© 2014 Dongjing He

# SECURITY THREATS TO ANDROID APPS

BY

DONGJING HE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Advisers:

Professor Carl A. Gunter
Professor Klara Nahrstedt

# ABSTRACT

Smartphones have become ubiquitous and smartphone users are increasingly relying on the mobile applications (*app* for short) to store and handle private information. The fluidity of mobile apps and mobile app markets has complicated mobile app security. Many new threats emerged are either because of the deficiency of mobile app development or the design ambiguities of the Android operating system.

In order to seek a better understanding of mobile app security, we present a systematic study on security threats to Android apps in two dimensions. *First*, we study Android apps from mobile health (*mHealth* for short) sector, in order to understand the prevalence of mobile app threats to that sector. In particular, we present a three-stage study of the mHealth apps to show that mHealth apps make widespread use of unsecured Internet communications and third party servers. Assuming that mobile apps are well protected by their developers, we ask a *second* question: are there any limitations in fundamental Android security design that can be used by malicious parties to disclose users' sensitive information? We study a newly discovered threat, side-channel information leaks on Android devices, in detail. Particularly, we discover an unexpected channel of information leaks from per-app data-usage statistics and demonstrate that a malicious app can infer users' identity or investment information with zero-permission by monitoring the channel. To mitigate these threats, we propose defense strategies for both widespread threats on mHealth apps and the side-channel information leaks on Android devices.

*To my parents, for their love and support.*

# ACKNOWLEDGMENTS

First and foremost, I would like to express my special appreciation to my advisers Professor Carl A. Gunter and Professor Klara Nahrstedt, who have been offering tremendous helps to me. The extreme kindness, valuable opportunities and learning experiences they have given me are deeply appreciated. Only under their guidance, I had the opportunity to be introduced to the current challenges in mHealth security and privacy which further motivated me to work in this field.

I am also grateful to Soteris Demtrious, Muhammud Naveed from the University of Illinois at Urbana-Champaign, Professor Xiaofeng Wang, Xiaoyong Zhou and Rui Wang from the Indiana University Bloomington for being wonderful collaborators. I cherish the opportunity to learn from them and gain deeper insights into Android security during our weekly discussions and project meetings.

I deeply appreciate the friendship and the support from the two research groups I have worked with. I enjoy all the conversations I had with my labmates in the Multimedia Operating System and Networking Group, including Debish Fesehaye, Haiming Jin, Zhenhuan Gao, Hongyang Li, Shannon Chen, Ting-yu Wang, Wenyu Ren, Rauol Rivas, Ahsan Arefin, Rehana Tabassum, Phuong V. Nguyen, Zixia Huang, Anjali Sridhar, as well as labmates in the Illinois Security Lab, including Soteris Demetrious, Muhammud Naveed, Xun(Sean) Lu, Ting Wu, Se Eun Oh, Gaurav Lahoti, Siddharth Gupta, Vincent Bindschaedler, Ravinder Shankesi, Ji Young Chun, Lingyu Xu and Aston Zhang, and the deputy director from SHARPS, Antonios Michalos.

Above all, I would also like to thank my parents, who have always been supportive and caring. I appreciate Liang Xia, for his encouragements, understandings, patience, and support during my research. Without their firm love and support, especially at those difficult moments, I would not have successfully made so far.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

With the growing popularity of Android operating system, an enormous number of Android apps has been developed in recent years. As smartphones are becoming a ubiquitous source of private and sensitive personal information, the quality of the booming apps can be a concern. Poorly programmed Android apps may contain information leakage that undermines mobile users' privacy and security. In this chapter, section 1.1 gives an overview of the phenomenal trend of Android mobile system. Along with this trend, security concerns are also expressed and their relevant research studies regarding Android app security are listed in section 1.1. Section 1.2 refines the initial motivation and focuses our research questions on specific components in Android app security. Section 1.3 presents the approach to address the problem in two different directions as elaborated in sections 1.3.1 and 1.3.2. The main contributions of this master thesis are summarized in section 1.4. Section 1.5 gives the organization of this master thesis.

## 1.1 Overview

Today's smartphones have as much computing resources and power as a high end computer only a few years ago and smartphones are becoming more and more popular these years. Current mobile operating system, such as Android, has gained tremendous popularity these years, with 81.4% market share in the mobile operating system market share [2] and more than one billion device activations [3]. Android allows users to run various apps developed by many independent developers downloaded from app markets, and more than one million apps are available on official Android app market, Google Play [4]. Unlike most of the PC apps, Android apps collect, process and store personal sensitive information, which may undermine users' security and privacy.

Indeed, in recent years, we have seen waves of Android-based malware [5, 6] that exploit different vulnerabilities within this new computing platform, highlighting the security and privacy threats it is facing. Furthermore, Android's security protection has been under scrutiny for years. A few weaknesses of the system, such as permission re-delegation [7] and capability leaks [8], are revealed by researchers. Particularly, a recent blog [9] from Leviathan Security Group describes the malicious activities that could be performed by an app without any permissions, including reading from SD card, accessing a list of installed apps, getting some device and system information (GSM and SIM vendor ID, the Android ID and kernel version) and pinging through a Linux Shell. Most of the problems reported here are either implementation weaknesses or design ambiguities that lead developers to inadvertent exposures. Examples include SD card read and unauthorized ping, which have all been fixed. The rest turns out to be less of a concern, as they are almost impossible to exploit in practice (GSM/SIM/Android ID, which leads to nothing more than mobile country code and mobile network code).

## 1.2 Problem Statement

Compared with the phenomenal progress in Android operating system and the apps on its platform, the ways Android devices are being used today are even more stunning. Increasingly, Android smartphones become primary devices not only for traditional phone calling, but for email checking, messaging, mapping/navigation, entertainment, social networking and even for activities as important as healthcare and investment management. The vast array of Android apps enables mobile users to store sensitive private information on their smartphones. As a large amount of new Android apps are constantly submitted to app markets and become instantly available to users, we believe Android device has become an increasingly attractive target for security attacks, which calls for several specific questions:

1. What are the most widespread and serious security concerns existing among Android apps?
2. Are the original security designs of smartphone systems, like Android, ready for those highly diverse, fast-evolving apps?

3. Given the findings of limitations in Android security design, what kind of vulnerabilities can be explored by malicious parties if they want to pilfer users' secrets?

4. How is it possible to protect mobile phone users against security threats such as information leakage on the Android apps?

## 1.3 Our Approach

To answer the first question, we will choose a specific sector of Android apps, for example, mHealth, which usually contains highly sensitive information. By studying the Android apps in mHealth sector, we try to answer what the most widespread security concerns are and the approach is illustrated in section 1.3.1. After understanding what the widespread threats are, we are further asking if any loopholes exist in fundamental security designs of Android. If so, we want to further explore the possibility to pilfer users' sensitive information with the finding of the security loopholes. We illustrate the approach to address the second and third questions in section 1.3.2. A fourth question naturally comes into our mind after exploring enough security threats on Android apps: how can we mitigate these threats? Two different mitigation techniques are to be proposed for the purpose of the above two distinct directions: one is for general security threats on Android, and another one is for a specific threat based on the finding of security loopholes on Android.

### 1.3.1 Study on Security Threats in mHealth

Advanced techniques based on mobile computing and communications technology significantly improve health care quality and efficiency. Mobile devices greatly enable individuals and their hospital staffs to better monitor their health and deliver health care services. However, implementing these mobile health (mHealth) technologies brings up certain privacy concerns about users' sensitive health care data. The proliferation of mHealth applications results in greater chance that applications are not being carefully examined in terms of privacy protection.

In our study, we choose mHealth sector as our study target, since new security and privacy risks particular to mobile computing and communications technology abound in mHealth apps [10, 11] and the data from mHealth apps is particularly private and sensitive to mobile users. Furthermore, the aspects of mHealth make it different from other health information systems: First, mHealth apps allow a much larger amount of data being collected from the patient, as mobile devices can collect data over extended periods of time. Second, a much broader range of health-related data is being collected, as many mHealth apps collect patient activities and lifestyle, not only physiological data, but also include physical activity, location tracking, eating habits and diet details, social interactions and so on. Third, the nature of communications technology and mobile computing exposes many new attack surfaces to the outside world.

To answer what the most widespread and serious security concerns are among Android mHealth apps, we conduct our study with a narrow-down methodology: firstly, we prepare a list of attack surfaces by researching literature and observing a large dataset of randomly sampled apps; second, from the above list, we study each one attack surface and find whether indeed the threat exists extensively in a small dataset; lastly, after locating the threats that are widespread in Android mHealth apps, we further explore how serious the threats can be by surveying a small dataset of randomly sampled apps. The study on security concerns of Android mHealth apps is elaborated in chapter 3.

### 1.3.2 Study Side Channel Threats against Android Apps

Actually, the design of Android takes security seriously. It is built upon a sandbox and permission model, in which each app is isolated from others by Linux user-based protection and required to explicitly ask for permissions to access the resources outside its sandbox before it can be installed on a phone. Compared with what security models are provided by conventional desktop-oriented OSes, even Linux on which Android is built, this security design looks pretty solid.

In our research, we are asking a different question: *assuming that Android's security design has been faithfully implemented and apps are well protected by* *their developers, what can a malicious app still learn about the user's private information without any permission at all?* For such a zero-permission app, all it can access are a set of seemingly harmless resources shared across users (i.e., apps), which are made publicly available by Android and its underlying Linux, for the purpose of facilitating coordination among apps and simplifying access control. However, the rapidly-evolving designs and functionalities of emerging apps, particularly their rich and diverse background information (e.g. social network, public online services, etc.), begin to invalidate such design assumptions, turning thought-to-be innocuous data into serious information leaks.

In chapter 4, we explore the possibility of Android turning public resources into sources of side-channel information leaks. For example, network data usage of an app is deliberately published by Android through its Linux public directory, aiming to help a smartphone user keep track of the app's mobile data consumption. In our research, we try to prove that this piece of apparently harmless and also useful data can actually be used to fingerprint a user's certain online activity, such as tweeting. This knowledge, combined with the background information that comes with the Twitter app (i.e., public tweets), can be used to infer the user's true identity, using an inference technique we developed as described in section 4.2 and 4.3. As another example, a user's investment interests can be inferred by fingerprinting the network usage statistics as described in section 4.4.

## 1.4 Main Contributions

Some of the major contributions are as follows:

- *Understanding major security issues in Android mHealth apps.* Through our multi-stage study, it shows some serious problems with the Android mHealth apps. The major issue is unencrypted communication over the Internet and the use of third party hosting and storage services. Our study shows that a significant number of mHealth apps from Google Play market have these issues. Many Android app developers are not security experts and using encrypted communication is more expensive than using unencrypted communication. Third party cloud and hosting services provide a very economical solution but have serious implications to host the app's services and store users' sensitive data. We believe that these issues need immediate attentions from both vendors and users.

- *Understanding of information leaks from Android public resources.* Different from prior research that mainly focuses on implementation flaws within Android, our study contributes to a better understanding of an understudied yet fundamental weakness in Android design: the information leaks from the resources that are not originally considered to be confidential and therefore made available for improving system usability. This has been achieved through a suite of new inference techniques designed to demonstrate that highly-sensitive user data can be recovered from the public resources, in the presence of rich background information provided by popular apps and online resources.

- *First step toward mitigating information leaks from Android public resources.* We have designed a new mitigation approach, aiming to preserve the utility of such public data to the legitimate parties while controlling the way that an adversary can use it to derive user secrets.

- *Proposing strategies for security concerns of mHealth apps.* To automatically detect data leaks in Android mHealth apps, we have proposed a static analysis framework for mHealth apps to detect any possible links between private data and leakage sinks. We also present general compliance recommendations for mHealth app vendors to minimize security risks to mHealth users.

## 1.5 Outline

Chapter 2 gives thorough background information on the mHealth technology, Android operating system and its security mechanisms. Section 2.1 explains various important concepts in the Android operating system such as four components, inter-component communication, and permissions. Section 2.2 explains Android security mechanisms.

Chapter 3 aims to understand the security threats to Android apps in a specific sector, mHealth. Section 3.1 introduces the related work. Section

3.2 explains the methods to build datasets for the three stages of the study respectively. Sections 3.3, 3.4 and 3.5 highlight the results for the three-stage study. Section 3.3 provides a classification of Android mHealth apps and forms a list of attack surfaces that need attention. Section 3.4 investigates the list of attack surfaces and forms a subset of the attack surfaces are the most widespread. Then section 3.5 study the problems that are the most serious in section 3.4 and concludes that Android mHealth apps make widespread use of unsecured Internet and third party servers.

Chapter 4 introduces a newly-discovered side channel attack on Android, which uses per-app data-usage statistics to cause information leaks. Section 4.1 illustrates the related work. Section 4.2 describes the techniques we use to infer private information from public resources on Android with zero permission. Section 4.3 and section 4.4 highlight two attack instances using the techniques described in section 4.2. Section 4.3 demonstrates an instance of inferring a mobile user's identity with combined usage of Twitter app data-usage statistics and publicly accessible Twitter API. Section 4.4 introduces an example of knowing a mobile user's investment interests by monitoring the data-usage statistics of Yahoo! Finance.

Chapter 5 discusses mitigation strategies for the security risks introduced in chapter 3 and 4. Section 5.1 introduces strategies and enforcement framework on Android to mitigate the side channel inference from public resources on Android. The methods introduced are round up or round down and aggregation of data usage statistics on Android. Section 5.2 proposes a static analysis framework to mitigate general information leaks from Android mHealth apps and provides compliance recommendations for the threats that are the most widespread and serious as concluded in chapter 3.

Chapter 6 discusses the summary of findings of this master thesis in section 6.1 and the limitations for our study in section 6.2.

Chapter 7 concludes the entire master thesis with the highlights of our work and potential future scope of this work.

# CHAPTER 2

# BACKGROUND

This chapter gives background information of the Android operating system and its security mechanisms. Section 2.1 explains the fundamentals of the Android operating system. Section 2.2 introduces Android security mechanisms.

## 2.1 Android Operating System

Android is an operating system based on Linux for mobile devices. It provides a rich application framework to allow developers to build apps written in Java. The Android operating system is a multi-user system, in which each app has a unique user ID (UID). All files in an app will be assigned to that apps UID and normally not accessible to other apps. Every app runs in its own Linux process on a separate Dalvik Virtual Machine isolated from other apps, so that apps must explicitly share data and resources. In this way, Android implements *the principle of least privilege*. That is, no application, by default, will have the permission to perform operations to adversely impact the system, other applications, or the user.

App components are the essential building blocks of an Android app. There are four different types of components, of which each serves a distinct purpose.

- An *Activity* component is a single screen with user interface, which can interact with the user via touchscreen or physical keyboard. An app commonly contains multiple Activities, one for each screen presented to the user. The Activities work together to form a cohesive user experience, and each one is independent of the others. A different app can start any one of these Activities (if it is allowed), possibly expecting a return value. Only one Activity at a time on the screen has input and processing focus.

- A *Service* component provides background long-running operations or performs work for remote processes that continues even after its application loses focus. A Service does not provide a user interface. Another component, such as an Activity, can start a Service or interact with it. In order to interact with other component, Services define arbitrary interfaces for Remote Procedure Call (RPC), including method execution and callbacks, which can only be called if the Service has been bound.

- A *Content Provider* component manages a shared set of app data. The app data could be either private to the app itself or shared with other apps, if they have the proper permissions. You can store the data in many different places: in the file system, a SQLite database, on the web, or any other persistent storage location. The Content Provider supports the basic CRUD (create, retrieve, update, and delete) functions, through which components in other apps can retrieve or modify the data according to the Content Providers schema. The data in a Content Provider is addressed via a content URI.

- A *Broadcast Receiver* component is asynchronous event mailbox that responds to system-wide broadcast announcements represented by action strings. Android defines many standard action string corresponding to system events  for example, the screen has turned off, the battery is low, or the system has booted. Developers can define their own action strings  for example, to let other apps know that some data has been downloaded is available for them to use.

Android uses *Intent* for inter-component communication in many ways: to start an Activity, to start a Service, or to deliver a Broadcast message. There are two types of Intents: *explicit intent* and *implicit intent*. Explicit intents specify the component to start by name, whereas implicit intents do not name a specific component, but instead declare a general operation to perform. An *intent filter* is an expression of action strings, in order to specify what type of intents the component would like to receive. Android provides a *permission* mechanism to enforce restrictions of inter-component communication and access to system resources. Per-URI permissions can be granted to specify ad hoc accesses to specific pieces of data. Permissions are requested per application at its installation time. Once installed, this security policy cannot change.

Each application includes a manifest file. The manifest file describes the component of the app, including their types and intent filters. Note that Broadcast Receiver can be dynamically created in runtime, in addition to being statically defined in the manifest file. The manifest file declares which permissions the app must have in order to access protected parts of the API and interact with other apps.

## 2.2 Android Security

### 2.2.1 Android Security Mechanisms

Android is designed and built based on the Linux and it seeks to provide additional security controls over: user data protection, system resources protection (including the network) and application isolation. To provide these additional security controls, Android supports these key extra security features:

**Security at the OS level through the Linux kernel.** As described in section 2.1, Android is based on Linux and the Linux kernel provides several key security features to Android, including a user-based permission model, process isolation and secure IPC and so on.

**Secure inter-process communication.** Processes are able to communicate using any of the standard UNIX-style mechanisms. Android also provides new inter-process communication (*IPC* for short) mechanisms, including *Binder, sssServices, Intents* and *Content Providers*. *Binder* is a lightweight remote procedure call mechanism designed for high-performance in-process and cross-process calls. *Services* can provide interfaces directly accessible using *Binder*. The concepts of *Intents* and *Content Providers* have been discussed in section 2.1.

**Android permission model.** By default, an Android app can only access a limited range of system resources. The access to sensitive resources is protected via a security mechanisms known as Permissions. It provides protected APIs for the sensitive resources, including *camera, location, Bluetooth, telephony, SMS/MMS* and *network*. To make use of the protected APIs, an app must declare APIs' associated permissions in a manifest file and the permissions are agreed upon at installation time by users. Once granted, the permissions are applied to the app as long as it's installed.

**Application sandbox.** All applications on Android run in an application sandbox. As described in section 2.1, the Android assigns UID to each Android app and runs it as an individual user in a separate process. The kernel enforces security between applications through standard Linux facilities, such as UID-based permissions and process isolation. The application sandbox is implemented in the kernel, this security model extends to native code and operating system apps. Since all Android applications are sandboxed at the OS level, memory corruption is not an issue in Android.

**Application signing.** Application signing allows developers to identify the author of the application and enables updating applications without using complicated interfaces and permissions. Application signing is the first step to ensure the application sandbox mechanism. It signs certificates to ensure which UID is associated with which app and different apps run under different UIDs. When an app is installed on an Android device, the system verifies that the app has been properly certified.

# CHAPTER 3

# PREVALENCE OF SECURITY CONCERNS IN ANDROID MHEALTH APPS

In this chapter, we aim to provide a deeper understanding of the security threats to Android apps by studying apps from a specific sector, mHealth. Section 3.1 illustrates the related work of mHealth technology and its security and privacy issues. Section 3.2 explains the main methods to conduct a systematic study on the security threats to Android mHealth apps. Specifically, it elaborates the methods of how we collect apps as our dataset for the three-stage study. Sections 3.3, 3.4 and 3.5 elaborate the findings for the three-stage study. Section 3.3 formulates a list of attack surfaces that need attention in mHealth apps. Section 3.5 focuses on the attack surfaces that are proven by section 3.4 for having significant issues in most of the apps. By analyzing a new randomly selected dataset, section 3.5 shows that mHealth apps make widespread use of unsecured Internet communications and third party servers, suggesting that increased use of mHealth apps could lead to less secure treatment of health data unless mHealth vendors make improvements in the way they communication and store data.

## 3.1 Related Work

### 3.1.1 mHealth Technology

Advanced techniques based on mobile computing and communications technology significantly improve health care quality, improve efficiency and reduce costs. Mobile health (mHealth) apps and medical devices greatly enable individuals to better monitor and manage their health conditions. Mobile apps for clinicians improve the way in which hospital professions interact with their patients and deliver health care services.

The mHealth trend is evident: as of March 2013, Research2Guidance reported that there were about 97,000 mHealth apps across 62 app stores [12]. According to a report from MarketsandMarkets, the global mHealth market is predicted to grow from $6.21 billion in revenue in 2013 to $23.49 billion by 2018 at a compound annual growth rate ($CAGR$ for short) of 30.5 percent over the five-year-period from 2013 to 2018. The mobile fitness and wellness market is expected to grow at a CAGR of 36.7 percent from 2013 to 2018 [13]. This rising mHealth market threatens changes in the way significant amounts of health data will be managed, with a paradigm shift from mainframe systems located in the facilities of healthcare providers to apps on mobiles and storage in shared cloud services. This trend is paralleled by a new openness in which devices that were once only available in hospitals become widely available to individuals outside the hospitals. Furthermore, flexible mHealth applications tempt clinicians away from the hospital-based systems they used in the past. This popular market will disruptively challenge traditional approaches by being cheap and accessible.

The proliferation of mHealth apps is radically changing the way healthcare services are delivered. In mHealth, mobile devices connected to portable or embedded medical sensors will enable long-term continuous medical or health monitoring for many purposes [14, 15, 16, 10]: for outpatients to monitor their chronic medical conditions, for individuals to increase physical activities and to change their eating habits, for athletes to track their training activities and performance, or for physicians to remotely monitor their patients' health conditions and to respond to emergencies in a timely manner.

### 3.1.2 Security and Privacy of mHealth

Despite the great potential and benefits that mHealth creates, new security and privacy risks abound in mHealth apps [10, 17, 18]. Security and privacy of health data could be significantly affected by the mHealth trend. Freed from the bonds of HIPAA, mHealth apps are free to handle data using lower assurances than those typically applied to HIPAA entities. However, the data they handle is often as sensitive as the data handled by HIPAA entities. Typical Google Play apps such as Self-help Anxiety Management, iCardio, Epocrates CME, and Clinical Advisor provide assurance with mental health concerns, activity monitoring, and information services that reveal user interests in particular symptoms or diseases. To exercise enforcement discretion on the emerging mHealth market, the Food and Drug Administration (*FDA* for short) issued guidelines for mobile medical applications [19] in September 2013, intending to apply its regulatory oversight to "mobile medical applications". Section III.C of the FDA guidance defines "mobile medical applications" as those *"to be used as accessories to regulated medical devices or to transform into a regulated medical devices"*. The FDA only regulates a small subset of mHealth apps that connect to and act as an extension of medical devices, and mobile apps that transform mobile platforms into regulated medical devices, such as an app that turns a mobile device into an electrocardiography (ECG) machine. It doesn't regulate mHealth apps in a broader range. Based on the incompleteness of FDA regulatory scopes, it is important to develop guidelines for the security and privacy of mHealth apps that suit a dynamic market while assuring that the growth of mHealth does not lead to a cavalier vendor attitude toward personal data.

## 3.2 Methodology

We carry out a three-stage study of the security and privacy status of free mHealth apps offered on Google Play. In the first study we classify the top 160 free mHealth apps in Google Play and examine them to formulate a list of six attack surfaces that need attention in mHealth apps. These are shown in Table 3.1. We then select a random sample of 27 apps from the top 1080 apps and analyze them with respect to these six attack surfaces. We find significant issues with three attack surfaces: Internet, Logging, and Third Party Services. The concerns we find with Logging will be addressed to a significant degree by the deployment of a version of Android, which is elaborated in section 6.2, so we focus our attention on the other two. We develop a random sample of additional 22 apps that involve Internet communications, analyze these 22 apps, and find a significant number of risks to security and privacy based on these two attack surfaces. Our primary conclusions are that the mHealth apps in Google Play commonly send sensitive data in clear text and store them on third party servers whose confidentiality rules may not be as strong as they need to be for the type of data being stored.

Specifically, we want to understand the threats against Android mHealth

Table 3.1: Description of attack surfaces

| Attack Surface | Description |
|---|---|
| Internet | Sending sensitive information over the Internet with insecure protocols, e.g., HTTP, misconfigured HTTPS, etc. |
| Third Party Services | Storing sensitive information on third party servers |
| Bluetooth | Sniffing or injecting sensitive information that is collected by Bluetooth-enabled health devices |
| Logging | Putting sensitive information into system logs where it is not secured |
| SD Card Storage | Storing sensitive information as unencrypted files on SD card, publicly accessible by any other app |
| Exported Components | Accessing Android app components from other apps, which are intended to be private, but set as exported |

apps by answering the following three questions:

1. What are the potential attack surfaces for Android mHealth apps?
2. How widespread is the threat?
3. How serious is the threat?

In the first study, in order to understand what the potential attack surfaces are, we identify a total number of 160 apps by selecting the top 80 free apps in Health & Fitness category and another top 80 free apps in Medical category on Google Play. To get a sense of the context of Android mHealth apps and formulate a list of attack surfaces that need attention, we divide the 160 apps into two groups with regard to their target users and classify them into eight categories by their functionalities. By reviewing research papers [20, 21] and online documents [22, 23], and analyzing the problems we found directly in these 160 mHealth apps, we develop the following six attack surfaces that represent the primary areas needing protections: *Internet, third party services, Bluetooth, logging, SD card storage*, and *exported components.*

After identifying the attack surfaces in Study 1, in Study 2, we aim to learn how widespread attack surfaces are. We choose the top 1080 free apps from the Medical and Health & Fitness categories on Google Play, 540 from each. By using a random number generator without replacement, we select 27 apps as our dataset for Study 2. Of the 27 apps we have selected, we analyze them one by one in details regarding to the six attack surfaces we have identified in Study 1. As a result, three attack surfaces are identified as important ones: *Internet, third party storage* and *logging*, because the majority of the 27 apps suffer from the problems. Figure 3.1 shows how we include and exclude apps for the app selection process in Study 2.

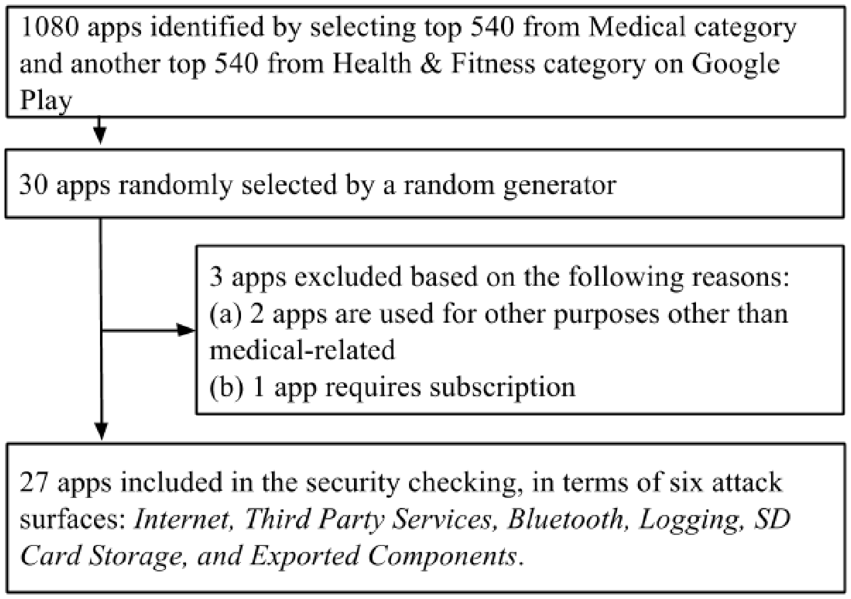

1080 apps identified by selecting top 540 from Medical category and another top 540 from Health & Fitness category on Google Play
30 apps randomly selected by a random generator
3 apps excluded based on the following reasons: (a) 2 apps are used for other purposes other than medical-related (b) 1 app requires subscription
27 apps included in the security checking, in terms of six attack surfaces: *Internet, Third Party Services, Bluetooth, Logging, SD Card Storage, and Exported Components*.

Figure 3.1: App selection flow graph for Study 2

For Study 3, we want to have a deeper understanding of how serious the threat is with regarding to the most common attack surfaces we have identified in Study 2. The app selection process here is similar to that of Study 2. We randomly select 120 apps from the top 1080 free mHealth apps on Google Play and exclude the apps that have already been studied in Study 2 and also exclude those are not sending sensitive information over the Internet. Then we analyze the included 22 apps in detail to understand how serious the threat is. Figure 3.2 shows how we include and exclude apps for the app selection process in Study 3.

Table 3.2: Classification of popular free mHealth apps on Google Play

| Target users | Category | Functionality examples | Modules used | Number of apps (%) |
|---|---|---|---|---|
| Patients | Lifestyle management | Count calories; track eating habits, exercise, sleep, period, pregnancy, etc. | gyroscope, accelerometer, GPS, network | 96(60%) |
| | Medical sensor-based monitoring | Monitor health metrics such as: heart rate, blood pressure, blood glucose, insulin, cholesterol, etc. | externally connected health sensors, network | 15(9.38%) |
| | Medical contact | Contact registered nurses, doctors or hospitals | network, phone call, email | 14(8.75%) |
| | Medication and disease | Manage prescription records; identify pills; shop medication online; look up symptoms; manage chronic diseases | Network | 27(16.88%) |
| | PHR management | Manage and/or synchronize PHR with health services | Network | 75(46.88%) |
| Healthcare professionals | Medical references | Look up drug, disease and condition; anatomy tool; medical calculator; medical dictionary | Network | 26(16.25%) |
| | Medical training | Aid medical students studying medical theories | Network | 9(5.63%) |
| | Clinical communication | Emergency alert; photo sharing | GPS, network | 2(1.25%) |

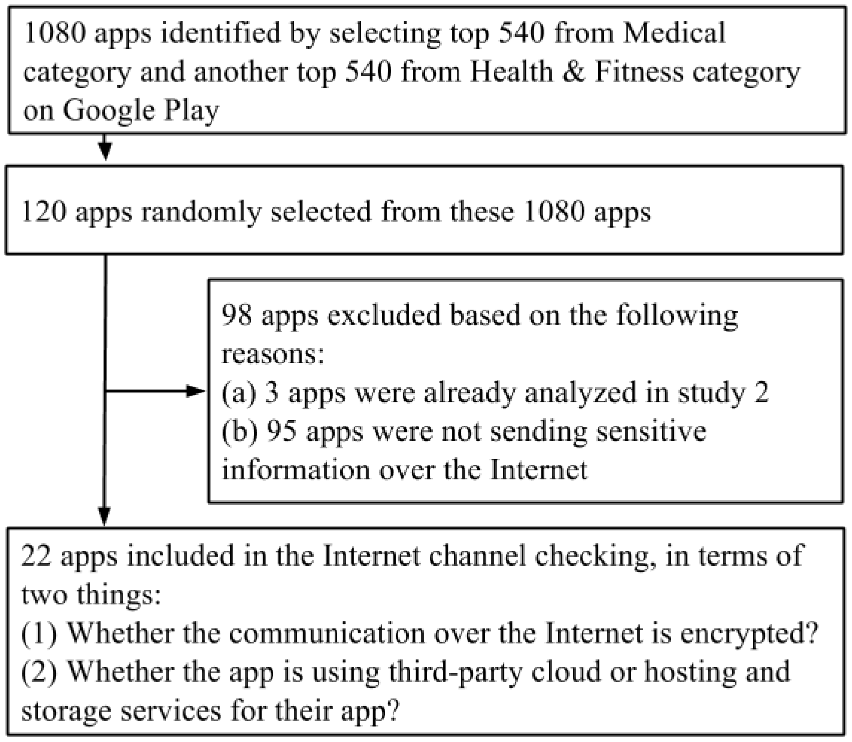

1080 apps identified by selecting top 540 from Medical category and another top 540 from Health & Fitness category on Google Play
120 apps randomly selected from these 1080 apps
98 apps excluded based on the following reasons:
(a) 3 apps were already analyzed in study 2
(b) 95 apps were not sending sensitive information over the Internet
22 apps included in the Internet channel checking, in terms of two things:
(1) Whether the communication over the Internet is encrypted?
(2) Whether the app is using third-party cloud or hosting and storage services for their app?

Figure 3.2: App selection flow graph for Study 3

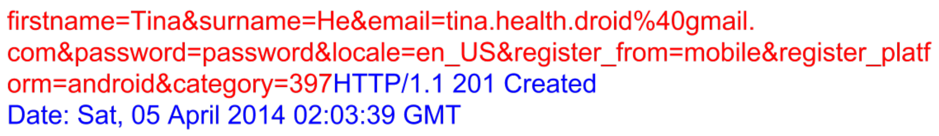
## 3.3 Study 1: What are the Potential Attack Surfaces?

To investigate the potential attack surfaces, we first try to understand the context of Android mHealth apps. We summarize the classification of Android mHealth apps by studying the 160 apps collected as described in section 3.2. In Table 3.2, we divide the top 160 free mHealth apps into two groups by the expected user. Patient apps are the ones mainly used by the individual whose health is being monitored. In most cases the monitoring is done by the individual herself. Healthcare professional apps are the ones mainly used by physicians, nurses, medical students and other healthcare professionals mainly to support their activities, including the monitoring of patients. We further classify the apps into 8 categories according to their functionalities. These categories include *Lifestyle management, Sensor-based health monitoring, Medical contact, Medication and disease management*, and *Personal health record (PHR for short) management (Here we define PHR management as patients to sync and manage health records on an online service provider.)* targeted for Patients, and *Medical references, Medical training*, and *Clinical communication* targeted for Professionals. An mHealth app may be useful for both the Patients and the Professionals (e.g., a pill identifier app can be used by a patient to organize pills or to be used by a pharmacist to prevent errors in dispensing medications). Besides, an mHealth app may belong to more than one category, since it may serve multiple functionalities (e.g., a fitness tracking app can monitor lifestyle data as well as manage PHR).

Most of the applications in the categories are appropriate for our study but we exclude one app because it lack a medical or healthcare purpose, and we exclude another app because its language is not English. Among the included 158 apps, we have 129 (81.65%) that are Patient-facing, 32 (20.25%) that are Professional-facing, and 3 (1.90%) apps, drug identifiers, are both. All the Patient-facing apps are from the Health & Fitness category and 41.03% of the apps from the Medical category are Professional-facing. From Table 3.2, we can see the majority (60%) of our selected apps belong to the Life management category. Nearly half (46.88%) of the apps manage and synchronize user health records to online service providers. The average rating score for the Patient-facing apps have almost 4 times more user installations than Professional-facing apps.

We identify several attack surfaces as important when we review previous literature [20, 21] and online documents [22, 23]. By studying the 160 apps, we have a sense of what commercial mHealth apps do and whether indeed there exist attack surfaces exposing vulnerabilities. We find real security threats against many Android mHealth apps, and identity six potential attack surfaces as the most important: *Internet, Third Party Services, Bluetooth, Logging, SD Card Storage*, and *Exported Components*. Here we show several example vulnerabilities found in Android commercial mHealth apps and demonstrate that they can lead to realistic and serious consequences.

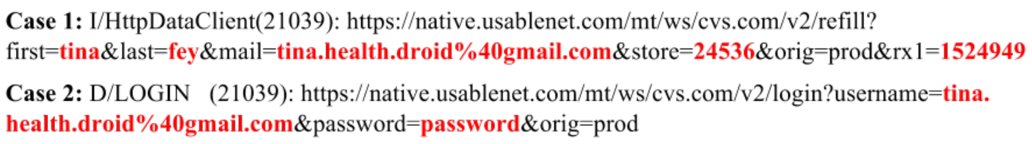
**Example 1(Unencrypted Internet).** From Table 3.2, many mHealth apps send information and most of them send unencrypted information over the Internet. For example, both Doctor Online [24] (patients can contact doctors online) and Recipes by Ingredients [25] (patients can search recipes according their illness or ingredients suitable for their disease), are not only sending sensitive information over the Internet unencrypted but also are sending user email and password in cleartext over the Internet. Figure 3.3 shows the network traffic from Doctor Online captured by Wireshark that contains user's name, email and password in cleartext.



```
firstname=Tina&surname=He&email=tina.health.droid%40gmail.
com&password=password&locale=en_US&register_from=mobile&register_platf
orm=android&category=397HTTP/1.1 201 Created
Date: Sat, 05 April 2014 02:03:39 GMT
```

Figure 3.3: Network traffic from Doctor Online containing sensitive information

**Example 2 (Logging vulnerability).** We find many mHealth apps put user's sensitive information into logs. For example, CVS/pharmacy [26], a popular app with millions of users on Google Play, put user login credentials and other sensitive information in their apps' log messages. Figure 3.4 shows the log messages with sensitive information from CVS/pharmacy. In Case 1, CVS/pharmacy logs prescription refill details from user input, including name, email address, store number, and Rx number. In Case 2, CVS/pharmacy puts login credentials in a debug log message. With the login credentials for CVS/pharmacy, anyone could view a user's profile and prescription history, which can potentially lead to a medical identity theft. A malicious party can even conduct pharmacy online shopping with a user's credit card information stored in the CVS/pharmacy online shopping store.



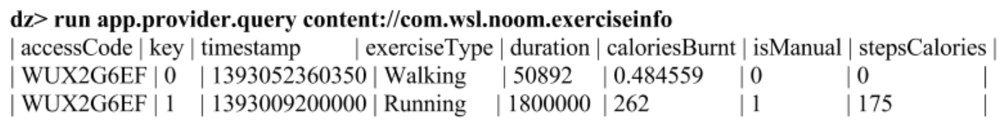
```
Case 1: I/HttpDataClient(21039): https://native.usablenet.com/mt/ws/cvs.com/v2/refill?
first=tina&last=fey&mail=tina.health.droid%40gmail.com&store=24536&orig=prod&rx1=1524949
Case 2: D/LOGIN   (21039): https://native.usablenet.com/mt/ws/cvs.com/v2/login?username=tina.
health.droid%40gmail.com&password=password&orig=prod
```

Figure 3.4: Log messages from CVS/pharmacy containing sensitive information

Note that in both cases, the sensitive information is in HTTPS URLs using a `GET` request method. Developers may have a misconception that all HTTPS requests, either using `GET` or `POST`, are sent over encrypted TCP connections so that sensitive information can be safely put into the HTTPS URLs. However, even if sensitive information won't be seen during transition, it still remains visible in places like mobile app logs, server logs, browser history and etc. Developers should avoid putting sensitive information in any form of logs.

**Example 3 (Exported components vulnerability).** We find that several mHealth apps have security threats rooted from component exposures.

For instance, Noom Weight Loss Coach [27], an app with more than 10 million installations, exposes its Content Providers to external apps, which means any app can access the exposed Content Providers without declaring any permission. After searching for `"content://"` paths in Manifest and the source code decompiled from Noom Weight Loss Coach apk, we get a list of content URIs defined in the app. Figure 3.5 demonstrates using an automatic security analysis tool called Drozer [28] to read user workout history from Content Provider with the content URI `"content://com.wsl.noom.exerciseinfo"`.



```
dz> run app.provider.query content://com.wsl.noom.exerciseinfo
| accessCode | key | timestamp     | exerciseType | duration | caloriesBurnt | isManual | stepsCalories |
| WUX2G6EF | 0   | 1393052360350 | Walking      | 50892    | 0.484559      | 0        | 0             |
| WUX2G6EF | 1   | 1393009200000 | Running      | 1800000  | 262           | 1        | 175           |
```

Figure 3.5: Access Noom Weight Loss Coach's user workout history using Drozer

**Example 4 (SD card storage vulnerability).** We discover many sleep monitoring apps, such as SnoreClock [29] and Sleep Talk Recorder [30], record sound when users sleep and store the sleep recordings as unencrypted audio files on an external storage. For example, the Sleep Talk Recorder explicitly stores sleep-recording audio files unencrypted on SD card as `YYYY-MM-DD-HH-MM-SS.wav`. A malicious app with read storage permission can read the users' sleep recordings; with internet permission, it can further send out the files. Another example is that Urgent Care [31] stores system logs in an unencrypted file on SD card, which leaks out what symptoms the patient looked up within this app.

## 3.4 Study 2: How Widespread is the Threat?

The complexity of Android system has led to numerous potential attack surfaces, which could be explored by a malicious party to gain unauthorized access to sensitive data in mHealth apps and cause serious consequences. Analyzing these attack surfaces help security specialists perform security assessments as well as help mHealth users/vendors understand and manage security risks. In Study 2, we analyze these attack surfaces with a new set of randomly selected apps. The 27 mHealth apps in Study 2 are identified as described in section 3.2.

**Internet.** Many apps access Internet either to transfer information to a remote server or to display ads. Information that is transferred over the Internet to the remote server is sensitive health information and ideally all such communication with the remote server should be encrypted. We analyze the randomly sampled apps to study why they require the Internet access (i.e. to transfer information or to display ads). Furthermore, we analyze if the encrypted communication is used to transfer the information to the remote server.

Any app can get access to the Internet using `INTERNET` permission. To study if the apps are using Internet for displaying ads or transferring information to the remote server, we study the description of the apps and check the functionality of the apps by installing and using them on a Samsung Galaxy SII phone. We observe that 85.2% (23/27) of all the apps have permissions to access the Internet. 70.4% (19/27) use the Internet permissions to display ads, while 29.6% (8/27) of them use the Internet permissions to communicate information over the Internet.

To study whether the communication with the remote server is encrypted, we install and use the apps. We capture their network traffic using "Shark for Root" app and analyze the network traffic in Wireshark to see if the traffic is encrypted. We find that 7.4% (2/27) of the apps allow the users to use the blog or social network associated with the app via Internet, of which only one of the apps is using encrypted communication. 25.9% (7/27) of all the apps transmit medical information to the remote server. 66.7% (4/6) of the apps use encrypted communication, while 50% (3/6) of the apps use unencrypted communication to transfer the sensitive health related information.

We further analyze if the three aforementioned apps sending unencrypted data over the Internet are sending sensitive information. And we find that one of the app is used to search for nearby pharmacies, doctors, etc., the second app is used to track exercise workouts, and the third (from Spain) is used to find and contact doctors online. The third app (Doctor Online) is sending email, username and even password unencrypted over the Internet.

**Third Party Services.** Android apps use storage and hosting services such as Amazon instead of maintaining their own infrastructure. This is an economical as well as scalable solution for mobile app vendors. But, storing sensitive health information on these third party services can have serious implications and that is why HIPAA does not allow storing unencrypted data on third party cloud services. 22.2% (6/27) apps are sending sensitive data over the Internet. To study if these 6 apps that are actually transferring information to remote servers are hosted on the cloud or in-premises servers owned by the app vendors, we analyze the IP addresses of the apps from their communication with their respective servers. IP addresses have a publicly available record of to whom it belongs to and we use that information to find out from where the traffic is coming. We find surprising results: all six apps are hosted on third party services. Three of them are hosted on Amazon and the rest three are hosted on other hosting services. Note that these apps are using encryption for the communication only, we are not sure if data on the remote third party services is stored in a encrypted fashion such that the hosting companies do not have access to the data.

**Bluetooth.** Many Bluetooth sensing apps use Bluetooth primarily to collect data from health sensors to mobile devices. Out of the 27 apps in our dataset, one app (3.7%) connects to a Bluetooth health device to collect extra health information. Supporting Bluetooth devices is more common among the 160 most popular Android mHealth apps, where 15 (9.5%) provide options to collect health data from external health sensors and all of them are Patient-facing apps. 12 of the 15 apps declare and use both `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions, so that they can use Bluetooth to connect to and collect data from external health sensors, while the remaining three apps collect health data via Internet or by connecting with other apps. All the apps we find with Bluetooth connectivity are Patient-facing apps. They collect various types of health information, including heart rate, respiration, pulse oximetry, electrocardiogram (*ECG* for short), blood pressure, body weight, body temperature, quality of sleep, exercise activities and etc. Apparently, Bluetooth is a major communication technology for sensor-based health monitoring in mHealth apps. Naveed et al. [20] present a problem of external-device misbinding (*DMB* for short), under the context of Bluetooth-enabled Android devices and health sensors. They show that a malicious app can stealthily collect user data from an Android device and also deploy a spoofed device that injects fake data into the original device's app. We find one of the 27 apps actually connects to an external health sensor and uses a default PIN code `0000`, which makes that app vulnerable to the DMB attack. To defend against the Bluetooth-based threats on mHealth apps, Naveed et al. [20] propose an OS-level protection, which generates secure binding policies between a device and its official app and enforces these rules when establishing and unpairing Bluetooth connections.

**Logging.** Android logging system is designed for app developers to collect and view app debug output. The logging facility allows a system-wide logging, including application information and system events. If an app is granted `READ_LOGS` permission, the app is allowed to read low-level system log messages. With the `READ_LOGS` permission, a malicious app can monitor system logs from mHealth apps in the background and extract sensitive information from the log messages. To discover logging vulnerability, we use a tool called `logcat` from Android Debug Bridge (*ADB* for short) shell to view system log messages.

In our dataset, we find 9 out of 27 apps (33.3%) include sensitive information in app logs. Among the 9 aforementioned apps, two (22.2%) disclose GPS coordinates, three (33.3%) disclose Facebook friend information, one (11.1%) discloses the direction to the dentist office that a user inquires about. One (11.1%) discloses sensitive data that could lead to more serious consequences, that is disclosing user sign up data, including name, location and profession. That app also leaks the medical tests that user takes on this app. All the disclosure of private data from this app can potentially lead to a medical identity theft. Three (33.3%) apps leak disease and drug browsing history in the app logs. From the study on our 27-app dataset, we find a large number (33.3%) of mHealth apps leak sensitive information in system logs and many of them could cause serious problems.

**SD Card Storage.** Each Android app gets a dedicated part of file system where it can write its private data. However, if an app writes files to an external storage, such as an SD card, the files are not guaranteed to be protected. With `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permissions, any app can read or write files from an external storage. Note that before API level 19, the `READ_EXTERNAL_STORAGE` permission is not enforced and all apps still have access to read from an external storage.

In our dataset, 66.7% (18/27) of the apps declare the `WRITE_EXTERNAL_STORAGE` permission, which means they write data to the external storage and the data is universally readable by any app with the `READ_EXTERNAL_STORAGE` permission. Then, we use an existing tool Dex2jar [32] to decompile the 27 apps' apk (application package) files to get their Java source code. After that, we search for `"ExternalStorage"` and `"ExternalFiles"` keywords in the source code to construct all possible paths for any file to be stored on the SD card. Then, we execute all possible operations with the studied apps and go through these directories to check their file contents. As a result, we find none of the apps stores sensitive information as files on the external storage.

**Exported Components.** An Android app developer has the rights to specify if an app component (i.e. *Activity, Service, Broadcast Receiver*, or *Content Provider*) is public to external apps or not. A component is declared as *exported*, or public, if its declaration sets the `EXPORTED` flag or includes at least one *Intent Filter* without any permission protection. However, setting a private component improperly as *exported* enables a malicious app to send unwanted *Intents* to the component, which can cause security problems [21]: *broadcast injection, activity launch*, or *service launch*. In addition, if the *Content Provider* is *exported*, a malicious app can read or write the *exported Content Provider* without declaring any particular permission. The *Content Provider* supports basic `"CRUD"` (create, retrieve, update and delete) functions and the data in a *Content Provider* is addressed via a `"content URI"`. Knowing the `"content URI"` from an *exported Content Provider*, a malicious app can retrieve or modify the data according to the *Content Provider*'s schema. We already showed an example of unauthorized access to an *exported Content Provider* to read its sensitive information in Study 1.

## 3.5 Study 3: How Serious is the Threat?

Study 2 reveals three vulnerabilities that are common and serious: sending sensitive information unencrypted over the Internet, using third party services, and logging containing private data. Since logging can be fixed by Android version upgrade, we focus on the other two threats, which are the Internet traffic and third party services.

As only 6 apps in Study 2 are actually sending sensitive information over the Internet, we perform another study to understand the prevalence of the two threats. We randomly sample another 120 apps from the 540 top Health & Fitness apps and 540 top Medical apps (1080 in total) on Google Play. Then we manually analyze these apps to filter those are not sending any sensitive information over the Internet. The filtering results show that 22 apps are actually sending sensitive data over the Internet and we could analyze their Internet traffic (some apps require subscription and we filter out those also). We install these 22 apps and capture their traffic using the same techniques described in Study 2. We find that 63.6% (14/22) of these apps are sending data unencrypted over the Internet and 81.8% (18/22) of them are using third party storage and hosting services such as Amazon's cloud services. One of our randomly selected apps is Fitbit app and it is using encryption over the Internet, but is also using third party storage and hosting services. The four apps that are using their own servers to store and host their app data are big companies such as Aetna, United Healthcare, Caring Bridge and US Dept. of Health and Human Services. If third party services receive data in cleartext, there is no point in sending data in encrypted fashion. We do not have the ground truth, but we suspect that the apps that are doing encryption for the communication are storing data unencrypted on the clouds.

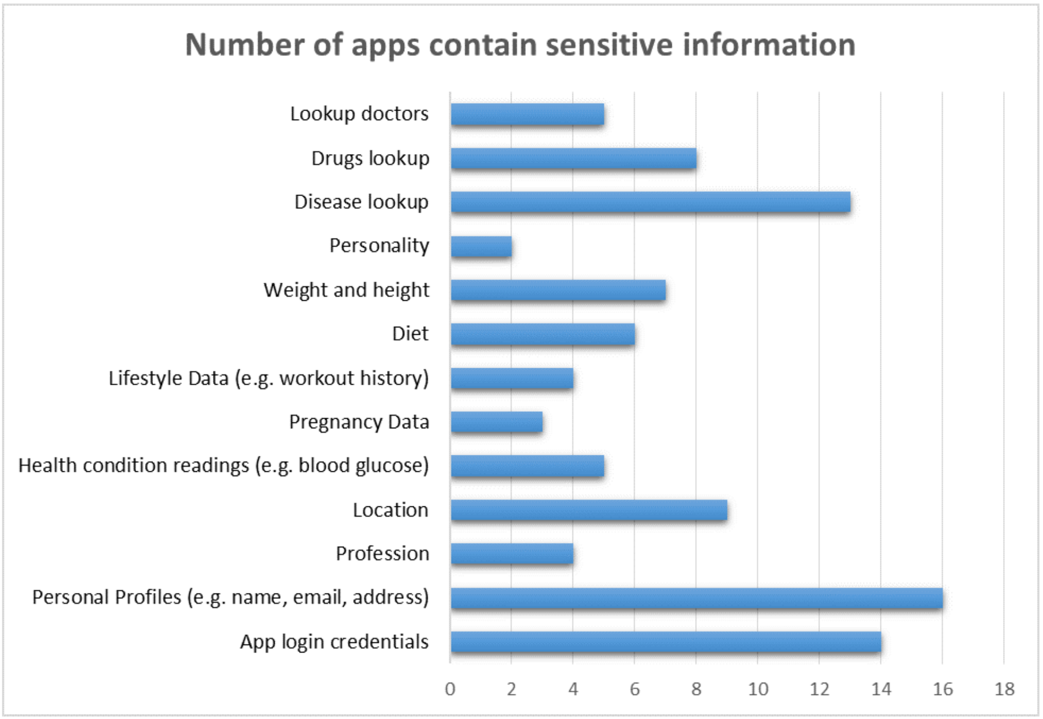


Figure 3.6: Sensitive information distribution in the 22-app dataset for Study 3

When used as intended, a variety of sensitive user data are collected, stored, or transmitted by these Android mHealth apps. Figure 3.6 shows the distribution of sensitive information in these 22 apps. Based on our study, these include at least personal profiles, health sensor data, lifestyle data, medical information browsing history, and third-party app data (e.g. Facebook account information). Depending on the type, sensitivity, and volume of mHealth data breaches, disclosure or tampering with these sensitive data may lead to serious consequences, such as loss of privacy, medical identity theft, and errors in healthcare decision-making. According to a report by World Privacy Forum [33], thefts have used stolen medical information for a resourceful collection of nefarious reasons. For example, a Colorado man whose Social Security number, name and address had been stolen received a bill for $44,000 he presumably owed to a hospital because his identity had been used by a thief to get medical services in his name. In another case, another identity thief in Missouri used the personal data of multiple victims to establish false driving licenses and was able to use them to obtain prescriptions in the victims' names at a regional health center. All the above examples prove that leakage of health information can lead to serious consequences.

# CHAPTER 4

# SIDE-CHANNEL THREATS AGAINST ANDROID APPS

After understanding the widespread security threats in Android mHealth apps in the last chapter, we ask a further question: *are there any limitations in Android security design that could be utilized by malicious parties to pilfer mobile users' private information on the phone?* In this chapter, we introduce a newly-discovered side channel that can cause information leaks on Android, which is per-app data-usage statistics. By utilizing this side channel, a malicious party can divulge users' private information on Android devices. This reveals the gap between the fundamental limitations of Android security design and the diversity in the ways the system is utilized by developers. Section 4.1 illustrates the related work of side-channel information leaks on general platforms and Android devices. Section 4.2 describes the techniques we develop for learning private information from Android public resources with zero permission. More specifically, in section 4.2.1 describes the public resources on Android leaking out information, section 4.2.2 and 4.2.3 show how a zero-permission malicious app monitors and analyzes the public resources. Sections 4.3 and 4.4 demonstrate that we can actually utilize the public resources on Android to infer users' identity (section 4.3) and investment interests (section 4.4).

## 4.1 Related Work

Side-channel information leaks have been studied for decades and new discoveries continue to be made in recent years [34, 35, 36]. Among them, most related to our work is the work on the information leaks from procfs, which includes using ESP/EIP data to infer keystrokes [37] and leveraging memory usage to fingerprint visited websites [38]. However, it is less clear whether those attacks pose a credible threat on Android, due to the high non-determinism of its memory allocation [38] and the challenges in keystroke analysis [37]. In comparison, our work shows that the usage statistics under procfs can be practically exploited to infer an Android user's sensitive information. The attack technique used here is related to prior work on traffic analysis [39]. However, those approaches assume the presence of an adversary who sees encrypted packets. Also, their analysis techniques cannot be directly applied to smartphone. Our attack is based upon a different adversary model, in which an app uses public resources to infer the content of the data received by a target app on the same device. For this purpose, we need to build different inference techniques based on the unique features of mobile computing, particularly the rich background information (i.e., social network) that comes with the target app.

Information leaks have been discovered on smartphone by both academia and the hacker community [7, 8, 9]. Most of known problems are caused by implementation errors, either in Android or within mobile apps. By comparison, the privacy risks come from shared resources in the presence of emerging background information have not been extensively studied on mobile devices. Up to our knowledge, all prior research on this subject focuses on the privacy implications of motion sensors or microphones [40, 41, 38, 42, 43]. What has never been done before is a systematic investigation on what can be inferred from the public resources exposed by both Linux and Android layers.

New techniques for better protecting user privacy on Android also continue to pop up [44, 45, 8, 46, 47, 48, 7]. Different from such research, our work focuses on the new privacy risks emerging from the fast-evolving smartphone apps, which could render innocuous public resources related to sensitive user information.

## 4.2 Mechanism

An in-depth understanding of information leaks from Android public resources is critical, as it reveals the gap between the fundamental limitations of Android security design and the diversity in the ways the system is utilized by app developers. This understanding will be invaluable for the future development of securer mobile OSes that support the evolving utility. However, a study on the problem has never been done before. In this chapter, we report our first study on this crucial yet understudied direction.

Our study inspects public resources disclosed at both the Android and its Linux layers and analyzes the impact such exposures can have on the private information maintained by a set of popular apps, in the presence of the rich background information they bring in. This research leads to a series of stunning discoveries on what can actually be inferred from those public resources by leveraging such auxiliary information. Specifically, by monitoring the network-data usage statistics of high-profile Android apps, such as Twitter and Yahoo! Finance (one of the most widely-used stock apps), one can find out a smartphone user's true identity and stocks one is interested in, without any permission at all.

All such information leaks are found to be strongly related to the fallacies of design assumptions instead of mere implementation bugs. Every piece of information here is actually meant to be disclosed by Android and most of such data has been extensively used by legitimate Android apps: for example, hundreds of data usage monitors are already out there [49], relying on the usage statistics to keep track of a user's mobile data consumption.

### 4.2.1 Leaks from Public Resources

Android is an operating system based on Linux kernel. Its security model is based on Linux's kernel level protection (process separation, file system access control). Specifically, each Android app is assigned with a unique user ID and runs as that user. Sensitive resources are usually mapped to special Linux groups such as inet, gps, etc. This approach, called *application sandboxing*, enables the Linux kernel to separate an app from other running apps. Within a sandbox, an app can invoke Android APIs to access system resources. The APIs that operate on sensitive resources, including camera, location, network, etc., are protected by permissions. An app needs to explicitly request (using AndroidManifest.xml) from the device's user such permissions during its installation so as to get assigned to the Linux groups of the resources under protection, before it can utilize such resources.

**Public resources on Android.** Like any operating system, Android provides a set of shared resources that underprivileged users (apps without any permission) can access. This is necessary for making the system easy to use for both app developers and end users. Following is a rough classification of the resources available to zero-permission apps:

- *Linux layer: public directories.* Linux historically makes available a large amount of resources considered harmless to normal users, to help them coordinate their activities. A prominent example is the process information displayed by the `ps` command (invoked through `Runtime.getRuntime.exec`, which includes each running process's user ID, Process ID (PID), memory and CPU consumption and other statistics. Most of such resources are provided through two virtual filesystems, the proc filesystem (procfs) and the sys filesystem (sysfs). The procfs contains public statistics about process's use of memory, CPU, network resources and other data. Under the sysfs directories, one can find device/driver information, network environment data (`/sys/class /net/`) and more. Android inherits such public resources from Linux and enhances the system with new ones (e.g. `/proc/uid_stat`). For example, the network traffic statistics (`/proc/uid_stat/tcp_snd` and `/proc/uid_stat/tcp_rcv`) are extensively utilized [49] to keep track of individual apps' mobile data consumption.

- *Android layer: Android public APIs.* In addition to the public resources provided by Linux, Android further offers public APIs to enable apps to get access to public data and interact with each other. An example is `AudioManager.requestAudioFocus`, which coordinates apps' use of the audio resource (e.g., muting the music when a phone call comes in).

**Privacy risks.** All such public resources are considered to be harmless and their releases are part of the design which is important to the system's normal operations. Examples include the coordination among users through `ps` and among the apps using audio resources through `AudioManager.requestAudio-Focus`. However, those old design assumptions on the public resources are becoming increasingly irrelevant in front of the fast-evolving ways to use smartphones. We find that the following design/use gaps are swiftly widening:

- *Gap between Linux design and smartphone use.* Linux comes with the legacy of its original designs for workstations and servers. Some of its information disclosure, which could be harmless in these stationary environments, could become a critical issue for mobile phones. For example, Linux makes the MAC address of the wireless access points (WAP) available under its procfs. This does not seem to be a big issue for a workstation or even a laptop back a few years ago. For a smartphone, however, knowledge about such information will lead to disclosure of a phone user's location, particularly with the recent development that databases have been built for fingerprinting geo-locations with WAPs' MAC addresses (called *Basic Service Set Identification, or BSSID*).

- *Gap between the assumptions on Android public resources and evolving app design, functionalities and background information.* Even more challenging is the dramatic evolution of Android apps. For example, an app is often dedicated to a specific website. Therefore, an adversary no longer needs to infer the website a user visits, as it can be easily found out by looking at which app is running (through `ps` for example). Most importantly, today's apps often come with a plethora of background information like tweets, public posts and public web services such as Google Maps. As a result, even very thin information about the app's behavior (e.g. posting a message), as exposed by the public resources, could be linked to such public knowledge to recover sensitive user data.

**Information leaks.** In our research, we carefully analyze the ways that public resources are utilized by the OS and popular apps on Android, together with the public online information related to their operations. Our study discovered a new source of information leak, which is *App network-data usage*. We find that the data usage statistics disclosed by the procfs can be used to precisely fingerprint an app's behavior and even infer its input data, by leveraging online resources such as tweets published by Twitter. To demonstrate the seriousness of the information leakage from those usage data, we develop a suite of inference techniques that can reveal a phone user's identity from the network-data consumption of Twitter app and the stock she is looking at from Yahoo! Finance app. We build a zero-permission app that stealthily collects information for these attacks.

### 4.2.2 Zero-Permission Adversary

**Adversary model.** The adversary considered in our research runs a zero-permission app on the victim's smartphone. Such an app needs to operate in a stealthy way to visually conceal its presence from the user and also minimize its impact on a smartphone's performance. On the other hand, the adversary has the resources to analyze the data gathered by the app using publicly available background information, for example, through crawling the public information released by social networks, etc. Such activities can be performed by ordinary Internet users.

**What the adversary can do.** In addition to collecting and analyzing the information gathered from the victim's device, a zero-permission malicious app needs a set of capabilities to pose a credible privacy threat. Particularly, it needs to send data across the Internet without the INTERNET permission. Also, it should stay aware of the system's *situation*, i.e., which apps are currently running? This enables the malicious app to keep a low profile, start data collection only when its target app is being executed. Here we show how these capabilities can be obtained by the app without any permission.

- *Networking.* Leviathan's blog describes a zero-permission technique to smuggle out data across the Internet [9]. The idea is to let the sender app use the `URI ACTION_VIEW` Intent to open a browser and sneak the payload it wants to deliver to the parameters of an HTTP `GET` from the receiver website. We re-implement this technique in our research and further make it stealthy. Leviathan's approach does not work when the screen is off because the browser is `paused` when the screen is off. We improve this method to smuggle data right before the screen is off or the screen is being unlocked. Specifically, our app continuously monitors `/lcd_power` (`/sys/class/lcd/panel/lcd_power` on Galaxy Nexus), an LCD status indicator released under the sysfs. Note that this indicator can be located under other directory on other devices, for example, `sys/class/backlight/s6e8aa0` on Nexus Prime. When the indicator becomes zero, the phone screen dims out, which allows our app to send out data through the browser without being noticed by the user. After the data transmission is done, our app can redirect the browser to Google and also set the phone to its home screen to cover this operation.

- *Situation awareness.* Our zero permission app defines a list of target applications such as social network app and stock applications and monitors their activities. It first checks whether those packages are installed on the victim's system (`getInstalledApplications()`) and then periodically calls `ps` to get a list of active apps and their PIDs. Once a target is found to be active, our app will start a thread that closely monitors the `/proc/uid_stats/[uid]` and the `/proc/[pid]/` of the target.

### 4.2.3 Usage Monitoring and Analysis

**Mobile-data usage statistics.** Mobile data usages of Android are made public under `/proc/uid_stat/` (per app) and `/sys/class/net/[interface]/statistics/` (per interface). The former is newly introduced by Android to keep track of individual apps. These directories can be read by *any* app directly or through `TrafficStats`, a public API class. Of particular interest here are two files `/proc/uid_stat/[uid]/tcp_rcv` and `/proc/uid_stat/[uid]/tcp_snd`, which record the total number of bytes received and sent by a specific app respectively. We find that these two statistics are actually aggregated from TCP packet payloads: for every TCP packet received or sent by an app, Android adds the length of its payload onto the corresponding statistics. These statistics are extensively used for mobile data consumption monitoring [49]. However, our research shows that their updates can also be leveraged to fingerpint an app's network operations, such as sending HTTP POST or GET messages.

**Stealthy and realtime monitoring.** To catch the updates of those statistics in real time, we build a data-usage monitor that continuously reads from `tcp_rcv` and `tcp_snd` of a target app to record increments in their values. Such an increment is essentially the length of the payload delivered by a single or multiple TCP packets the app receives and sends, depending on how fast the monitor samples from those statistics. Our current implementation has a sampling rate of 10 times per second. This is found to be sufficient for picking up individual packets most of the time, as illustrated in Figure 4.1, in which we compare the packet payloads observed by Shark for Root (a network traffic sniffer for 3G and WiFi), when the user is using Yahoo!

Table 4.1: Performance overhead of the monitor tool: there the baseline is measured by AnTuTu [1]

| | Total | CPU | GPU | RAM | I/O |
|---|---|---|---|---|---|
| Baseline | 3776 | 777 | 1816 | 588 | 595 |
| Monitor Tool | 3554 | 774 | 1606 | 589 | 585 |
| Overhead | 5.8% | 0.3% | 11.6% | -0.1% | 1.7% |

Finance, with the cumulative outbound data usage detected by our usage monitor.

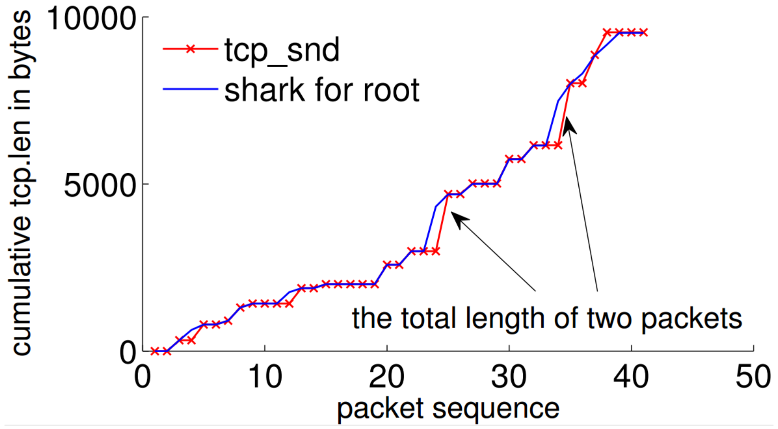


Figure 4.1: Monitor tool precision

From the figure, we can see that most of the time, our monitor can separate different packets from each other. However, there are situations in which only the cumulative length of multiple packets is identified (see the markers in the figure). This requires an analysis that can tolerate such non-determinism, which we will discuss later.

In terms of performance, our monitor has a very small memory footprint, only 28 MB, even below that of the default Android keyboard app. When it is running at its peak speed, it takes 7% of a core's cycles on a Google Nexus S phone. Since all the new phones released today are armed with multi-core CPUs, the monitor's operations will not have noticeable impacts on the performance of the app running in the foreground as demonstrated by a test described in Table 4.1 measured using AnTuTu [1] with a sampling rate of 10Hz for network usage. The scores in Table 4.1 measured by AnTuTu benchmark imply the performances of a mobile device when or when not running our monitor tool. The higher the scores are, the better the performance of a mobile device are. To make this data collection stealthier, we adopt a strategy that samples intensively only when the target app is being executed, which is identified through `ps`.

**Analysis methodology.** The monitor cannot always produce deterministic outcomes: when sampling the same packet sequence twice, it may observe two different sequences of increments from the usage statistics. To obtain a reliable traffic fingerprint of a target app's activity we design a methodology to bridge the gap between the real sequence and what the monitor sees.

Our approach first uses Shark for Root to analyze a target app's behavior (e.g., click on a button) offline - i.e. in a controlled context - and generate a payload-sequence signature for the behavior. Once our monitor collects a sequence of usage increments from the app's runtime on the victim's Android phone, we compare this usage sequence with the signature as follows. Consider a signature $(\cdots, s_i, s_{i+1}, \cdots, s_{i+n}, \cdots)$, where $s_{i,\cdots,i+n}$ are the payload lengths of the TCP packets with the same direction (inbound/outbound), and a sequence $(\cdots, m_j, \cdots)$, where $m_j$ is an increment on a usage statistic (`tcp_rcv` or `tcp_snd`) of the direction of $s_i$, as observed by our monitor. Suppose that all the elements before $m_j$ match the elements in the signature (those prior to $s_i$). We say that $m_j$ also matches the signature elements if either $m_j = s_i$ or $m_j = s_i + \cdots + s_{i+k}$ with $1 < k \leq n$. The whole sequence is considered to *match* the signature if all of its elements match the signature elements.

The payload-sequence signature can vary across different mobile devices, due to the difference in the User-Agent field on the HTTP packets produced by these devices. This information can be acquired by a zero-permission app through the `android.os.Build` API.

## 4.3 Identity Inference

A person's identity, such as name, email address, etc., is always considered to be highly sensitive [50, 51] and should not be released to an untrusted party. For a smartphone user, unauthorized disclosure of her identity can immediately reveal a lot of private information about her (e.g., disease, sex orientation, etc.) simply from the apps on her phone. Here we show one's identity can be easily inferred using the shared resources and rich background information from Twitter.

Twitter is one of the most popular social networks with about 500 million users worldwide. It is common for Twitter users to use their mobile phones to tweet extensively and from diverse locations. Many Twitter users disclose their identity information which includes their real names, cities and sometimes homepage or blog URL and even pictures. Such information can be used to discover one's accounts on other social networks, revealing even more information about the victim according to prior research [52]. We also perform a small range survey on the identity information directly disclosed from public Twitter accounts to help us better understand what kind of information users disclose and at which extend. By manually analyzing randomly selected 3908 accounts (obvious bot accounts excluded), we discover that 78.63% of them apparently have users' first and last names there, 32.31% set the users' locations, 20.60% include bio descriptions and 12.71% provide URLs. This indicates that the attack we describe below poses a realistic threat to Android users' identity.

**The idea.** In our attack, a zero-permission app monitors the mobile-data usage count `tcp_snd` of the Twitter 3.6.0 app when it is running. When the user send tweets to the Twitter server, the app detects this event and send its timestamp to the malicious server stealthily. This gives us a vector of timestamps for the users' tweets, which we then use to search the tweet history through public Twitter APIs for the account whose activities are consistent with the vector: that is, the account's owner posts her tweets at the moments recorded by these timestamps. Given a few of timestamps, we can uniquely identity that user. An extension of this idea could also be applied to other public social media and their apps, and leverage other information as vector elements for this identity inference: for example, the malicious app could be designed to figure out not only the timing of a blogging activity, but also the number of characters typed into the blog through monitoring of the CPU usage of the keyboard app, which can then be correlated to a published post.

To make this idea work, we need to address a few technical challenges. Particularly, searching across all 340 million tweets daily is impossible. Our solution is using less protected, the coarse location (e.g., city) of the person who tweets, to narrow down the search range.

**Fingerprinting tweeting event.** To fingerprint the tweeting event from the Twitter app, we use the aforementioned methodology to first analyze the app *offline* to generate a signature for the event. This signature is then compared with the data usage increments our zero-permission app collects *online* from the victim's phone to identify the moment she tweets.

Specifically, during the offline analysis, we observed the following TCP payload sequence produced by the Twitter app: $(420|150, 314, 580 - 720)$. The first element here is the payload length of a TLS Client Hello. This message normally has 420 bytes but can become 150 when the parameters of a recent TLS session are reused. What follow are a 314-byte payload for Client Key Exchange and then that of an encrypted HTTP request, either a `GET` (download tweets) or a `POST` (tweet). The encrypted `GET` has a relatively stable payload size, between 541 and 544 bytes. When the user tweets, the encrypted `POST` ranges from 580 to 720 bytes, due to the tweet's 140-character limit. So, the length sequence can be used as a signature to determine when a tweet is sent.

As discussed before, what we want to do here is to use the signature to find out the timestamp when the user tweets. The problem here is that our usage monitor running on the victim's phone does not see those packets and can only observe the increments in the data-usage statistics. Our offline analysis shows that the payload for Client Hello can be reliably detected by the monitor. However, the time interval between Key-Exchange message and `POST` turns out to be so short that it can easily fall through the cracks. Therefore, we have to resort to the aforementioned analysis methodology to compare the data-usage sequence collected by our app with the payload signature: a tweet is considered to be sent when the increment sequence is either $(420|150, 314, 580 - 720)$ or $(420|150, 894 - 1034)$.

**Identity discovery.** From the tweeting events detected, we obtain a sequence of timestamps $T = [t_1, t_2, \cdots, t_n]$ that describe when the phone user tweets. This sequence is then used to find out the user's Twitter ID from the public index of tweets. Such an index can be accessed through the Twitter Search API [53]: one can call the API to search the tweets from a certain geo-location within 6 to 8 days. Each query returns 1500 most recent tweets or those published in the prior days (1500 per day). An unauthorized user can query 150 times every hour.

To collect relevant tweets, we need to get the phone's geo-location, which is specified by a triplet (latitude, longitude, radius) in the twitter search

Table 4.2: City information and Twitter identity exploitation

| Location | Population | City size | Time interval covered (radius) | # of timestamps |
|---|---|---|---|---|
| Urbana | 41,518 | 11.58 mi$^2$ | 243 min(3 mi) | 3 |
| Bloomington | 81,381 | 19.9 mi$^2$ | 87 min (3 mi) | 5 |
| Chicago | 2,707,120 | 234 mi$^2$ | 141 sec (3 mi) | 9 |

API. Here all we need is a *coarse location* (at city level) to set these parameters. Android has permissions to control the access to both coarse and fine locations of a phone. Our zero-permission app can invoke the mobile browser to visit a malicious website, which can then search her IP in public IP-to-location databases [54] to find her city. This allows us to set the query parameters using Google Maps. Note that smartphone users tend to use Wi-Fi whenever possible to conserve their mobile data, which gives our app chances to get their coarse locations. Please note that we do not require the user to geo-tag each tweet. The twitter search results include the tweets in a area as long as the user specifies her geo-location in her profile.

As discussed before, our app can only sneak out the timestamps it collects from the Twitter app when the phone screen dims out. This could happen minutes away from the moment a user tweets. For each timestamp $t_i \in T$, we use the twitter API to search for the set of users $u_i$ who tweet in that area in $t_i \pm 60s$ (due to the time skew between mobile phone and the twitter server). The target user is in the set $U = \cap u_i$. When $U$ contains only one twitter ID, the user is identified. For a small city, oftentimes 1500 tweets returned by a query are more than enough to cover the delay including both the $t_i \pm 60s$ period and the duration between the tweet event and the moment the screen dims out. For a bigger city with a large population of Twitter users, however, we need to continuously query the Twitter server to dump the tweets to a local database, so when our app report a timestamp, we can search it in the database to find those who tweet at that moment.

**Attack evaluation.** We evaluate the effectiveness of this attack at three cities, Urbana, Bloomington and Chicago. Table 4.2 describes these cities' information.

We first study the lengths of the time intervals the 1500 tweets returned by a Twitter query can cover in these individual cities. To this end, we examine the difference between the first and the last timestamps on 1500 tweets downloaded from the Twitter server through a single API call, and present the results in Table 4.2. As we can see here, for small towns with populations below 100 thousand, all the tweets within one hour and a half can be retrieved through a single query, which is sufficient for our attack: it is conceivable that the victim's phone screen will dim out within that period after she tweets, allowing the malicious app to send out the timestamp through the browser. However, for Chicago, the query outcome only covers 2 minutes of tweets. Therefore, we need to continuously dump tweets from the Twitter server to a local database to make the attack work.

In this experiment, we run a script that repeatedly calls the Twitter Search API, at a rate of 135 queries per hour. All the results without duplicates are stored in a local SQL database. Then, we post tweets through the Twitter app on a smartphone, under the surveillance of the zero-permission app. After obvious robot Twitter accounts are eliminated from the query results, our Twitter ID are recovered by merely 3 timestamps at Urbana, 5 timestamps at Bloomington and 9 timestamps in Champaign, which is aligned with the city size and population (number of people).

## 4.4 Investment Inference

**Knowing your personal investment.** A person's investment information is private and highly sensitive. Here we demonstrate how an adversary can infer her financial interest from the network data usage of Yahoo! Finance, a popular finance app on Google Play with nearly one million users. We discover that Yahoo! Finance discloses a unique network data signature when the user is adding or clicking on a stock.

**Stock search autocomplete.** Similar to all aforementioned attacks, here we consider that a zero-permission app running in the background collects network data usage related to Yahoo! Finance and sends it to a remote attacker when the device's screen dims out. Searching for a stock in Yahoo! Finance generates a unique network data signature, which can be attributed to its network-based autocomplete feature (i.e. suggestion list) that returns suggested stocks according to the user's input. Consider for example the case when a user looks for Google's stock (GOOG). In response to each letter she enters, the Yahoo! Finance app continuously updates a list of possible autocomplete options from the Internet, which is characterized by a sequence of unique payload lengths. For example, typing "G" in the search box produces 281 bytes outgoing and 1361 to 2631 bytes incoming traffic. We find that each time the user enters an additional character, the outbound HTTP `GET` packet increases by one byte. In its HTTP response, a set of stocks related to the letters the user types will be returned, whose packet size depends on the user's input and is unique for each character combination.

**Stock news signature.** From the dynamics of mobile data usage produced by the suggestion lists, we can identify a set of candidate stocks. To narrow it down, we further study the signature when a stock code is clicked upon. We find that when this happens, two types of HTTP `GET` requests will be generated, one for a chart and the other for related news. The HTTP response for news has more salient features, which can be used to build a signature. Whenever a user clicks on a stock, Yahoo! Finance will refresh the news associated with that stock, which increases the `tcp_rcv` count. This count is then used to compare with the payload sizes of the HTTP packets for downloading stock news from Yahoo! so as to identify the stock chosen by the user. Also note that since the size of the HTTP `GET` for the news is stable, 352 bytes, our app can always determine when a news request is sent.

**Attack evaluation.** In our study, we run our zero-permission app to monitor the Yahoo! Finance app on a Nexus S 4G smartphone. From the data-usage statistics collected while the suggestion list is being used to add 10 random stocks onto the stock watch list, we manage to narrow down the candidate list to 85 possible stocks that match the data-usage features of these 10 stocks. Further analyzing the increment sequence when the user clicks on a particular stock code, which downloads related news to the phone, we are able to uniquely identify each of the ten stocks the user selects among the 85 candidates.

# CHAPTER 5

# MITIGATION TECHNIQUES

To address the security concerns as described in chapter 3 and 4, we proposes mitigation techniques in this chapter. Section 5.1 discusses the strategies for side-channel information leakage and its permission enforcement framework on Android. Section 5.2 proposes a mitigation strategy and introduces a static analysis framework, and also discusses compliance recommendations for mHealth apps.

## 5.1 Strategies for Side-channel Information Leakage

Given the various public resources on Android, the information leaks we found are very likely to be just a tip of the iceberg. Finding an effective solution to this problem is especially challenging with rich background information of users or apps gratuitously available on the web. To mitigate such threats, we first take a closer look at the attacks discovered in our research. To address the availability mechanism of the data usage statistics, which have already been used by hundreds of apps to help Android users keep track of their mobile data consumption, merely removing them from the list of public resources is not an option. In this section, we report our approach on mitigating the threat deriving from the statistics availability, while maintaining their utility.

### 5.1.1 Mitigation Strategies

To suppress information leaks from statistics available through `tcp_rcv` and `tcp_snd` as described in Chapter 4, we can release less accurate information. Here we analyze a few strategies designed for this purpose.

**Round up and round down.** One strategy is to reduce the accuracy of the available information by rounding up or down the actual number of bytes sent or received by an app to a multiple of a given integer before disclosing that value to the querying process. This approach is reminiscent of a predominant defense strategy against traffic analysis, namely packet padding [39, 55]. The difference between that and our approach is that we can not only round up but also round down to a target number and also work on accumulated payload lengths rather than the size of an individual packet. This enables us to control the information leaks at a low cost, in terms of impact on data utility.

Specifically, let $d$ be the content of a data usage counter (`tcp_rcv` or `tcp_snd`) and $\alpha$ an integer. When the counter is queried by an app, our approach first finds a number $k$ such that $k\alpha \leq d \leq (k+1)\alpha$ and reports $k\alpha$ to the app when $d - k\alpha < 0.5\alpha$ and $(k+1)\alpha$ otherwise.

**Aggregation.** A limitation of the simple rounding strategy results from the fact that it still gives away the payload size of each packet, even though the information is perturbed. As a result, it cannot hid packets with exceedingly large payloads. To address this issue, we can accumulate the data usage information of multiple queries, for example, stocks on Yahoo! Finance the user looks at, and only release the cumulative result when a time interval expires. This can be done, for example, by updating an app's data usage to the querying app once every week, which prevents the adversary from observing individual packets.

### 5.1.2 Enforcement Framework

To enforce the aforementioned policies, we design a preliminary framework, which is elaborated below.

A naive idea would be adding yet another permission to Android's already complex permission system and have any data monitoring app requesting this permission in AndroidManifest.xml. However, prior research shows that the users do not pay too much attention to the permission list when installing apps, and the developers tend to declare more permissions than needed [56]. On the other hand, the traffic usage data generated by some applications (e.g., banking applications) is exceptionally sensitive, at a degree that the app developer might not want to divulge them even to the legitimate data monitoring apps. To address this problem, our solution is to let an app specify ”permissions” to Android, which defines how its network usage statistics should be released. Such permissions, which are essentially a security policy, was built into the Android permission system in our research. Using this usage counters as an example, our framework supports four policies: *NO_ACCESS, ROUNDING, AGGREGATION* and *NO_PROTECTION*. These policies determine whether to release an app's usage data to a querying app, how to release this information and when to do that. They are enforced at a `UsageService`, a policy enforcement mechanism by holding back the answer and adding noise to it or periodically updating this information.

## 5.2 Strategies for Security Concerns in mHealth Apps

We have discussed several vulnerabilities discovered in Android mHealth apps in chapter 3 - *unsecured Internet, third party storage, Bluetooth, logging, SD card storage*, and *exported components*. Given the variety of existing Android mHealth apps and the number of upcoming new apps in the market, the vulnerabilities we found and discussed in chapter 3 is just a tip of the iceberg. Furthermore, even though applications in the markets were not designed to be malicious and were carefully programmed, they may still surfer from data leakage threats, for instance, when their source code contains unnecessary system logs that disclose sensitive information. Many developers are not able to fully understand the privacy implications, nor are they not be able to take fully control which data flows to which dangerous channel.

### 5.2.1 Static Analysis

To circumvent the data leakage threats in Android mHealth apps, we need a way to automatically assess and detect both newly discovered and existing known threats in the market. *Static taint analysis* addresses this problem by analyzing tainted data flows through Android applications and sending outputs to human analysts or to automated tools which can make security-decisions. This *static taint analysis* approach can keep track of sensitive “tainted” information through the application by starting at any one from a list of pre-defined sources (e.g., an API method returning users' contact list or a source labeled in source code) and then following the data flow until it reaches any one from a list of pre-defined sinks (e.g., an API method storing the information on a SD card). It gives precise information about which sensitive data may leak to which sink channel, as shown in figure 5.1. Many previous research works [57, 58, 59] work on the static taint analysis, while some work on *dynamic program analysis* such as TaintDroid [44]. Both *static analysis* and *dynamic analysis* can be used for this task. But *dynamic analysis* may require many test runs to reach appropriate code coverage. On the other side, malware can be developed to be able to recognize the behavior of *dynamic analysis* and pose as a benign application to bypass the detection. For the above reasons, we choose *static analysis* over *dynamic analysis* for data leakage detection.

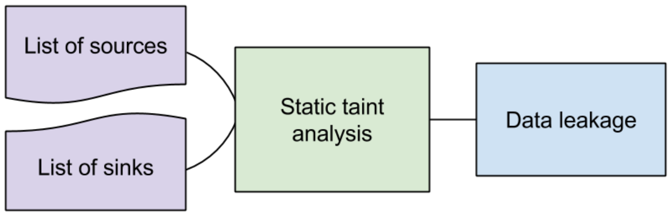


Figure 5.1: Data leakage detection with static analysis

Many challenges exist for implementing static analysis on Android due to the special design of Android operating system. Existing data-flow analysis techniques and modeling methods are not directly applicable to Android applications. Difficulties are triggered because of Android programming paradigm's special multiple entry points. Unlike a Java program, an Android app doesn't have a single entry point. Developers can define many *entry points* for a single Android app. As noted in section 2.1, there are four types of components developers can define: *Activity, Service, Content Provider* and *Broadcast Receiver.* The Android framework calls the methods associated with these components, to start, stop, pause, or to resume the components, depending on the environment needs. Data flow analysis can be expensive on Android apps because of its asynchronous execution and inter-component data flow. To be able to effectively predict the data flow, static analysis must not only precisely model the life-cycles of components but also integrate callbacks for system-event handling (e.g., for camera sensor), UI interaction and so on.

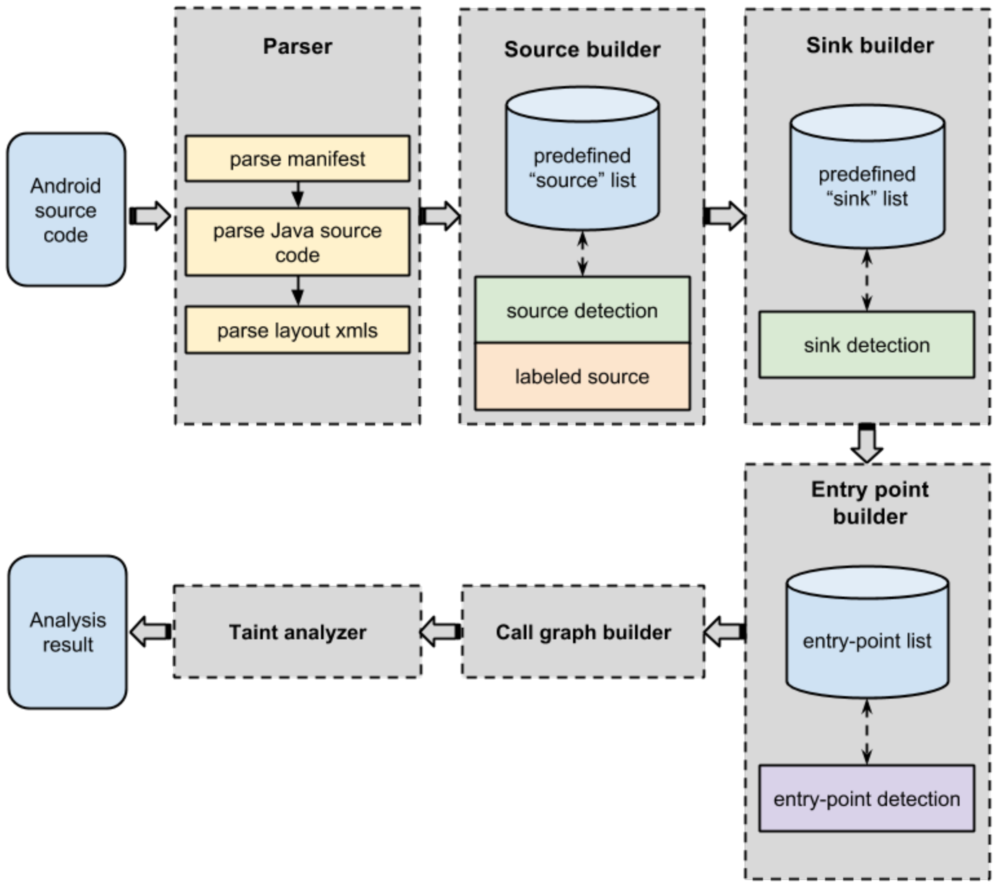


Figure 5.2: Static analysis system design framework

From our observation, many real-world vulnerabilities we discovered in chapter 3 have sources beyond the scope of FlowDroid's pre-defined source list, including account, Bluetooth, browser, calendar, contact, database, file, network, nfc, settings, sync, and unique-identifiers. The FlowDroid's sources are all returned from API methods, but for sensitive information (e.g., personal information like name, age and weight, or health information) that is from user input is hard to be categorized and captured with the tool. Neither false positive or false negative is acceptable in this scenario, since false positive may lead to too much inaccurate warnings and false negative can make the warnings incomplete. So we add a source builder into the architecture, with which the sources can be collected from labels in the source code or automated detection tool based on machine-learning technique [57]. Figure 5.2 shows our proposed framework for the static taint analysis tool on Android apps. This architecture is derived from FlowDroid [58], and we improve it by adding source builder into the architecture. After unzipping the app's app file, our tool searches through the application for lifecycle and call back methods by parsing various Android-specific files, including manifest, Java source code, layout xmls and so on. The source builder then constructs a list of sources from automated source detection tool built upon machine-learning technique [57] or from labels defined by developers in source code. Sinks are built with automated sink detection from pre-defined sink list. Next, the tool generates a dummy main method as a single entry point for the Android program from the list of lifecycle and callback methods. This main method is then used to generate a call graph for the taint analyzer. The taint analyzer reports any possible links between the sources and sinks as warnings of potential vulnerabilities to the developers for a further check.

### 5.2.2 Compliance Recommendations

The increased use of mHealth in various scenarios results in greater risks to health-related information on mobile devices. MHealth vendors including app developers and healthcare service providers should make efforts to ensure that mHealth apps facilitate security compliance. Based on our study on the risks from Android mHealth apps, here are some important compliance recommendations:

- Encryption is essential to secure personal data stored on mobile devices.
- When accessing web-based services for syncing users' sensitive data, TLS/SSL is necessary to be deployed throughout the Internet transmission session.
- Even though the network transmission session is protected and encrypted, using third party services to store users' sensitive data must be closely reviewed and users should be informed when it is happening.
- Developer guidelines or training can be helpful in avoiding many of the common mistakes that are rooted from development with poor secure practices.
- Risk assessment provided by authorities can further minimize the security risks that may harm users.

# CHAPTER 6

# DISCUSSION

In this chapter, we summarize the findings from the study of Android mHealth apps and the investigation on Android side channel exploits in section 6.1, and discuss the limitations of the study for these two studies respectively in section 6.2.

## 6.1 Summary of Findings

In this section, we summarize the findings from the studies in this thesis.

**Study of Android mHealth apps.** Our three-stage study in chapter 3 shows some serious problems with the Android mHealth apps. The major issue is unencrypted communication over the Internet and the use of third party services. Our study shows that a significant amount of Android mHealth apps on Google Play suffer from these issues. We believe that these issues need further attention from vendors, users and administration authorities, since these issues cannot be easily fixed. Many Android mHealth app developers are not security experts and lack of necessary security senses. It is not economical for the app vendors to maintain their own servers and it not clear how encryption could be used to store data on the third-party services. App vendors tend to use unencrypted communication and build services on third party hosts. For Android mHealth apps, more adequate security and privacy guarantees are in urgent demand.

**Understanding and proving data leakage from public resources on Android.** In chapter 4, we make a first step to understand the fundamental design deficiencies of Android: the Android operating system is designed based on a set of shared resources, which could be utilized by a malicious party to infer sensitive information. We discover an unexpected channel on Android: per-app data-usage statistics. Our study reveal that the per-app data-usage statistics channel can be a threat to user privacy by showing two attack instances - to infer a user's identity with Twitter app and public databases and to infer a user's investment interests with Yahoo! Finance app.

**Mitigation strategies for the side-channel data leakage.** The side-channel threat discussed in chapter 4 can hardly be mitigated, but it is still possible to circumvent this threat with Android version upgrades. As discussed in section 5.1, we add the Android kernel with additional permissions control for the data-usage statistics readings, such that developers can have the rights to define how the network usage statistics should be released for each app. We propose round up or round down and aggregation strategies and further discuss the enforcement framework for permissions control in Android kernel.

**Proposal of static analysis framework.** As the number of Android apps is exploding, many new security vulnerabilities emerge in the market. It is becoming hard for us to clearly define and detect all the data leakage threats for new channels and new data types, so we propose an automated static analysis technique in section 5.2. We add an additional source detection mechanism into an existing static analysis tool called FlowDroid and enable the tool to find sources that are labeled by developers.

## 6.2 Limitations

In this section, we discuss the limitations of our study in this thesis.

**App version upgrade.** In chapter 4, we describe a discovery of a new data leakage channel: per-app data-usage statistics and we give two instances of private inferences: identity inference from Twitter app and investment interest inference from Yahoo! Finance app. The success of data inferences is highly dependent on the accuracy of data-usage statistics from pre-analysis. If the target app has any version upgrade, the data-usage statistics may change. It will cause our data monitoring tool to be inaccurate or even to become unable to work. The only way to circumvent this problem is to create a variety of signature data entries for various different versions of target apps, so that even the version changes, we can still detect correctly the behavior with the data-usage signature.

**Android library behavior change.** Android has been making efforts by doing system behavior changes to circumvent newly found threats. Some of the security issues discussed in chapter 3 can be mitigated by Android version updates. For example, to mitigate the storage information leakage problem, starting from Android 4.4, the `WRITE_EXTERNAL_STORAGE` permission has been modified that it only allows apps to write on an external storage within its app-specific directory. To mitigate the logging information leakage problem, since Jelly Bean (Android 4.1), an app can only collect and view log messages originating from itself. However, on a rooted device (i.e., a device allows any app to run administration permissions on Android), a malicious app can, by executing a `pm grant` command, grant itself a `READ_LOGS` permission. This means that it is still dangerous for an app to keep sensitive information in the system logs. According to the Android platform distribution [60] collected in March, 2014, almost 40% of the overall Android devices are under the version of Jelly Bean. Due to a large number of Android devices users and mHealth apps, it is still highly lucrative for malicious to investigate ways to harvest sensitive personal healthcare information from mHealth apps.

**Side-channel information leakage mitigation strategies.** To mitigate the side-channel information leaks from data-usage stats as described in chapter 4, we propose data-usage stats mitigation strategies in chapter 5. The strategies are to round up, round down or aggregate statistics available through `tcp_rcv` and `tcp_snd` in order to suppress information leaks. However, there are some mobile apps on Android that are operating highly dependent on the statistics collected via these channels. For example, hundreds of data usage monitors are already out there [49], relying the usage statistics to keep track of a user's mobile data consumption. If we suppress the information leaks by making these statistics less accurate, it would affect the data monitor apps' current level of precision.

# CHAPTER 7

# CONCLUSION

In this master thesis, we explore the security threats in Android apps in two different directions: studying vulnerabilities in the mHealth sector and discussing a specific side-channel attack on Android. With a three-stage study on Android mHealth apps, we discover that many Android mHealth apps have issues of using unsecured Internet and third party hosting services. We present our compliance recommendations for these problems needing attention from vendors, users, and authorities. To mitigate the general data leakage problem in mHealth apps, we propose a static analysis framework, with a source detection mechanism, to enable developers labeling sensitive data that is easily to be missed by automated tools without human interaction. Our second study on the side-channel attack discovers an unexpected information leakage channel: per-app data-usage statistics. Our study reveals that highly sensitive user data, such as identity and investment interests, can be inferred from public resources by a malicious app with zero permission on Android devices. Our findings call into question the design assumptions made by Android developers on public resources and demand new efforts to address such privacy risks. To this end, we further propose a preliminary design for mitigating the threats from the selected public resources, while preserving their utility.

# REFERENCES

[1] "Antutu benchmark," https://play.google.com/store/apps/details?id=com.antutu.ABenchMark, 2014, [Online; accessed 3-April-2014].

[2] "Comparison of mobile operating systems," http://en.wikipedia.org/wiki/Comparison_of_mobile_operating_systems, 2014, [Online; accessed 16-April-2014].

[3] "Google announces android has surpassed 1 billion device activations," http://phandroid.com/2013/09/03/android-device-activations-1-billion/, Sept. 2014, [Online; accessed 16-April-2014].

[4] "Number of available android applications," https://www.appbrain.com/stats/number-of-android-apps, AppBrain, April 2014, [Online; accessed 16-April-2014].

[5] B. Singer, "Fbi issues android smartphone malware warning," http://www.forbes.com/sites/billsinger/2012/10/15/fbi-issues-android-smartphone-malware-warning/, October 2012, [Online; accessed 6-April-2014].

[6] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.16 pp. 95–109.

[7] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[8] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *NDSS*, 2012.

[9] P. Brodeur, "Zero-permission android applications," http://www.leviathansecurity.com/blog/zero-permission-android-applications/, 2012, [Online; accessed 2-April-2014].

[10] S. Avancha, A. Baxi, and D. Kotz, "Privacy in mobile technology for personal healthcare," *ACM Computing Surveys*, vol. 45, no. 1, November 2012. [Online]. Available: http://www.cs.dartmouth.edu/ dfk/papers/avancha-survey.pdf

[11] D. Kotz, "A threat taxonomy for mhealth privacy," in *COMSNETS*, 2011, pp. 1–6.

[12] "Mobile health market report 2013-2017," http://www.research2guidance.com/shop/index.php/mhealth-report-2, Research2Guidance, March 2013, [Online; accessed 6-April-2014].

[13] "Mobile health apps & solutions market by connected devices (cardiac monitoring, diabetes management devices), health apps (exercise, weight loss, women's health, sleep and medication), medical apps (medical reference) - global trends & forecast to 2018," MarketsandMarkets, Sep 2013.

[14] R. Murthy and D. Kotz, "Assessing blood-pressure measurement in tablet-based mHealth apps," in *Workshop on Networked Healthcare Technology (NetHealth)*. IEEE Press, January 2014, accepted for publication. [Online]. Available: http://www.cs.dartmouth.edu/ dfk/papers/murthy-bp.pdf

[15] R. Istepanian, S. Laxminarayan, and C. S. Pattichis, "M-health: emerging mobile health systems," *M-Health: Emerging Mobile Health Systems, Edited by R. Istepanian, S. Laxminarayan, and CS Pattichis. 2006 XXX, 624 p. 182 illus. 0-387-26558-9. Berlin: Springer, 2006.*, vol. 1, 2006.

[16] Y. Anokwa, N. Ribeka, T. Parikh, G. Borriello, and M. C. Were, "Design of a phone-based clinical decision support system for resource-limited settings," in *Proceedings of the Fifth International Conference on Information and Communication Technologies and Development*, ser. ICTD '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2160673.2160676 pp. 13–24.

[17] D. Kotz, S. Avancha, and A. Baxi, "A privacy framework for mobile health and home-care systems," in *Workshop on Security and Privacy in Medical and Home-Care Systems (SPIMACS)*. ACM Press, November 2009. [Online]. Available: http://www.cs.dartmouth.edu/ dfk/papers/kotz-mhealth-spimacs.pdf pp. 1–12.

[18] C. C. Poon, Y.-T. Zhang, and S.-D. Bao, "A novel biometrics method to secure wireless body area sensor networks for telemedicine and m-health," *Comm. Mag.*, vol. 44, no. 4, pp. 73–81, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/MCOM.2006.1632652

[19] "Mobile medical applications guidance for industry and food and drug administration staff," US. Food and Drug Administration, Sept. 2013.

[20] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside job: Understanding and mitigating the threat of external device mis-bonding on android," 2014.

[21] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018 pp. 239–252.

[22] "Mobile threat report," http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf, F-Secure Labs, Helsinki, Finland, 2013, [Online; accessed 5-April-2014].

[23] E. C. Yekaterina Tsipenyuk O'Neil, "Seven ways to hang yourself with google android," http://www.cs.berkeley.edu/ emc/slides/SevenWaysToHangYourselfWithGoogleAndroid.pdf, 2011, [Online; accessed 5-April-2014].

[24] "Doctor online," https://play.google.com/store/apps/details?id=com.airpersons.airpersonsmobilehealth, 2014, [Online; accessed 5-April-2014].

[25] "Recipes by ingredients," https://play.google.com/store/apps/details?id=com.abMobile.recipebyingredient, 2014, [Online; accessed 5-April-2014].

[26] "Cvs/pharmacy," https://play.google.com/store/apps/details?id=com.cvs.launchers.cvs, 2014, [Online; accessed 5-April-2014].

[27] "Noom weight loss coach," https://play.google.com/store/apps/details?id=com.wsl.noom, 2014, [Online; accessed 5-April-2014].

[28] "Drozer," https://www.mwrinfosecurity.com/products/drozer/, 2014, [Online; accessed 5-April-2014].

[29] "Snoreclock," https://play.google.com/store/apps/details?id=de.ralphsapps.snorecontrol, 2014, [Online; accessed 5-April-2014].

[30] "Sleep talk recorder," https://play.google.com/store/apps/details?id=com.madinsweden.sleeptalk, 2014, [Online; accessed 5-April-2014].

[31] "Urgent care," https://play.google.com/store/apps/details?id=com.greatcall.urgentcare, 2014, [Online; accessed 5-April-2014].

[32] "Dex2jar," https://code.google.com/p/dex2jar/, 2014, [Online; accessed 5-April-2014].

[33] P. Dixon, "Medical identity theft: The information crime that can kill you," http://www.worldprivacyforum.org/wp-content/uploads/2007/11/wpf_medicalidtheft2006.pdf, World Privacy Forum, 2006, [Online; accessed 5-April-2014].

[34] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687 pp. 199–212.

[35] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382230 pp. 305–316.

[36] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson, "Uncovering spoken phrases in encrypted voice over ip conversations," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 4, pp. 35:1–35:30, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1880022.1880029

[37] K. Zhang and X. Wang, "Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855770 pp. 17–32.

[38] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.19 pp. 143–157.

[39] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.20 pp. 191–206.

[40] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *NDSS*, 2011.

[41] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion," in *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, ser. HotSec'11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028040.2028049 pp. 9–9.

[42] L. Cai and H. Chen, "On the practicality of motion based keystroke inference attack," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, ser. TRUST'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 273–290.

[43] J. Han, E. Owusu, L. T. Nguyen, A. Perrig, and J. Zhang, "Accomplice: Location inference using accelerometers on smartphones." in *COMSNETS*, K. K. Ramakrishnan, R. Shorey, and D. F. Towsley, Eds. IEEE, 2012, pp. 1–9.

[44] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. 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*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971 pp. 1–6.

[45] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028088 pp. 21–21.

[46] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. 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*, ser. CCS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046780 pp. 639–652.

[47] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2184489.2184500 pp. 49–54.

[48] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653691 pp. 235–245.

[49] "Google play," https://play.google.com/store/search?q=traffic+monitor&c=apps, 2014, [Online; accessed 2-April-2014].

[50] D. J. Solove, "Identity theft, privacy, and the architecture of vulnerability," *Hastings Law Journal*, vol. 54, p. 1227, 2003.

[51] J. Camenisch, a. shelat, D. Sommer, S. Fischer-Hübner, M. Hansen, H. Krasemann, G. Lacoste, R. Leenes, and J. Tseng, "Privacy and identity management for everyone," in *Proceedings of the 2005 Workshop on Digital Identity Management*, ser. DIM '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1102486.1102491 pp. 20–27.

[52] T. Govani and H. Pashley, "Student awareness of the privacy implications when using facebook," *Draft*, Jan. 2005. [Online]. Available: http://lorrie.cranor.org/courses/fa05/tubzhlp.pdf

[53] "Get search, twitter api," https://dev.twitter.com/docs/api/1/get/search, 2013, [Online; accessed 4-April-2014].

[54] "Ip address lookup," http://whatismyipaddress.com/ip-lookup, 2014, [Online; accessed 4-April-2014].

[55] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu, "Statistical identification of encrypted web browsing traffic," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, ser. SP '02. Washington, DC, USA: IEEE Computer Society, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=829514.830535 pp. 19–.

[56] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779 pp. 627–638.

[57] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[58] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” 2014.

[59] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “Chex: Statically vetting android apps for component hijacking vulnerabilities,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382223 pp. 229–240.

[60] “Android historical version distribution,” https://developer.android.com/about/dashboards/index.html, 2014, [Online; accessed March-2014].