand how permissions are used within the apps themselves.

## 2.3 Permission Usage Within Apps

On Android, apps and all their necessary resources are packaged into a single `apk` file. As a result, apps and any libraries they use are tightly integrated, and any permissions granted to apps are available to embedded libraries as well. In what follows, I highlight related work on understanding permission usage in apps and their libraries.

### 2.3.1 By the Apps Themselves

Enck et al. [71] propose Kirin, a lightweight certification system that is designed to mitigate malware. Kirin uses rules to statically match undesirable properties in apps. Using a sample of 311 apps, their system found 5 popular apps that implemented dangerous functionality. In a similar vein, Sarma et al. [131] analyse permission usage within apps as well as the permissions used by apps in the same app store category. They argue that machine learning techniques can be used to

identify malware in this way, by looking at the rareness of certain permissions. The authors claim a malware identification rate of up to 81% at a warning rate of 8%. Similarly, Peng et al. [122] use probabilistic models at the centre of their risk scoring scheme and outperform existing approaches. Gorla et al. [85] propose the CHABADA framework and cluster apps by description topic to identify outliers as it relates to permission usage.

Other authors leverage permission request and usage patterns to detect potentially malicious activity. Chia et al. [57] observe that some insidious app developers attempt to mislead users into granting permissions. Papamartzivanos et al. [118] detect privacy violations in apps using a cloud-based crowdsourcing approach. Zhang et al. [168] developed a framework, called VetDroid, which identifies information leaks by leveraging permission use analysis. Along similar lines, Frank et al. [82] examine permission request patterns and use a clustering-based approach to determine that low-reputation apps deviated from the permission request patterns of high-reputation apps.

Wijesekera et al. [164] study the contextual integrity of permission usage. That is, they examined the usage of protected resources when users were not expecting it. The authors found that 80% of users would have wanted to block more than one-third of permission requests based on the context under which the permissions were being used. In a similar vein, Wang et al. [156] use text mining on decompiled apps to understand the purpose of permission use within them. The authors achieved approximately 80% accuracy by using static analysis to infer the reason for the use of location permissions and 94% accuracy for inferring why apps accessed a user's contacts. Using dynamic analysis, the authors improved the accuracy of inferring the purpose of location data access to 90%.

Several authors leveraged natural language processing (NLP) to identify whether app permission usage could be inferred from app descriptions. Pandita et al. [117] proposed WHYPER, a system that is able to identify sentences

from an app's app store description that could suggest the need for a particular permission. Qu et al. [127] later outperformed WHYPER with a system called AutoCog, and the authors conclude that there is a general problem of "low description-to-permission" fidelity. Similar in performance to WHYPER is ACODE, a system proposed by Watanabe et al. [159]. These approaches all use coarse-grained information from app descriptions to attempt to identify apps using unnecessary permissions.

Along similar lines, Chen and Zhu [56] propose DroidJust, an automated analysis framework that can identify privacy leakage from Android apps. DroidJust infers justification of sensitive data transmissions by examining the functionality of the app itself and the context under which sensitive data is transmitted.

The danger of apps using unnecessary permissions, and the granting of unnecessary permissions by users in general, is cause for alarm. However, this is exacerbated by the fact that libraries embedded within apps have access to these granted permissions as well and may exploit them for profit. Additionally, while prior work has looked at identifying apps with anomalous permission request patterns, it remains unclear the extent to which these apps can be substituted, if necessary, for others with more reasonable request patterns.

### 2.3.2 By Embedded Libraries

Many apps are made available to users free of charge, with the developer embedding ads in the app to earn revenue. If an ad network is able to tailor ads specifically to a user's interests, they are more likely to generate a positive response to the ad from that user. As a result, ad networks use various means to obtain a clearer picture of user interests through profiling. To better profile users, ad networks typically leverage permissions that have been granted to apps to access data useful for profiling from a device [88].

Book et al. [45] perform a longitudinal analysis of permission requirements in Android ad libraries. They did this by downloading, disassembling, and

analysing 114,000 apps from the Google Play Store. The authors found that there is an increase in the number of libraries requiring access to dangerous permissions over time. They argue that most of the increases in permission usage do not come from making the ad libraries more efficient, but rather seem to facilitate the extraction of additional personal data. This work uncovers an alarming trend in ad libraries using more permissions over time, but fails to examine permission usage increases in the apps that embed these ad libraries. This is critical, since the ad libraries are unable to leverage permissions unless the app itself declares (and has been granted) these permissions.

Grace et al. [88] developed AdRisk and used it to investigate the privacy risks posed by ad libraries. The authors found that most ad libraries collect sensitive user data, such as their location. Additionally, the authors found evidence of invasive data collection by ad libraries whereby some ad libraries were collecting a user's call logs. To make matters worse, other ad libraries dynamically downloaded and executed code, adding security risks to devices.

In a similar vein, Stevens et al. [139] examined user privacy in ad libraries. The authors uncovered the use of 'permission probing' by libraries to determine whether permissions were granted to apps before attempting to access related permission-protected APIs. Some of the permissions probed for by libraries were not even mentioned in the ad library SDK documentation. Thus, there is evidence to suggest that some ad libraries surreptitiously attempt to gain access to additional sensitive user data.

The danger, however, not only comes from libraries accessing permissions through standard Android API calls. Book and Wallach [46] identify direct collusion between app developers and libraries, whereby sensitive user data is directly passed from an app to an ad library through internal ad library APIs. From a study of 114,000 apps, the authors found that the popularity of an app correlated with its propensity for privacy leakage. They argue that the marginal increase in revenue contributed by invasive techniques, when taken across a large

set of users, gives financial incentives to app developers for violating user privacy.

To counter the problems of libraries being able to access the permissions granted to apps, several authors propose privilege separation mechanisms. Shekhar et al. [136] proposed AdSplit, an Android extension that allows apps and libraries to run as separate processes (with separate UIDs and thus permissions). The AdSplit framework automatically separates library code from app code by decompiling the app and replacing the library with a 'stub library' that presents the same APIs. This stub library then interacts with an advertisement service running on the device. Along similar lines, AFrame provides features of process and permission isolation, but also provides display and input isolation [167].

Finally, Pearce et al. [119] propose AdDroid. AdDroid eliminates the need for ad networks and ad libraries completely. Instead, the Android platform directly supports advertising. The AdDroid system handles the fetching of ads, thus obviating the need for ad libraries that are bundled within apps. While these approaches address the problem of privilege separation between apps and libraries, it remains unclear whether they can be successfully implemented in practice. For one, disentangling ad libraries from apps makes it easier for users to disable advertising on their devices altogether. For example, a user can simply disable the AdDroid or AdSplit service on their device to prevent advertising. A lack of advertising can harm the app ecosystem, since many app developers are motivated to publish apps for free, with the hope of monetising their apps through ad revenue [98].

## 2.4 Assessing the Real-World Impact of Apps

With security and privacy concerns on devices coming from many directions, it is increasingly important to measure the real-world impact of these security and privacy concerns.

### 2.4.1 Using Smartphone Data

Ferreira et al. [80] present Securacy, an app that analyses users' security and privacy concerns with Android apps. The authors first do a study to understand users' concerns about app behaviour. Securacy works by allowing users to specify what permissions they find concerning and then notifying them when an app that they install requires these permissions. The tool also identifies the destination of connections made by apps and determines whether the connection is encrypted using TLS. From data retrieved from 216 users, the authors determined that the majority (65.5%) of apps communicated with servers in North America regardless of where the user was based. The authors also discovered that just over a half (56.4%) of app connections are insecure, i.e., not using TLS. Interestingly, users' concern for permissions had a bimodal distribution with 70 of 218 participants not caring about any permissions at all, while 59 of 218 were concerned about all permissions. Users were most apprehensive about giving access to contacts, messages, call history, their profile, and user account information.

Along similar lines, several authors investigated tracking sensitive data flows off a device. In seminal work, Enck et al. [69] propose TaintDroid, a dynamic taint tracking system that can identify sensitive sources of data and the destinations that they are leaked to. TaintDroid provides real-time analysis and feedback, and only incurs 14% CPU overhead on devices. As it relates to static taint tracking, Arzt et al. [37] present FlowDroid, a highly precise static taint analysis tool. FlowDroid improves on precision by leveraging an accurate model of the Android app lifecycle. Similar to FlowDroid, Wei et al. [160] propose Amandroid, which outperforms prior work on static taint analysis. Amandroid is able to handle inter-component data flows and can thus do taint tracking within app components or across apps.

In a tangential direction, Zhou et al. [172] present work on inferring privileged

information from an Android device by using a malicious app that requires no permissions. The app obtains data by accessing public resources on a device and makes inferences based on these resources. For example, Android makes available some public directories and files that can be read by apps without requiring any explicit permission. The authors' malware takes advantage of files such as the network traffic statistics files for each app, to understand how much data an app has sent. By polling these files at a sufficiently high frequency, malware can identify network traffic flows and their sizes whenever they happen. Using this information, the authors are able to infer when a user sent a tweet or what illnesses a user researched using a medical app. Using publicly readable address resolution protocol (ARP) data from a device, the authors could also infer a user's location based on the service set identifier (SSID) of the Wi-Fi network that the user was connected to.

Similar to public readable files, the list of installed apps on a smartphone does not require special permissions to access. Seneviratne et al. [134] leverage this information to predict user traits such as religion, marital status, whether a parent, native language, and countries of interest. During their study, the authors found that more than 50% of the users in their test set of over 200 users had more than 20 apps installed. Thus, the list of apps installed on the average device is a large and potentially rich source of data useful for predicting user traits. The authors built their classification system using the Weka machine learning tool [89]. For user traits such as native language, country, and religion, the authors achieved over 85% precision on their test set depending on the threshold condition. Other traits, such as whether the user was a parent, or a user's marital status, had very high precision (more than 95%). The proposed system suffers is in its recall, whereby the classifier had many false negatives. This is understandable, since not everyone with a particular trait will have a smartphone app to match it. Conversely, if a person has a particular app typical of a certain trait, it is likely that they also have that trait.

Seneviratne et al. [133] present a study on the use of tracking libraries in apps. The authors focus mainly on the extent to which tracking happens in paid apps. Since paid apps typically operate under a different business model and (usually) do not contain ads, one might expect that they would not have tracking libraries either. The authors discovered that 60% of paid apps are still connected to trackers that collect personal information compared to approximately 85% for free apps. They further discovered that 20% of paid apps had more than three trackers. Additionally, the authors found that eight of the 22 top trackers tested accessed the user's current location. Unique identifiers from the Android device were collected by 17 of the 22 top trackers. The authors also found that 50% of users were connected to more than 25 trackers and 20% of those same users were connected to more than 40 trackers. The prevalence of trackers in paid apps, apps that typically do not use advertisements, highlights the fact that profiling and harvesting information about users remains of great value to app developers even outside of the context of delivering ads.

In a similar vein, Leontiadis et al. [98] examine a way of balancing privacy leakage in the Android app ecosystem. They found that 77% of the 50 most popular free apps in the Google Play Store were ad-supported. The authors observed that free apps, on average, "request 2-3 additional permissions compared to paid applications of the same category". Permissions were requested by 73% of free apps compared to 41% of paid apps. The authors acknowledge that advertisements are needed to maintain the current free app business model, and argue that allowing users to withhold much private information may lead to ineffective advertisement targeting that may eventually lead to an unsustainable app market. Instead, what the authors propose is a strategy that decouples privacy control between the app and the advertisement components. They contend that this separation would allow users to commit to different data sharing agreements with the different parties. Additionally, the advertisers themselves could then be subject to separate (and potentially stricter) privacy regulations.

In related work, Tongaonkar et al. [149] showed how it was possible to understand mobile app usage patterns using in-app advertisements. Their work hinges on the fact that ad libraries rely on unique identifiers to identify the publisher to the ad network so that they can be paid. By observing how these unique identifiers are passed in HTTP requests, the authors can identify the usage of a particular app. By analysing a sample of 55,000 apps from the Google Play Store, the authors observed that the vast majority of apps have only one ad library, while fewer apps had two or more, with some apps having more than five ad libraries.

### 2.4.2 Using the Device Analyzer Project

Device Analyzer is a research project from the University of Cambridge that began in 2010. It is concerned with capturing usage data from smartphones [154]. The aim of the project is to collect data to facilitate the answering of a wide variety of research questions. Contributors participate by downloading the Device Analyzer app and running it on their smartphones. The Device Analyzer app is designed to run on Android 2.1 or higher (more than 99% of devices connecting to the Google Play Store). The app collects a wide variety of data from a device but anonymises the user as well as any user-identifiable data such as nearby Wi-Fi SSIDs, phone numbers, and the like. The app is designed to be minimally invasive to the user and to consume as little resources as possible. As of September 2017, the Device Analyzer dataset has over 100 billion records of smartphone usage data from over 31,134 contributors to the project [12].

The developers of Device Analyzer argue that their system is the "largest, most detailed dataset of Android smartphone usage publicly available to date" [155]. The data gathered allowed the developers to observe particular user behaviour such as the majority of users keeping their Wi-Fi off most of the time and other behaviour such as a particular user discharging their battery by 38% in one day and 298% another day. Device Analyzer aids in the study of

user-device interaction by collecting data points such as the number of calls and texts received per day. The data I am interested in from the Device Analyzer dataset is lists of apps installed on devices. With knowledge of the apps installed on devices, risks and improvements on real-world devices can be quantified at large scale.

While the Device Analyzer dataset is large and useful, it also has limitations. The Device Analyzer app will likely be installed by more technical users since the Device Analyzer project is mainly advertised through scientific publications. Technical users may use some apps that would not likely be used by the general public. Technical users may also be more adept at choosing apps that are safer from a security and privacy perspective. If the latter is the case, this may cause the results obtained by leveraging Device Analyzer data to actually understate the severity of any observed problems. In any case, the Device Analyzer dataset is arguably a good starting point for real-world analysis due to its sheer size and the worldwide coverage of its install-base.

## 2.5 Network Traffic Analysis

Network traffic analysis on smartphones takes many forms, and has been used in the literature to invade user privacy [172][60]. In this thesis, I use network traffic analysis in a positive way, i.e., to uncover the existence of potentially undesirable apps on user devices. Network traffic analysis on smartphones is related to traditional network traffic analysis on workstations but there are several important nuances that must be accounted for. In what follows, I introduce related work on traditional network traffic analysis, and then summarise related work on smartphone traffic analysis.

### 2.5.1 On Traditional Workstations

Danezis and Clayton [64] introduce traffic analysis succinctly by presenting a survey on various traffic analysis tools and techniques that have been used throughout history to achieve various objectives. They begin by discussing how traffic analysis was used in a military context in the Second World War to assess the size of enemy air forces. The authors highlight traffic analysis from those early periods up to more current work that has uncovered weaknesses in the encrypted channels of SSH and SSL. They survey the ways in which web privacy can be violated and the way that network devices can be identified and mapped. For SSL in particular, the authors highlight that the key weaknesses come from inadequate padding and concealment which reveal the 'shape' of the encrypted traffic. This shape can potentially be used to identify common actions across users and is especially efficient when the activity space is limited. This points to the general utility of traffic analysis for uncovering course information and general patterns in both encrypted and unencrypted traffic.

Nguyen and Armitage [114] go a step further and survey more specific techniques for internet traffic classification using machine learning. They examine how both supervised and unsupervised learning may be used to identify or aggregate traffic. Their research lends support to the idea that machine learning can be used as a means of automating traffic analysis, allowing it to be used on large scales.

Several authors examined traffic analysis by considering the threat model of a strong adversary such as a government or large ISP. Raymond [128] considers adversaries able to capture entire flows of network traffic through communication links and highlights traffic analysis attacks on protocols. Liberatore et al. [100] showed that it is possible to identify web pages from encrypted HTTP requests and responses using traffic analysis. Along similar lines, Herman et al. [90] produced a better system that relied on text mining techniques leveraged with the

normalised frequency distribution of packet sizes from IP traffic. They achieved up to 97% accuracy when classifying HTTP requests. Panchenko et al. [116] went a step further and correctly identified web pages even when onion routing anonymisation techniques, such as Tor, were used.

Cai et al. [50] achieved 90% accuracy in identifying a victim's use of a particular website in the presence of an application-level defence system called HTTPOS [104] running over Tor. These techniques, while useful, all consider the adversary to be interested in HTTP web browsing traffic, and in particular, identifying what web pages are being visited. Web page traffic is full of very rich information since web pages are composed of a number of smaller documents. For example, a typical web page may feature an HTML document, one or more stylesheets, one or more scripts, and one or more images. All these page assets require a separate HTTP transfer and gives the adversary additional side-channel information that can be used for profiling/fingerprinting the page. Additionally, these assets usually have greatly varying sizes (HTML pages are small, images are large) which gives further coarse information to an adversary intending to identify web pages.

Smartphone apps, though they routinely communicate using HTTP requests, seldom receive actual web resources in the corresponding HTTP responses. Usually, HTTP is used simply to send text-based queries to APIs using standard formats such as JSON and XML. The corresponding response is usually also text-based but may contain binary resources. For this reason, smartphone traffic delivers less side-channel information than standard web traffic. On these grounds, it is expected that the traditional traffic analysis methods presented above will fail or under-perform when applied to smartphones.

Chen et al. [55] give a report on side-channel leaks observed from real-world, high-profile web applications through their encrypted traffic. Side-channel leaks are common in web applications and mobile applications because of fundamental characteristics such as "stateful communication, low entropy input for better

interaction and significant traffic distinctions". The authors gather their side-channel information in the form of network traffic flows, i.e., the number of bytes sent and their direction. They conceptualise an attack as an ambiguity-set reduction process and use their knowledge of the workflow (finite state machine) of the web application to narrow down the possible user choices that led to a particular traffic flow. The authors did not rely on machine learning or data mining techniques, but instead made inferences based on traffic flow vectors and the fact that the applications had specific workflows which could be used to easily train a rudimentary classifier. They argue that strategies such as padding packets, faking packets (to add noise) and merging/splitting application workflows exist, but may not be generalisable enough to be useful out-of-the-box for all web applications. While the authors' system works, it is mostly suitable for scenarios where an attacker wants to profile a particular web application and takes the time to map its workflow and corresponding traffic movement. Their approach is not suitable for traffic analysis on smartphones because it does not scale well. The approach would require a finite state machine mapping of the universe of apps to be able to detect apps from their network traffic. This significant overhead for training is not desirable and may well be intractable if the system were to be scaled up.

Dyer et al. [66] demonstrate why efficient countermeasures against traffic analysis fail. In their work, they argue that none of the countermeasures that were studied were effective against traffic analysis, including the countermeasures specified by the TLS, SSH, and IPSec RFCs. The authors further argue that hiding packet lengths alone is not sufficient to frustrate traffic analysis. A classifier using only the very coarse feature of total bandwidth still performed with up to 98% accuracy in classifying HTTP traffic.

Several schemes, such as Buffered Fixed Length Obfuscation (BuFLO), exist to hide coarse features such as bandwidth. BuFLO works by sending fixed size packets at fixed intervals. Dyer et al. observe that 27% accuracy in identifi-

cation can still be achieved if the BuFLO scheme fails to hide total time and total bandwidth of a flow of traffic. If BuFLO is tuned to consume a bandwidth overhead of 400%, identification accuracy then drops to 5%. On the basis of this evidence, a system like BuFLO may be effective in properly masking network flows in some settings. However, a bandwidth overhead of 400% is naive and would contribute significant (perhaps unreasonable) overhead to a communication channel for typical applications. On smartphones with limited batteries, processing, and bandwidth, BuFLO is impractical except in the most niche scenarios.

Muehlstein et al. [113] show that browsers, applications, and operating systems can be identified through analysis of HTTPS traffic. The authors generate features from network traffic using the concept of network flows. Miller et al. [110] was able to identify individual web pages within websites using traffic analysis with up to 90% accuracy. Their work hinged on two assumptions: (i) they assume that all traffic flows within a burst of traffic belongs to the same webpage; (ii) they assume that they can rely on multiple bursts to build the model of a website by leveraging Hidden Markov Models (HMM). While these assumptions may be reasonable on workstations, they do not usually hold on smartphones. On smartphones, the existence of multiple installed apps running simultaneously (and making network requests simultaneously) frustrates traffic analysis that assumes that a burst of network traffic belongs to a single entity.

### 2.5.2 On Smartphones

Falaki et al. [73] was among the first to look at smartphone traffic. They did this by studying network logs from 43 users to gain a better understanding of the types of traffic that could be expected from smartphones. They found that 81% of the traffic went to port 80 (HTTP) or port 443 (HTTPS). The authors identify this as a trend for smartphone apps to tunnel their traffic through these protocols although what the apps are transmitting/receiving would not be considered a

typical HTTP request/response. By analysing TCP flows, the authors observe that smartphones typically make many small data transfers. Small transfers are consistent with apps doing things such as loading recent updates from a server, checking in with a 'ping', fetching ads, or pulling new content as a result of user interaction. Interestingly, the authors observed that the use of TLS does not add much overhead beyond what is already caused by TCP. Thus there is little performance penalty coming from the use of TLS. Given that there are also free certificate authorities, such as *Let's Encrypt* [18], there is no longer a financial barrier to app developers deploying TLS across their apps.

Wei et al. [162] present ProfileDroid, "a multi-layer system for monitoring and profiling apps". They evaluated their system on 27 free and paid apps. From their analysis, the authors were able to identify a lack of transparency, whereby some apps used resources that they did not declare that they needed to access. At the time of their research (2012), the authors discovered that most mobile network traffic was unencrypted. It is not possible to say whether this was representative of apps at that time, since the authors chose an insufficient sample size in a Google Play Store they reported as having 550,000 apps at the time. The authors also observed that free apps have a cost. They showed that free apps may end up costing a user more in terms of data consumed than paid versions of the same apps. This comes from the fact that free apps tend to have increased advertising/analytics traffic and this may incur non-trivial costs, especially to users with limited data subscriptions. They quantify one app in particular, Angry Birds, and find that the free version sends 13 times more traffic than the paid version. ProfileDroid is a useful step towards an automated framework for the profiling of apps but unfortunately does not focus in great detail at the network level. Indeed, they authors only focused on metrics such as total bandwidth used per app, traffic in/out ratio, and the HTTP/HTTPS traffic split of apps.

Along similar lines, Dai et al. [63] took a concrete step towards automatically

fingerprinting Android apps with a system called NetworkProfiler. The authors presented an automated approach to identifying Android apps using dynamic methods to elicit traffic flows from apps. They use a user interface (UI) fuzzing technique to automatically try different execution paths in an app while the network traffic is being monitored. The authors parse HTTP GET methods and build state machines to understand the various query keys and parameters that are sent by apps to the server they are communicating with. From these query keys the authors build profiles about particular apps, which are later used to identify the apps from traffic dumps. This technique is a good start for fingerprinting apps but suffers from the fact that it only works on unencrypted traffic. In a more privacy-centric future, the usefulness of NetworkProfiler is expected to decrease. Indeed, the most popular messaging app, WhatsApp, does end-to-end encryption by default [125] with many other popular apps following suit. Any app that sends/receives only encrypted traffic is immediately immune to profiling by NetworkProfiler.

Qazi et al. [126] presented a framework for identifying apps using network flows. Their system, called Atlas, uses crowdsourcing to obtain ground truth. The authors achieve 94% accuracy for a corpus of 40 apps. It is unclear, however, if Atlas maintains good accuracy as the number of apps is increased. Le et al. [95] use crowdsourcing as well, but for the fine-grained collection of network data. Their system is called AntMonitor.

Stöber et al. [140] go in a tangential direction and propose a scheme to identify entire smartphones based on characteristic patterns in the background traffic coming from devices. Their work hinges on the fact that different devices have different sets of apps with different configurations. They motivate their work by showing that 70% of smartphone traffic comes from apps running in the background doing things such as synchronising with a server, checking for updates, and other maintenance tasks. They argue that this background traffic is unique per device and may be leveraged to make a fingerprint for an entire

device. To fingerprint a device, the authors target 3G transmissions coming from the device and argue that they can be realistically intercepted and demodulated to obtain side-channel data, such as the number of bytes transmitted and packet timing. Based on their testing, the authors determined that approximately 15 minutes of captured traffic was sufficient to obtain a classification accuracy in excess of 90%. The system, however, suffers from the fact that approximately six hours of traffic is needed for reliable fingerprinting.

Mongkolluksamee et al. [112][111] use communication patterns and packet size distributions for identifying app traffic. The authors report an F-score of 95% but they only consider five apps so it is unclear how scalable their approach is. Additionally, the authors fail to collect perfect ground truth. In a similar vein, Alan and Kaur [35] present a system for app identification using TCP/IP headers. Using packet size information from the first 64 packets generated when an app is launched, the authors achieve 88% accuracy. The authors find that accuracy decreases when different devices are used for training and testing and when time has passed between training and testing. Similarly, Wang et al. [157] use packet-level traffic analysis to fingerprint and identify 13 arbitrarily chosen apps. Due to a small sample size (and arbitrarily chosen apps), it is unclear whether their approach scales well in the real-world.

Conti et al. [60][61] take a step away from fingerprinting Android apps as a whole and instead examine the extent to which they can identify specific actions that a user has performed within an app. The authors classify specific actions by leveraging hierarchical clustering and supervised learning on the network flows that are observed from each app following particular actions. Given that the authors use only packet direction and size from a flow, their system works in the presence of TLS. The authors achieve 95% accuracy for most of the actions that were considered. Their system performs well, but it is not known how the system performs when a larger number of apps is used. Indeed, classification becomes harder when there are more classes for the classifier to choose from and

the authors only analysed three apps and up to eight actions from each. Finally, the authors' approach is not very scalable since specific actions to profile have to be chosen within each app. Consequently, there is no way to automate the fingerprinting process. This motivates the need for a highly-scalable system that is able to fingerprint entire apps in an automatic way.

## 2.6 Summary

From the examination of related work, it is clear that users face substantial threats to their privacy and security as a result of the apps that they install and use on their devices. These threats exist even when the protections provided by the Google Play Store and the Android operating system itself are working properly. Secure app development is hard since the developer needs to be perfect in the way their code is written and in other defensive techniques, while the attacker only needs one mistake to have a foot-hold from which to launch an attack. Moreover, given the failure of Android to separate the privileges of apps and their embedded libraries, additional risks may be introduced into otherwise secure apps by libraries; black-boxes of code that promise the easy deployment of advanced functionality.

This chapter has identified gaps in our understanding of security and privacy in app ecosystems that this thesis is designed to shrink. It is unclear how vulnerabilities and permission usage in Android apps have evolved over time (Chapter 3). An understanding of the extent to which users can obtain less risky apps to replace the ones they use is also lacking (Chapter 4), as is an understanding of the emergent risks coming from having multiple apps installed on a device (Chapter 5). While traffic analysis has been used previously for attacking user privacy, it is unclear whether it can also be used for transparently and unobtrusively helping users to make their devices safer (Chapter 6).

# CHAPTER 3

## LongLook: Understanding App Evolution

*The more things change, the more they stay the same.*

– Jean-Baptiste Alphonse Karr

This chapter starts by showing that across the Google Play Store, developers are more likely to add permissions (as opposed to remove permissions) to apps as they are updated. It quantifies for the first time, the number and types of permissions that are being added to (and removed from) apps and shows that free apps and popular apps are more likely to facilitate privacy erosions as they are updated. These insights were the result of analysing three years of app metadata collected periodically from the Google Play Store.

The chapter later addresses the unstudied areas of vulnerability evolution in apps and the evolution of the privacy erosion caused by libraries that are embedded within apps. This study was facilitated by 30,000 apps that were collected over a two-year period. Apps are seen to become more vulnerable as they are updated. Additionally, libraries are seen to be able to access more

sensitive data as apps are updated to use new permissions.

Overall, I demonstrate that there is a trend of apps having increased access to user data, while at the same time being more vulnerable to attack. If this trend continues, the utility of the permission-based access control model will decrease over time, since more and more apps are asking for increasing numbers of permissions. Moreover, more apps asking for permissions may lead to users being conditioned into accepting permissions. The insights from this chapter highlight the need to rein in permission usage by apps, as well as demonstrate that vulnerabilities are being introduced into apps through app updates.

Only apps available in the Google Play Store were used in this study, but prior work has shown that unofficial marketplaces usually contain greater incidences of undesirable apps such as malware, cloned apps, and greyware apps [174, 87, 84, 171]. For this reason, the results presented in this chapter may actually represent a conservative estimate of the threats coming from apps across the entire Android ecosystem as a whole.

Most of the material in this chapter formed the basis of a paper published at the ACM Asia Conference on Computer and Communications Security (ASIACCS '17) 2017 [146]. Tools and techniques presented in this chapter were also used in a 'case-study' paper on financial apps, published at Financial Cryptography and Data Security (FC '17) 2017 [145].

## 3.1 Permission Usage Evolution

Permissions form one of the main lines of defence on Android by allowing access to sensitive user data to only those apps that have been granted the appropriate permission. Thus, an understanding of privacy erosion by apps over time can be informed by permission usage patterns in apps. Moreover, trends in the usage of new permissions by apps can pinpoint areas worth focusing on when seeking to preserve user privacy. The first focus in this chapter is understanding how per-

mission usage in apps has changed over the three-year period from October 2014 to September 2017.

### 3.1.1 Data Collection

At the time of writing, there is no public database of the permissions used by apps in the Google Play Store. Historic data on permissions used by apps is also not available. To compile the dataset of permission usage analysed in this chapter, I built a crawler that could obtain full metadata (including permission usage information) on apps by requesting the Google Play Store webpage of each app. This crawler[1] took 'snapshots' of the Google Play Store over the course of 30 months during my research. The crawler was first run in March 2015 with snapshots taken subsequently in three month intervals (at the beginning of each month) up until September 2017.

The architecture for this crawler is shown in Fig. 3.1. The crawler itself is informed of all the apps in the Google Play Store using a list provided by the (now defunct) Google Play Store Crawler project [15]. Geographically distributed worker nodes are used to parallelise the crawling process. Using three to four worker nodes, snapshots can be taken in less than 48 hours.

To obtain an early (pre-March 2015) snapshot of the Google Play Store, I downloaded a dataset [6] of apps that was compiled using the PlayDrone tool [153]. The apps in this dataset were originally downloaded in October 2014. Permission information was extracted from the manifest[2] files of these apps to obtain a snapshot of permission usage in the Google Play Store as at October 2014. At the end of the crawling and data compilation process, 12 snapshots of permission usage in the Google Play Store were available, covering a three-year period: `OCT-2014`, `MAR-2015`, `JUN-2015`, `SEP-2015`, `DEC-2015`, `MAR-2016`,

[1] Note that it is possible to obtain permission usage information from the Google Play Store website using a crawler without violating the `robots.txt` file of the site.

[2] The app manifest, `AndroidManifest.xml`, is a mandatory file that is found in the root directory of all apps. The manifest file provides essential information about an app, including the permissions it uses [9].

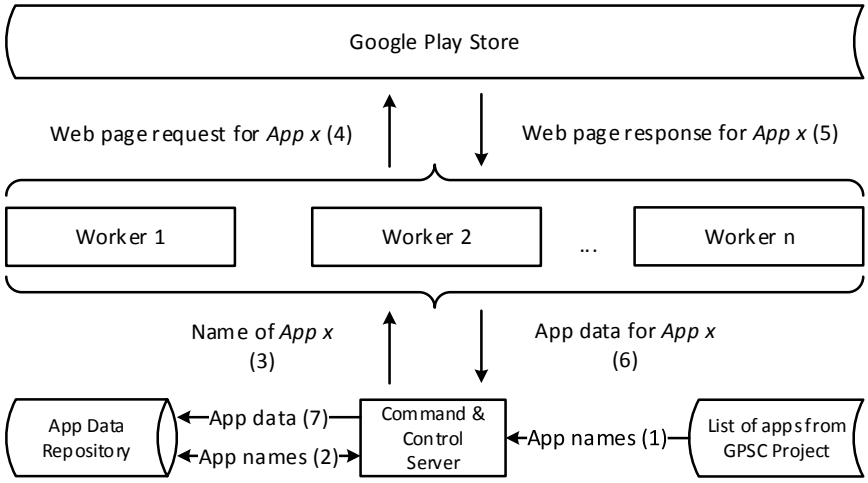



Figure 3.1: Highly-scalable cloud-based crawler architecture. Numbers in brackets indicate the sequence of events.

JUN-2016, SEP-2016, DEC-2016, MAR-2017, JUN-2017, SEP-2017.

### 3.1.2 Changes in Number of Permissions

Increasing numbers of permissions indicate that apps have increasing access to sensitive user data. Therefore, even measuring coarse-grained metrics such as mean permission usage in apps in the Google Play Store over time can signal large-scale erosions in user privacy. Apps in my dataset were divided into three categories, by number of downloads, to understand differences in mean permission usage over time based on the popularity of apps. Apps were considered as having low downloads if they had 1-1K installs, medium downloads if they had 1K-1M installs, and high downloads if they had 1M-5B installs[3]. Since apps are frequently added to and removed from the Google Play Store, in the following permission usage analyses I only consider apps that were present in all snapshots over the entire three-year period ($n = 344,142$).

Table 3.1 shows how mean permission usage increased over the studied period. Across all three categories of apps, mean permission usage increased be-

[3] K = thousand, M = million, and B = billion.

Table 3.1: Mean permission usage (and percentage change) across apps over the three-year period.

| Downloads | OCT-2014 | SEP-2015 | SEP-2016 | SEP-2017 | Change |
|---|---|---|---|---|---|
| 1-1K | 3.03 | 3.06 | 3.06 | 3.04 | +0.3% |
| 1K-1M | 2.31 | 2.38 | 2.39 | 2.37 | +2.6% |
| 1M-5B | 3.37 | 3.55 | 3.55 | 3.50 | +3.9% |

tween OCT-2014 and SEP-2017. That is, across the Google Play Store as a whole, apps had a tendency to increase permission usage over time. Additionally, apps with high downloads, i.e., those having 1M-5B installs, used the highest absolute number of permissions (3.5 as at SEP-2017) and also had the highest percentage increase (3.9%) in the number of permissions used, when compared to the baseline dataset of OCT-2014 from three years prior. While the increase in mean permission usage may seem small overall, it is important to remember that this is the mean permission usage for 344,142 apps. This includes a large number of apps that were not updated at all between snapshots.

Interestingly, while there is an upward trend in permission usage overall, apps across all categories decreased permission usage, on average, between SEP-2016 and SEP-2017. While this is a welcome observation, it is unclear whether this trend will continue. Indeed, future studies should seek to examine the progression of mean permission usage over the long-term.

Since apps with high downloads had the largest increase in permission usage, Table 3.2 shows a detailed breakdown of permission changes in this category of apps. It can be seen that some apps added (and removed) several permissions when they were updated. From the table, approximately 3.6% of apps with high downloads added three or more permissions between OCT-2014 and SEP-2015, 1.9% added three or more permissions in the subsequent year, and 1.3% added three or more permissions in the year after that. This corresponds to over a thousand apps (with more than one million downloads) adding three or more permissions over a three-year period.

Table 3.2: Granular breakdown of permission usage changes across snapshots of the Google Play Store. Each snapshot shows measurements of permission changes from the preceding year.

| | SEP-2015 | SEP-2016 | SEP-2017 |
|---|---|---|---|
| **Total Increase** | **17.0%** | **10.9%** | **8.9%** |
| +3 or More | 3.6% | 1.9% | 1.3% |
| +2 | 4.8% | 3.0% | 2.5% |
| +1 | 8.6% | 6.0% | 5.1% |
| **No Change** | **75.0%** | **78.6%** | **79.8%** |
| -1 | 4.3% | 5.9% | 6.2% |
| -2 | 2.0% | 2.3% | 2.7% |
| -3 or Less | 1.7% | 2.3% | 2.4% |
| **Total Decrease** | **8.0%** | **10.5%** | **11.3%** |

### 3.1.3 Which Permissions Changed

In addition to merely quantifying apps varying their permission usage, it is also important to assess which permissions are being added to (and removed from) apps. This will give insight on specific areas of user privacy that are being eroded. Fig. 3.2 shows which permissions were added by apps, broken down by the number of downloads of the app. The top five permissions (aggregated across all three categories of apps) that were added are `WRITE_CALENDAR`, `READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`, `GET_ACCOUNTS`, and `ACCESS_COARSE_LOCATION`.

It can be seen that the `WRITE_CALENDAR` permission is added in disproportionately many cases in apps with `1-1K` downloads. Manual analysis suggests that this permission is predominantly added in apps that are automatically generated using the PhoneGap [2] and Appcelerator [10] frameworks. Moreover, it does not seem that these apps use the permission to provide actual app functionality. Thus, app generator frameworks seem to actually contribute to apps being unnecessarily permission-hungry. There were no other notable differences in which permissions were added across the categories of apps.

The most common permission added by apps with high downloads is `ACCESS_COARSE_LOCATION`. Overall, apps are seen to predominantly add per-

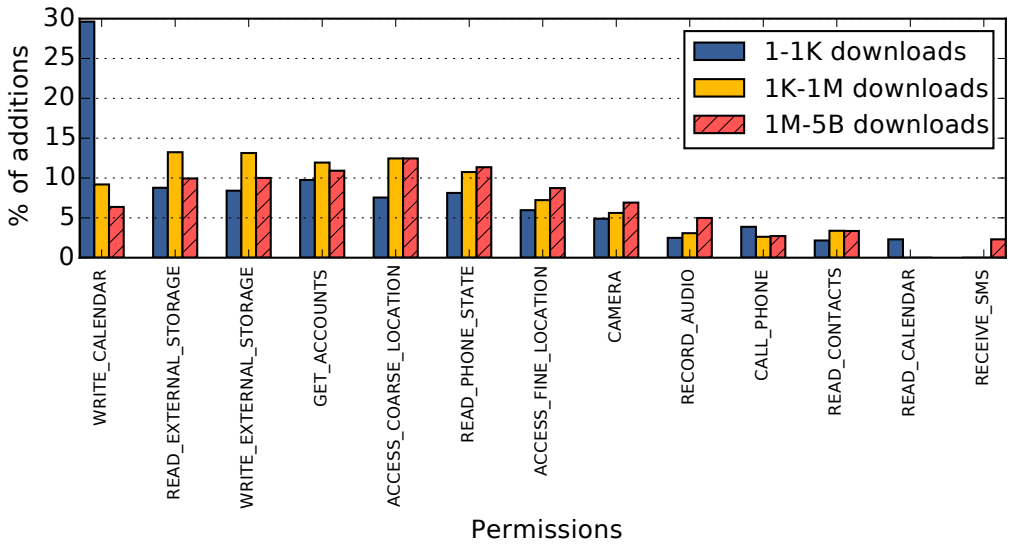



Figure 3.2: Breakdown of the most commonly added permissions. For clarity, permissions that accounted for less than 2% of additions are omitted.

missions relating to accessing device storage, user accounts, phone state, and location. These permissions can be used to read files on the external storage of a device, get email addresses associated with a device, get permanent device identifies (such as the IMEI), and get the user's coarse location. While coarse location may not seem as worrying as precise location, users are known to underestimate how 'precise' coarse location is, with twice as many users thinking coarse location does not offer enough protection, after being educated about its granularity [83].

Although permission additions dominated, some apps removed permissions as well. Fig. 3.3 shows which permissions were removed, broken down by app popularity. The most commonly removed permission overall was `WRITE_CALENDAR`, accounting for approximately 37% of permission removals in apps with `1-1K` downloads. Once again, app generators seem to be responsible for the large fluctuations relating this permission.

Across my snapshots of the Google Play Store, `ACCESS_COARSE_LOCATION`, `READ_EXTERNAL_STORAGE`, and `WRITE_EXTERNAL_STORAGE` were the main permissions that were removed in fewer cases than they were added. Thus, when per-

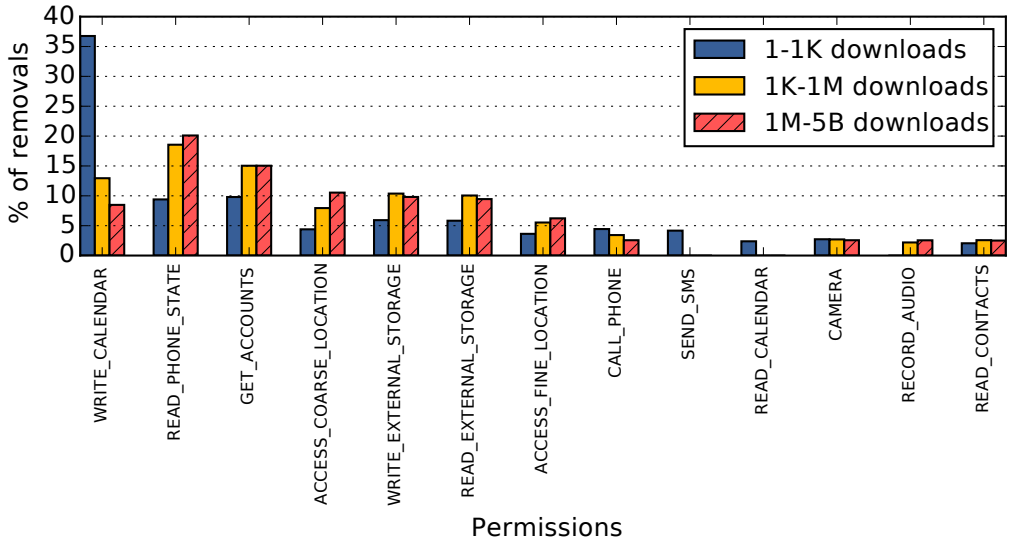



Figure 3.3: Breakdown of the most commonly removed permissions. For clarity, permissions that accounted for less than 2% of removals are omitted.

mission additions are compared to permission removals across the Google Play Store over time, apps are largely retaining access to a device's external storage and the user's current location.

### 3.1.4 Which Apps are Adding Permissions

It is important to understand whether particular groups of apps are responsible for permission additions. This will help to target further privacy erosion analysis. To this end, I conducted two hypothesis tests to determine whether i) free apps and ii) popular apps are more likely to add permissions than their corresponding counterparts, paid and unpopular apps. The behaviour of free and popular apps is important to consider, since these apps constitute the majority of apps that are installed by users.

Both hypotheses were tested using a two-proportion z-test with a sample of 20,000 apps that added permissions. Two-proportion z-tests are useful to determine whether two groups differ significantly on a single characteristic. The significance level for both tests was chosen as 0.01.

**Hypothesis 1: Free apps are more likely to add permissions than paid apps**

The aim of this hypothesis test is to determine whether there is a statistically significant difference between free and paid apps as it relates to adding new permissions. Specifically, it is the null hypothesis that whether an app is free or paid has no bearing on its likelihood of adding permissions. The test returns $p < 0.01$ thus rejecting the null hypothesis. That is, free apps were seen to be statistically significantly more likely to add new permissions than paid apps.

Free apps predominantly earn revenue for their developers through advertisements, delivered by ad libraries. Ad libraries are a source of privacy concerns since they are able to leverage any permissions granted to a host app. Indeed, they are known to sometimes leverage additional permissions beyond what they declare to developers, since they benefit from better profiling of users [88]. Thus, the addition of permissions in free apps greatly facilitates privacy erosion by third-parties, as later discussed in § 3.3.

**Hypothesis 2: Popular apps are more likely to add permissions than unpopular apps**

Popular apps are installed on a large cross-section of devices in the app ecosystem. Thus, it is important to understand the addition of permissions by popular apps. As in § 3.1.2, apps with `1M-5B` installs were considered as being popular apps, because of their high download count.

The aim of this hypothesis test is to determine whether there is a statistical basis to support the idea that popular apps are more likely to add new permissions. Specifically, it is my null hypothesis that there is no difference in the proportion of popular apps and unpopular apps adding permissions. The hypothesis test returns $p < 0.01$, thus rejecting the null hypothesis. That is, popular apps are statistically significantly more likely to add new permissions.

Table 3.3: Datasets used in the analysis.

| Dataset | # of APKs | Source | Date |
|---|---|---|---|
| TOP-OLD | 5,000 | PlayDrone | Oct-2014 |
| TOP-NEW | 5,000 | Google Play | Sep-2016 |
| RANDOM-OLD | 10,000 | PlayDrone | Oct-2014 |
| RANDOM-NEW | 10,000 | Google Play | Sep-2016 |

On one hand, apps adding new permissions may open the door to new features for users. On the other hand, adding permissions may contribute to privacy erosion. The negative effect of new permissions, however, may be magnified if the apps themselves contain vulnerabilities that may be exploited by an attacker. Vulnerability evolution across apps in the Google Play Store is covered in the following section.

## 3.2 Vulnerability Evolution

The aim of this section is to understand how the presence of vulnerabilities within apps has changed over time. Apps containing fewer vulnerabilities over time suggests that app developers are getting better at writing secure code. On the other hand, if apps are getting more vulnerable over time, it suggests that app developers continue to write insecure code and leave users at risk.

### 3.2.1 Dataset Collection and Overview

To understand how the vulnerabilities contained within apps has evolved, I collected a dataset of the current versions (at the time of testing) of 15,000 apps and old versions of those same 15,000 apps from two years earlier. Current versions of apps were downloaded using physical smartphones via the Google Play Store app, i.e., without violating the Terms of Service of the Google Play Store. Old versions of apps were downloaded from a publicly available dataset [6] of apps that was compiled using the PlayDrone tool [153].

A total of 30,000 apps were analysed: 15,000 old versions and 15,000 new

versions. Old versions of apps are hereafter referred to as belonging to the OLD dataset and new versions of apps are referred to as belonging to the NEW dataset. To gain a more detailed understanding of vulnerability evolution across different types of apps, each set of 15,000 apps was assembled with 5,000 TOP apps and 10,000 RANDOM apps.

TOP apps are the 5,000 most popular apps in the Google Play Store (by number of downloads at the time of data collection) that also had an older version in the PlayDrone dataset. Measuring TOP apps gives insight on the characteristics of the most popular apps across the Google Play Store. RANDOM apps were apps selected randomly from a list of all apps in the Google Play Store with more than 100 downloads. The Google Play Store has a long tail of apps with fewer than 100 downloads, so this threshold was chosen to reduce bias from unrepresentative[4] apps. Measuring RANDOM apps allows us to compare the characteristics of TOP apps to the average app in the Google Play Store. The datasets used are summarised in Table 3.3.

Table 3.4 shows a summary of file size statistics across apps in the datasets. TOP apps had larger file sizes than RANDOM apps. This is in line with expectations, since one would expect more popular apps to contain more functionality, thus necessitating larger file sizes. Similarly, NEW versions of apps were larger than OLD versions of apps. This is also in line with expectations, since one would expect updates to apps to deliver improvements or new functionality, thus necessitating larger file sizes. While larger code size may mean additional features for users, it also means that there is a greater surface where bugs or vulnerabilities may be present.

The update frequency of apps can offer insight on latent app vulnerabilities. On one hand, apps receiving frequent updates may mean that vulnerabilities get fixed more quickly. On the other hand, app updates may introduce more

[4]Unrepresentative in this sense means that these apps are installed on so few devices that their overall impact on the app ecosystem would be negligible.

Table 3.4: File size statistics across datasets.

| | TOP-OLD | TOP-NEW | RANDOM-OLD | RANDOM-NEW |
|---|---|---|---|---|
| mean | 12.9MB | 17.5MB | 6.4MB | 7.4MB |
| std | 13.0MB | 16.9MB | 8.7MB | 9.8MB |
| med | 8.0MB | 11.7MB | 3.1MB | 3.8MB |
| min | 21.2KB | 22.2KB | 15.6KB | 15.6KB |
| max | 52.4MB | 104.6MB | 52.4MB | 86.0MB |

vulnerabilities without fixing old ones. To understand app update frequency, I leveraged the snapshots of the Google Play Store (as described in § 3.1) to obtain apps' *updated* date, version number, and file size. These data points, when combined, serve as a robust indication of whether an app has been updated, since a change in any indicates that the app has been updated.

From my snapshots of the Google Play Store, 45% of `TOP` apps received four or more updates in the two-year period that was studied. Conversely, only 5% of `RANDOM` apps had four or more updates. This update behaviour is in line with expectations, since developers of `TOP` apps have financial motivations to try to continuously improve their apps. Frequent updates are better from a security perspective, but only if vulnerabilities are patched during updates. Unfortunately, as shown in § 3.2.4, app updates tend to introduce, as opposed to fix, app vulnerabilities.

### 3.2.2 Vulnerabilities Considered

The Open Web App Security Project (OWASP) compiled a list of the Top 10 vulnerabilities affecting mobile devices and apps [23]. This thesis examines vulnerabilities on this list that are present in the binary of apps themselves, i.e., in the `apk` file of an app. Other vulnerabilities, such as those in code on the server-side are out of scope. The vulnerabilities studied are intentionally general[5] to allow for an understanding of whether app developers are making

[5]General, meaning vulnerabilities coming from well-known Android programming errors, as opposed to obscure programming errors.

Table 3.5: List of the vulnerabilities that are considered.

| Identifier | Description | Tool Used |
|---|---|---|
| INF-DISC-WRLRD | App leverages world readable/writeable files | AndroBugs |
| INF-DISC-PRVDR | ContentProvider exported but not secured | |
| INF-DISC-KSNPW | Keystores not protected by a password | |
| SSL-TLSX-PLAIN | Sending data over plain HTTP | AndroBugs |
| SSL-TLSX-INVLD | Invalid certificate verification | |
| SSL-TLSX-WVIEW | Improper WebView certificate validation | |
| BRK-CRYP-ECBMD | Use of the ECB cryptographic mode | MobSF |
| BRK-CRYP-RANDG | Use of insecure random number generators | |
| OTH-MISC-INTNT | Starting services with implicit Intents | AndroBugs |
| OTH-MISC-DEBUG | App is debuggable | |
| BIN-ROOT-DTECT | App does not seem to have root detection | MobSF |

common mistakes and thus getting even the basics of Android app security wrong. The vulnerabilities studied are summarised in Table 3.5, and briefly described in what follows.

**Information Disclosure:** Sensitive data may be unintentionally leaked from apps that contain weaknesses in program logic or access control. The vulnerabilities examined are: apps creating world readable/writeable files (`INF-DISC-WRLRD`), exported but unsecured `ContentProviders` (`INF-DISC-PRVDR`), and apps using keystores but failing to password-protect them (`INF-DISC-KSNPW`).

**Insecure Network Communication:** Insecure network communication is network communication that fails to use encryption at all, or fails to implement encryption properly. Fahl et al. [72] demonstrated that some apps fail to properly implement SSL/TLS connections and thus are vulnerable to active MITM attacks. The vulnerabilities examined are: apps not using encryption at all (`SSL-TLSX-PLAIN`), verifying SSL certificates in an insecure way (`SSL-TLSX-INVLD`), and improperly validating certificates in WebViews (`SSL-TLSX-WVIEW`).

**Broken Cryptography:** Broken cryptography is the use of available cryptographic primitives or protocols in a weak or insecure way. Encrypting in ECB mode (`BRK-CRYP-ECBMD`) is known to be weak since it is not semantically secure [166]. Additionally, using insecure random generators (or pseudo-random number generators) is a weakness (`BRK-CRYP-RANDG`), since these generators produce predictable values.

**Miscellaneous:** Apps may communicate with each other and the Android system using `Intents`, `ContentProviders`, and `BroadcastReceivers`. Vulnerabilities arise if apps improperly handle external input. These vulnerabilities may allow an attacker to read/write from/to sensitive data stores or trigger an unexpected action within an app. The vulnerabilities considered in this category are: apps using implicit intents to start services (OTH-MISC-INTNT) and apps being debuggable (`OTH-MISC-DEBUG`).

**Binary Protection:** Attackers may attempt to modify the binary of apps using techniques such as repackaging/cloning. Attempts to do this may be frustrated using protection methods such as rooted device detection, debugger detection, and validating app checksums. The vulnerability considered in this category is apps not containing root detection mechanisms (`BIN-ROOT-DTECT`).

### 3.2.3 Tools Used

Several open-source static vulnerability scanning tools were used in this analysis. I modified the tools to make them suitable for the large-scale analysis that was conducted. Specifically, the tools were stripped down to their identification engines and wrapper code was added to allow parallelisation.

**AndroGuard:** AndroGuard [4] is an open-source analysis framework that sup-

ports `apk` analysis. It is easily extensible to facilitate custom analysis. Several of the remaining tools leverage AndroGuard internally.

**AndroBugs:** Androbugs [3] is an open-source vulnerability scanning framework that detects a variety of known Android vulnerabilities.

**Mobile Security Framework:** The Mobile Security Framework [20], MobSF, allows for penetration testing of apps using static or dynamic analysis.

**QARK:** QARK [24] is a tool that scans Android apps for known vulnerabilities. QARK is also able to generate ADB commands or proof-of-concept `apk` files that can exploit the vulnerabilities it finds.

**Mallodroid:** Mallodroid [19] is a research tool specifically designed to detect insecure SSL certificate validation within apps.

### 3.2.4 Vulnerability Measurements

Table 3.6 shows the prevalence of identified vulnerabilities within apps in each of the studied datasets. The most common vulnerabilities were apps sending network traffic unencrypted (`SSL-TLSX-PLAIN`) and using insecure random number generators (`BRK-CRYP-RANDG`). Approximately 95% of `TOP` apps sent network traffic in plaintext, compared to approximately 80% of `RANDOM` apps that did the same.

Approximately 30% of `TOP` apps failed to properly do TLS certificate validation compared to approximately 14% of `RANDOM` apps. Thus, a non-trivial portion of apps are vulnerable to MITM attacks. This problem is elevated since apps have great (and increasing) access to sensitive user data through permissions, and can also get additional sensitive data from user input. Thus, susceptibility to MITM attacks, if sensitive data is transmitted over the network, adds another dimension to privacy risks.

Table 3.6: Percentage of apps from each dataset that contained vulnerabilities. Numbers in brackets show the measurements when only considering apps that were updated between OLD and NEW datasets.

| **Vulnerability** | TOP-OLD | TOP-NEW | RANDOM-OLD | RANDOM-NEW |
|---|---|---|---|---|
| INF-DISC-WRLRD | 32.7 (34.1) | 62.6 (69.1) | 16.4 (19.6) | 24.8 (42.6) |
| INF-DISC-PRVDR | 5.02 (5.44) | 11.2 (12.6) | 2.36 (2.87) | 3.14 (5.01) |
| INF-DISC-KSNPW | 3.06 (3.42) | 2.90 (3.24) | 2.27 (3.33) | 2.25 (3.28) |
| SSL-TLSX-PLAIN | 94.2 (95.8) | 95.3 (97.0) | 79.4 (87.1) | 80.3 (89.5) |
| SSL-TLSX-VERIF | 30.1 (31.4) | 31.7 (33.2) | 14.5 (20.0) | 14.3 (19.3) |
| SSL-TLSX-WVIEW | 18.4 (20.4) | 20.7 (23.0) | 9.87 (13.3) | 9.35 (11.9) |
| BRK-CRYP-ECBMD | 30.2 (27.5) | 29.2 (26.3) | 12.3 (6.61) | 12.5 (6.61) |
| BRK-CRYP-RANDG | 83.7 (73.7) | 91.1 (80.7) | 59.1 (26.3) | 63.6 (30.6) |
| OTH-MISC-INTNT | 12.0 (13.1) | 22.3 (25.0) | 3.07 (3.76) | 5.02 (9.04) |
| OTH-MISC-DEBUG | 0.46 (0.19) | 0.30 (0.33) | 2.21 (0.76) | 1.93 (n/a) |
| BIN-ROOT-DTECT | 83.7 (84.4) | 70.6 (71.5) | 95.6 (97.3) | 93.3 (97.3) |

For all vulnerabilities except OTH-MISC-DEBUG and BIN-ROOT-DTECT, a greater proportion of TOP apps were vulnerable compared to RANDOM apps. One explanation for this is that TOP apps typically have larger code sizes and thus have a larger codebase where vulnerabilities may be present. TOP apps have a large install base and this means that a large cross-section of users will have apps installed that contain vulnerabilities. OTH-MISC-DEBUG and BIN-ROOT-DTECT are the only studied vulnerabilities where TOP apps were less vulnerable than RANDOM apps. One explanation for this is that fixing these vulnerabilities may lead to improved safeguarding of the app intellectual property, an outcome likely favoured by the developers of popular apps.

Along similar lines, with the exception of INF-DISC-KSNPW, OTH-MISC-DEBUG, and BIN-ROOT-DTECT, NEW versions of apps had an increased prevalence of vulnerabilities over OLD versions, for both TOP and RANDOM apps. Alarmingly, for several of the studied vulnerabilities (INF-DISC-WRLRD, INF-DISC-PRVDR, and OTH-MISC-INTNT), the prevalence of vulnerabilities approximately doubled between OLD and NEW versions of apps.

In addition to simply measuring vulnerability prevalence, I also measured how the number of distinct types of vulnerabilities within apps changed between

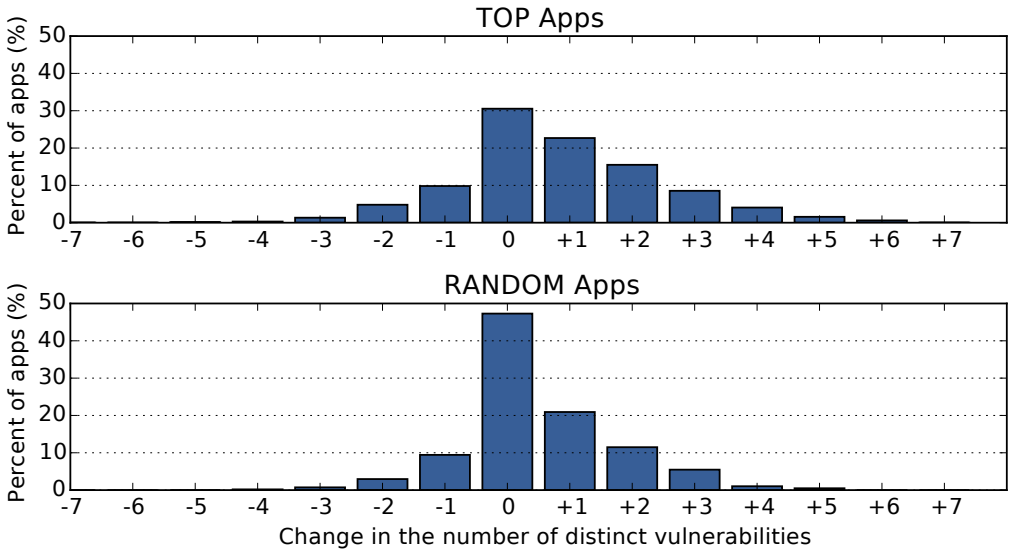



Figure 3.4: How the number of distinct types of vulnerabilities in apps varied between `OLD` and `NEW` versions of apps.

versions. For this measurement, only apps that were updated between `OLD` and `NEW` datasets were examined. This result is shown in Fig. 3.4. Both `TOP` apps and `RANDOM` apps had an overall increase in the number of distinct types of vulnerabilities that they contained. `RANDOM` apps were more likely than `TOP` apps to have no change in the number of distinct vulnerabilities. This demonstrates that in addition to having increasing prevalence for vulnerabilities, `TOP` apps are also more likely to have new types of vulnerabilities introduced as they are updated.

### 3.2.5 Vulnerability Provenance

Fig. 3.5 shows how vulnerabilities were distributed in terms of where in app code they were found. Textured areas represent the occurrences of vulnerabilities in *library code* (`LIB`) and non-textured areas represent vulnerabilities found in *developer-written code* (`NON-LIB`). Library code was identified using the method detailed in § 3.3.1. The vulnerabilities `BIN-ROOT-DTECT` and `OTH-MISC-DEBUG` are omitted for brevity since they usually appear in `NON-LIB` code.

From the figure, vulnerabilities such as INF-DISC-PRVDR, SSL-TLSX-VERIF, and SSL-TLSX-WVIEW are seen to be more prevalent in LIB code. On the other hand, vulnerabilities such as INF-DISC-WRLRD and OTH-MISC-INTNT are almost exclusively found in NON-LIB code. This highlights the fact that a targeted approach can be taken both when looking for particular vulnerabilities, and when educating app developers and library developers on writing secure code.

The INF-DISC-PRVDR vulnerability was much more prevalent in NON-LIB code for apps in the TOP-OLD dataset compared to apps in the TOP-NEW dataset. Manual analysis of a sample of the apps in question revealed that the increase in the vulnerabilities in LIB code came from updates to several libraries that were embedded within these apps. This highlights the potential impact that a few libraries can have on a large number of apps, and also lends support to the recommendations made to app developers to keep their libraries constantly updated.

Another vulnerability, BRK-CRYP-ECBMD, saw more instances in TOP apps than in RANDOM apps. Manual analysis suggests that this is due to more instances of this vulnerability per app in TOP apps. This is likely due to the fact that TOP apps have larger code sizes. It further suggests that these vulnerabilities are not due to 'one off' mistakes, but rather a systematic lack of understanding of how to prevent the vulnerability. The vulnerability OTH-MISC-INTNT overwhelmingly came from NON-LIB code. Once again, this insight can narrow down areas of focus when training app developers.

### 3.2.6 Limitations of Vulnerability Analysis

The aim of my vulnerability analysis is not to analyse an exhaustive list of vulnerabilities. Rather, several common Android vulnerabilities were examined instead to understand whether app developers were getting the basics right. If app developers continue to make common mistakes, it stands to reason that they will also make more serious mistakes. Thus, my observations of vulnerability

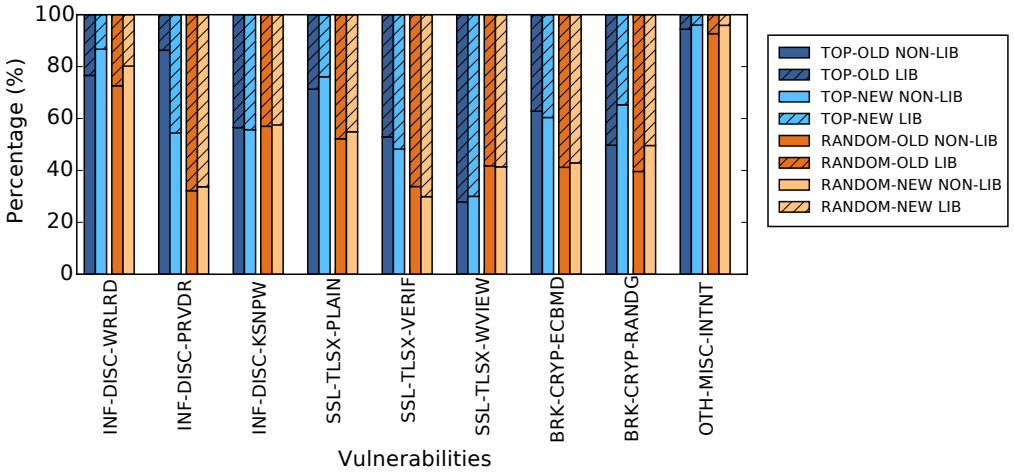



Figure 3.5: How vulnerabilities changed over the two-year period for the four datasets. Areas with texture indicate the fraction of vulnerabilities that come from library code (LIB) and vice-versa.

evolution may represent a conservative estimate of the actual situation in the app ecosystem.

For scalability, lightweight static analysis tools were used for vulnerability scanning. Static analysis tools, however, may yield false positives. This is why I used several tools that detect the same vulnerabilities. These tools often detected vulnerabilities using different approaches. Thus, the results from one tool can be validated using the other tool(s) that detect similar vulnerabilities. This reduces false positives and thus improves the precision of the results.

Static analysis tools may lack completeness (i.e., have reduced recall and more false negatives) if programming techniques such as reflection, dynamic code loading, or native code are used. Thus, the actual number of vulnerabilities within apps may be higher than what was measured.

## 3.3 Library Empowerment

An `apk` file is a compressed archive that houses all of an app's code and other resources internally. On Android, the libraries used by apps are tightly integrated with the code of an app itself. A consequence of this is that libraries

enjoy the permissions granted to the host app since they share the same running process and UID. As a result, third-party library developers are also able to leverage the permissions that have been granted to apps. As apps evolve to request more and more permissions (§ 3.1), these libraries obtain increased privileges as well. I call this phenomenon *library empowerment*. In what follows, I define two different forms of library empowerment.

`OLD-LIBRARY-EMPOWERMENT`: The older version of an app contains a library that internally has code to make permission-protected API calls. The library, however, cannot make these API calls because the host app has not declared the relevant permission in its manifest. The newer version of the app, however, now declares additional permissions that empower the library to call the permission-protected APIs that it could not access before.

`NEW-LIBRARY-EMPOWERMENT`: The older version of an app contains a library, but the library does not contain code to make additional permission-protected API calls. However, the new version of the app contains an updated version of the library that now has code that can make additional API calls. Additionally, the new version of the app now declares additional permissions that allow the library to access these additional APIs.

Measuring library empowerment is critical because whereas apps may be updated to add new functionality that requires additional permissions, the user granting the permissions is likely not aware that one or more pieces of third-party code is also obtaining new privileges on their devices at the same time.

As a pragmatic way to measure library empowerment, I assume that apps compatible with run-time permissions have been granted any new permissions that they request. This may, however, cause measurements of library empowerment to be overestimated for those users who do not necessarily grant all run-

time permissions. In this case, the measurements will capture the 'worst-case' scenario on devices.

### 3.3.1 Library Permission Usage in Apps

To understand the potential impact of library empowerment, it is important to understand library permission usage in apps. The do this, the `apk` file for each app was first decompiled using `apktool` [8]. This transformed the `classes.dex` Dalvik Executable (DEX) code into `smali` code. Smali is an intermediate register-based language that is human-readable.

In Android apps, there are several types of operations that involve permissions, including explicit calls to standard Android APIs, managing `ContentProviders`, and managing `Intents` [156]. I developed a static analysis tool that identifies these permission-related operations within app code, and used PScout permission mappings [39] to identify the permission guarding the operation. For example, my static analysis tool would identify the Android API `Landroid/location/LocationManager;->getLastKnownLocation` and determine that it needs location permissions to be accessed.

By looking at the Java hierarchy in the decompiled `smali` files, one can infer whether a file belongs to app code or library code. To identify library code, I used a whitelist of library signatures provided by the authors of FlexDroid [135]. As a sanity check, I used LibRadar [40] to validate this output. Library usage provided by LibScout showed results with a strong general agreement with those obtained by using the signature approach.

By performing the aforementioned steps, it is possible to identify and distinguish permission usage in app code from permission usage in library code. Importantly, it is possible to determine which particular permissions are being used in each type of location.

Library usage across `OLD` and `NEW` versions of apps was fairly consistent as shown in Table 3.7. `TOP` apps used approximately two more libraries than `RANDOM`

Table 3.7: Number of libraries used in apps across the datasets.

| | TOP-OLD | TOP-NEW | RANDOM-OLD | RANDOM-NEW |
|---|---|---|---|---|
| mean | 6.6 | 6.4 | 4.1 | 4.1 |
| std | 5.2 | 4.6 | 4.9 | 4.6 |
| med | 6 | 6 | 2 | 3 |
| min | 0 | 0 | 0 | 0 |
| max | 31 | 30 | 34 | 34 |

Table 3.8: Prevalence of library empowerment.

| | TOP | RANDOM |
|---|---|---|
| OLD-LIBRARY-EMPOWERMENT | 9.8% | 2.8% |
| NEW-LIBRARY-EMPOWERMENT | 14.3% | 4.5% |

apps, with average usage of 6.5 and 4.1 respectively. Some apps had no detectable libraries, while other apps had up to 34 libraries detected. For those apps with greater library usage, the possibility of library empowerment increases accordingly.

### 3.3.2 Library Empowerment Measurements

Table 3.8 shows the prevalence of library empowerment across the studied datasets. Overall, 9.8% of TOP apps suffered from OLD-LIBRARY-EMPOWERMENT and 14.3% of TOP apps suffered from NEW-LIBRARY-EMPOWERMENT. RANDOM apps had less library empowerment with 2.8% suffering from OLD-LIBRARY-EMPOWERMENT and 4.5% suffering from NEW-LIBRARY-EMPOWERMENT. This asymmetry in the rates of library empowerment may be expected given that TOP apps use more libraries on average than RANDOM apps to begin with. However, after correcting for the number of libraries used per app, TOP apps were seen to be twice as likely to suffer from library empowerment than RANDOM apps.

Fig. 3.6 shows the additional permissions that libraries are able to use due to library empowerment. For both categories of apps, the majority of cases involved allowing libraries to read from and write to external storage and access a user's

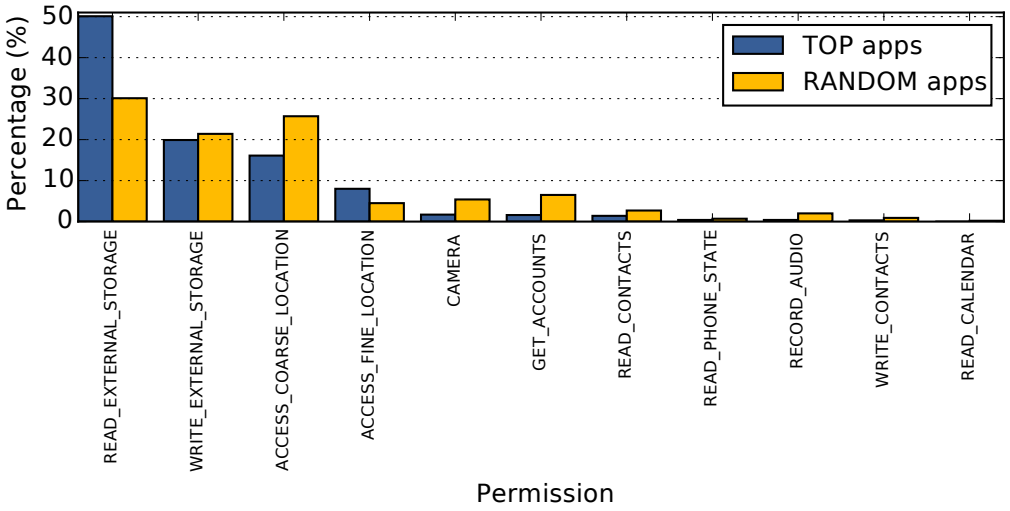



Figure 3.6: Additional permissions that libraries were empowered to use.

location. `TOP` apps and `RANDOM` apps were most frequently empowered with the `READ_EXTERNAL_STORAGE` permission in 50.1% and 30.1% of cases respectively. Access to a user's coarse location happened in 16.1% and 25.7% of cases for `TOP` apps and `RANDOM` apps respectively. Camera, accounts, and contacts list permissions were also newly available to libraries in fewer cases.

Overall, I have observed a trend in libraries being able to access new sensitive data on a device through library empowerment. This additional access predominantly relates to a device's external storage and its location services, but also to the camera, accounts on the device, and a user's contacts to a lesser extent. While the empowerment through the latter permissions may seem small, I remind the reader that the apps affected have millions of installs across the app ecosystem and thus the risk from library empowerment is significant.

## 3.4 Summary

This chapter presented the results of a large-scale longitudinal study detailing how apps have evolved over time, in terms of the permissions they use, the vulnerabilities they contain, and access to sensitive data by their embedded li-

braries. Overall, apps were seen to be getting more permission-hungry over time, with popular apps and free apps more likely to add new permissions. Embedded libraries were seen to benefit from these permission increases, since they are able to leverage the permissions granted to their host app. Apps also had increases in the number of vulnerabilities they contained as they were updated. Some vulnerabilities came predominantly from developer-written code, while others mostly came from embedded libraries. Understanding the source of vulnerabilities can help scope the training given to the respective developers and can also narrow areas of security analysis.

This chapter shows that there is an overall trend towards the erosion of privacy and security in apps, making apps more risky to use over time. However, if no suitable alternatives to risky apps exist, users may simply choose to accept the added danger. This is undesirable, since it erodes security and privacy across the app ecosystem as a whole. Thus, it raises the question of the extent to which users can replace risky apps with less risky alternatives, to mitigate the negative impact on their security and privacy. This question is investigated in the next chapter.

# CHAPTER 4

## SecuRank: Finding Functionally-Similar Apps

*Whenever you give up something, you must replace it with something.*

– Lou Holtz

As shown in the previous chapter, apps are getting more permission-hungry and more vulnerable over time. This is undesirable from both a security and privacy standpoint. It does not suffice to merely highlight issues within apps, as users may be inclined to accept risks within the apps that they use if they are unable to replace them with suitable alternatives. Some apps, such as the Facebook app [13], can be considered as one-of-a-kind since they offer a unique service. On the other hand, *general-purpose* apps may be replaced with functionally-similar alternatives to the benefit of the user, if they are found to be risky.

Risky, as used in this chapter, is defined as having the possibility to contribute danger or cause loss. Thus, apps may be considered risky if they are permission-hungry, contain vulnerabilities, contain invasive libraries, or simply do not follow security and privacy best practices.

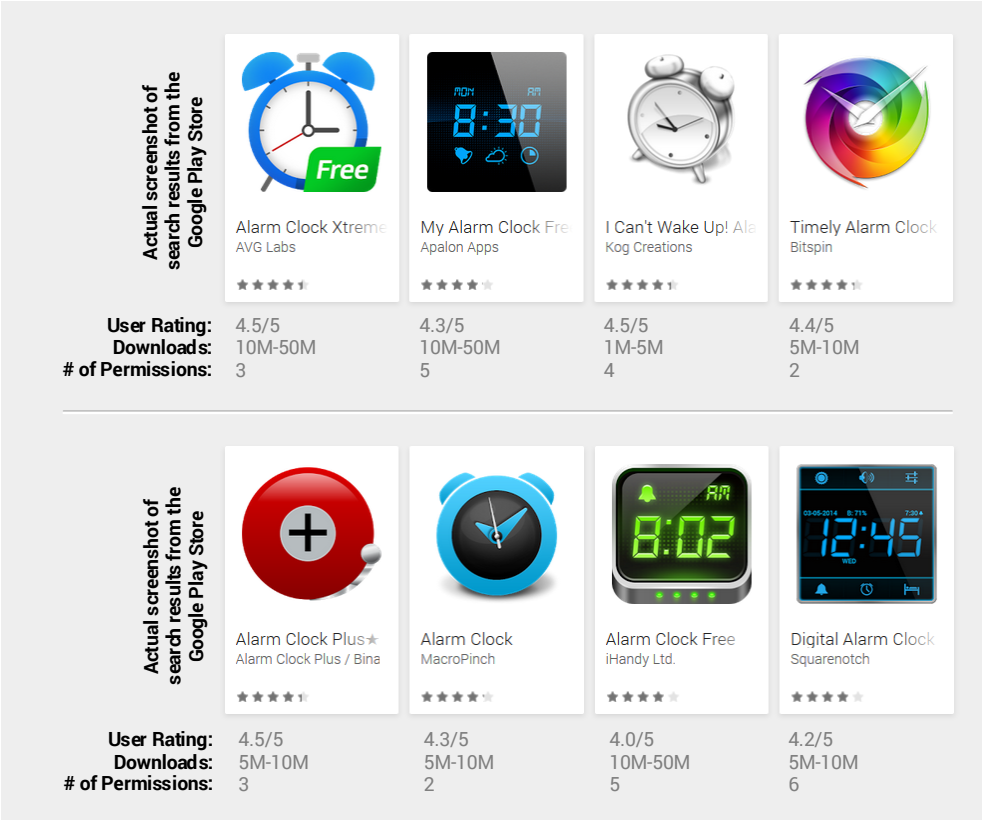



Figure 4.1: Snapshot of the Top 8 Google Play Store results for the search query "alarm clock".

General-purpose apps, such as *flashlight* or *alarm clock* apps, provide generic functionality, and as such typically have several competing alternatives from other app developers in the app store. The main idea in this chapter is that security and privacy on devices can be improved if risky general-purpose apps that users currently have installed are replaced with less risky alternatives.

In this chapter, I perform a large-scale study of the Google Play Store to understand the prevalence of general-purpose apps, before grouping them by functionality. Using the number and type of permissions requested by an app as a proxy for its riskiness, I measure the extent to which apps can be replaced with less risky, i.e., 'preferable' alternatives. The importance of this analysis is motivated by Fig. 4.1, which shows the top search results for the *'alarm clock'*

search query. All apps have high-ratings and provide similar functionality, but the most permission-hungry search result uses three times as many permissions as the least permission-hungry search result. As a proof-of-concept, this chapter considers permission usage as the sole metric that determines the riskiness of an app. However, other metrics can be easily combined as well.

The chapter begins with a description of the methodology that was used to identify and group functionally-similar general-purpose apps. I then show how preferable alternative apps can be used to replace more risky ones and argue for the quality of the recommended alternative apps. Finally, I describe a publicly available tool in the form of an app, called SecuRank, which I developed while conducting the research presented in this chapter. SecuRank can be used to scan smartphones and automatically identify and suggest preferable alternative apps to users.

Most of the material in this chapter formed the basis of a paper published at the ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '16) 2016 [143]. Additionally, the SecuRank app was accepted as a demo at the ACM Conference on Computer and Communications Security (CCS '16) 2016 [142].

## 4.1 Approach to Finding Functionally-Similar Apps

My overall aim is to identify general-purpose apps in the Google Play Store and determine whether less risky functionally-similar alternative apps exist that can replace them. My approach is to first group apps by functionality, and then compare the riskiness of apps within groups to determine what the most preferable choice would be.

The search feature in app stores can be leveraged to find groups of functionally-similar apps, since the search results that are returned from search queries are usually similar. However, not all search queries will return

functionally-similar apps. A specific query like 'alarm clock' is likely to be suitable, but a more general query such as 'football' would likely not be suitable. Thus a two-pronged approach to identifying search queries and then filtering them is necessary. The main steps in this process are summarised below:

- Obtain metadata for all apps in the Google Play Store. This metadata contains app descriptions, which are keyword-rich pieces of text that describe app functionality. Obtaining app metadata for all apps in the Google Play Store was previously described in § 3.1.1.

- Identify search queries on the Google Play Store by leveraging keywords from the previous step (§ 4.1.1).

- Filter search queries to retain only those that return groups of functionally-similar apps (§ 4.1.2).

In what follows, I describe the latter two of the aforementioned steps in detail. For the analysis presented in this chapter, only apps with descriptions written in English were considered. This approach, however, can also be applied to apps with descriptions written in other languages. Additionally, I focused on the most popular search queries in the Google Play Store to understand the extent to which highly-downloaded apps had preferable alternatives.

### 4.1.1 Identify Popular Search Queries

Since keywords in app descriptions factor significantly in the app store search[1] ranking algorithm, an understanding of keywords in app descriptions can be used to 'reverse-engineer' popular search queries on the app store. This approach to identifying popular search queries is summarised in *Stage 1* of Fig. 4.2 and detailed below:

[1] Here I assume that search ranking in the Google Play Store works similarly to how it works in search engines on the world-wide web [17].

1. Identify all apps in the Google Play Store with more than 100,000 installs. This threshold was arbitrarily chosen so that the data later derived represented popular apps.

2. Obtain the app store descriptions of these apps and filter them using the NLP techniques of tokenization, stemming, and *stop word* removal.

3. Combine these processed app store descriptions into a large corpus of text and extract the 20,000 most frequently occurring words. The number 20,000 was chosen arbitrarily to limit the size of the final dataset.

4. Input each of these frequently occurring words to the Google Play Store search form (on its website) and obtain its autosuggestions[2] without actually conducting a search. Using autosuggestions is a useful way to 'convert' words of interest into actual search queries.

5. Add the highest ranking autosuggestion for each word to a list of popular search queries.

After performing the aforementioned steps, I had a list of popular search queries on the Google Play Store. These search queries can be assumed to be popular, since they were autosuggested by the app store itself.

### 4.1.2 Filter Popular Search Queries

There is no guarantee, however, that the list of popular search queries obtained in the previous step will give search results that consist of only functionally-similar apps. For example, the query "strategy game" (an actual popular search query) would yield games, but not necessarily those that were the same type of strategy game. Thus, additional filtering is needed to identify those queries that return groups of functionally-similar apps. Since the Google Play Store

[2]Google Play Store autosuggestions were programmatically retrieved using the following URL: `https://market.android.com/suggest/SuggRequest?json=1&query=QUERY`.

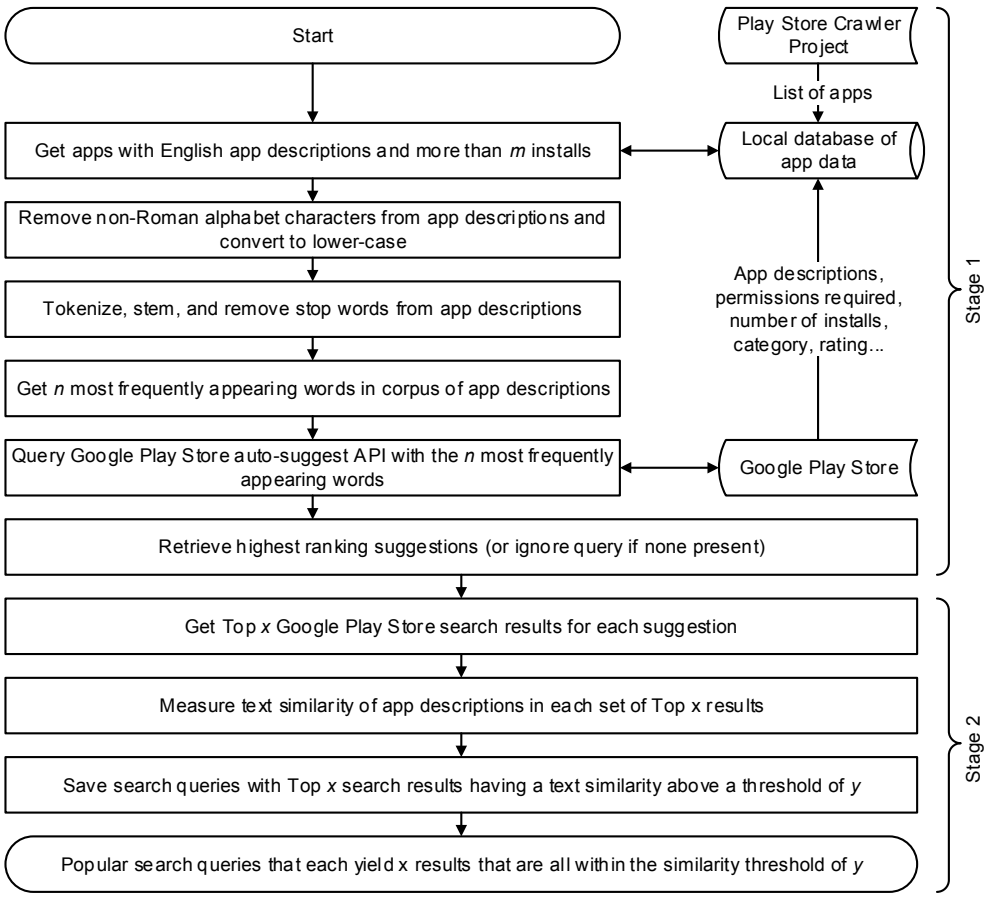



Figure 4.2: Flow chart showing the various stages in my data collection methodology.

returns results in groups of 20, I considered only search queries that returned 20 functionally-similar apps to be valid during filtering.

An app's description in the app store is usually a comprehensive summary of the functionality that the app provides. This is because app developers describe the functionality of their apps in their app store description in order to entice users, attract downloads, and provide keywords for the app store search ranking algorithms. Thus, apps similar in functionality will likely have similar app store descriptions. By taking text similarity measurements across the descriptions of the 20 apps in a set of search results, one can determine whether the group consists of functionally-similar apps.

Text similarity measures for short segments of text, such as app descriptions, have been examined in the literature [108, 109]. Some approaches are lexical and rely only on matching terms. Other approaches are probabilistic in nature and involve modelling frameworks. One popular measure, *Term Frequency-Inverse Document Frequency*, transforms documents into a vector space model. However, this approach does not work well for comparing documents of different lengths. The *cosine similarity* measure alleviates this problem by normalising the vectors before measurement. For this reason, cosine similarity was chosen as the text similarity measure to be used. Cosine similarity has been used previously in the field of smartphone security for malware detection [130]. A higher cosine similarity measure means that two samples of text being compared are more similar, and vice-versa.

I used the Python NLTK library [43] to implement the *cosine similarity* measurement function. This function leveraged a Porter stemmer for text pre-processing. Each popular search query identified in § 4.1.1 was run on the Google Play Store and the Top 20 search results and their descriptions were obtained. App descriptions of apps ranked #2 to #20 were each compared to the app ranked #1 for the search query. Apps were compared to the #1 ranked app since I assume that it is the best match for the search query. If the remaining 19 apps had descriptions within a prescribed similarity threshold of the #1 app, I assumed that the apps in the search results were functionally-similar. These steps to filter queries to identify only those that return groups of functionally-similar apps are summarised in *Stage 2* of Fig. 4.2.

#### Choosing a Similarity Threshold

In order to filter queries, an appropriate text similarity threshold needs to be chosen. If the similarity threshold is too low, apps will be less related and I hypothesise that search queries will more likely return apps from a wider variety

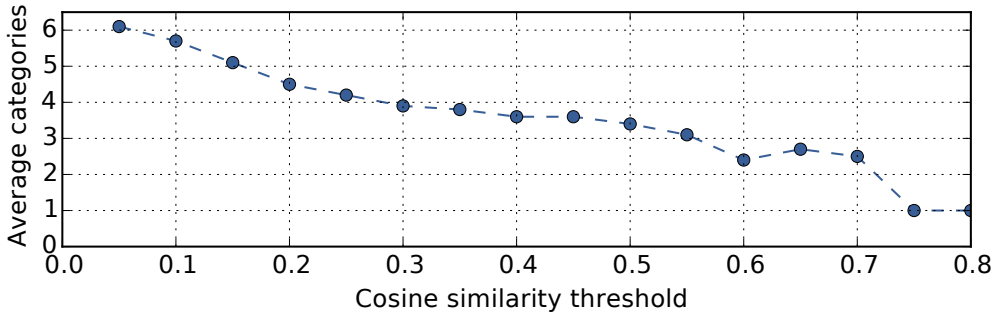



Figure 4.3: Impact of the cosine similarity threshold on the average number of categories that apps in the search results belonged to.

of app store categories. To test this, I varied the cosine similarity threshold and analysed the average number of categories that the search results belonged to. This result is shown in Fig. 4.3. There were no queries for which the average cosine similarity of the results was above 0.8. Setting the cosine similarity threshold to 0.75 or 0.8 yielded search results that belonged to one category. However, there were very few search queries that returned results with this high level of similarity. One example is "slot machines", with the results unsurprisingly belonging to the category of `GAME_CASINO`.

A cosine similarity threshold of 0.25 was found to be suitable by manual inspection. At this threshold, there were 2,620 search queries that each returned 20 functionally-similar apps as search results. The functional-similarity of these results is validated in § 4.3.3. I randomly selected 2,500 of these queries and used them for the remainder of my analysis. These 2,500 queries and their corresponding 50,000 search results are hereafter called the *search query dataset*. The 20 functionally-similar apps returned by a query are hereafter called the *search result set* for that query.

Fig. 4.4 shows a breakdown of the Top 25 categories that the 50,000 search results from my search query dataset belonged to. The largest category was `TOOLS` with 6,458 apps. This lends support to the suitability of my methodology for delivering functionally-similar general-purpose apps, since `TOOLS` is arguably the most likely category that general-purpose apps would be placed in.

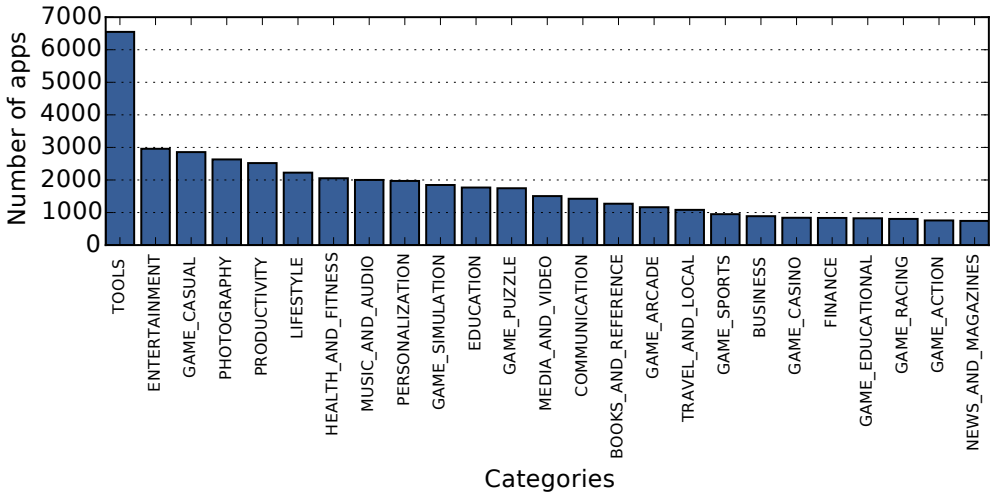



Figure 4.4: Breakdown of the Top 25 categories that the apps in the search results belonged to.

## 4.2 Analysis of the Search Query Dataset

As a first step, I measured changes in the position of apps in app store search results over time. This is to understand whether highly-ranked apps tend to hold on to their search rankings. To take this measurement, the Top 20 search results for each of the queries generated in § 4.1.2 was fetched three times. The first time was to establish the baseline of app rankings, and the second and third time were two weeks and two months after the baseline respectively.

The stability of search results is shown in Fig. 4.5. The plot shows the percentage of search queries that have the same app in the same search result position. After two weeks, 86% of queries had the same app in the #1 position. Approximately 71% of queries had the same pair of apps in positions #1-2. After two months, the percentages fell to 75% and 51% respectively. Thus, the search ranking of apps changes at times, especially over longer periods of time. However, highly-ranked apps tend to hold on to their positions, while less highly-ranked apps are less likely to hold on to their positions.

The most highly-ranked search results are known to command a significant majority of search engine traffic in regular web searches [97]. This is likely true

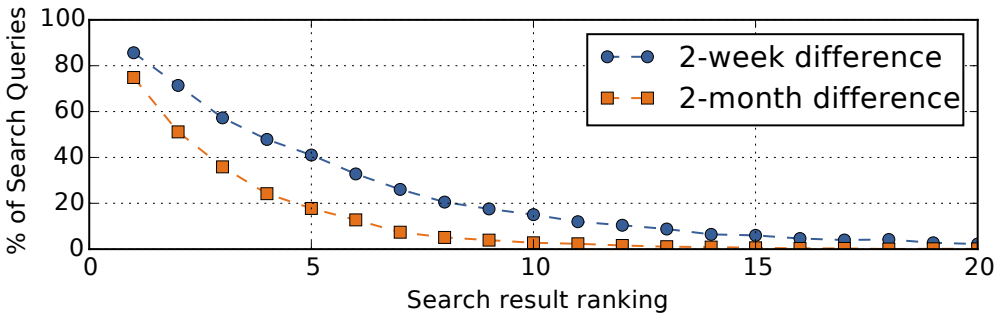



Figure 4.5: Difference in search results over a 2-week/2-month period. The most highly ranked apps tend to hold on to their exact search rankings or were slightly reshuffled.

in app stores are well. Thus, highly-ranked apps, if undesirable from a security or privacy perspective, may persistently affect users users due to their prominent positions and the stability of their rankings.

### 4.2.1 App Ratings and Permission Usage

To motivate the utility of a system that can suggest functionally-similar alternative apps, I examined how the user rating of an app varied with its position in the search results for a query. The rating of an app (out of five stars) is a user-generated number that gives insight into the quality of an app as perceived by users of that app. The relationship between app rating and position in search results is shown in Fig. 4.6. This plot is largely flat, suggesting that there is no significant decrease in the rating of an app even if it is not highly-ranked in the search results (up to position 20, which was the limit of my analysis) for a search query. Given that even low-ranked search results are highly-rated, it suggests that lower-ranked alternative apps may be found satisfactory by users if they are recommended to them.

Fig. 4.7 shows how the mean number of permissions used per app varied with the position of the app in search results. Apps ranked #1 for their search query required more permissions than apps that ranked lower. This observation was confirmed to be statistically significant using single-factor ANOVA. Note that an

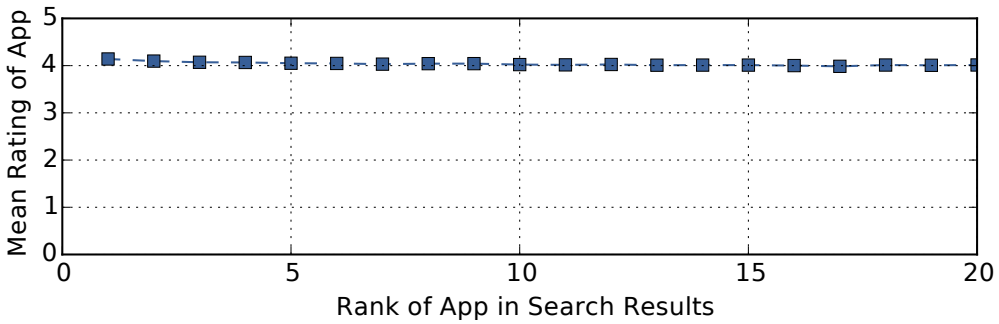



Figure 4.6: How the mean user rating per app varies with the rank of that app.

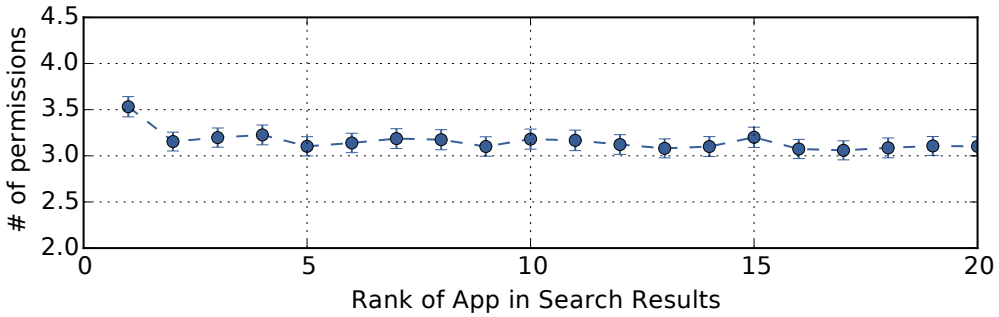



Figure 4.7: How the mean number of permissions used per app varies with the rank of the app in the search results.

app using more permissions than functionally-similar apps does not necessarily indicate malfeasance from the app developer. However, it likely indicates that the app is permission-hungry or not following the principle of least privilege [129].

Additional permissions, though they may be used to provide secondary functionality to an app's core functionality, open avenues for greater privacy leaks as shown in Chapter 5. App developers can mitigate this privacy leakage and still provide secondary functionality within their apps by using `Intents` to have an existing app perform the privileged task [22]. However, many app developers fail to do this, leading to highly-privileged permission-hungry apps in app stores.

For the remainder of this analysis, permissions used by apps that are not required for providing core app functionality are called *extraneous permissions*. This is because their usage can contribute negative effects to a user's privacy and security, while not being necessary for the functioning of the app. For

example, a *flashlight* app that uses the `READ_CONTACTS` permission will be using it extraneously, since reading a user's contacts does not directly facilitate the core function of the flashlight.

The analysis so far demonstrates that low-ranking apps, on average, are approximately as highly-rated as highly-ranking apps. Additionally, we have seen that #1 ranked apps use more permissions on average than the apps below them in search results. This motivates the next stage of my analysis, where I examine the extent to which less permission-hungry apps can be offered as functionally-similar replacements to permission-hungry apps using extraneous permissions.

## 4.3 Fine-Grained Contextual Permission Analysis

As a way to measure how permission-hungry an app is, I developed a strategy called *fine-grained contextual permission analysis*. The idea is that permission usage within an app can be more accurately judged if it is compared to that of other functionally-similar apps. Among functionally-similar apps, permission usage should cluster around a central point, and any significant deviation from that can be considered evidence of an app being permission-hungry.

Subsequent to the research in this chapter being conducted and published, Google's Security and Privacy team announced a similar approach to be used in the Google Play Store called "peer group analysis" [121]. This approach uses deep learning performed on app metadata and user metrics to group apps by functionality. The behaviour of apps within each group is then examined to identify anomalous behaviour. Peer group analysis is an equally valid approach, but incurs greater overhead because of the involvement of machine learning.

### 4.3.1 Algorithm for Contextual Permission Analysis

Central to my contextual permission analysis scheme are two metrics that measure the rareness of permissions within groups of functionally-similar apps.

- **Individual Permission Prevalence (IPP)**: This is a per permission metric that is the fraction of apps in a search result set that use the particular permission.
- **App Overall Permission Prevalence (AOPP)**: This is the average of the IPPs for all permissions used by a particular app.

As a concrete example, consider four apps: $app_1 = \{A, B, C\}$, $app_2 = \{A, B\}$, $app_3 = \{A, C\}$, $app_4 = \{A\}$, where A, B, and C are permissions. The IPP and AOPP metrics for each app are calculated as follows:

1. Make list of all permissions used, i.e., [A, B, C, A, B, A, C, A].
2. Count the number of occurrences of each permission in the list, i.e., $p_A = 4$, $p_B = 2$, $p_C = 2$.
3. IPP is the fraction of occurrences of a permission to the number of apps that use the permission, i.e., $IPP_A = \frac{4}{4} = 1.0$, $IPP_B = \frac{2}{4} = 0.5$, $IPP_C = \frac{2}{4} = 0.5$.
4. AOPP is the mean of the IPPs for those permissions used by an app, i.e., $AOPP_1 = 0.66$, $AOPP_2 = 0.75$, $AOPP_3 = 0.75$, $AOPP_4 = 1.0$.

Intuitively, lower IPPs signify that not many apps in a group use that permission. Since apps in a group are functionally-similar and thus have similar core functionality, less used permissions are likely ones that are not vital for enabling core app functionality. Similarly, since AOPP is the average of the IPPs of permissions used by an app, an app with a low AOPP is likely using

permissions that are unnecessary for providing core app functionality and can thus be considered permission-hungry, i.e., using extraneous permissions.

Algorithm 1 formally outlines how IPP and AOPP are calculated for a group of apps in a search result set.

**Algorithm 1:** Calculate IPP and AOPP for a list of apps

**Input:** List of apps $\beta = \beta_1, \ldots, \beta_n$
**Output:** IPP, AOPP per app
$permList \leftarrow [\,]$
**foreach** *app* in $\beta$ **do**
  $permList \leftarrow permList + getPermissions(app)$
$IPP \leftarrow \emptyset$
**foreach** *perm* in $GetUniqueItems(permList)$ **do**
  $IPP[perm] \leftarrow permList.count(perm) \div len(\beta)$
$AOPP \leftarrow \emptyset$
**foreach** *app* in $\beta$ **do**
  $temp \leftarrow [\,]$
  **foreach** *perm* in $getPermissions(app)$ **do**
    $temp \leftarrow temp + IPP[perm]$
  $AOPP[app] \leftarrow mean(temp)$
**return** $AOPP, IPP$

### 4.3.2 Identifying Rare Permissions

Building on the idea of apps using extraneous permissions, I wanted to examine apps using *rare permissions*. I defined a rare permission as any permission with an IPP of 5%, i.e., only one app in a group of 20 functionally-similar apps used that permission.

Across the search query dataset, approximately 6% of #1 ranked apps used rare permissions. Of these apps, the most common rare permissions were `ACCESS_FINE_LOCATION` (10.5%), `CAMERA` (9.0%), and `READ_CONTACTS` (8.5%). At this point, I remind the reader that there is no claim that rare permission usage signifies malicious intent from an app developer. However, it can point to bad programming practices, since apps can use `Intents` to have permission-protected

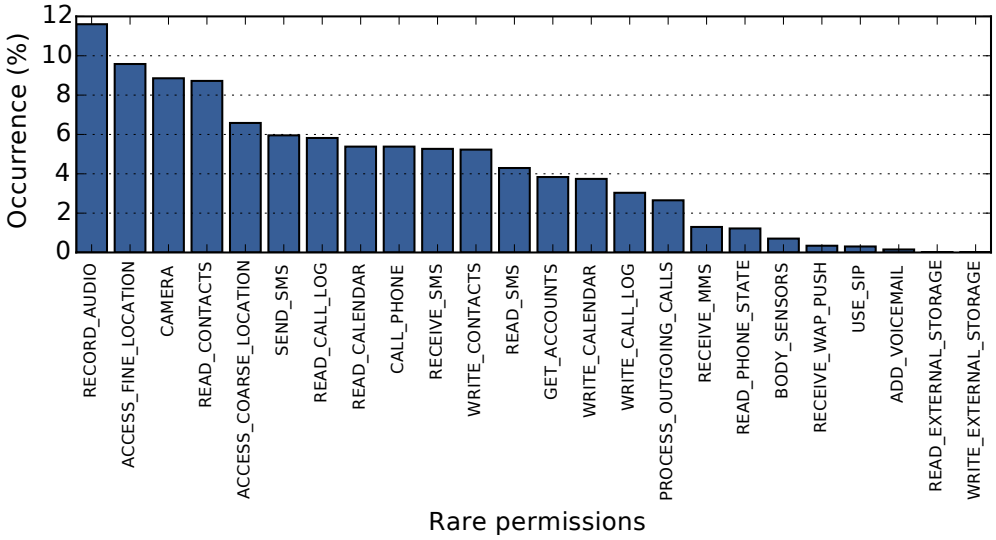



Figure 4.8: Breakdown of the usage of rare permissions, i.e., permissions with an IPP of 5%.

functions performed for them, without themselves requiring a particular permission.

The rare permissions identified above are those used by #1 ranked apps. The same approach was repeated across the entire search query dataset. The rare permissions identified across the entire dataset are shown in Fig. 4.8. In approximately 12% of cases, the rare permission was `RECORD_AUDIO`. Accessing a user's precise location (9.6%), the device camera (8.9%), and a user's contact list (8.7%) were other 'common' instances of rare permissions.

Since rare permission usage is unusual in apps with a particular functionality, apps using rare permissions are suspicious and could potentially be subject to more intensive vetting using static or dynamic analysis. Using contextual permission analysis to identify apps with rare permission usage can significantly reduce the resources required for scanning an app store with millions of apps.

### 4.3.3 Case Study of Rare Permissions

False positives, i.e., apps that are not functionally-similar but somehow made their way into search result sets can be the cause of the observation of rare per-

Table 4.1: Results of manual inspection of 125 apps that used rare permissions. Most apps were relevant to their search query and did not justify their usage of the rare permission.

| Permission | Relevant to search | Permission justified? |
|---|---|---|
| ACCESS_FINE_LOCATION | 96% | 8% |
| CALL_PHONE | 96% | 4% |
| CAMERA | 92% | 61% |
| RECORD_AUDIO | 92% | 30% |
| SEND_SMS | 88% | 9% |
| **Average** | **92.8%** | **22.4%** |

missions. For this reason, I manually inspected 125 apps found to be using rare permissions to determine whether they were apps that were not functionally-similar but erroneously got into search result sets. Specifically, I manually inspected 25 apps using each of the following permissions rarely: CAMERA, RECORD_AUDIO, ACCESS_FINE_LOCATION, SEND_SMS, and CALL_PHONE. I chose these permissions because they have the potential to allow an app to spy on a user or cost the user financially.

The results of the manual inspection is shown in Table 4.1. Approximately 93% of apps flagged for using rare permissions were relevant to their search query. That is, these were not irrelevant apps that were erroneously thought to be functionally-similar to the other apps in their search result set. This also lends support to the accuracy of my approach in generating queries that return functionally-similar apps, since irrelevant apps would likely show up as using rare permissions.

In addition to confirming the relevance of apps to their search query, I examined whether their rare permission usage was justified. I manually searched for an app's justification for its use of a rare permission from the app store description of the functionality of the app, within the app itself, the feature listing of the app, and the "What's New" section of an app's app store page. Fig. 4.9 shows an example of a children's app, air.com.classteacher.main, that fails to justify its usage of the ACCESS_FINE_LOCATION permission. This app is con-

sidered to have not justified its usage of the permission because an examination of the app's description and an investigation of the functionality provided within the app itself does not explain why the app requires access to the location of the user (who in this case is likely to be a child), let alone their precise location. Other apps returned by the same search query, *numbers games*, do not use this permission at all.

The reader should note that there is a distinction made between apps using rare permissions and apps using rare permissions but have not justified their need to do so. Some apps that use rare permissions were seen to justify their rare permission usage. One example is a phonebook app that used the `CAMERA` permission. This permission is rare among phonebook apps and may seem suspicious at first glance. However, this app justified access to the camera by providing a feature that allows users to take a picture of their contact when making a new phonebook entry. This feature was also explained in the app's description.

Of the 125 apps that were manually audited, rare permission usage was justified only 22.4% of the time. This average is, however, inflated by the rare usage of the `CAMERA` permission, which was justified in 61% of cases. The more surreptitious `CALL_PHONE` and `SEND_SMS` permissions had their usage justified by the apps that used them in 4% and 9% of cases respectively.

Malware is especially known to abuse these two permissions by making calls and sending text messages to premium rate numbers [77][173]. While no malicious usage of these two permissions was observed in the sampled apps, 'phalibs' (potentially harmful libraries infecting 6.8% of Google Play Store apps) are especially poised to exploit them [54].

Apps using rare permissions were analysed using historical data from the Google Play Store. In total, there were 3,452 apps using rare permissions after duplicates (the same app appearing more than once for the same rare permission) were removed. Approximately 9% (301) of apps using rare permissions added the rare permission within the ten months before the study was done. Of these 301

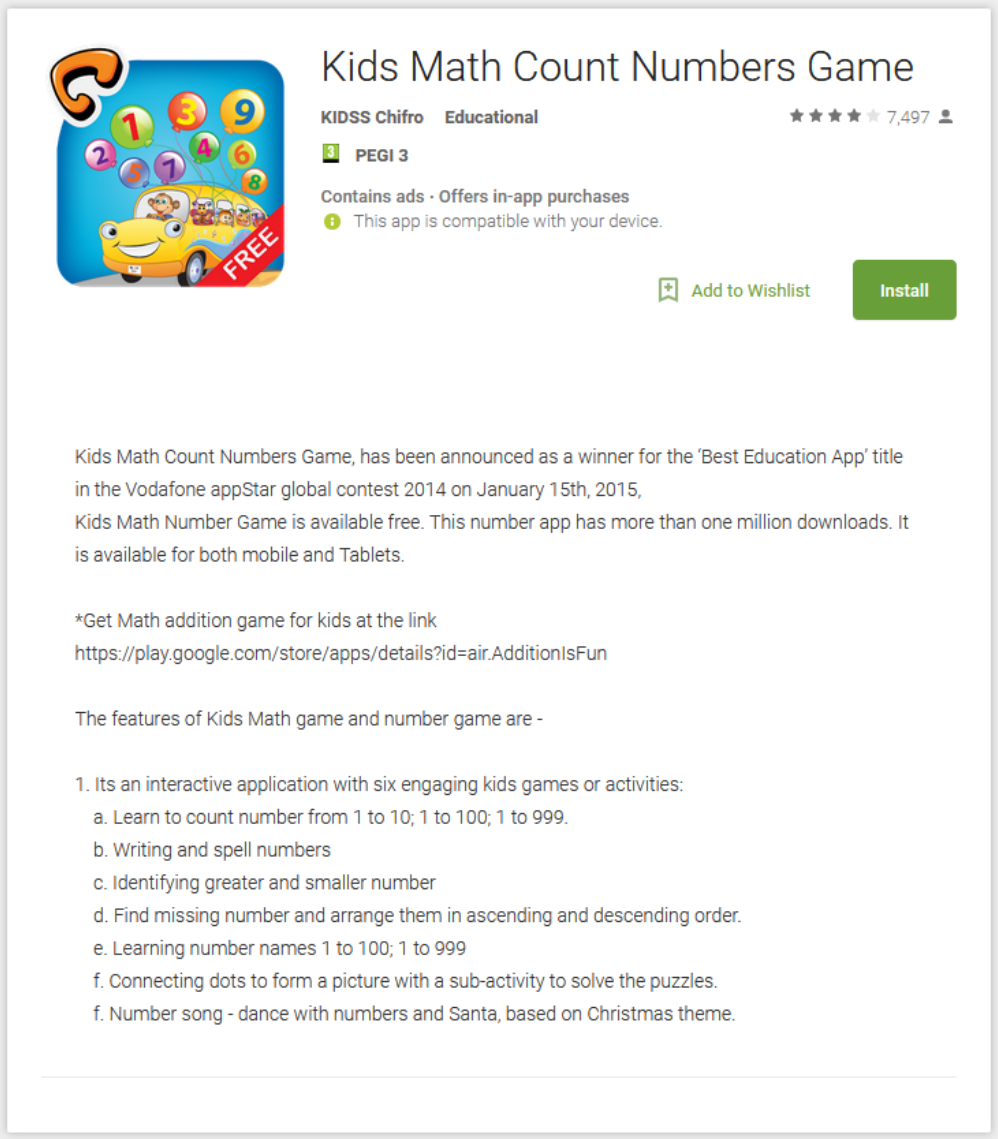



Figure 4.9: An example of a children's app using the `ACCESS_FINE_LOCATION` permission without justification.

apps, 95 (31.6%) of them did not have any update to their app store page to explain the reason for a *rare* permission being added. Of these apps, 77 had more than 100,000 installs and 36 had more than 1,000,000 installs. While my observations are inconclusive as it relates to malicious intent from app developers, they suggest that contextual permission analysis and historical data can be useful when narrowing down a set of apps to receive enhanced security analysis.

## 4.4 SecuRank Framework and Tool

Given the observation of rare permission usage, and apps being permission-hungry in general, I applied the concept of contextual permission usage to analyse the entire Google Play Store. The intention was to understand the extent to which apps in the official app store had 'preferable' alternatives. In this case, preferable simply means whether there is a functionally-similar app to a target app that has a higher AOPP than the target app. Because of the way AOPP is calculated, in that less common permissions are penalised, AOPP can be used as a proxy for whether an app is following the principle of least privilege [129]. Thus, a preferable app will have a higher AOPP than the target app and will (in most cases) be less permission-hungry.

My contextual analysis approach described in § 4.3 was built into a framework and tool called SecuRank[3] (*Secu*rity *Rank*). Given an input app, SecuRank can determine if there are preferable functionally-similar alternatives to that app. The database used by the SecuRank system was populated using several additional steps:

1. For each app in the app store, obtain all apps that are suggested by the app store as being similar to that app.

2. Use the text similarity measurement function described in § 4.1.2 to compare the descriptions of suggested similar apps to the target app. Suggested similar apps that do not meet the similarity threshold are considered to be falsely suggested as being similar and are removed.

3. Using AOPP, compare permission usage between the target app and apps that are inferred to be similar.

Fig. 4.10 shows the likelihood of apps across the Google Play Store having preferable alternatives. This data is broken down by the cost of an app and

[3]SecuRank is available as an Android app and is free to use online at https://securank.me/

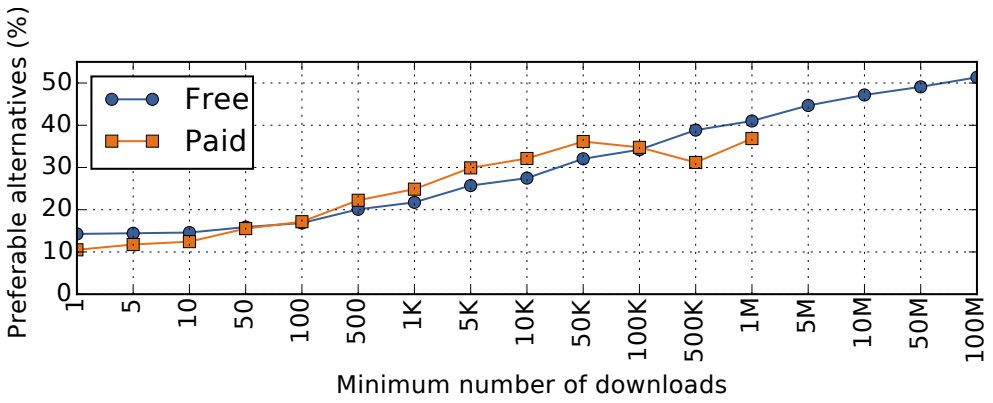
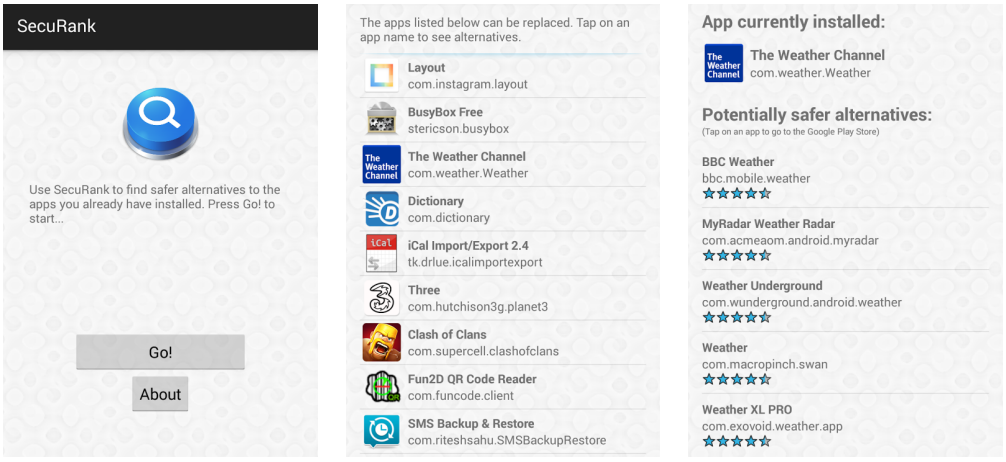



Figure 4.10: Percentage of apps having functionally-similar alternatives with a higher AOPP. *Note that (at the time of data collection) there were no paid apps with five million downloads or more.*

the number of downloads that the app has. The number of downloads an app has positively correlates with its likelihood of having a preferable alternative, regardless of the cost of the app. Approximately 20% of apps with 500 downloads or more had a preferable alternative. Very popular apps, those having 50 million downloads or more, had a 50% chance of having a preferable alternative. That is, the most popular apps across the app ecosystem are also most likely to have preferable alternatives.

Finally, I measured the likelihood of an app to have a preferable alternative depending on whether the app was free or paid. Specifically, it was my null hypothesis that the cost (free or paid) of an app is independent of its likelihood to have a preferable alternative. This hypothesis was evaluated using a 2-proportion z-test with a sample of 20,000 apps. The test returned $p < 0.01$, confirming that the results of this test was statistically significant. That is, free apps were statistically significantly more likely than paid apps to have preferable alternatives.




(a) Home screen (b) Results screen (c) Suggestions screen

Figure 4.11: Screenshots of the main activities in the SecuRank Android app.

### 4.4.1 Real-World Deployment of SecuRank

The SecuRank tool was packaged as an Android app and released to the general public via the Google Play Store[4]. Screenshots of the SecuRank app can be seen in Fig. 4.11. The app is able to scan an Android device and give the user feedback on preferable functionally-similar alternative apps that they can install and use instead of others currently installed on their device. The SecuRank app itself requires no permissions to run on a device. Anecdotal feedback from users of SecuRank and public ratings (50-100 installs with a rating of 4 stars out of 5) in the Google Play Store suggest that it is a useful tool.

The usefulness of SecuRank can be more robustly measured by doing user studies. Specifically, study participants could be asked to install and run SecuRank on their own devices. Based on the output of SecuRank, users could evaluate recommended alternative apps and indicate whether they would be satisfied with them as an alternative. A user's inclination to use an alternative app could also be evaluated in relation to the perceived privacy improvements to be gained.

[4] `https://play.google.com/store/apps/details?id=me.securank.jov`

## 4.5 Summary

In this chapter, I presented results of a large-scale analysis of the Google Play Store to understand the existence of groups of functionally-similar general-purpose apps. The main idea is that apps that are considered risky for whatever reason can be replaced by functionally-similar alternatives with more desirable security and/or privacy characteristics. From observations in Chapter 3, apps could be considered risky because of their permission usage or because of the vulnerabilities or libraries they contain. In this chapter, I used contextual permission usage in functionally-similar apps (based on the AOPP metric) as the discriminating factor. Due to the robust and extensible nature of the SecuRank framework, additional or combined metrics can also be easily used to rank apps.

Overall, it was found that a large number of risky apps in the Google Play Store have functionally-similar alternatives that can be used to replace them. It, however, remains unclear what the overall risk to users is if they happen to not be using the least risky apps or if they choose not to use less risky apps. Additionally, since devices typically have many apps installed on them, the total security and privacy impact of having more than one risky apps installed could increase through emergent factors. In the next chapter, I shed light on these issues by using real-world data from a large corpus of devices to measure real-world risks on devices coming from the use of risky apps.

# CHAPTER 5

## DeviceAnalyzer: Measuring Real-World Risks

*What gets measured, gets managed.*

– Peter Drucker

Having demonstrated in the previous chapters that many apps contribute potential risks to user security and privacy because of their use of extraneous permissions, this chapter measures real-world risks that comes as a result of this. This is important, because while permission-hungry apps (or apps otherwise undesirable from a security and privacy standpoint) are known to exist, their real-world impact on smartphones and the app ecosystem as a whole is an understudied area.

Additionally, it remains unclear what the emergent risks coming from multiple apps installed on devices are. Indeed, prior work (§ 2.1, § 2.3, § 2.4) has typically examined apps in isolation, without taking into consideration any additional issues or amplification of issues that may arise due to having multiple apps installed (and sometimes running) at the same time.

In this chapter, I leverage real-world data from approximately 30,000 smartphones to paint a clearer picture of the overall risk to real-world devices coming from the aforementioned dimensions. The real-world data I use comes from the Device Analyzer Project, which was previously described in § 2.4.2. My contribution in this chapter is two-fold:

- § 5.1 uses insights and data from Chapter 4 to measure improvements that can be made on real-world devices if less risky functionally-similar apps are used instead of the ones that are currently installed.
- § 5.2 quantifies an emergent privacy problem, called *intra-library collusion*, whereby a single library may obtain greater access to a device by virtue of being embedded within multiple apps that each have distinct sets of permissions granted to them.

Some of the material in this chapter formed the basis of a paper published at the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (*WiSec '17*) 2017 [141].

## 5.1 Improvements Using SecuRank

Chapter 4 shows that many apps in the Google Play Store are permission-hungry by virtue of their permission usage and the core functionality that they provide. In this section, I measure how libraries benefit from these permission-hungry apps (§ 5.1.1) and the context under which permission-hungry apps use their extraneous permissions (§ 5.1.2). Additionally, I measure the improvements that can be made on devices if users install and use less risky functionally-similar alternative apps instead of the ones they currently use (§ 5.1.3). In parts of this chapter, I perform static analysis on apps to understand where in app code permissions are used. The steps for identifying permission use and its provenance is the same as described in § 3.3.1.

As in Chapter 4, I consider groups of 20 functionally-similar apps. Within groups of functionally-similar apps, an app was considered to be using extraneous permissions if 10% or fewer apps in its group use that particular permission. The idea is that apps in a group of functionally-similar apps provide the same core functionality, and thus, if a small percentage of apps in the group use a particular permission, this permission can be assumed to not be necessary to support core app functionality. Manual inspection showed that a threshold of 10% gave a conservative estimate of extraneous permission usage.

### 5.1.1 How Libraries Benefit from Extraneous Permissions

Static analysis was used to explore the extent to which embedded libraries benefit from the extraneous permissions used by the apps that embed them. To this end, I randomly generated a list (from the dataset in Chapter 4) of 1400 apps that used extraneous permissions. These apps were downloaded from the Google Play Store and subjected to static analysis. If the app contained a library with code that leveraged any of an app's extraneous permissions, I consider the library to have also benefited from the app's use of those extraneous permissions.

Across my dataset of 1400 apps, 25.6% of apps used extraneous permissions that their embedded libraries were also able to leverage. Of these apps, 72% had libraries that were able to access one permission and 28% had libraries that could access two or more permissions. As an interesting fringe case, one app[1] in the dataset used at least five extraneous permissions that its embedded libraries were also able to access.

Fig. 5.1 summarises the extraneous permissions that embedded libraries were able to access. In most cases, these were fine and coarse location permissions (39% and 31% respectively). In fewer cases, camera (10%), microphone (8%), and access to the address book (4%) were also seen. I take this

[1]This was a *tax inquiry* app that requested permission to access the user's contacts, microphone, location, and camera.

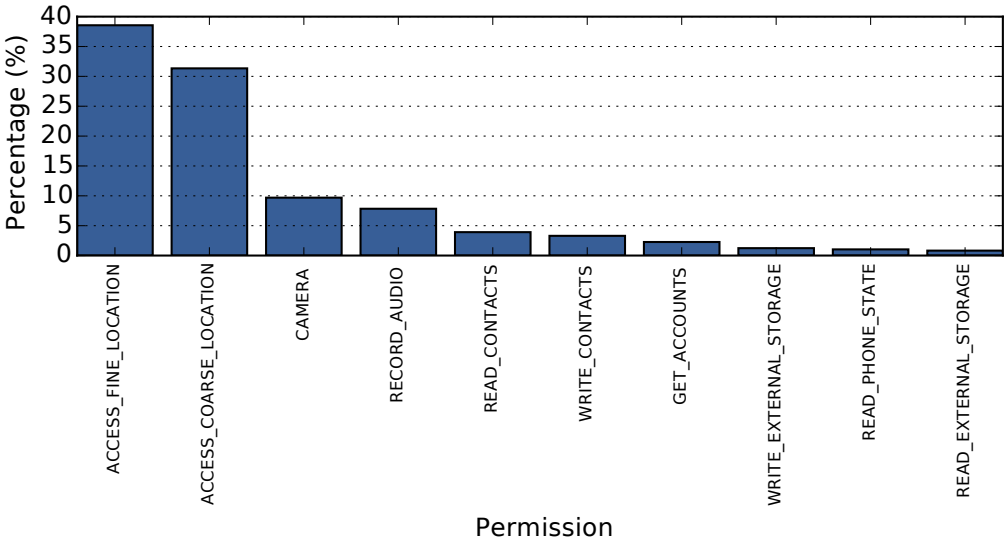



Figure 5.1: Breakdown of which extraneous permissions libraries were able to gratuitously leverage.

opportunity to remind the reader that extraneous permissions do not support core app functionality, but as a result of their usage by an app, existing code within embedded libraries can now access additional sensitive user data.

### 5.1.2 Context of Extraneous Permission Usage

The contextual integrity of permission usage was explored by Wijesekera et al. [164]. Context refers to the circumstances under which permissions are used, such as in the background when the user is not interacting with the app. The authors wanted to measure whether "information flows [were] appropriate, as determined by the user". They observed that many apps used permissions in contexts that users were unhappy with. Moreover, users wanted to block 35% of sensitive data access, mainly because the access did not seem to relate to app functionality or they were uncomfortable sharing that information with the app in question.

Along similar lines, I examined the context under which extraneous permissions were being used in apps. This analysis is motivated by the fact that while extraneous permission usage is undesirable, it warrants increased attention if this

usage is happening in the background. Extraneous permissions being used in the background is not necessarily malicious, but such incidences should certainly be given more careful scrutiny by app stores and security practitioners. Based on the observations of Wijesekera et al. [164], users would likely want to know of extraneous permission usage on their devices, and especially when it happens in the background.

EviCheck is a static analysis tool that can certify and verify fine-grained permission usage policies within Android apps [132]. For example, an EviCheck policy for checking that an app does not record audio in the background can be written as `EVICHECK_ENTRY_POINT EVICHECK_DO_INBACKGROUND : ~RECORD_AUDIO`. I modified the EviCheck tool to identify the context under which permission-protected API calls were used within apps in my dataset. EviCheck was modified by stripping it down to its identification engine, and writing wrapper code to make it parallelisable.

Concretely, I used EviCheck to determine whether an app used any of `ACCESS_FINE_LOCATION`, `CAMERA`, `READ_CALENDAR`, `READ_CONTACTS`, `RECORD_AUDIO`, or `SEND_SMS` when the app was not visible to the user. Not visible to the user includes situations where the permission-protected API call is in a service or receiver used by the app or when the app is otherwise in the background. The aforementioned permissions were chosen since they can cost a user money, spy on a user, track a user, and in general protect operations that can be considered suspicious if they are happening without user interaction [76][164].

Table 5.1 summarises the results of the analysis of background usage of extraneous permissions within apps. These results are calculated as the percentage of apps using an extraneous permission in the background of the total number of apps using that extraneous permission in the first place. The extraneous permissions overwhelmingly used in the background are `SEND_SMS` (26%), `ACCESS_FINE_LOCATION` (22%) and `READ_CONTACTS` (16%). Camera and microphone access happened in fewer cases with 4% and 2% respectively. Background

Table 5.1: Percentage of apps using extraneous permissions without user interaction.

| Permission | Percentage |
|---|---|
| ACCESS_FINE_LOCATION | 22% |
| CAMERA | 4% |
| READ_CALENDAR | 0% |
| READ_CONTACTS | 16% |
| RECORD_AUDIO | 2% |
| SEND_SMS | 26% |

access to the device calendar was not observed in the studied dataset.

Although background usage of ACCESS_FINE_LOCATION is undesirable, it can be considered to be less of a problem, since users often receive visual cues when their location is being accessed and can also see a history of apps that have recently accessed their location. On the other hand, the usage of SEND_SMS and READ_CONTACTS without user interaction has the potential to be more dangerous. Given the concerns of users highlighted by Wijesekera et al. [164], identifying suspicious background usage of permissions is indeed important and could also serve as a risk metric that could be added to app stores or the SecuRank system described in Chapter 4.

### 5.1.3 Improvement on Real-World Devices

Apps using extraneous permissions, and permission-hungry apps in general, contribute security and privacy risks to the app ecosystem. I used lists of apps installed on devices (from the Device Analyzer dataset) to understand the improvements that can be achieved if users favour less permission-hungry apps when choosing which of a group of functionally-similar apps to install. To do this, I married knowledge of functionally-similar but less permission-hungry apps (Chapter 4) with lists of installed apps on real-world devices to measure the potential improvement that can be made on devices.

An analysis of the Device Analyzer dataset reveals that devices had a mean of 201 apps installed. However, this total includes OEM/Android system apps

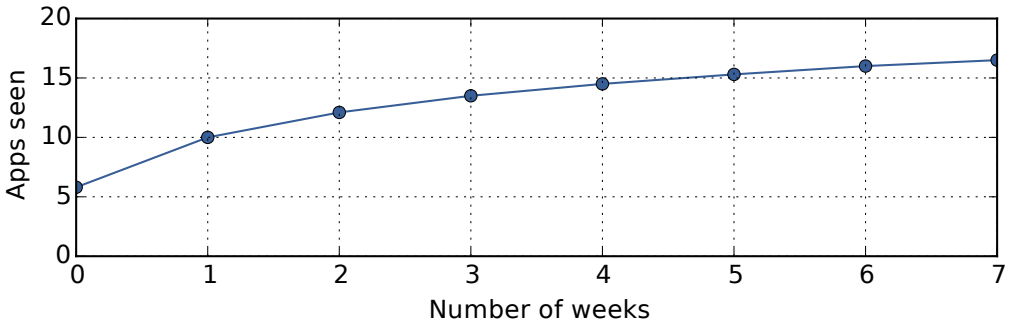



Figure 5.2: Number of distinct third-party apps with visible/foreground or foreground priority on devices over time.

that should not be included in the analysis. The intersection of apps installed on devices and third-party apps available in the Google Play Store (using data from Chapter 3) was taken to filter non-third-party apps. After filtering, devices were seen to have a mean of 56 third-party apps installed. However, some of these apps may be unused, and thus while theoretically able to contribute security or privacy risks, their lack of use makes them inert from a security and privacy erosion perspective.

For this reason, additional Device Analyzer data was used to identify the apps that were 'used' on a device. An app was considered as being used if it had *foreground* or *visible* priority according to Android's `RunningAppProcessInfo.importance` level. This allows a conservative measurement of the improvement on devices that can be achieved, since unused apps and apps that run exclusively in the background are ignored.

Fig. 5.2 shows the number of visible/foreground apps on devices over time. Unsurprisingly, the cumulative number of distinct apps observed on devices increased over time, since users typically do not use every single app on their device every day. I chose a threshold of four weeks for whether an app was used on a device. Four weeks is suitable since the plot approaches its asymptotic limit (of $\approx 17$) at that point. Any app not seen within four weeks of the most recently seen app was considered to be inactive and not used in the analysis.

In total, I was able to obtain lists of apps installed on 28,476 devices. This

was interfaced with data on groups of functionally-similar apps in the Google Play Store from Chapter 4. In this way, I could measure the number of apps per device that had functionally-similar alternatives and, importantly, the extent to which the functionally-similar alternatives were less permission-hungry.

From the data, 43.5% of third-party apps on devices could be replaced with a functionally-similar app that was less permission-hungry. It is important, however, to understand whether users might be satisfied with the suggested alternative apps. To get a sense of this, I used the app store ratings (out of five stars) of the suggested replacement apps to understand whether they were considered to be of high quality by those persons that used them. Recommended alternative apps were rated within +/-0.5 stars of the apps they were intended to replace in 81.5% of cases. This suggests that replacement apps are also of equivalent quality and lends support to the idea that functionally-similar alternative apps would be accepted by users.

### Number of Permissions Saved

The next step is to see what overall impact on permission usage could be achieved on entire devices if all suggested replacement apps were used to replace currently installed apps. The *Current Permission List* is the union of the sets of all permissions used by apps currently installed on a device. The *Predicted Permission List* is the simulated permission list on a device, assuming that all recommendations of alternative apps are adopted by a user. The set difference of *Current Permission List* and *Predicted Permission List* will reflect any changes in the overall number of permissions used on a device.

Fig. 5.3 shows the number of permissions that could be saved and the proportion of devices that they could be saved on. No permission savings was possible on 37% of devices. Saving a permission requires all third-party apps on a device using that particular permission to have a functionally-similar alternative that does not use that permission. Some apps using a permission may simply not

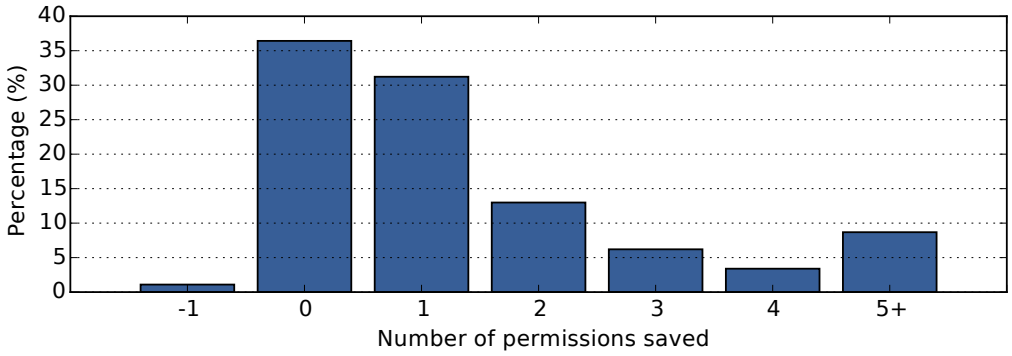



Figure 5.3: Change in the number of permissions no longer used on devices if all recommendations are followed.

have a replacement, or the permission in question may be necessary for providing core app functionality in one or more apps. However, approximately 31% of devices could save one permission altogether, and another 31% of devices could save two or more permissions.

A small proportion of devices would actually use more permissions (-1 in the figure) if all recommended alternative apps are used. This is because recommended alternative apps are chosen using the AOPP metric (as defined in Chapter 4). Using AOPP, an app that uses more permissions overall can still be considered to be less permission-hungry than another app if the permissions it uses are more similar to permissions used by functionally-similar apps than the other app. For example, a phonebook app using only the camera permission will be considered as being more permission-hungry than another phonebook app using two permissions for reading and writing contacts. This is because permissions for reading/writing contacts are more common among phonebook apps than the camera permission, and thus the former app would be penalised more heavily.

### Type of Permissions Saved

The next step is to understand what permissions would be saved if less permission-hungry apps were used according to SecuRank's suggestions. The

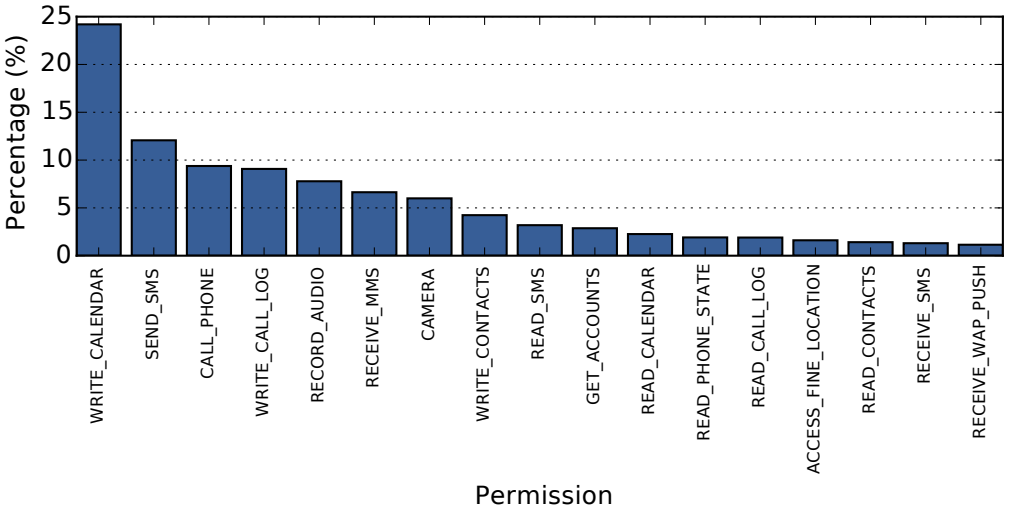



Figure 5.4: Details of which permissions were saved.

results of this analysis is summarised in Fig. 5.4. For clarity, permissions with less than 1% occurrence are omitted. `WRITE_CALENDAR` was the most commonly saved permission with 24.2%. Because of permission groups, this would save the `READ_CALENDAR` permission as well. Other notable permissions that could be saved include `SEND_SMS` (12.1%), `CALL_PHONE` (9.4%), `RECORD_AUDIO` (7.8%), and `CAMERA` (6.0%). These four permissions can be used to spy on a user or cost them financially, so being able to eliminate their usage on an entire device (while preserving the overall functionality of the device) is notable.

These results demonstrate that privacy risks on devices can be reduced by simply using functionally-similar alternative apps that are less permission-hungry. If permission-hungry apps continue to be used, however, their libraries will benefit from this extraneous permission usage as shown in § 5.1.1. Motivated by the observation that libraries benefit from extraneous permissions, in the following section I demonstrate how the problem of libraries benefiting from permissions is actually much worse than it would appear a first glance.

## 5.2 The Danger of Intra-Library Collusion

In § 5.1, we observed that some apps use extraneous permissions and that libraries embedded within these apps contained code that could leverage these permissions as well. However, there is a larger problem caused by the failure of the Android OS to separate privileges between apps and their libraries. Not only do libraries obtain permissions from their host app, but the same libraries, if installed in other apps on the same device, may obtain additional permissions from those apps as well. This opens the possibility for libraries to remotely aggregate multiple streams of sensitive data from a variety of apps on a single device. I call this phenomenon *intra-library collusion* (ILC).

ILC is not to be confused with collusion among or between apps, i.e., inter-app collusion. Inter-app collusion has been explored in prior work [47, 52, 79, 86, 103]. With inter-app collusion, apps collude locally to leak sensitive data from devices by leveraging inter-component communication (ICC) or other overt/covert local communication channels. Real-world attacks using inter-app collusion are limited, since they require one[2] or more malicious apps to be installed on devices to facilitate the attack.

On the other hand, ILC is more likely. This is because ILC does not require malicious apps to be installed on devices. Indeed, libraries that are embedded within apps already installed on devices are the attack vector. These libraries only need to access sensitive data (guarded by permissions) across the apps that they are embedded within, and transmit this data off the device where it may be aggregated remotely.

An example of how ILC could happen is shown in Fig. 5.5. From the figure, `library-2` is common across all apps, and consequently has access to a total of four permissions (`A,B,C,F`), although in any one app it has access to a maximum of two permissions (`A,B` or `A,C`).

[2] A single malicious app is sufficient for a confused deputy attack [49] to be performed if a benign, but vulnerable, app exists on a device.

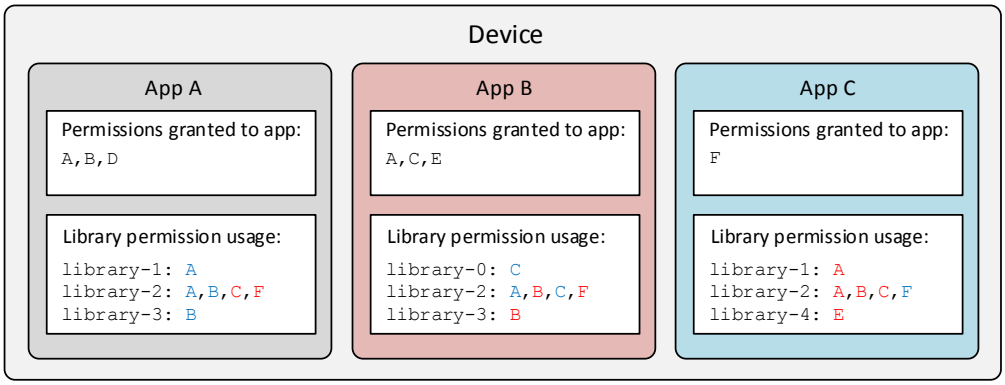



Figure 5.5: How intra-library collusion (ILC) happens in practice. Libraries are able to use permissions in blue because they have been granted to the app, while permissions in red are unavailable for libraries within that app.

Concretely, ILC is possible under the following constraints:

1. Two or more apps installed on a device, $A_1, ..., A_n$, have the same library, $L$, embedded inside them.

2. Apps $A_1, ..., A_n$ have sets $S_1, ..., S_n$ of permissions granted to them such that the union of the sets of permissions $U$ contains more permissions than any of the individual sets of permissions.

3. $L$ contains code that allows it to make API calls that are guarded by two or more permissions in $U$. One or more of these permissions should be unique across sets $S_1, ..., S_n$.

4. $L$ has the ability to uniquely identify a device.

5. $L$ transmits permission-protected data to a remote server.

Libraries escalating their combined privileges through ILC pose a greater threat to user privacy than apps colluding through ICC. This is because libraries are intended to be embedded within multiple apps. This in turn increases the likelihood that multiple apps on a device will contain the same library. Moreover, the existence of popular libraries with an aggregated tens or hundreds of millions

of installs suggests that the potential for ILC should be very common on devices in practice.

The problem of libraries leaking sensitive data is difficult to solve since libraries may have plausible reasons as to why sensitive data needs to be transmitted from a device. If the library provider then aggregates multiple streams of data from a user on their servers, this behaviour will be opaque to users and industry regulators alike.

Risks from traditional libraries are only one facet of the problem of ILC. Chen et al. [54] observed that libraries were also being repackaged to propagate malicious code. In their study, the authors estimated that 6.8% of apps in the Google Play Store were infected with potentially harmful libraries. Thus, libraries have also become an attack vector used by adversaries. Harmful libraries can serve to amplify the effect of ILC dramatically.

### 5.2.1 Methodology for Measuring Intra-Library Collusion

The following steps were taken to measure the potential for ILC on real-world devices:

1. Obtain the `apk` files for popular apps in the Google Play Store.

2. Identify libraries within popular apps and the permissions that these libraries are able to use, by virtue of the permission-protected API calls present in their code and the permissions used by the apps themselves.

3. Use lists of apps installed on real-world devices to understand the combined permissions that libraries have access to across devices.

In what follows, each of these steps in described in detail.

### Obtaining Apps for Analysis

Statically analysing all apps in the Google Play Store requires substantial processing and storage resources, and may ultimately lead to *diminishing returns*. For this reason, I opted to analyse only popular apps. Similar to previous chapters, apps having more than one million downloads were considered to be popular. By focusing on popular apps, the study captures a large cross-section of apps likely to be installed on the average smartphone, while limiting the amount of computing resources required for conducting the study. This will give a good indication of the major threats to a majority of smartphone users. While additional data is lost by not analysing unpopular apps, unpopular apps have more limited install-bases in the first place and thus will have a more limited contribution to ILC in any case. Since some devices will have unpopular apps installed, which may contribute additional risks that are not measured, the results presented in this chapter will actually represent a conservative estimate of the potential for intra-library collusion.

In the end, I was able to obtain 15,052 apps with more than one million downloads. This is all the apps in the Google Play Store with more than one million downloads, with the exception of several apps that failed to download due to geographical restrictions.

### Library Permission Usage

Identifying the permissions that libraries are able to use is a multi-step process. Merely analysing library SDK documentation does not paint a full picture of permission usage within libraries. Indeed, libraries may use undocumented permissions internally, and the app itself needs to declare (and have approved) the necessary permissions before the library can use them. Conversely, an app having been granted permissions does not mean that the library contains code that leverages these permissions. The steps for accurately identifying what permis-

sions libraries are able to access are already described in § 3.3.1. These permissions were then intersected with the permissions that are actually used by the app. Not all apps in my dataset of popular apps were successfully analysed due to technical limitations of the decompiler that was used. I was able to obtain library permission usage information for a total of 14,976 apps.

### Obtaining Real-World App Lists

Lists of apps installed on real-world devices were obtained from the Device Analyzer dataset as described in § 2.4.2.

### Assumptions

With the introduction of run-time permissions, users can selectively accept (or reject) permissions used by apps. Unfortunately, the Device Analyzer project does not collect per app data on which permissions have been accepted or rejected by users. Thus, in measuring ILC, I assumed that apps are granted all permissions listed in their manifest file as a matter of practicality. This may lead to an over-estimate of the number of permissions that are granted to apps and, in turn, to their libraries.

At the time of writing however, as shown in § 1.3, approximately 60% of apps in my dataset have a `targetSDK` of 22 or lower, meaning the aforementioned assumption is valid in a majority of cases. For the remaining 40% of apps, they need to be run on devices with Android 6.0 or higher for run-time permissions to be triggered. The majority of devices in Device Analyzer are not running Android 6.0, leading to the assumption being valid in even more cases. Finally, even with compatible apps and devices, users are known to accept run-time permissions for little reward [68]. Thus, I believe that this assumption regarding run-time permissions gives a fair estimate of the current problem of ILC.

Some of the data in the Device Analyzer dataset is old, and thus represents

devices with old versions of apps. However, in doing the necessary static analysis, I downloaded the newest versions of these apps. New versions of apps may have since been updated to use different permissions and libraries. For the measurement of ILC, I treat old devices as if they were running the versions of apps that were statically analysed. Of course this is not always the case, but in the worst case this assumption allows insight into what ILC would look like on devices with similar app lists but with the latest app versions installed. Moreover, given that apps are set to auto-update by default within the Google Play Store app, this assumption holds on devices with default settings.

### 5.2.2 Intra-Library Collusion Measurements

Table 5.2 quantifies the number of distinct libraries detected within apps in the dataset of popular apps in the Google Play Store. Devices had a mean of 23.6 detectable libraries within the popular apps installed on them. A library is considered popular if it is observed in 1% or more of the apps that were studied. Overall, there were 18 popular libraries across apps in the dataset. The most popular libraries were Facebook, Google Analytics, Flurry, and Chartboost. Popular ad libraries observed include InMobi, MoPub, Millenial Media, Heyzap, and TapJoy. Utility libraries were also popular, providing graphics rendering facilities (`com/unity3d` and `org/cocos2d`) and image loading/caching services (`com/nostra13/universalimageloader`).

Fig. 5.6 shows the prevalence of libraries that are able to benefit from ILC. Prevalence was calculated as the number of instances of distinct libraries able to benefit from ILC divided by the total number of instances where libraries were able to benefit from ILC. In most cases, `com/facebook` was able to benefit from ILC at 31.3%. Other libraries that can exploit ILC are `com/mopub` (21.8%), `com/flurry` (14.0%), `com/amazon` (10.8%), and `com/inmobi` (8.4%).

The Top 5 libraries able to benefit from ILC include MoPub, Flurry Analytics, and InMobi; these are known advertising/analytics providers. Note that this

Table 5.2: Listing of the most popular libraries detected within apps with more than one million downloads. Note that libraries detected in less than 1% of apps are omitted.

| **Library** | **% of apps** |
|---|---|
| com/facebook | 11.9 |
| com/google/android/gms/analytics | 9.8 |
| com/flurry | 6.3 |
| com/chartboost/sdk | 5.9 |
| com/unity3d | 5.2 |
| com/applovin | 3.5 |
| com/mopub | 3.1 |
| com/inmobi | 3.0 |
| com/google/ads | 3.0 |
| com/google/android/gcm | 2.7 |
| com/tapjoy | 2.4 |
| org/cocos2d | 2.4 |
| com/amazon | 2.0 |
| com/millennialmedia | 1.6 |
| org/apache/commons | 1.4 |
| com/heyzap | 1.4 |
| com/nostra13/universalimageloader | 1.3 |
| com/adobe/air | 1.0 |

merely demonstrates the potential for libraries to benefit from ILC, and does not quantify the extent of ILC in practice. However, given the fierce competition between ad libraries, it is conceivable that wily ad networks could exploit ILC by aggregating user data on ad library servers, maximising profits while evading detection.

I downloaded, decompiled, and manually investigated several popular ad library SDKs and was able to confirm that they already transmit multiple types of sensitive data (inclusive of nearby Wi-Fi networks) from devices. This means that data aggregation using ILC is already possible if these ad networks use existing data that is (already) being transmitted to library servers.

Table 5.3 shows the number of distinct libraries per device that are able to benefit from ILC. Approximately 42% of devices are not susceptible to the problem of ILC. However, note that a device can go from not being susceptible to being susceptible with the installation of a single new app. Approximately three

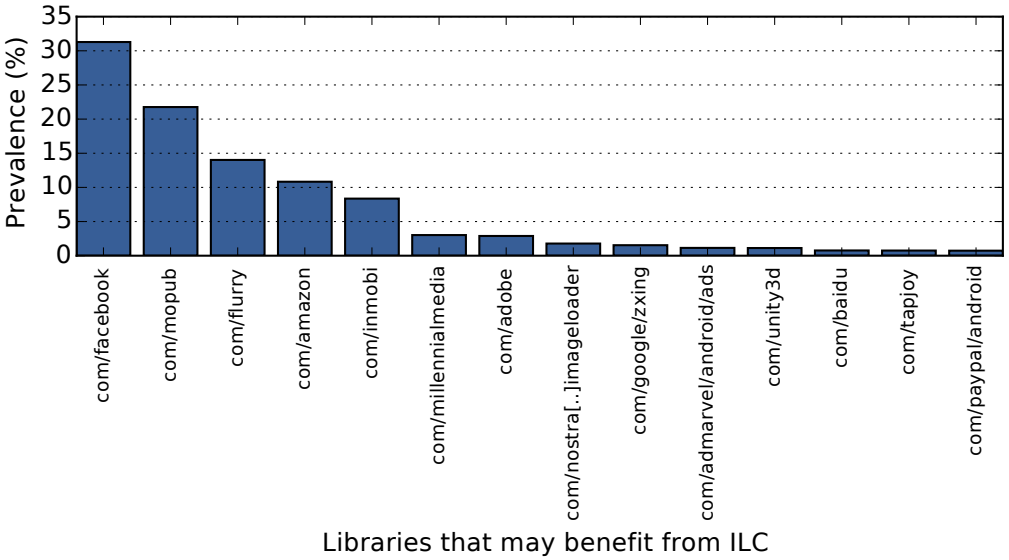



Figure 5.6: Libraries that are potentially able to benefit from ILC. For clarity, libraries appearing less than 0.5% of the time are omitted.

Table 5.3: Number of libraries per device that had increased access to permissions.

| **Number of Libraries** | **% of devices** |
|---|---|
| 0 | 42.4 |
| 1 | 20.7 |
| 2 | 16.5 |
| 3 | 10.5 |
| 4 | 5.8 |
| 5+ | 4.1 |

in five (57.6%) devices had one or more libraries that are able to benefit from ILC. At the extreme end of the scale, one in five (20.4%) devices had three or more libraries that are able to benefit from ILC. Translated to the real-world, this would amount to hundreds of millions of devices being vulnerable. At this scale, it becomes clear that exploiting ILC can reap a windfall for ad networks (and indeed other library developers) with the requisite guile.

Table 5.4 shows what increase in permissions libraries may gain by exploiting ILC. This is calculated as the difference between the total number of permissions a library can access when exploiting ILC and the maximum number of permissions the library has access to from any single app. In 69.2% of cases, libraries

Table 5.4: Number of additional permissions a library had access to beyond the single-app maximum.

| Number of Permissions | % of cases |
|---|---|
| 1 | 69.2 |
| 2 | 21.2 |
| 3 | 5.9 |
| 4 | 2.8 |
| 5+ | 0.9 |

were able to access one additional permission. There was a monotonic decrease in the number of devices vulnerable as the number of additional permissions increased. On 9.6% of devices, libraries would be able to access three or more permissions. This translates to hundreds of millions of devices in the real-world with libraries able to access three or more permissions through ILC.

### 5.2.3 Intra-Library Collusion Evolution

In addition to merely measuring the potential for ILC, it is also important to understand how it has changed over time. To measure this, I once again leveraged the freely available dataset [6] of historical versions of apps compiled using the PlayDrone tool [153]. This dataset was collected in October 2014. Thus, by measuring the potential for ILC on devices running these old versions of apps, and comparing it to the same devices running new versions of apps, we can see how the potential for ILC has changed over approximately two-and-a-half years (at the time of measurement).

Not all 14,976 apps in the current app dataset were available in the historic dataset of apps. This is simply because some apps in the current dataset did not exist at the time when the historic dataset was compiled. Overall, I was able to obtain library permission usage in 11,821 historic versions of apps. To properly calculate the evolution of the potential for ILC, it is necessary to take the intersection of the apps that exist in both the current and historic datasets (and were decompiled successfully). In the end, I was able to measure how the potential

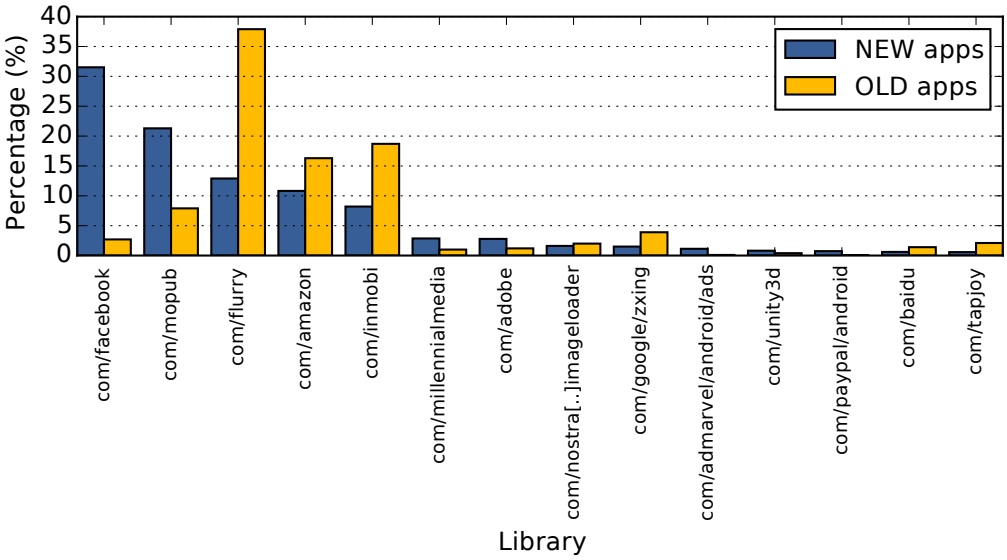



Figure 5.7: Longitudinal look at changes in the libraries that are able to benefit from ILC. For clarity, libraries appearing less than 0.5% of the time are omitted.

for ILC changed for 11,774 popular apps over a two-and-a-half year period.

Increases in the potential for ILC are not only caused by apps being updated to use more permissions. The libraries themselves embedded within the apps may have also been updated with additional code that makes use of new permissions. App developers may also change what libraries they use in their apps across app versions by adding new libraries (common) or removing libraries (less common). Adding and removing whole libraries can change the potential for ILC greatly, especially if these libraries use many permissions.

In what follows, `NEW` refers to ILC measurements using the current versions of apps, while `OLD` refers to ILC measurements coming from historic versions of apps. Fig. 5.7 shows the longitudinal changes in the potential for libraries to exploit ILC. The library `com/facebook` saw the largest increase in potential, increasing from 2.7% to 31.5%. The library `com/flurry` had the largest decrease, falling to 12.9% from 37.9%.

I manually investigated the four libraries with the greatest changes over the period to determine the reasons for the change. These four libraries

Table 5.5: Longitudinal look at the number of libraries per device that had increased access to permissions.

| Number of Libraries | OLD % | NEW % | % Change |
|---|---|---|---|
| 0 | 53.8 | 43.2 | -19.7% |
| 1 | 24.7 | 21.3 | -13.8% |
| 2 | 12.5 | 16.4 | +31.2% |
| 3 | 5.6 | 10.1 | +80.4% |
| 4 | 2.1 | 5.3 | +152.4% |
| 5+ | 1.3 | 3.7 | +184.6% |

were `com/facebook`, `com/mopub`, `com/flurry`, and `com/inmobi`. The library `com/facebook` had an increase in prevalence due to an increase in the number of apps using the library, as well as increased permission usage by the library itself. The ad library `com/mopub` saw increased prevalence for the same reasons: more apps used the library over time and the library's permission usage also increased. When it came to decreases in prevalence, `com/flurry` and `com/inmobi` saw decreases because of less apps using the respective libraries. For these two libraries, no change in the number of permissions used was observed.

Table 5.5 summarises the longitudinal results of the number of libraries per device that have the potential to benefit from ILC. There was a 19.7% decrease in the number of libraries that were *not* able to benefit from ILC. This suggests that library usage and permission evolution is such that it facilitates increases in the potential for ILC. There was a 13.8% decrease in the case of one library being able to benefit from ILC. At first glance, this might seem like a positive observation, but it comes at the expense of two or more libraries being able to benefit from ILC. In total, the case of two or more libraries benefiting increased from 21.5% to 35.5%, a 65% increase in two-and-a-half-years. This suggests that the potential for ILC increasing as well as its consequences.

Table 5.6 summarises the increase in the number of permissions that libraries are able to access over time if they exploit ILC. In the case of one new permission, the cases where this was possible fell from 86.5% to 68.5%. However, this decrease comes at the expense of libraries being able to access two or more addi-

Table 5.6: Longitudinal look at the number of additional permissions a library had access to beyond the single-app maximum.

| Number of Permissions | OLD % | NEW % | % Change |
|---|---|---|---|
| 1 | 86.5% | 68.5% | -20.8% |
| 2+ | 13.5% | 31.5% | +133.3% |

tional permissions, with an increase from 13.5% to 31.5% of cases. Once again, this highlights increasing consequences.

## 5.3 Summary

This chapter presented results from two main research threads. In the first research thread, improvements that can be made on real-world devices using my SecuRank tool was measured. The results showed that while extraneous permission usage in apps is prevalent, it can be mitigated by using functionally-similar alternative apps that are less permission-hungry. I further showed that extraneous permission usage provided non-trivial benefit to embedded libraries, underscoring the severity of the problem. The second research thread took this idea a step further and examined the combined access that libraries have to sensitive user data, if they leverage ILC. Over time, libraries were seen to be able to benefit from ILC across the apps they are installed within in more cases, and with increasing consequences.

This chapter confirms that smartphone users face several risks during the day-to-day usage of their devices. However, while these risks can be measured by experts, they are not readily apparent to users who are the primary victims. Requiring users to install and operate additional tools to measure and mitigate their risks will likely not be a scalable, effective, or well-received solution. In the next chapter, I investigate the feasibility of a system that can transparently and unobtrusively identify apps on user devices. Once apps have been identified, users can be automatically notified of the presence of risky apps on their devices, and can then take any necessary corrective action.

# CHAPTER 6

## AppScanner: Transparently Identifying Apps

*Patterns repeat themselves in history.*

– Rick Riordan, The Last Olympian

As shown in Chapters 3-5, many apps contribute to security and privacy risks on smartphones. In Chapter 3, we saw that apps are seeking greater access to devices while also getting more vulnerable at the same time. Users are often unaware of these risks, and continue to use their installed apps without knowning the extent of the problem. In Chapter 4, I demonstrated that many risky apps can be replaced with less risky alternatives that are functionally-similar.

In order to replace risky apps on devices, however, they must first be identified. Identifying risky apps currently requires users to manually iterate through their list of installed apps. This approach is both burdensome and error-prone for the average user. The SecuRank app [26] that I developed automates this process to an extent, but still requires manual effort from the user since they have to install the app and initiate the scanning process. Scanning also needs to

be repeated whenever the user installs new apps. What is needed is a scalable way of automatically scanning devices on behalf of users.

This chapter argues that apps can be transparently and unobtrusively identified on devices by analysing the network traffic that they generate. Thus, a user's device may be 'scanned' for risky apps transparently, without manual effort required on the part of the user. Importantly, the network traffic analysis techniques that I use to identify apps do not inspect packet payloads and only rely on coarse packet length information. Thus, transparently identifying apps on a device can be done while maintaining a high-level of user privacy.

At this point, I would like to remind the reader that the list of apps installed on a device is data that is itself not guarded by a permission on Android. Thus, my app identification system does not capture any information that is actively protected by the Android OS.

Since apps are identified using network traffic analysis, it follows that only apps that generate network traffic can be identified with the proposed system. While this is a limitation, it should be noted that (from my analysis) only 3.8% of apps with more than one million downloads do not have the `INTERNET` permission. Thus, only a small fraction of apps are incapable of sending network traffic. While these apps may also contribute security and privacy risks, they are inherently safer[1], since they have less opportunity to exfiltrate data and there is less opportunity for adversaries to attack them.

Most of the material in this chapter formed the basis of a journal paper, which was accepted for publication in the IEEE Transactions on Information Forensics and Security 2017 [148]. This journal paper is an extension of a conference paper published at the IEEE European Symposium on Security and Privacy (EuroS&P '16) 2016 [147].

[1] I note that apps without network connectivity can still be a part of attacks (such as confused deputy and collusion attacks), but this requires more effort from the adversary.

## 6.1 Design Requirements

The objective is to build an app fingerprinting system that can transparently identify apps from their network traffic. A secondary aim is for the system to preserve privacy as much as possible. For this reason, the system was engineered to not inspect packet payloads. Additionally, since there are millions of apps in the Google Play Store, with many of them receiving updates periodically, the app identification system must be highly-scalable and not require human intervention if and when apps need to be re-fingerprinted. Concretely, the design objectives of the system are outlined below:

1. The system should be able to identify apps from the network traffic that they generate.
2. The system should be able to identify apps without inspecting packet payloads.
3. The system should be highly-scalable, being able to create and update fingerprints for apps automatically.
4. The system should be able to identify apps in real-time or near to real-time.

I opted to use a machine learning framework that can be trained on patterns in the traffic coming from apps. Similar patterns, when later seen in unlabelled traffic, are used to identify the apps that generated them. This machine learning framework for the fingerprinting and identification of apps from their network traffic is called AppScanner.

## 6.2 System Overview

To perform traffic analysis, it is useful to divide network traffic into manageable pieces from which features can be generated and used to train classifiers. In AppScanner, the concepts of a *burst* and *flow* are used to divide traffic. Bursts

and flows have been used in prior work on traffic classification [140][60]. In what follows, bursts and flows are explained to the reader.

**Burst** - A burst is a group of packets (independent of source or destination IP address) that are transmitted/received in quick succession. Within a burst, the most recent packet must occur within a threshold of time, the *burst threshold*, of the previous packet. That is, a burst is a set of packets that are grouped temporally and a new group is only started if no new packet has arrived within the time set as the burst threshold.

This is illustrated in Fig. 6.2(b), where Burst A and Burst B can be seen separated by the burst threshold. No packets have arrived for sufficient time after the last packet of Burst A, so the next packet that arrives is automatically the beginning of Burst B. Bursts are used to divide network traffic into discrete, manageable portions that are then sent for further processing.

**Flow** - A flow is a subset of packets within a burst that have the same remote IP address. A flow is similar to a TCP session, except that a flow ends at the end of a burst while a TCP session can span multiple bursts. Flows typically have a duration of a few seconds, while TCP sessions can continue indefinitely. Flows are shown in Fig. 6.2(c), where Burst A can be seen to have three flows: Flow A1, Flow A2, and Flow A3. A single burst can have multiple flows if more than one app (or even the same app) initiates TCP connections in parallel. Thus, bursts need to be disentangled into their constituent flows before they can be used for further analysis.

### 6.2.1 Traffic Generation

The setup used to generate and collect network traffic from apps is shown in Fig. 6.1. The smartphone connects to the internet via a Wi-Fi access point (AP) that is connected to a workstation with internet access. The workstation is con-

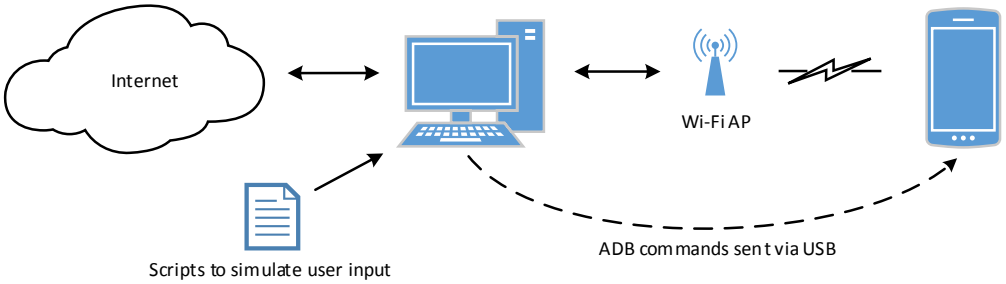



Figure 6.1: How network traffic is automatically elicited from apps.

figured to forward traffic from the AP to the internet. At the same time, scripts are run on the workstation that communicate with the smartphone over USB via the Android Debug Bridge (ADB). These scripts launch apps and simulate user actions within apps, thus eliciting network traffic from the apps. User actions such as button presses, swipes, and keypresses are sent to apps in an automated fashion. This is a technique called UI fuzzing.

The internet-bound traffic from the smartphone is intercepted and stored by the workstation before being passed to the internet. Details of the traffic such as time, source/destination IP address, packet size, protocol, port, and TCP/IP flags are captured. During this research, physical devices were used for data collection, but this process is highly-automatable by running apps within Android emulators. High automation of this process is facilitated by the aforementioned scripts that simulate user input, thus obviating the need for human intervention.

### 6.2.2 Fingerprint Making

The fingerprint making phase of AppScanner consists of several parts as summarised in Fig. 6.2. In what follows, I describe them in detail.

**Traffic capture**: One app at a time is run on the target device during traffic collection. This is to minimise 'noise' from other apps that may be running on the system. However, this does not guarantee a noise-free traffic dump, since the Android OS itself may generate traffic. To combat this, the Network Log

tool [21] was also installed on the target device. Network Log is a tool that can be used to identify the app responsible for each network flow on an Android device. By using data from Network Log and a 'demultiplexing' script, all traffic not originating from an app can be removed from the traffic dump for that app. In this way (and in contrast to related work), I am able to obtain perfect ground truth of network traffic coming from apps. After collection, network traffic was filtered to include only error-free TCP traffic. As an example, packet retransmissions caused by network errors were filtered.

**Traffic burstification and Flow separation**: Network traffic is then parsed to obtain bursts that can then be sent for flow separation. Falaki et al. [73] observed that 95% of packets sent by smartphones "are received or transmitted within 4.5 seconds of the previous packet". This suggests that a burst threshold of 4.5 seconds might be suitable. During analysis, I observed that a burst threshold of one second only slightly increased the overall number of bursts in network dumps when compared to a burst threshold of 4.5 seconds. This suggests that network performance has increased since the original study. Thus, I opted to use a burst threshold of one second to favour more overall bursts and allow bursts to be sent for flow separation more quickly. This increases the speed with which AppScanner can identify apps.

Bursts are separated into their constituent flows using remote IP address. Additionally, a maximum valid flow length was arbitrarily chosen to ensure that the system ignores abnormal flows in the real-world. While remote IP addresses are used for flow separation, they are not used as features to assist app identification. Information that could be gleaned from DNS queries is also not used. These decisions were taken to avoid relying on domain-specific knowledge, and thus make AppScanner more general-purpose and robust. They can, however, be used to increase system accuracy at the expense of robustness.

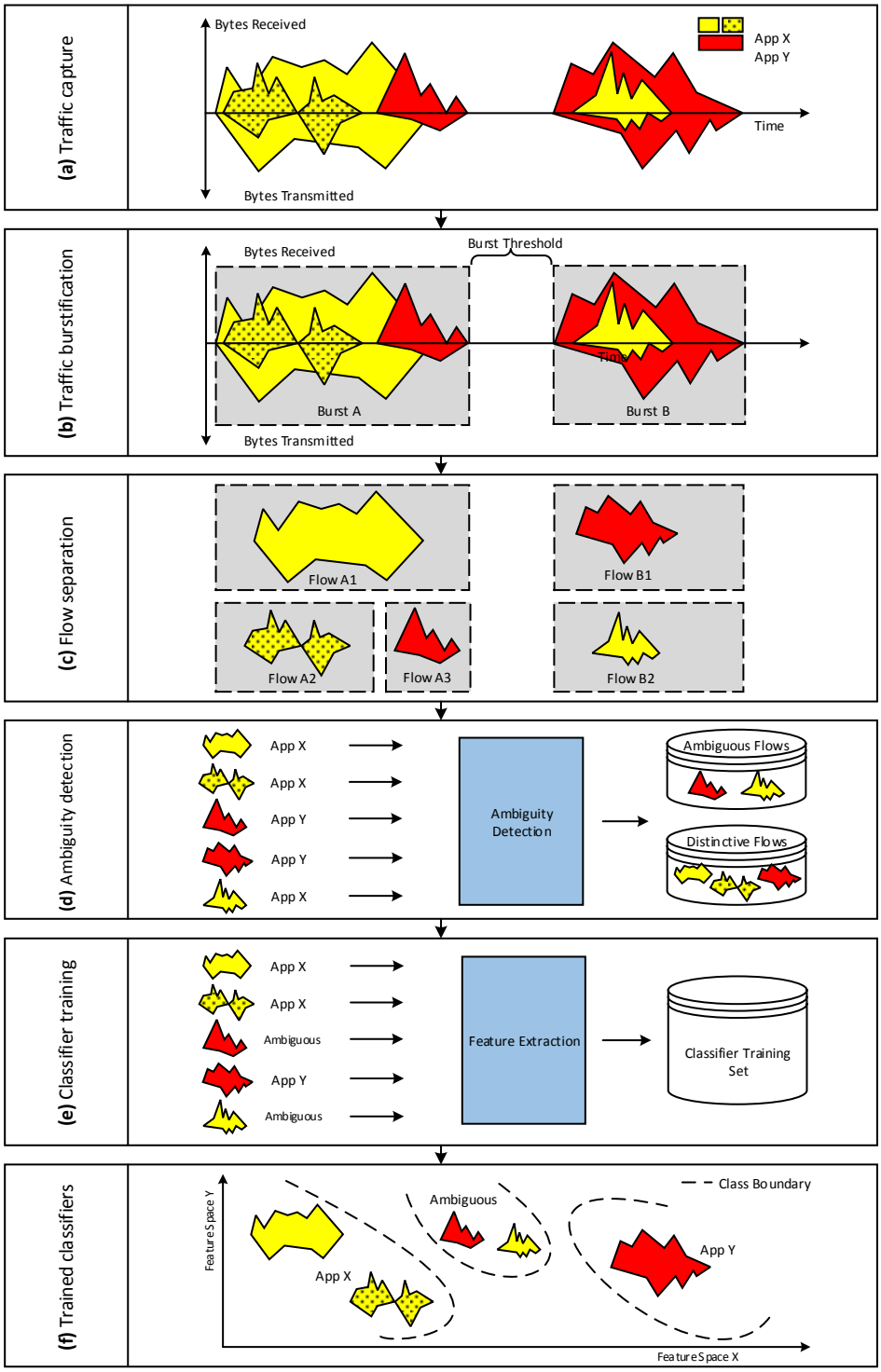



Figure 6.2: High-level representation of the steps in classifier training. (a) Network trace capture. (b) Traffic burstification. (c) Flow separation. (d) Ambiguity detection. (e) Classifier training. (f) Trained classifiers.

**Ambiguity detection**: Apps sometimes have traffic patterns in common because they share libraries, such as ad libraries, that generate similar traffic across apps. If left unchecked, this can frustrate app classification that relies on traffic analysis, since classifiers may be fed with conflicting labels for similar flows. Thus, I developed a strategy for detecting traffic that is similar across apps, so that it can be appropriately handled before being used to train classifiers. This traffic that is similar across apps is hereafter called *ambiguous* traffic. Traffic that is unique to an app is called *distinctive* traffic.

Ambiguous traffic poses a challenge for naive machine learning approaches. To counteract this problem, I devised an *ambiguity detection* strategy as detailed in § 6.3. This strategy uses simple reinforcement learning techniques to identify similar network traffic between apps. During classifier training, ambiguous flows are identified and relabelled as such, so that the classifiers are later able to identify and handle them.

**Classifier training**: The classifiers are trained with statistical features that are generated from the flows of each app. Statistical feature generation is depicted in Fig. 6.3. For each flow, three series of data generated from packets are considered: sizes of outgoing packets, sizes of incoming packets, and sizes of both incoming and outgoing packets. For each series (3 in total), the following statistics were computed: minimum, maximum, mean, median absolute deviation, standard deviation, variance, skew, kurtosis, percentiles (from 10% to 90%), and the number of packets in the series (18 in total). Statistical feature generation was facilitated by the Python `pandas` library [106]. At the end of statistical feature generation, arbitrary length flows are transformed into feature vectors with a length of 54. These feature vectors, along with their corresponding ground truth, are used as training examples for the classifiers.

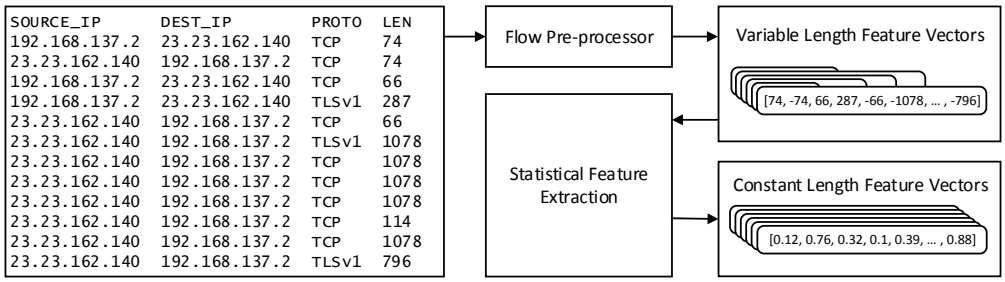



Figure 6.3: Generating features from flows for classifier training.

### 6.2.3 Fingerprint Matching

When identifying apps from unknown network traffic, flows are extracted and passed to the trained classifiers. Ambiguous flows are identified immediately, while unambiguous flows go through a final processing stage called *classification validation* as detailed in § 6.6.2. The classification validation stage is important because machine learning algorithms will attempt to assign a label to an unknown example, regardless of the algorithm's confidence in the match. Given that AppScanner will likely not be trained with the universe of flows[2] from apps, some unknown flows presented to the system may simply be unknown. These flows will be labelled as belonging to the app that is the closest match (as far as the classifier is concerned), but with low confidence. Blindly accepting low confidence predictions may cause an undesirable increase in AppScanner's false positive rate.

To implement classification validation, the *prediction probability* metric of the classifier is used to understand how confident the classifier is in its classification. If this value is below what I call the *prediction probability threshold*, AppScanner will not accept that classification as valid. If this value exceeds the threshold, AppScanner will report the flow as belonging to a particular app.

[2] Apps may also generate flows that are not observed during training and thus will not have been seen by the system. New or updated apps may also generate never-before seen flows. For these reasons, it is unlikely that AppScanner will be trained with the universe of possible flows from apps, thus motivating the need for classification validation.

## 6.3 Ambiguity Detection

Some apps will have flows in common with other apps, such as flows generated by the same library embedded within multiple apps. If left unchecked, this can degrade system performance since the classifier would be trained with conflicting labels for what is effectively the same training example. The ambiguity detection phase of AppScanner is responsible for identifying and relabelling such ambiguous flows. This phase uses a simple reinforcement learning strategy that is invoked during classifier training.

Ambiguity detection is outlined in Fig. 6.4. There are two classifiers: a *preliminary classifier* and a *reinforced classifier*. Ambiguity detection uses two independent datasets for training: Dataset 1 and Dataset 2. Dataset 1 is first randomly shuffled and divided into halves: the *preliminary training set* and the *preliminary testing set*. The preliminary classifier is trained with the preliminary training set. The preliminary testing set is used to validate the accuracy of the preliminary classifier and forms the basis of the reinforced training set.

The *Relabel Engine* leverages accuracy results from the preliminary classifier to identify ambiguous flows. Flows that are incorrectly classified by the preliminary classifier are relabelled "ambiguous" by the Relabel Engine. Flows that were correctly classified keep their original label, i.e., the app where they originated from. This relabelled dataset is the reinforced training set and is used to train the reinforced classifier. Thus, the reinforced classifier benefits from knowledge of ambiguous flows that the preliminary classifier failed to correctly classify. I manually validated that ambiguous flows largely came from IP addresses that suggest that they were indeed generated by libraries.

No flows from the preliminary training set were used in the reinforced training set. The preliminary training set and corresponding classifier are only used as a means of identifying ambiguous flows so that additional knowledge can be passed to the reinforced classifier during training.

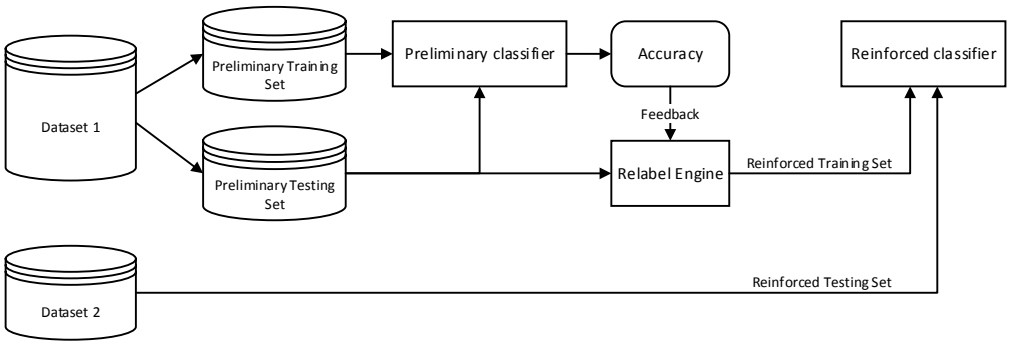



Figure 6.4: Using reinforcement learning to obtain robustness against ambiguous flows.

## 6.4 Dataset Collection

The performance of the AppScanner system was tested using a random 110 of the 200 most popular free apps as listed by the Google Play Store at the time of data collection. Popular apps were chosen since they constitute a significant proportion of the install-base of apps across the world. Free apps were used, because free apps are more likely to use ad libraries. There is a limited number of major ad libraries and thus free apps are more likely than paid apps to have shared libraries and thus generate more ambiguous flows. If AppScanner performs well in identifying free apps, it suggests that the system will also perform well on paid apps, since paid apps are less likely to have ambiguous flows. This is a reasonable assumption, because while other libraries that may generate ambiguous flows exist, from observation ad libraries are seen to send/receive significantly more traffic.

The physical smartphones used to generate my datasets were connected to the internet through a Linksys E1700 router/wireless AP. The router/wireless AP was connected to the internet through a Dell Optiplex 9020 workstation running Windows 7 Professional. UI fuzzing was performed on each of the 110 apps in turn for 30 minutes using `MonkeyRunner` from the Android SDK. The `MonkeyRunner` tool invoked touchscreen events simulating user interactions such

as swipes, touches, keypresses, and button presses. These touchscreen events were generated at random and caused navigation through the activities[3] of an app, thus eliciting network traffic.

Some apps required a login on first launch. In these cases, I manually created accounts for these apps and logged in. This was to ensure that the UI fuzzing process was not hindered due to a login screen. Automated login to apps that require it can be achieved heuristically, and this may be assisted in practice given that many apps offer the convenience of using social networks, such as Facebook to login.

Greater coverage of the activities within an app can be obtained using smart UI fuzzing techniques such as those offered by Dynodroid [105]. Human participants can also be recruited, but this is not a scalable approach. I consider these approaches that aim to increase the coverage of flows elicited from apps to be out of scope for this research.

### 6.4.1 Datasets Collected

To assess the suitability of AppScanner in the real-world, several datasets[4] were collected to understand how app fingerprinting is affected by time, device used, app versions, and various combinations of these variables. Thus, several datasets were collected to allow the measurement of system performance under these scenarios. The datasets that were collected are summarised in Table 6.1.

The first dataset collected is `Dataset-1`. This dataset was collected using *Device-A*, a Motorola XT1039 (Moto G) running Android 4.4.4. This dataset contains traffic generated by all 110 randomly chosen apps. All apps were the latest version of the app at the time of data collection. This time of initial data collection is known as $T_0$. The remainder of the datasets (`Dataset-2` to `Dataset-5`) were collected six months after the initial data collection, i.e., $T_0$ + 6 months.

[3] Activities refer to the various 'screens' that form the basis of an app's user interface.
[4] Datasets are available for download at `https://github.com/vftaylor/appscanner`.

Table 6.1: Descriptions of the devices, number of apps, app versions, and time of data collection for each dataset used. The Moto G (Motorola XT1039) device was running Android 4.4.4 and the LG E960 was running Android 5.1.1.

| **Name** | **Device** | **Apps** | **App versions** | **Collected at** |
|---|---|---|---|---|
| `Dataset-1` | Moto G | 110 | Current at $T_0$ | $T_0$ |
| `Dataset-1a` | Moto G | 65 | Current at $T_0$ | $T_0$ |
| `Dataset-2` | Moto G | 65 | Current at $T_0$ | $T_0$ + 6 months |
| `Dataset-3` | LG E960 | 65 | Current at $T_0$ | $T_0$ + 6 months |
| `Dataset-4` | Moto G | 110 | Current at $T_0$ + 6 months | $T_0$ + 6 months |
| `Dataset-4a` | Moto G | 65 | Current at $T_0$ + 6 months | $T_0$ + 6 months |
| `Dataset-5` | LG E960 | 110 | Current at $T_0$ + 6 months | $T_0$ + 6 months |
| `Dataset-5a` | LG E960 | 65 | Current at $T_0$ + 6 months | $T_0$ + 6 months |

`Dataset-2` differs from `Dataset-1` by time of data collection and the number of apps tested. `Dataset-2` only has data on 65 apps because the remaining 45 apps refused to run unless they were updated. These 65 apps that ran without being updated are hereafter called the *run-without-update* subset.

`Dataset-3` was collected with *Device-B*, an LG E960 smartphone running Android 5.1.1. `Dataset-3` used the *run-without-update* subset of apps. That is, the only difference between `Dataset-2` and `Dataset-3` is the device used to collect the data.

`Dataset-4` and `Dataset-5` were collected using the latest versions (as at $T_0$ + 6 months) of the 110 apps that were considered. These datasets were collected using *Device-A* and *Device-B* respectively.

Three variants of the datasets containing the full 110 apps were also considered. These variants contain only apps from the *run-without-update* subset, and thus have 65 apps each. These variants are called `Dataset-1a`, `Dataset-4a`, and `Dataset-5a` for `Dataset-1`, `Dataset-4`, and `Dataset-5` respectively. These datasets were constructed to allow balanced analysis when compared to the other datasets, `Dataset-2` and `Dataset-3`, that also had 65 apps.

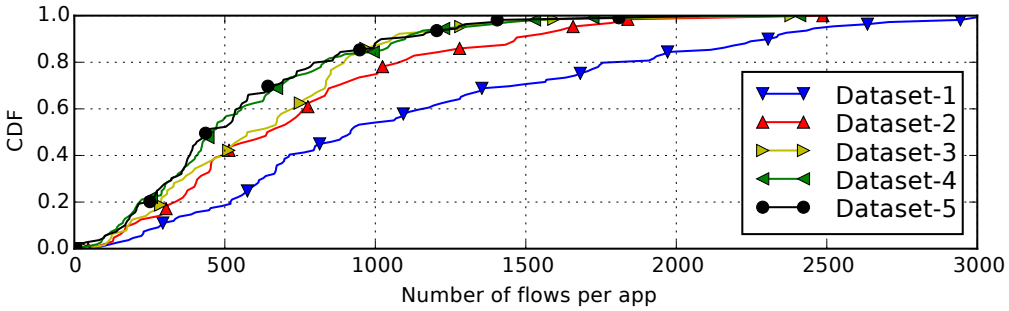



Figure 6.5: CDF plot showing the number of flows per app in each dataset.

### 6.4.2 Overview of Datasets

Fig. 6.5 shows a cumulative distribution function (CDF) of the distribution of the number of flows per app in each of the five datasets. `Dataset-1` had an average of 1132 flows per app, with 80% of apps having 500 or more flows. The remaining datasets had fewer flows per app on average. Note however, that since AppScanner works by analysing individual flows, even a single flow is suitable for app identification if that flow is unambiguous.

Note that all the considered apps generated a non-trivial number of network flows. This confirms that UI fuzzing is a useful technique for eliciting network flows from apps.

## 6.5 Evaluation

The *scikit-learn* machine learning library [120] was used to implement the classifiers used in AppScanner. The default scikit-learn parameters were used for all classifiers. In my previously published work on AppScanner [147], a variety of machine learning algorithms and classification approaches were explored. Random forest classifiers gave markedly better performance than support vector machines (SVMs). Thus, for brevity, I discuss only random forest classifiers in this thesis. For an in-depth analysis of various classification algorithms and approaches, I would like to direct the reader to the aforementioned paper. Ran-

Table 6.2: Summary of the suite of tests used to measure the performance of the app classification system. All training and testing sets were completely independent of each other. The independent variables (IV) for each test are identified. AV = app versions, DEV+AV = Device and app versions.

| **Test** | **Training** | **Testing** | **P(%)** | **R(%)** | **F1(%)** | **A(%)** | **IV** |
|---|---|---|---|---|---|---|---|
| TIME | Dataset-1a | Dataset-2 | 44.5 | 43.1 | 42.6 | 40.9 | Time |
| D-110 | Dataset-4 | Dataset-5 | 40.9 | 36.6 | 36.3 | 37.6 | Device |
| D-110A | Dataset-4a | Dataset-5a | 38.4 | 35.2 | 35.1 | 37.7 | Device |
| D-65 | Dataset-2 | Dataset-3 | 43.5 | 38.3 | 39.0 | 39.6 | Device |
| V-LG | Dataset-3 | Dataset-5a | 33.0 | 31.0 | 30.0 | 30.3 | AV |
| V-MG | Dataset-2 | Dataset-4a | 34.8 | 32.1 | 32.1 | 32.7 | AV |
| DV-110 | Dataset-1 | Dataset-5 | 23.3 | 19.5 | 19.3 | 19.2 | DEV+AV |
| DV-65 | Dataset-1a | Dataset-5a | 22.3 | 19.7 | 19.3 | 19.5 | DEV+AV |

dom forest classifiers are suitable for large multi-class classification tasks such as app identification, since they are inherently multi-class classifiers [48]. Random forest classifiers also use aggregated decision trees, which reduce bias. Finally, random forest classifiers natively output the probability of their decision being correct; a feature used in classification validation (§ 6.6.2).

In the results that follow, AppScanner is trained and tested using completely independent datasets, to give a better idea of the real-world performance of the system under various scenarios. Datasets were randomly shuffled and each test was run 50 times. The results presented give the average of the runs of each test. Tests were conducted to understand the impact on performance in various scenarios such as training and testing with different devices, app versions, or after time has passed. The various tests that were conducted and the performance of AppScanner in each case is summarised in Table 6.2 and discussed in what follows.

Note that the results presented in this section are baseline results only. That is, strategies for improving performance, such as ambiguity detection and classification validation, have not been applied. The aim of this section is to show how performance is impacted under different classification scenarios. Results when strategies for improving performance are utilised are given in § 6.6.

### 6.5.1 Effect of Time

The elapsed time between making app fingerprints and matching them can negatively affect classifier performance, especially considering that apps can be updated and change their behaviour as a result of the update. To measure the effect of time on classifier performance, AppScanner was trained using `Dataset-1a` and tested using `Dataset-2`. This kept device and app versions constant, while allowing six months to elapse between training and testing. This test, called `TIME`, had an overall accuracy of 40.9% and was the highest performing test in the suite of tests.

This test having the highest performance is not unexpected since app versions and device (including Android version) were constant. This suggests that as long as the logic (app and to a lesser extent operating system) remains the same, the network traffic produced can be expected to be similar even if time, six months in this case, has elapsed between training and testing.

### 6.5.2 Effect of a Different Device

It is important to understand how changing the device used between training and testing impacts app classification performance. This is because during fingerprint matching in the real-world, network traffic will come from a large set of devices. Three tests were conducted to assess the impact of changing the device used: `D-110`, `D-110A`, and `D-65`. Test `D-110` had 110 apps and used a training set of `Dataset-4` and a testing set of `Dataset-5`. The overall accuracy for this test was 37.6%.

The remaining two tests used the *run-without-update* subset of 65 apps. Test `D-110A` used a training set of `Dataset-4a` and a testing set of `Dataset-5a` and had an overall accuracy of 37.7%. Test `D-65` used a training set of `Dataset-2` and a testing set of `Dataset-3`. This test had an overall accuracy of 39.6%. The accuracy of these tests that used a different device is only 2.6% less (on

average) than the TIME test. This suggests that changing device does not have a significant negative impact on classification performance. This lends support to the real-world suitability of AppScanner, since a wide variety of Android devices exist in the real-world, and it would be computationally expensive if fingerprint making had to be done on a variety of devices.

### 6.5.3 Effect of Different App Versions

Across the app ecosystem, apps are constantly updated to fix bugs and provide improvements to users. Updating apps, however, can impact the network traffic that an app generates. To understand how changes in app versions affected classification performance, two tests were conducted: V-LG and V-MG. These tests used the same device, but with different app versions installed on training and testing devices. For test V-LG, a training set of Dataset-3 and a testing set of Dataset-5a was used. The overall accuracy of this test was 30.3%. Test V-MG used a training set of Dataset-2 and a testing set of Dataset-4a. This test had an overall accuracy of 32.7%.

Both tests in this category had similar performance, but their performance was notably lower than the TIME, D110, D110A, or D65 tests. This suggests that changes in app versions negatively impact app classification performance. This is expected, since newer versions of apps may have introduced or changed internal app logic, thus causing changes in the network traffic sent by the apps. This motivates the need for an automatic and scalable system, such as AppScanner, which easily handles app updates since human intervention is not needed for re-fingerprinting apps.

### 6.5.4 Effect of a Different Device and Different App Versions

Finally, I used two tests to measure the impact that changing both device and app versions had on classification accuracy. These tests were DV-110 and DV-65. These tests are the most challenging for the classifiers, since they use different

devices and app versions between training and testing. Test `DV-110` used all 110 apps that were considered and had a training set of `Dataset-1` and a testing set of `Dataset-5`. The overall accuracy for this test was 19.2%. Test `DV-65` used apps in the *run-without-update* subset and had a training set of `Dataset-1a` and a testing set of `Dataset-5a`. This test had an overall accuracy of 19.5%.

From the results, it is clear that changing both device and app versions at the same time has a more severe impact on classification performance. Interestingly, the two tests considered 110 and 65 apps respectively, but there was no notable difference in performance. This suggests that app classification can be a scalable process. Overall, these two challenging tests gave an overall accuracy of approximately half of that of the `TIME` test. However, while there was a performance penalty, these tests perform approximately 20 times better than random guessing (0.91%).

## 6.6 Improving Accuracy

The reader will recall from § 6.2.2 that the Network Log tool was used to filter noise from the traffic collected. While filtering noise is possible during training, it is infeasible during testing. Thus, this section examines the impact of noise under three (two real-world and one laboratory) experimental settings. The experimental settings used in this analysis are as follows:

1. Under the *noise-filtered* setting, all noise is removed from training and testing sets using Network Log as described in § 6.2.2. This is the 'perfect world' scenario where devices only have network traffic being generated by apps, and none from the operating system or other sources.

2. Under the *noise-ignored* setting, noise is removed from training sets but is left in testing sets. This setting is more real-world since it mimics the capabilities of an attacker in terms of when they are able to filter noise.

3. Under the *noise-managed* setting, classifiers are trained to understand what noise looks like. That is, noise is not filtered, but is instead identified and labelled as such. The classifiers will thus be able to identify noise coming from the Android OS itself. By training a classifier with examples of noise, additional accuracy can be extracted, since the classifier can now more accurately identify noise.

For brevity, in what follows I report on the performance of AppScanner using the *noise-managed* experimental setting (Table 6.5). The results for the other settings are similar and the reader can find them in Table 6.3 and Table 6.4.

In the previous section (§ 6.5), results were shown without AppScanner having the benefit of ambiguity detection and classification validation. In this section, the benefits of both strategies are highlighted.

### 6.6.1 Ambiguity Detection

Tables 6.3, 6.4, and 6.5 present results on the performance of AppScanner for each of the three experimental settings when ambiguity detection is applied. Ambiguity detection, as described in § 6.3, was applied to the training sets in each test and the performance of the resulting reinforced classifiers was measured. Each test was run 50 times with training and testing sets randomly shuffled and the results averaged.

Ambiguity detection improved system accuracy approximately two-fold. The lowest performing tests, `DV-110` and `DV-65` received the greatest boost in performance when ambiguity detection was applied. For example, `DV-110` saw accuracy increase from 19.2% to a maximum of 46.3%. This highlights the fact that ambiguous flows are present in network traffic, and motivates the need for systems that can identify and handle them. There is no free lunch, however, since ambiguity detection improves accuracy at the expense of the total number

Table 6.3: How the reinforcement learning strategy improved classifier performance for each of the tests that were conducted under the **noise filtered** setting.

| Test | Noise filtered | | | | Testing Set Details | |
|---|---|---|---|---|---|---|
| | P(%) | R(%) | F1(%) | A(%) | # Flows | % Ambiguous |
| TIME | 66.9 | 66.4 | 65.7 | 72.9 | 54240 | 58.3 |
| D-110 | 62.2 | 57.2 | 56.3 | 66.2 | 73811 | 59.6 |
| D-110A | 60.5 | 55.9 | 55.6 | 65.9 | 51995 | 60.4 |
| D-65 | 64.4 | 59.8 | 59.4 | 67.5 | 47018 | 58.6 |
| V-LG | 47.8 | 47.3 | 44.6 | 52.8 | 51995 | 61.9 |
| V-MG | 52.4 | 50.6 | 49.5 | 58.1 | 51731 | 61.8 |
| DV-110 | 37.0 | 35.0 | 33.2 | 41.0 | 73811 | 67.5 |
| DV-65 | 35.8 | 34.8 | 32.9 | 39.8 | 51995 | 67.2 |

Table 6.4: How the reinforcement learning strategy improved classifier performance for each of the tests that were conducted under the **noise ignored** setting.

| Test | Noise ignored | | | | Testing Set Details | |
|---|---|---|---|---|---|---|
| | P(%) | R(%) | F1(%) | A(%) | # Flows | % Ambiguous |
| TIME | 61.4 | 65.3 | 61.9 | 64.5 | 54240 | 58.3 |
| D-110 | 56.1 | 56.9 | 53.0 | 55.4 | 73811 | 59.6 |
| D-110A | 53.7 | 55.0 | 51.6 | 54.5 | 51995 | 60.4 |
| D-65 | 59.4 | 59.0 | 56.0 | 59.0 | 47018 | 58.6 |
| V-LG | 41.4 | 46.6 | 41.4 | 43.9 | 51995 | 61.9 |
| V-MG | 48.9 | 50.0 | 47.2 | 50.2 | 51731 | 61.8 |
| DV-110 | 32.8 | 34.8 | 31.0 | 32.4 | 73811 | 67.5 |
| DV-65 | 30.9 | 34.0 | 30.2 | 31.3 | 51995 | 67.2 |

Table 6.5: How the reinforcement learning strategy improved classifier performance for each of the tests that were conducted under the **noise managed** setting.

| Test | Noise managed | | | | Testing Set Details | |
|---|---|---|---|---|---|---|
| | P(%) | R(%) | F1(%) | A(%) | # Flows | % Ambiguous |
| TIME | 66.4 | 65.3 | 64.6 | 74.8 | 54240 | 58.3 |
| D-110 | 59.5 | 53.5 | 53.0 | 64.7 | 73811 | 59.6 |
| D-110A | 57.3 | 54.0 | 53.1 | 63.4 | 51995 | 60.4 |
| D-65 | 63.8 | 58.8 | 58.5 | 68.1 | 47018 | 58.6 |
| V-LG | 44.7 | 45.6 | 42.5 | 54.5 | 51995 | 61.9 |
| V-MG | 52.8 | 50.3 | 49.3 | 62.4 | 51731 | 61.8 |
| DV-110 | 35.0 | 33.7 | 32.0 | 46.3 | 73811 | 67.5 |
| DV-65 | 33.8 | 33.8 | 31.5 | 44.0 | 51995 | 67.2 |

of flows that the system is able to classify.

In a situation where an app only generates ambiguous flows, using ambiguity detection would cause the system to ignore all flows from that app and the app would be undetectable. Thus, it is important to evaluate the number of unambiguous flows per app to understand if AppScanner will be useful for detecting that app in the real-world. Fig. 6.6 shows the number of unambiguous flows per app for the *noise-managed* experimental setting. Other experimental settings gave similar plots and are omitted for brevity. For clarity, `D-110A` and `DV-65` are omitted since they are scaled down versions of other datasets. With the exception of test `DV-110`, all apps generated some amount of unambiguous flows.

There was a single app in `DV-110` that did not have any flows remaining after ambiguity detection was applied. To identify this app, one may re-fingerprint the app in the hope of eliciting additional potentially unambiguous flows. If that approach fails, ambiguity detection may be abandoned altogether, but this would come at the expense of system performance.

It is possible that some apps simply do not produce unambiguous flows and thus cannot be readily identified using the techniques employed by AppScanner. This is a limitation of app classification using network traffic analysis. However, note that the vast majority of apps indeed generate unambiguous flows and thus the system is able to identify the vast majority of apps in practice.

### 6.6.2 Classification Validation

Classification validation was another strategy used to improve classification performance. Classification validation takes the confidence that a classifier outputs with its prediction into account and uses it to determine whether the prediction will be considered as valid. In a situation where class boundaries are distinct, a classifier will be very confident in its predictions since the separation between classes is large. In other cases, class boundaries may be less clear and a classifier

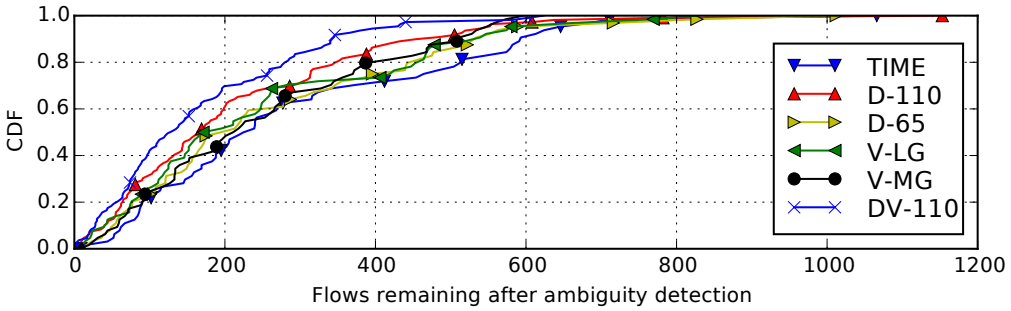
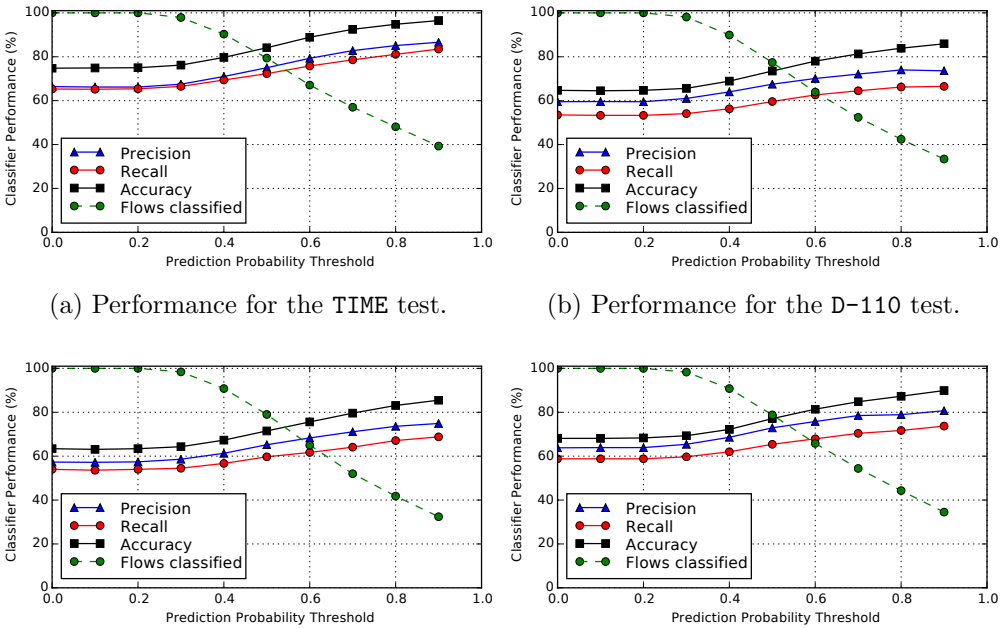



Figure 6.6: CDF plot showing the number of flows remaining per test after ambiguity detection was applied.

may be less confident in a prediction if an unknown example falls close to a class boundary.

Using classification validation, AppScanner only considers classifications as valid if the classifier itself gives a prediction probability (the measure of classifier confidence) that is higher than the cut-off threshold. This cut-off threshold is called the prediction probability threshold (PPT). A high PPT will cause the system to ignore all but the most confident classifications. This leads to a more conservative system with improved accuracy, but this performance increase comes at the cost of the total number of flows that the system is confident enough to classify. Conversely, a low PPT leads to a less conservative classifier with reduced performance. The classifier in this case, however, will be able to classify a greater proportion of flows. Note that while classification validation reduces the number of flows that are classified, there is no inherent requirement to classify all flows. Indeed, apps typically send tens of flows per minute and even a single distinctive flow is enough to classify an app. Thus, classification validation can be a useful strategy for improving performance while incurring negligible drawback.

Fig. 6.7 shows the performance improvement obtained using classification validation for the `TIME`, `D-110`, `D-110A`, and `D-65` tests. The `TIME` test had a baseline accuracy of 74.8% that was subsequently improved to 96.5% using classification validation at a PPT of 0.9. At this PPT, the system was still able




(a) Performance for the `TIME` test.

(b) Performance for the `D-110` test.

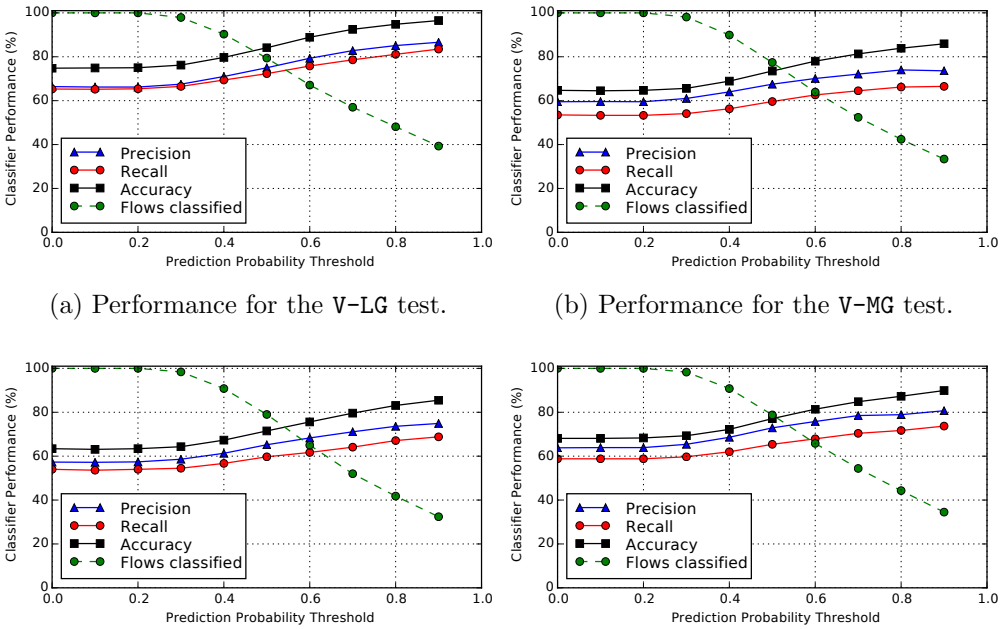
(c) Performance for the `D-110A` test.

(d) Performance for the `D-65` test.

Figure 6.7: Performance of the reinforced classifiers on the `TIME`, `D-110`, `D-110A`, and `D-65` tests.

to classify approximately 40% of flows. The improvement in performance when using classification validation at a similar PPT of 0.9 is also shown for the `D-110` and `D-110A` tests. Test `D-110` saw accuracy improve from 64.7% to 85.9% and test `D-110A` had accuracy increase from 63.4% to 85.5%. Finally, the `D-65` test has an increase in accuracy from 68.1% to 89.9% when using a PPT at 0.9. At a PPT of 0.9 with the increased performance, AppScanner was able to classify an average of 33% of flows for these final three tests.

Fig. 6.8 shows the performance improvement obtained using classification validation during the `V-LG`, `V-MG`, `DV-110`, and `DV-65` tests. For brevity, I report results considering a PPT of 0.9. For the `V-LG` test, accuracy was improved from 54.5% to 83.9%. For the `V-MG` test, classification validation improved accuracy from 62.4% to 85.5%. For the most challenging tests, `DV-110` and `DV-65`, classification validation was also useful in improving accuracy, increasing accuracy to 76.7% and 73.5% respectively. For these four tests, accuracy was




(a) Performance for the `V-LG` test.

(b) Performance for the `V-MG` test.

(c) Performance for the `DV-110` test.

(d) Performance for the `DV-65` test.

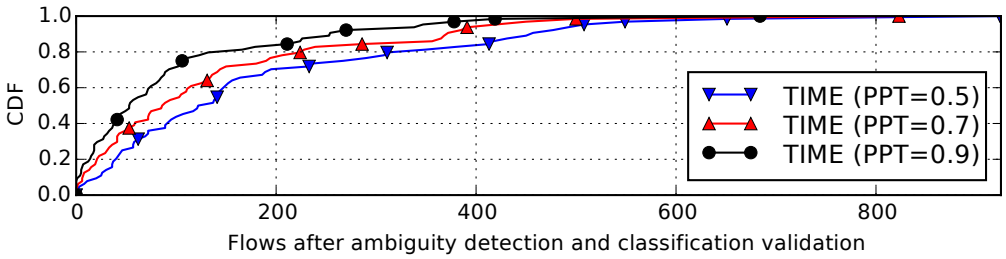
Figure 6.8: Performance of the reinforced classifiers on the `V-LG`, `V-MG`, `DV-110`, and `DV-65` tests.

improved markedly using classification validation, while the system was able to classify approximately 35% of the unlabelled flows that it saw.

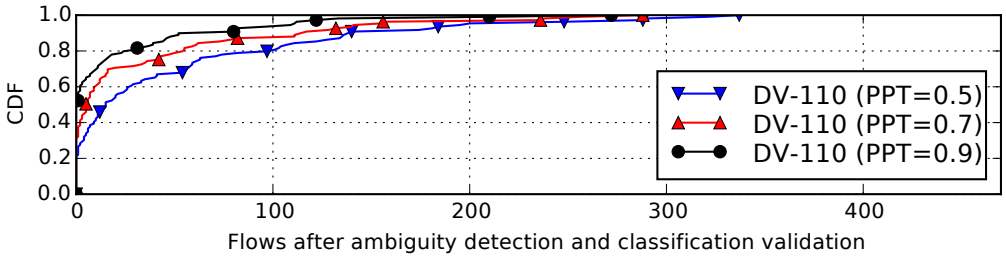
### 6.6.3 Considerations for Parameter Tuning

As mentioned previously, classification validation improves accuracy at the expense of the number of flows that the system can classify. Thus, it is important to quantify the extent to which classification validation affects the number of apps that can be classified by AppScanner. For example, it is possible that some apps have only flows that the classifier is not confident in classifying. For these apps, setting the PPT too high will render them unidentifiable.

Fig. 6.9 shows CDFs of the number of apps that AppScanner is able to classify at various PPTs (0.5, 0.7, 0.9) for the best and worst performing tests, `TIME` and `DV-110` respectively. Higher PPTs negatively impacted the number of flows that remained and were correctly classified across my dataset of apps.




(a) `TIME` test.



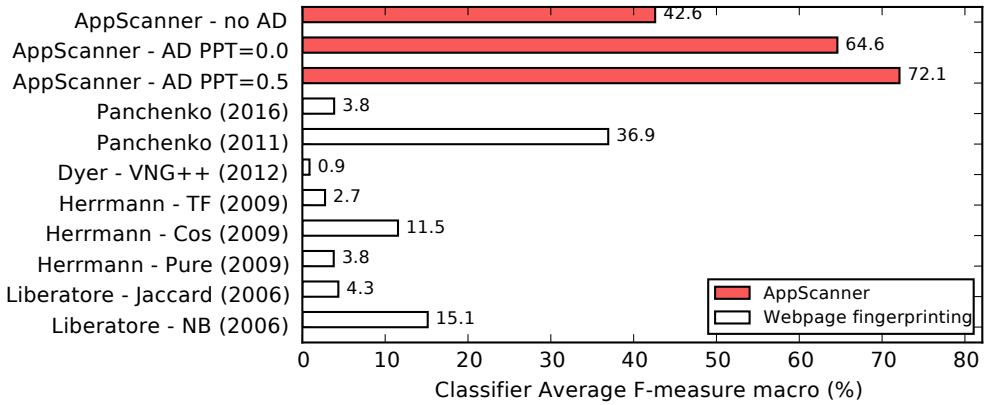

(b) `DV-110` test.

Figure 6.9: CDF plots showing the number of flows correctly classified per app after ambiguity detection and classification validation.

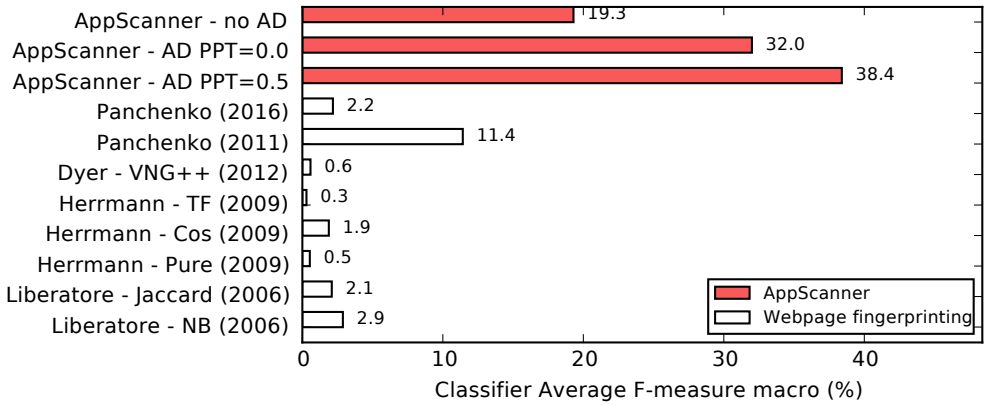
Indeed, setting the PPT to 0.9 for the `DV-110` test resulted in AppScanner no longer being able to identify approximately half of the considered apps. The reader should remember, however, that `DV-110` does not represent a real-world scenario. Indeed, `DV-110` uses outdated app fingerprints since the versions of apps have changed between training and testing without the fingerprints being updated. In practice, AppScanner would be automatically updated with new fingerprints as apps got updated.

The `TIME` test thus gives a more realistic indication of the real-world performance of AppScanner. This test failed to accurately identify only six apps at a PPT of 0.9 and two apps at a PPT of 0.5. In a case where it is vital to identify particular apps, the PPT needs to be tuned after understanding how the system performs for those particular apps. Thus, there is a trade-off in the maximum accuracy achievable and the number of apps that the system can classify.

Like with ambiguity detection, apps that are difficult to accurately identify can be subjected to more intensive UI fuzzing (and even manual exercising of




(a) Performance for the `TIME` test.




(b) Performance for the `DV-110` test.

Figure 6.10: Comparison with related work for the `TIME` and `DV-110` tests. AD = Ambiguity Detection and PPT = Prediction Probability Threshold.

the UI) in an effort to obtain more distinctive flows. New distinctive flows will likely improve the ability of the system to identify these apps. However, there will likely be rare cases where some apps pose a significant challenge for identification. Additional features, such as packet timing, remote IP address, and the like could be employed to attempt to identify these apps.

### 6.6.4 Comparison with Related Work

The literature exemplifies traffic analysis techniques in the domain of website fingerprinting. At first glance, one might think that traffic analysis of smartphone apps is a simple translation of existing work. In what follows, I show

how existing work on website fingerprinting suffers degraded performance when applied to smartphone traffic datasets. The comparison is shown in Fig. 6.10. Results for the best and worst performing tests (`TIME` and `DV-110`) are shown.

Panchenko et al. [116] performed best achieving F-scores of 36.9% and 11.4% in the `TIME` and `DV-110` tests respectively. Without ambiguity detection and classification validation, AppScanner already outperformed this approach. Adding ambiguity detection and classification validation widened the gap in performance.

## 6.7 Summary

In this chapter, I demonstrated that transparently identifying apps from their network traffic is possible and argued that this process can be automated and highly-scalable. This opens the door to transparently identifying apps of concern on devices. Combining this approach with insight into app behaviour and characteristics from Chapters 3-5, user devices can be automatically scanned and a determination made on the riskiness of apps on their devices.

A system such as AppScanner can be employed at various levels. For example, if AppScanner is implemented in a smart-home router, users can be automatically notified, perhaps via email or text message, whenever a concerning app is identified on their device. At a different level, AppScanner can be implemented in enterprise networks, whereby devices can be scanned automatically and action taken against offending apps running over the enterprise network, according to whatever company policy is in place. Identification of apps could also happen at a larger scale if done by ISPs.

A client-side tool like SecuRank is the preferred approach to scanning user devices for apps, since it would be able to get perfect ground truth. AppScanner, on the other hand, does not have perfect accuracy, but can achieve reasonably good accuracy especially if it is able to observe devices over a longer period

of time. While a client-side tool is ideal, it is challenging to motivate users to install and routinely operate such a tool in a way that gives them the full benefit. Additionally, a client-side tool can highlight undesirable apps to a user, but the final decision rests with the user and their determination of the costs and benefits to make the change. Using AppScanner, a network administrator can directly take action against offending apps if a user is not motivated to do so themself.

Admittedly, a system like AppScanner has the potential to diminish user privacy since it can uncover apps that are installed on a device. Installed apps can leak information about a user's religion, sexual orientation, political affiliation, and the like. Thus the benefits and risks need to be carefully measured before deployment. On the positive side, malicious, insecure, or otherwise undesirable apps can be identified and blocked automatically at the network level without any manual effort required from the user. This will serve to automatically and transparently improve their security and privacy. Further investigation on the disadvantages of AppScanner in the real world is considered out of scope.

Since users often fail to realise the risks they face when they use their devices on a day-to-day basis, a system that is able to transparently and unobtrusively scan their devices and offer solutions to mitigate their risks is very useful. AppScanner, when combined with the insights and systems presented in Chapters 3-5 does exactly this, and can be a critical step in improving user privacy and security in app ecosystems.

# CHAPTER 7

# Conclusion

*The plural of anecdote is not data.*

– Roger Brinner

Smartphones have revolutionised the way we communicate, do business, and receive entertainment. Billions of users rely on millions of apps on a daily basis. Apps, however, come at a price. This price may not necessarily be an upfront monetary cost, but may be in the form of security risks or privacy leakages that come about from simply using apps. As Chapter 3 shows, apps are getting riskier over time, both in terms of permission usage and vulnerabilities contained, but end users are not necessarily aware of this fact.

While the Google Play Store offers several layers of protection against outrightly malicious apps, ostensibly benign apps also have their own risks, which the Google Play Store is not currently equipped to properly address. These risks may come about from intentional means, such as app developers requesting additional permissions to facilitate new functionality, or unintentional means, such as

app developers unwittingly publishing apps containing vulnerabilities. Equally concerning are libraries that are embedded within apps, which obtain the privileges granted to the apps. These libraries, in many cases, offer no transparency as to what goes on behind the scenes, while being known to sometimes abuse the privileges they have been granted.

These shortcomings of the Android app ecosystem need to be understood using data-driven approaches if they are to be addressed properly. While there have been observations from time-to-time of apps behaving suspiciously [163], or a sense that apps are becoming more permission-hungry, these are largely anecdotes or small-scale observations that need to be investigated using real-world data.

## 7.1 Key Findings and Implications

In what follows, my original contributions and main observations from the systematic analysis of real-world data are summarised for the reader.

**1) Apps are using more permissions and getting more vulnerable.**

My first original contribution is a longitudinal analysis of app evolution across the Google Play Store. In the longitudinal study, apps were seen to become more permission-hungry over time. Free apps and popular apps were most likely to add new permissions. Apps were seen to favour adding location permissions and permissions to access a device's external storage. While run-time permissions attempt to fix the problem of permission-hungry apps, apps not implementing run-time permissions and a lack of user comprehension of the permission system make permission-creep a challenge.

More difficult to address is the problem of apps becoming more vulnerable over time. As the Google Play Store gets better at identifying malware and the

Android OS gets more resilient against attacks, attackers may well turn their attention to exploiting vulnerabilities within apps and use them as a proxy to access the permissions that have been granted to the apps.

One potential solution to the problem of vulnerabilities within apps is to have app stores perform vulnerability scanning of apps to determine the level of risk they pose to a device. The permissions used by an app, embedded libraries, and whether the app supports run-time permissions could also be factored into this risk signal. App store search results could then be revised by using the riskiness of an app as a ranking signal. This would offer the necessary incentives to app developers to make their apps more secure and privacy-friendly.

Many general-purpose apps exist, such as the now-infamous 'flashlight' app, which are blatantly over-privileged, but will continue to rank highly in search queries because of the sheer number of downloads they have received so far and the fact that they were among the first to offer that functionality in the app store. Surely these are less important metrics than app stores make them out to be, especially when considering the importance of user privacy and security.

**2) Many risky apps can be replaced with alternatives.**

Until now, it was unclear the extent to which general-purpose apps have less risky functionally-similar alternatives that could replace them. It is one thing to identify undesirable characteristics within apps, but if there are no suitable alternatives, a user may be forced to continue using that app. I found that approximately 6% of apps ranked #1 for a search query used rare permissions when compared to functionally-similar apps. Apps that are ranked #1 in search results receive a significant proportion of user downloads and it therefore follows that many apps using rare permissions have received significant downloads.

Across the Google Play Store, the likelihood of an app having a less permission-hungry functionally-similar alternative monotonically increases with

the number of downloads that the app has. At the upper end of popularity, approximately 50% of apps have a less permission-hungry functionally-similar alternative. Thus, there is significant opportunity to improve the security and privacy of apps on user devices at a large scale.

To address the challenges of permission-hungry apps, especially those that fail to support run-time permissions, or those that contain libraries that use extraneous permissions, several steps could be taken. Users could arm themselves with the SecuRank app that I developed while conducting the research for this thesis. SecuRank reduces the burden required for finding functionally-similar alternative apps, and makes it clear why a particular app is being suggested over the one that the user is currently using.

SecuRank can be a valuable tool to empower users and its approach to identifying permission-hungry apps could be applied to app stores in general. Subsequent to the release of SecuRank, the Google Security and Privacy team announced that they are heading in this direction [121]. It remains unclear, however, how app developers will respond to this development, and whether they may attempt to game search ranking algorithms when this feature is introduced. In the case of developers wanting to reduce how permission-hungry their apps are, in order to comply with new app store guidelines, SecuRank can be a valuable tool since it gives immediate and actionable insight on how a developer's app compares to other functionally-similar apps.

**3) Permission-hungry apps and libraries cause emergent problems.**

Prior to this work, the combined and emergent risks coming from the group of apps installed on devices was an understudied area. Using a corpus of approximately 30,000 devices, I demonstrated that permission-hungry apps are a problem, even in cases where their extraneous permissions are being used to provide legitimate features and the app itself does not misuse the data.

Mitigating extraneous permission usage while allowing apps to provide additional functionality can be facilitated by using `Intents` to trigger helper apps, without needing the app itself to request new permissions. Having apps request permissions themselves opens the door to their embedded libraries accessing these permissions. I showed that approximately 26% of apps contain libraries that can gratuitously leverage extraneous permissions. In most cases, these extraneous permissions allowed libraries to get the user's current location. I further showed that users could achieve permission savings on their devices in general, if they replaced their permission-hungry apps with less permission-hungry functionally-similar alternatives.

Along similar lines, I looked at an emergent risk that comes from libraries called intra-library collusion, and showed that 57.6% of devices are susceptible. The problem of intra-library collusion mainly comes from the failure of the Android OS to separate the privileges between libraries and apps.

Until a privilege separation mechanism becomes reality, users will continue to be vulnerable to aggressive data collection by libraries. However, privilege separation between apps and libraries is easier said than done, because it opens the door to users disabling advertising on their devices altogether. This may end up hurting the app ecosystem [98].

**4) Devices can be transparently scanned for risky apps.**

Identifying risky apps on devices currently requires manual effort from the user. If done by the user themselves, this is an error-prone and burdensome task, which has to be repeated whenever the user installs new apps. I contributed an approach to transparently and unobtrusively identify apps installed on devices by leveraging the network traffic that they generate. Traffic analysis, typically a tool used by adversaries for malfeasance, was used in this thesis for the opposite effect.

I showed that it is possible to use traffic analysis to identify the apps running on a smartphone. Crucially, my system is able to do this in the presence of encrypted traffic, since it does not inspect packet payloads. I showed that fingerprints for apps can be built in a scalable way and that apps can be identified at a later date with up to 96% accuracy. If implemented, traffic analysis could obviate the need for users to routinely scan their devices to determine whether undesirable apps were present. This increases coverage, usability, and deployability. However, it comes at the cost of reduced accuracy, since an alternative like SecuRank is the 'gold standard', with 100% accuracy.

If employed at an enterprise or ISP level, users could automatically be scanned and notified of their use of risky apps. Whether users take heed and replace these apps is another challenge, which is outside the scope of this research. However, disincentives could certainly be imposed at the enterprise level, for example, by disallowing BYOD devices from the company network if they have risky apps (potentially posing a risk to the company network) installed.

While the reader may have concerns about the privacy implications of being able to identify the list of apps installed on a device, I reiterate that this is information that is already available to any app running on a device, without the need for a permission to be granted. Thus, no data considered to be sensitive by the Android OS is uncovered by my traffic analysis.

## 7.2 Applicability to Other App Ecosystems

In this thesis, I examined security and privacy concerns in the official Android app ecosystem, which is supported by the Google Play Store. While this is the largest app ecosystem at the time of writing, other large app ecosystems also exist. There is the iOS app ecosystem supported officially by the Apple App Store, as well as other app ecosystems for both Android and iOS supported by unofficial app stores. Several of the tools and techniques described in this thesis

could potentially be adapted to improving the security and privacy of these ecosystems as well.

In general, app ecosystems featuring apps that are governed by a permission-based access control model can have their apps investigated from a permission-use standpoint. The apps can also be compared to their functionally-similar alternatives. Given that apps are typically indexed in app stores and listed with descriptions of their functionality, the approach to finding functionally-similar apps that I presented will also largely remain applicable.

On non-Android platforms, what constitutes undesirable characteristics in apps may need to be revisited, as would what constitutes security vulnerabilities. In general, however, the metrics used would likely relate to how well apps directly and indirectly treat user data. As it relates to libraries embedded within apps, these will continue to contribute risks if they obtain the privileges that have been granted to the apps. Given that advertising is a popular revenue generation mechanism across existing app ecosystems, the problem of advertising networks trying to obtain greater profiling information is likely to remain. Thus static and dynamic analysis tools that can uncover and track permission usage in libraries will remain important.

Identifying apps using traffic analysis is likely to translate across platforms as well, since apps typically send and receive data via the internet regardless of platform. While the general principles of fingerprinting apps would remain the same, platform-specific tools for UI fuzzing and identifying what network flows came from what app would need to be leveraged.

In contexts where sideloading of apps is commonplace, users will not receive the benefit of the vetting offered by official app stores. This introduces additional risks, since unofficial app markets may have higher rates of malware. To combat malware on devices with sideloaded apps (and on devices in general), the AppScanner system could be trained to identify malicious apps from their network traffic. Users could then be notified of the presence of malware on their

devices. Better yet, malicious traffic could also be directly dropped by a firewall that leverages AppScanner, thus mitigating the effect of the malware.

## 7.3 Directions for Future Work

The work presented in this thesis could be directly extended in several ways. In Chapter 3, permission usage evolution was studied with a granularity of three months over a three-year period. While this is useful for getting a general impression of the phenomenon, more fine-grained snapshots of the Google Play Store would be able to reveal more ephemeral events that may be interesting. Additionally, differences in permission usage across the Google Play Store could be correlated with major real-world events, such as the introduction of run-time permissions. My study of vulnerability evolution involved apps with versions taken two years apart. Being able to analyse new versions of apps as soon as they are updated would allow for more rapid discovery of vulnerabilities, and this fine-grained data may yield additional insights.

In Chapter 4, functionally-similar apps were discovered by performing text mining on app descriptions. From there, permission-hungry apps were identified using contextual permission analysis. While this approach works in general, it sometimes yields false positives in grouping functionally-similar apps, and thus more advanced strategies could be employed for finding functionally-similar apps, such as using peer group analysis [92]. Additionally, user feedback about app suggestions from SecuRank could be incorporated to identify and remove inappropriate suggestions. Since the chapter only focused on ranking apps based on their permission usage, additional metrics that capture a wider variety of security and privacy risks could be explored for determining the riskiness of an app. Additionally, an app's support for run-time permissions could be an interesting metric.

The observations of Chapter 5 could also be extended along several direc-

tions. It was observed that many apps installed on devices could be replaced with functionally-similar alternatives that were safer from a permission-use perspective. However, it remains unclear whether users would be motivated to make changes and whether they would be happy with using alternative apps. User studies could be conducted to determine whether users would be likely to implement recommendations of alternative apps. Additionally, user studies could shed light on the major factors that would influence a decision to replace an app with a functionally-similar alternative in the first place.

As it relates to intra-library collusion, it would be interesting to measure the extent to which it is actually exploited in the real-world. This may be challenging, since aggregation of user data would be happening on third-party servers, and thus would be mostly opaque. However, differential analysis techniques, such as those provided in [96], could be utilised. In a similar vein, it is worth exploring privacy-preserving ways of allowing libraries to access sensitive user data [102] so as not to harm the economics of app development [98].

Finally, Chapter 6 demonstrated a scalable method for identifying apps from their network traffic. Future work could involve more thorough methods of UI fuzzing that give greater code coverage of the activities within apps. This would lead to more network traffic from apps overall, which would likely assist app classification. Additionally, since apps were identified from single flows, it would be interesting to determine whether accuracy could be further improved by considering groups of flows that happen together to obtain additional features.

While Android was the focus of my research, many of the analyses presented could also be performed on the Apple App Store and indeed other app stores and mobile operating systems. It would be interesting to measure the real-world differences in the privacy and security characteristics across app ecosystems.

## 7.4 Closing Remarks

Apps ecosystems have provided a wide range of benefits to users. However, they have also provided the means for third-parties to have their code delivered to billions of devices. Malicious apps are not the only problem, since even benign apps can contribute security and privacy risks.

This work has uncovered several shortcomings in protecting user security and privacy in the Android ecosystem. Importantly, this work has also shed light on large-scale security and privacy characteristics of the ecosystem, and highlighted strategies that can be employed to improve its health. If the app ecosystem model of software delivery remains dominant on smartphones or becomes commonplace on other form-factors of device (such as traditional workstations), the issues identified will need to be addressed if we are to preserve the security and privacy of users now and into the future.

# References


[1] Access Control Cheat Sheet. https://www.owasp.org/index.php/Access_Control_Cheat_Sheet. [Accessed: 25-Sep-2017].

[2] Adobe PhoneGap. https://phonegap.com/. [Accessed: 07-Jul-2016].

[3] AndroBugs Framework. https://github.com/AndroBugs/AndroBugs_Framework. [Accessed: 01-Jul-2016].

[4] Androguard. https://github.com/androguard/androguard. [Accessed: 15-May-2016].

[5] Android and Security. http://googlemobile.blogspot.co.uk/2012/02/android-and-security.html. [Accessed: 15-Jan-2015].

[6] Android Apps. https://archive.org/details/android_apps. [Accessed: 07-Jan-2016].

[7] Android Manifest. https://developer.android.com/guide/topics/manifest/manifest-element.html. [Accessed: 07-Nov-2016].

[8] Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/. [Accessed: 12-May-2016].

[9] App Manifest. https://developer.android.com/guide/topics/manifest/manifest-intro.html. [Accessed: 03-Oct-2017].

[10] Appcelerator Open Source. http://www.appcelerator.org/. [Accessed: 07-Jul-2016].

[11] Common App Rejections. https://developer.apple.com/app-store/review/rejections/. [Accessed: 08-Mar-2017].

[12] Device Analyzer for Android. https://deviceanalyzer.cl.cam.ac.uk/. [Accessed: 05-Sep-2017].

[13] Facebook - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.facebook.katana. [Accessed: 25-Jan-2015].

[14] Google Android 6.0 Marshmallow Release Date, Price and Specs. https://www.cnet.com/uk/products/google-android-6-0-marshmallow/preview/. [Accessed: 02-Oct-2017].

[15] Google Play Apps Crawler. https://github.com/MarcelloLins/GooglePlayAppsCrawler. [Accessed: 13-Jan-2015].

[16] History of Computers. https://homepage.cs.uri.edu/faculty/wolfe/book/Readings/Reading03.htm. [Accessed: 06-Sep-2017].

[17] How Google Search Works. https://www.google.com/search/howsearchworks/. [Accessed: 03-Oct-2016].

[18] Let's Encrypt - Free SSL/TLS Certificates. https://letsencrypt.org/. [Accessed: 05-Sep-2017].

[19] Mallodroid: Find broken SSL certificate validation in Android Apps. https://github.com/sfahl/mallodroid. [Accessed: 30-Jun-2016].

[20] Mobile Security Framework. https://github.com/ajinabraham/Mobile-Security-Framework-MobSF. [Accessed: 28-Jun-2016].

[21] Network Log. https://play.google.com/store/apps/details?id=com.googlecode.networklog. [Accessed: 09-Apr-2014].

[22] Permissions Usage Notes. https://developer.android.com/training/permissions/usage-notes.html. [Accessed: 20-Sep-2017].

[23] Projects/OWASP Mobile Security Project - Top Ten Mobile Risks. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks. [Accessed: 02-Jul-2016].

[24] QARK: Tool to look for several security related Android application vulnerabilities. https://github.com/linkedin/qark. [Accessed: 27-Jun-2016].

[25] Requesting Permissions at Run Time. http://developer.android.com/training/permissions/requesting.html. [Accessed: 09-Nov-2016].

[26] SecuRank - Android Apps on Google Play. https://play.google.com/store/apps/details?id=me.securank.jov. [Accessed: 15-Apr-2016].

[27] Smartphone OS Market Share, 2016 Q3. http://www.idc.com/promo/smartphone-market-share/os. [Accessed: 07-Feb-2017].

[28] Smartphones: So Many Apps, So Much Time. http://www.nielsen.com/us/en/insights/news/2014/smartphones-so-many-apps--so-much-time.html. [Accessed: 15-Dec-2014].

[29] So Many Apps, So Much More Time for Entertainment. http://www.nielsen.com/us/en/insights/news/2015/so-many-apps-so-much-more-time-for-entertainment.html. [Accessed: 24-Aug-2017].

[30] System Permissions. http://developer.android.com/guide/topics/security/permissions.html. [Accessed: 07-Nov-2016].

[31] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, S&P '16, pages 433–451, May 2016. [Online]. Available: https://dx.doi.org/10.1109/SP.2016.33.

[32] A. Acquisti and J. Grossklags. Privacy and Rationality in Individual Decision Making. *IEEE Security and Privacy*, 3(1):26–33, Jan 2005. [Online]. Available: https://dx.doi.org/10.1109/MSP.2005.22.

[33] Alessandro Acquisti. Privacy in Electronic Commerce and the Economics of Immediate Gratification. In *Proceedings of the 5th ACM Conference on Electronic Commerce*, EC '04, pages 21–29, New York, NY, USA, 2004. ACM. [Online]. Available: https://dx.doi.org/10.1145/988772.988777.

[34] Alessandro Acquisti and Jens Grossklags. Losses, Gains, and Hyperbolic Discounting: An Experimental Approach to Information Security Attitudes and Behavior. In *Proceedings of the 2nd Annual Workshop on Economics and Information Security*, WEIS '03, pages 1–27, 2003.

[35] Hasan Faik Alan and Jasleen Kaur. Can Android Applications Be Identified Using Only TCP/IP Headers of Their Launch Time Traffic? In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '16, pages 61–66, New York, NY,

USA, 2016. ACM. [Online]. Available: https://dx.doi.org/10.1145/2939918.2939929.

[36] Benjamin Andow, Adwait Nadkarni, Blake Bassett, William Enck, and Tao Xie. A Study of Grayware on Google Play. In *Proceedings of the 5th Mobile Security Technologies Workshop*, MoST '16, pages 224–233, 2016. [Online]. Available: https://dx.doi.org/10.1109/SPW.2016.40.

[37] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM. [Online]. Available: https://dx.doi.org/10.1145/2594291.2594299.

[38] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. Short Paper: A Look at Smartphone Permission Models. In *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 63–68, New York, NY, USA, 2011. ACM. [Online]. Available: https://dx.doi.org/10.1145/2046614.2046626.

[39] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2382196.2382222.

[40] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and Its Security Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS

'16, pages 356–367, New York, NY, USA, 2016. ACM. [Online]. Available: https://dx.doi.org/10.1145/2976749.2978333.

[41] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM. [Online]. Available: https://dx.doi.org/10.1145/1866307.1866317.

[42] 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*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM. [Online]. Available: https://dx.doi.org/10.1145/2184489.2184500.

[43] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. O'Reilly Media, Inc., 2009. ISBN: 0596516495.

[44] Denis Bogdanas. DPerm: Assisting the Migration of Android Apps to Runtime Permissions. *CoRR*, abs/1706.05042, 2017. [Online]. Available: http://arxiv.org/abs/1706.05042.

[45] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal Analysis of Android Ad Library Permissions. *CoRR*, abs/1303.0857, 2013. [Online]. Available: http://arxiv.org/abs/1303.0857.

[46] Theodore Book and Dan S. Wallach. A Case of Collusion: A Study of the Interface Between Ad Libraries and Their Apps. In *Proceedings of the 3rd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '13, pages 79–86, New York, NY, USA, 2013. ACM. [Online]. Available: https://dx.doi.org/10.1145/2516760.2516762.

[47] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security*, ASIACCS '17, pages 71–85, New York, NY, USA, 2017. ACM. [Online]. Available: `https://dx.doi.org/10.1145/3052973.3053004`.

[48] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, October 2001. [Online]. Available: `https://dx.doi.org/10.1023/A:1010933404324`.

[49] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, NDSS '12, 2012.

[50] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 605–616. ACM, 2012. [Online]. Available: `https://dx.doi.org/10.1145/2382196.2382260`.

[51] Bogdan Carbunar and Rahul Potharaju. A Longitudinal Study of the Google App Market. In *Proceedings of the 7th IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '15, pages 242–249, New York, NY, USA, 2015. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2808797.2808823`.

[52] Patrick P.F. Chan, Lucas C.K. Hui, and S. M. Yiu. DroidChecker: Analyzing Android Applications for Capability Leak. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Net-*

*works*, WISEC '12, pages 125–136, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2185448.2185466.

[53] Hao Chen, Daojing He, Sencun Zhu, and Jingshun Yang. Toward Detecting Collusive Ranking Manipulation Attackers in Mobile App Markets. In *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security*, ASIACCS '17, pages 58–70, New York, NY, USA, 2017. ACM. [Online]. Available: https://dx.doi.org/10.1145/3052973.3053022.

[54] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, S&P '16, pages 357–376, May 2016. [Online]. Available: https://dx.doi.org/10.1109/SP.2016.29.

[55] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, S&P '10, pages 191–206, May 2010. [Online]. Available: https://dx.doi.org/10.1109/SP.2010.20.

[56] Xin Chen and Sencun Zhu. DroidJust: Automated Functionality-aware Privacy Leakage Analysis for Android Applications. In *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '15, pages 5:1–5:12, New York, NY, USA, 2015. ACM. [Online]. Available: https://dx.doi.org/10.1145/2766498.2766507.

[57] 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*,

WWW '12, pages 311–320, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2187836.2187879.

[58] Erika Chin, Adrienne Porter Felt, Vyas Sekar, and David Wagner. Measuring User Confidence in Smartphone Security and Privacy. In *Proceedings of the 8th Symposium on Usable Privacy and Security*, SOUPS '12, pages 1:1–1:16, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2335356.2335358.

[59] Catalin Cimpanu. HummingBad Android Malware Found in 20 Google Play Store Apps. https://www.bleepingcomputer.com/news/security/hummingbad-android-malware-found-in-20-google-play-store-apps/. [Accessed: 27-Feb-2017].

[60] M. Conti, L.V. Mancini, R. Spolaor, and N.V. Verde. Analyzing Android Encrypted Network Traffic to Identify User Actions. *IEEE Transactions on Information Forensics and Security*, 11(1):114–125, Jan 2016. [Online]. Available: https://dx.doi.org/10.1109/TIFS.2015.2478741.

[61] Mauro Conti, Luigi V. Mancini, Riccardo Spolaor, and Nino Vincenzo Verde. Can'T You Hear Me Knocking: Identification of User Actions on Android Apps via Traffic Analysis. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 297–304. ACM, 2015. [Online]. Available: https://dx.doi.org/10.1145/2699026.2699119.

[62] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-related Policy Enforcement for Android. In *Proceedings of the 13th International Conference on Information Security*, ISC '10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949317.1949355.

[63] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. NetworkProfiler: Towards automatic fingerprinting of Android apps. In *Proceedings of the 32nd IEEE International Conference on Computer Communications*, INFOCOM '13, pages 809–817, April 2013. [Online]. Available: https://dx.doi.org/10.1109/INFCOM.2013.6566868.

[64] George Danezis and Richard Clayton. *Digital Privacy: Theory, Technologies, and Practices*, chapter Introducing Traffic Analysis, pages 95–116. Auberbach Publications, 2007.

[65] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, ISC '10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949317.1949356.

[66] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 332–346. IEEE, 2012. [Online]. Available: https://dx.doi.org/10.1109/SP.2012.28.

[67] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM. [Online]. Available: https://dx.doi.org/10.1145/2508859.2516693.

[68] N. Eling, S. Rasthofer, M. Kolhagen, E. Bodden, and P. Buxmann. Investigating Users' Reaction to Fine-Grained Data Requests: A Market Experiment. In *Proceedings of the 49th Hawaii International Conference*

*on System Sciences*, HICSS '16, pages 3666–3675, Jan 2016. [Online]. Available: https://dx.doi.org/10.1109/HICSS.2016.458.

[69] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971.

[70] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Conference on Security*, SEC '11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028088.

[71] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM. [Online]. Available: https://dx.doi.org/10.1145/1653662.1653691.

[72] 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 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2382196.2382205.

[73] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A First Look at Traffic on Smartphones. In

*Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 281–287, New York, NY, USA, 2010. ACM. [Online]. Available: https://dx.doi.org/10.1145/1879141.1879176.

[74] 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 Asia Conference on Computer and Communications Security*, ASIACCS '16, pages 747–758, New York, NY, USA, 2016. ACM. [Online]. Available: https://dx.doi.org/10.1145/2897845.2897914.

[75] 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*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM. [Online]. Available: https://dx.doi.org/10.1145/2046707.2046779.

[76] 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 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 33–44, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2381934.2381943.

[77] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM. [Online]. Available: https://dx.doi.org/10.1145/2046614.2046618.

[78] 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 8th Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2335356.2335360.

[79] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Conference on Security*, SEC '11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028089.

[80] Denzil Ferreira, Vassilis Kostakos, Alastair R Beresford, Janne Lindqvist, and Anind K Dey. Securacy: An Empirical Investigation of Android Applications Network Usage, Privacy and Security. In *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '15, 2015. [Online]. Available: https://dx.doi.org/10.1145/2766498.2766506.

[81] Conner Forrest. 1.2 million infected: Android malware 'Hummer' could be biggest trojan ever. http://www.techrepublic.com/article/1-2-million-infected-android-malware-hummer-could-be-biggest-trojan-ever/. [Accessed: 30-Jun-2016].

[82] M. Frank, Ben Dong, A.P. Felt, and D. Song. Mining Permission Request Patterns from Android and Facebook Applications. In *Proceedings of the 12th International Conference on Data Mining*, ICDM '12, pages 870–875, Dec 2012. [Online]. Available: https://dx.doi.org/10.1109/ICDM.2012.86.

[83] Huiqing Fu and Janne Lindqvist. General Area or Approximate Location?: How People Understand Location Permissions. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, WPES '14, pages 117–120, New York, NY, USA, 2014. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2665943.2665957`.

[84] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 431–444, New York, NY, USA, 2013. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2462456.2464461`.

[85] 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*, ICSE '14, pages 1025–1035, New York, NY, USA, 2014. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2568225.2568276`.

[86] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, NDSS '12, 2012.

[87] 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*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2307636.2307663`.

[88] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '12, pages 101–112, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2185448.2185464.

[89] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter*, 11(1):10–18, November 2009. [Online]. Available: https://dx.doi.org/10.1145/1656274.1656278.

[90] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial NaïVe-bayes Classifier. In *Proceedings of the ACM Workshop on Cloud Computing Security*, CCSW '09, pages 31–42. ACM, 2009. [Online]. Available: https://dx.doi.org/10.1145/1655008.1655013.

[91] 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*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM. [Online]. Available: https://dx.doi.org/10.1145/2046707.2046780.

[92] Suman Jana, Úlfar Erlingsson, and Iulia Ion. Apples and Oranges: Detecting Least-Privilege Violators with Peer Group Analysis. *CoRR*, abs/1510.07308, 2015. [Online]. Available: http://arxiv.org/abs/1303.0857.

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

[94] 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 *Proceedings of the 16th Financial Cryptography and Data Security*, FC '12, pages 68–79, Berlin, Heidelberg, 2012. Springer-Verlag. [Online]. Available: `https://dx.doi.org/10.1007/978-3-642-34638-5_6`.

[95] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. AntMonitor: A System for Monitoring from Mobile Devices. In *Proceedings of the ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*, C2B(I)D '15, pages 15–20, New York, NY, USA, 2015. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2787394.2787396`.

[96] Mathias Lécuyer, Guillaume Ducoffe, Francis Lan, Andrei Papancea, Theofilos Petsios, Riley Spahn, Augustin Chaintreau, and Roxana Geambasu. XRay: Enhancing the Web's Transparency with Differential Correlation. In *Proceedings of the 23rd USENIX Security Symposium*, SEC '14, pages 49–64, Berkeley, CA, USA, 2014. USENIX Association. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2671225.2671229`.

[97] Jessica Lee. No.1 Position in Google Gets 33% of Search Traffic [Study]. `http://searchenginewatch.com/sew/study/2276184/no-1-position-in-google-gets-33-of-search-traffic-study`. [Accessed: 21-Oct-2015].

[98] Ilias Leontiadis, Christos Efstratiou, Marco Picone, and Cecilia Mascolo. Don'T Kill My Ads!: Balancing Privacy in an Ad-supported Mobile Application Market. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '12, pages 2:1–2:6, New York, NY,

USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2162081.2162084.

[99] 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*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press. [Online]. Available: https://dx.doi.org/10.1109/ICSE.2015.48.

[100] Marc Liberatore and Brian Neil Levine. Inferring the Source of Encrypted HTTP Connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 255–263. ACM, 2006. [Online]. Available: https://dx.doi.org/10.1145/1180405.1180437.

[101] Charles Zhechao Liu, Yoris A. Au, and Hoon Seok Choi. Effects of Freemium Strategy in the Mobile App Market: An Empirical Study of Google Play. *Journal of Management Information Systems*, 31(3):326–354, 2014. [Online]. Available: https://dx.doi.org/10.1080/07421222.2014.995564.

[102] Yang Liu and Andrew Simpson. Privacy-preserving Targeted Mobile Advertising: Requirements, Design and a Prototype Implementation. *Software: Practice and Experience*, 46(12):1657–1684, December 2016. [Online]. Available: https://dx.doi.org/10.1002/spe.2403.

[103] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2382196.2382223.

[104] Xiapu Luo, Peng Zhou, Edmond WW Chan, Wenke Lee, Rocky KC Chang, and Roberto Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium*, NDSS '11, 2011.

[105] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '13, pages 224–234, New York, NY, USA, 2013. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2491411.2491450`.

[106] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Conference on Python in Science*, SciPy '10, pages 51 – 56, 2010.

[107] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, 2003. ISBN: 0262134322.

[108] Donald Metzler, Susan Dumais, and Christopher Meek. Similarity Measures for Short Segments of Text. In *Proceedings of the 29th European Conference on IR Research*, ECIR '07, pages 16–27, Berlin, Heidelberg, 2007. Springer-Verlag. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1763653.1763660`.

[109] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and Knowledge-based Measures of Text Semantic Similarity. In *Proceedings of the 21st National Conference on Artificial Intelligence*, AAAI '06, pages 775–780. AAAI Press, 2006. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1597538.1597662`.

[110] Brad Miller, Ling Huang, A. D. Joseph, and J. D. Tygar. I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis. In

Emiliano De Cristofaro and Steven J. Murdoch, editors, *Proceedings of the 14th International Symposium on Privacy Enhancing Technologies*, PETS '14, pages 143–163, Cham, 2014. Springer International Publishing. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-08506-7_8.

[111] S. Mongkolluksamee, V. Visoottiviseth, and K. Fukuda. Enhancing the Performance of Mobile Traffic Identification with Communication Patterns. In *Proceedings of the 39th Annual Computer Software and Applications Conference*, volume 2 of *ACSAC '15*, pages 336–345, 2015. [Online]. Available: https://dx.doi.org/10.1109/COMPSAC.2015.50.

[112] Sophon Mongkolluksamee, Vasaka Visoottiviseth, and Kensuke Fukuda. Combining Communication Patterns &Traffic Patterns to Enhance Mobile Traffic Identification Performance. *Journal of Information Processing*, 24(2):247–254, 2016. [Online]. Available: https://dx.doi.org/10.2197/ipsjjip.24.247.

[113] J. Muehlstein, Y. Zion, M. Bahumi, I. Kirshenboim, R. Dubin, A. Dvir, and O. Pele. Analyzing HTTPS Encrypted Traffic to Identify User Operating System, Browser and Application. In *Proceedings of the 14th IEEE Annual Consumer Communications Networking Conference*, CCNC '17, pages 1–6, Jan 2017. [Online]. Available: https://dx.doi.org/10.1109/CCNC.2017.8013420.

[114] T.T.T. Nguyen and G. Armitage. A Survey of Techniques for Internet Traffic Classification Using Machine Learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, October 2008. [Online]. Available: https://dx.doi.org/10.1109/SURV.2008.080406.

[115] Patricia A. Norberg, Daniel R. Horne, and David A. Horne. The Privacy Paradox: Personal Information Disclosure Intentions versus Behaviors.

*Journal of Consumer Affairs*, 41(1):100–126, 2007. [Online]. Available: https://dx.doi.org/10.1111/j.1745-6606.2006.00070.x.

[116] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, WPES '11, pages 103–114. ACM, 2011. [Online]. Available: https://dx.doi.org/10.1145/2046556.2046570.

[117] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Security Symposium*, SEC '13, pages 527–542, Berkeley, CA, USA, 2013. USENIX Association. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534812.

[118] Dimitrios Papamartzivanos, Dimitrios Damopoulos, and Georgios Kambourakis. A Cloud-based Architecture to Crowdsource Mobile App Privacy Leaks. In *Proceedings of the 18th Panhellenic Conference on Informatics*, PCI '14, pages 59:1–59:6, New York, NY, USA, 2014. ACM. [Online]. Available: https://dx.doi.org/10.1145/2645791.2645799.

[119] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Asia Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 71–72, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2414456.2414498.

[120] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duch-

esnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, November 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078195.

[121] Martin Pelikan, Giles Hogben, and Ulfar Erlingsson. Identifying Intrusive Mobile Apps using Peer Group Analysis . https://android-developers.googleblog.com/2017/07/identifying-intrusive-mobile-apps-using.html. [Accessed: 13-Jul-2017].

[122] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 241–252, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2382196.2382224.

[123] Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos, and Thomas Karagiannis. Rise of the Planet of the Apps: A Systematic Study of the Mobile App Ecosystem. In *Proceedings of the 13th Internet Measurement Conference*, IMC '13, pages 277–290, New York, NY, USA, 2013. ACM. [Online]. Available: https://dx.doi.org/10.1145/2504730.2504749.

[124] Andrey Polkovnichenko and Oren Koriat. Viking Horde: A New Type of Android Malware on Google Play. http://blog.checkpoint.com/2016/05/09/viking-horde-a-new-type-of-android-malware-on-google-play/. [Accessed: 16-Jun-2016].

[125] Dan Price. WhatsApp Encryption: It's Now the Most Secure Instant Messenger (Or is it?). http://www.makeuseof.com/tag/whatsapp-encryption-now-secure-instant-messenger/. [Accessed: 22-Apr-2015].

[126] Zafar Ayyub Qazi, Jeongkeun Lee, Tao Jin, Gowtham Bellala, Manfred Arndt, and Guevara Noubir. Application-awareness in SDN. In *Proceedings of ACM SIGCOMM*, SIGCOMM '13, pages 487–488, New York, NY, USA, 2013. ACM. [Online]. Available: https://dx.doi.org/10.1145/2486001.2491700.

[127] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS '14, pages 1354–1365, New York, NY, USA, 2014. ACM. [Online]. Available: https://dx.doi.org/10.1145/2660267.2660287.

[128] Jean-François Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In *Proceedings of the 6th International Workshop on Design Issues in Anonymity and Unobservability*. Springer, 2001. [Online]. Available: https://dx.doi.org/10.1007/3-540-44702-4_2.

[129] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. [Online]. Available: https://dx.doi.org/10.1109/PROC.1975.9939.

[130] Borja Sanz, Igor Santos, Xabier Ugarte-Pedrero, Carlos Laorden, Javier Nieves, and Pablo García Bringas. Anomaly Detection using String Analysis for Android Malware Detection. In *Proceedings of the International Joint Conference SOCO'13-CISIS'13-ICEUTE'13*, SOCO/CISIS/ICEUTE '13, pages 469–478, 2014. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-01854-6_48.

[131] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android Permissions: A Perspective Combining Risks and Benefits. In *Proceedings of the 17th ACM*

*Symposium on Access Control Models and Technologies*, SACMAT '12, pages 13–22, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2295136.2295141.

[132] Mohamed Nassim Seghir and David Aspinall. EviCheck: Digital Evidence for Android. In *Proceedings of the 13th International Symposium on Automated Technology for Verification and Analysis*, ATVA '15, pages 221–227. Springer, 2015. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-24953-7_17.

[133] Suranga Seneviratne, Harini Kolamunna, and Aruna Seneviratne. A Measurement Study of Tracking in Paid Mobile Applications. In *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. [Online]. Available: https://dx.doi.org/10.1145/2766498.2766523.

[134] Suranga Seneviratne, Aruna Seneviratne, Prasant Mohapatra, and Anirban Mahanti. Predicting User Traits from a Snapshot of Apps Installed on a Smartphone. *ACM SIGMOBILE Mobile Computing and Communications Review*, 18(2):1–8, June 2014. [Online]. Available: https://dx.doi.org/10.1145/2636242.2636244.

[135] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, NDSS '16, 2016.

[136] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX Security Symposium*, SEC '12, pages 553–567. USENIX, 2012.

[137] Aaron Smith. Record shares of Americans now own smartphones, have home broadband. http://www.pewresearch.org/fact-tank/2017/01/12/evolution-of-technology/. [Accessed: 09-Feb-2017].

[138] Chad Spensky, Jeffrey Stewart, Arkady Yerukhimovich, Richard Shay, Ari Trachtenberg, Rick Housley, and Robert K Cunningham. SoK: Privacy on Mobile Devices - It's Complicated. *Proceedings on Privacy Enhancing Technologies*, 2016(3):96–116, 2016. [Online]. Available: https://dx.doi.org/10.1515/popets-2016-0018.

[139] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating User Privacy in Android Ad Libraries. In *Proceedings of the 1st Workshop on Mobile Security Technologies*, MoST '12, page 10, 2012.

[140] Tim Stöber, Mario Frank, Jens Schmitt, and Ivan Martinovic. Who Do You Sync You Are?: Smartphone Fingerprinting via Application Behaviour. In *Proceedings of the 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 7–12. ACM, 2013. [Online]. Available: https://dx.doi.org/10.1145/2462096.2462099.

[141] Vincent F. Taylor, Alastair R. Beresford, and Ivan Martinovic. There Are Many Apps for That: Quantifying the Availability of Privacy-preserving Apps. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '17, pages 247–252, New York, NY, USA, 2017. ACM. [Online]. Available: https://dx.doi.org/10.1145/3098243.3098266.

[142] Vincent F. Taylor and Ivan Martinovic. DEMO: Starving Permission-Hungry Android Apps Using SecuRank. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS '16, pages 1850–1852, New York, NY, USA, 2016. ACM. [Online]. Available: https://dx.doi.org/10.1145/2976749.2989032.

[143] Vincent F. Taylor and Ivan Martinovic. SecuRank: Starving Permission-Hungry Apps Using Contextual Permission Analysis. In *Proceedings of the 6th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '16, pages 43–52, New York, NY, USA, 2016. ACM. [Online]. Available: https://dx.doi.org/10.1145/2994459.2994474.

[144] Vincent F. Taylor and Ivan Martinovic. *Intrusion Detection and Prevention for Mobile Ecosystems*, chapter Attacking Smartphone Security and Privacy, pages 25–64. Taylor & Francis, 2017. ISBN: 978-1-138-03357-3.

[145] Vincent F. Taylor and Ivan Martinovic. Short Paper: A Longitudinal Study of Financial Apps in the Google Play Store. In *Proceedings of the 21st Financial Cryptography and Data Security*, FC '17, 2017. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-70972-7.

[146] Vincent F. Taylor and Ivan Martinovic. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security*, ASIACCS '17, pages 45–57, New York, NY, USA, 2017. ACM. [Online]. Available: https://dx.doi.org/10.1145/3052973.3052990.

[147] Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, EuroS&P '16, pages 439–454, March 2016. [Online]. Available: https://dx.doi.org/10.1109/EuroSP.2016.40.

[148] Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Robust Smartphone App Identification Via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security*, 13(1):63–78, January 2018. [Online]. Available: https://dx.doi.org/10.1109/TIFS.2017.2737970.

[149] Alok Tongaonkar, Shuaifu Dai, Antonio Nucci, and Dawn Song. Understanding Mobile App Usage Patterns Using In-App Advertisements. In *Proceedings of the 14th International Conference on Passive and Active Measurement*, PAM '13, pages 63–72, 2013. [Online]. Available: https://dx.doi.org/10.1007/978-3-642-36516-4_7.

[150] Vasilis Tsiakos and Constantinos Patsakis. AndroPatchApp: Taming Rogue Ads in Android. In Selma Boumerdassi, Éric Renault, and Samia Bouzefrane, editors, *Proceedings of the 2nd International Conference on Mobile, Secure, and Programmable Networking*, MSPN '16, pages 183–196, Cham, 2016. Springer International Publishing. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-50463-6_15.

[151] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing Android Permission Creep. In *Proceedings of 5th Workshop on Web 2.0 Security and Privacy*, volume 2 of *W2SP '11*, 2011.

[152] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT '11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028052.2028062.

[153] Nicolas Viennot, Edward Garcia, and Jason Nieh. A Measurement Study of Google Play. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 221–233, New York, NY, USA, 2014. ACM. [Online]. Available: https://dx.doi.org/10.1145/2591971.2592003.

[154] Daniel T. Wagner, Andrew Rice, and Alastair R. Beresford. Device Analyzer: Large-scale Mobile Data Collection. *ACM SIGMETRICS Perfor-*

*mance Evaluation Review*, 41(4):53–56, April 2014. [Online]. Available: https://dx.doi.org/10.1145/2627534.2627553.

[155] Daniel T. Wagner, Andrew Rice, and Alastair R. Beresford. Device Analyzer: Understanding Smartphone Usage. In *Proceedings of the 10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, MOBIQUITOUS '13, pages 195–208. Springer, 2014. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-11569-6_16.

[156] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I. Hong. Understanding the Purpose of Permission Use in Mobile Apps. *ACM Transactions on Information Systems*, 35(4):43:1–43:40, July 2017. [Online]. Available: https://dx.doi.org/10.1145/3086677.

[157] Qinglong Wang, Amir Yahyavi, Mettina Kemme, and Wenbo He. I Know What You Did On Your Smartphone: Inferring App Usage Over Encrypted Data Traffic. In *Proceedings of the 3rd IEEE Conference on Communications and Network Security*, CNS '15, 2015. [Online]. Available: https://dx.doi.org/10.1109/CNS.2015.7346855.

[158] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the 22nd USENIX Security Symposium*, SEC '13, pages 559–572, Berkeley, CA, USA, 2013. USENIX Association. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534814.

[159] Takuya Watanabe, Mitsuaki Akiyama, Tetsuya Sakai, and Tatsuya Mori. Understanding the Inconsistencies between Text Descriptions and the Use of Privacy-sensitive Resources of Mobile Apps. In *Proceedings of the 11th Symposium On Usable Privacy and Security*, SOUPS '15, pages 241–255, Ottawa, July 2015. USENIX Association.

[160] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM. [Online]. Available: `https://dx.doi.org/10.1145/2660267.2660357`.

[161] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 31–40. ACM, 2012. [Online]. Available: `https://dx.doi.org/10.1145/2420950.2420956`.

[162] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. ProfileDroid: Multi-layer Profiling of Android Applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, MobiCom '12, pages 137–148, 2012. [Online]. Available: `https://dx.doi.org/10.1145/2348543.2348563`.

[163] Zack Whittaker. AccuWeather caught sending user location data, even when location sharing is off. `http://www.zdnet.com/article/accuweather-caught-sending-geo-location-data-even-when-denied-access/`. [Accessed: 24-Aug-2017].

[164] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proceedings of the 24th USENIX Security Symposium*, SEC '15, pages 499–514, Washington, D.C., August 2015. USENIX Association.

[165] Zhen Xie and Sencun Zhu. AppWatcher: Unveiling the Underground Market of Trading Mobile App Reviews. In *Proceedings of the 8th ACM Con-*

*ference on Security and Privacy in Wireless and Mobile Networks*, WiSec '15, pages 10:1–10:11, New York, NY, USA, 2015. ACM. [Online]. Available: https://dx.doi.org/10.1145/2766498.2766510.

[166] Medien Zeghid, Mohsen Machhout, Lazhar Khriji, Adel Baganne, and Rached Tourki. A Modified AES Based Algorithm for Image Encryption. *International Journal of Computer Science and Engineering*, 1(1):70–75, 2007.

[167] Xiao Zhang, Amit Ahlawat, and Wenliang Du. AFrame: Isolating Advertisements from Mobile Applications in Android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 9–18, New York, NY, USA, 2013. ACM. [Online]. Available: https://dx.doi.org/10.1145/2523649.2523652.

[168] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 611–622. ACM, 2013. [Online]. Available: https://dx.doi.org/10.1145/2508859.2516689.

[169] Yury Zhauniarovich and Olga Gadyatskaya. Small Changes, Big Changes: An Updated View on the Android Permission System. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID '16, pages 346–367, Cham, 2016. Springer International Publishing. [Online]. Available: https://dx.doi.org/10.1007/978-3-319-45719-2_16.

[170] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Pro-*

*ceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 185–196, New York, NY, USA, 2013. ACM. [Online]. Available: https://dx.doi.org/10.1145/2435349.2435377.

[171] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM. [Online]. Available: https://dx.doi.org/10.1145/2133601.2133640.

[172] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 1017–1028, New York, NY, USA, 2013. ACM. [Online]. Available: https://dx.doi.org/10.1145/2508859.2516661.

[173] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 95–109, May 2012. [Online]. Available: https://dx.doi.org/10.1109/SP.2012.16.

[174] 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 *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS '12, 2012.

[175] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustwor-*

*thy Computing*, TRUST '11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag. [Online]. Available: http://dl.acm.org/citation.cfm?id=2022245.2022255.