ANALYZING & DESIGNING THE SECURITY OF SHARED RESOURCES ON SMARTPHONE OPERATING SYSTEMS

BY

SOTERIS DEMETRIOU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

ABSTRACT

Smartphone penetration surpassed 80% in the US and nears 70% in Western Europe. In fact, smartphones became the de facto devices users leverage to manage personal information and access external data and other connected devices on a daily basis. To support such multi-faceted functionality, smartphones are designed with a multi-process architecture, which enables third-party developers to build smartphone applications which can utilize smartphone internal and external resources to offer creative utility to users. Unfortunately, such third-party programs can exploit security inefficiencies in smartphone operating systems to gain unauthorized access to available resources, compromising the confidentiality of rich, highly sensitive user data.

The smartphone ecosystem, is designed such that users can readily install and replace applications on their smarpthones. This facilitates users' efforts in customizing the capabilities of their smartphones tailored to their needs. Statistics report an increasing number of available smartphone applications—in 2017 there were approximately 3.5 million third-party apps on the offifial application store of the most popular smartphone platform. In addition we expect users to have approximately 95 such applications installed on their smartphones at any given point. However, mobile apps are developed by untrusted sources. On Android—which enjoys 80% of the smarpthone OS marketshare—application developers are identified based on self-sign certificates. Thus there is no good way of holding a developer accountable for a malicious behavior. This creates an issue of *multi-tenancy* on smartphones where principals from diverse untrusted sources share internal and external smartphone resources. Smartphone OSs rely on traditional operating system process isolation strategies to confine untrusted third-party applications. However this approach is insufficient because incidental seemingly harmless resources can be utilized by untrusted tenants as side-channels to bypass the process boundaries. Smartphones also introduced a permission model to allow their users to govern third-party application access to system resources (such as camera, microphone and location functionality). However, this permission model is both coarse-grained and does not distinguish whether a permission has been declared by a trusted or an untrusted principal. This allows malicious applications to perform privilege escalation attacks on the mobile platform. To make things worse, applications might include third-party libraries, for advertising or common recognition tasks. Such libraries share the process address space with their host apps and as such can inherit all the privileges the host app does. Identifying and mitigating these problems on smartphones is not a trivial process. Manual analysis on its own of all mobile apps is cumbersome and impractical, code analysis techniques suffer from scalability and coverage issues, ad-hoc approaches are impractical and sucseptible to mistakes, while sometimes vulnerabilities are well hidden at the interplays between smartphone tenants and resources.

In this work I follow an analytical approach to discover major security and privacy issues on smartphone platforms. I utilize the Android OS as a use case, because of its open-source nature but also its popularity. In particular I focus on the multi-tenancy characteristic of smartphones and identify the resources each tenant within a process, across processes and across devices can access. I design analytical tools to automate the discovery process, attacks to better understand the adversary models, and introduce design changes to the participating systems to enable robust fine-grained access control of resources. My approach revealed a new understanding of the threats introduced from third-party libraries within an application process; it revealed new capabilities of the mobile application adversary exploiting shared filesystem and permission resources; and shows how a mobile app adversary can exploit shared communication mediums to compromise the confidentiality of the data collected by external devices (e.g. fitness and medical accessories, NFC tags etc.). Moreover, I show how we can eradicate these problems following an architectural design approach to introduce backward-compatible, effective and efficient modifications in operating systems to achieve fine-grained application access to shared resources. My work has let to security changes in the official release of Android by Google.

To my parents Yiannis and Maria, my brothers Michael and Alexandros, my sister-in-law Georgia and my niece Florentia for their unconditional love and support.

ACKNOWLEDGMENTS

This thesis would have not been possible to realize without the invaluable support and guidance of my advisor and academic father Dr Carl A. Gunter. Professor Gunter trusted me to explore issues that I find interesting without any constraints. I consider myself privileged to be given the opportunity to work and learn from Dr. Gunter. I can not imagine having a better advisor and mentor.

I would like to also thank Dr Xiaofeng Wang, whose indispensable guidance was continuous throughout my work. I was lucky during the beginning of my studies to meet with Professor Wang who got me involved in smartphone security, a topic which became the focus of my thesis. Unequivocally I learnt a lot during my interactions with Professor Wang both on how to find interesting problems and how to approach solving them.

Special thanks to Dr Klara Nahrstedt and Adam M. Bates for their feedback and guidance. The final version of this thesis is shaped based on their astute observations.

My sincere thanks also goes to Dr. Michael Grace and Dr. Xiaoyong Zhou who provided me with an opportunity to join their team as intern at Samsung Research America (SRA). At SRA I got exposed to real problems modern smartphone operating system vendors face and the approaches they take to tackle them.

Lastly, I would like to thank all the people I got to work with during my Ph.D. studies: Dr. Carl A. Gunter; Dr. XiaoFeng Wang; Dr. Klara Nahrstedt; Dr. Kyu-Han Kim; Dr. Landon Cox; Dr. Patrick Tague; Dr. Xiaoyong Zhou; Dr. Muhammad Naveed; Dr. Michael Grace; Dr. Puneet Jain; Dr. Wei Yang; Dr. Aston Zhang; Dr. Yueh-Hsun Lin; Dr. Yuan Tian; Dr. Animesh Shrivastava; Dr. Wenrui Diao; Dr. Feng Qian; Dr. Kai Chen; Dr. Xinhui Han; Dr. Kehuan Zhang; Dongjing He; Xiaorui Pan; Yeonjoon Lee; Kan Yuan; Whitney Merrill; Gliz Seray Tuncay; Tongxin Li; Mingming Zha; Nan Zhang; Xianghang Mi; Peiyuan Zong; Karan Ganju. My interactions with all of them shaped me both as an academic and as a person.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Motivation 1
1.2 Problem Statement 3
1.3 Approach 4
1.4 Thesis Contributions 8
1.5 Thesis Organisation 10

CHAPTER 2 BACKGROUND 11
2.1 Android OS 11
2.2 Android Security Model 15
2.3 Backround on Techniques and Methodologies Used 23
2.4 Android's Shared Resources 24

CHAPTER 3 LITERATURE REVIEW 28
3.1 Advertising Libraries 28
3.2 Information Leaks through Filesystem Resources 29
3.3 IPC, System Resources and Shared Application Components 30
3.4 Shared Communication Channels 31
3.5 Shared IoT Devices 32

CHAPTER 4 SHARING PROCESS PRIVILEGES 34
4.1 Introduction 34
4.2 Analysis 36
4.3 Detection Design 46
4.4 Detection Evaluation 55
4.5 Utility and Limitations 62
4.6 Summary 65

CHAPTER 5 SHARING FILESYSTEM RESOURCES 67
5.1 Introduction 67
5.2 Analysis 68
5.3 Mitigation Design 87
5.4 Mitigation Evaluation 89
5.5 Summary 90

CHAPTER 6 SHARING SYSTEM AND APPLICATION RESOURCES 92
6.1 Introduction 92
6.2 Analysis 93
6.3 Mitigation Design 97
6.4 Mitigation Evaluation 102

6.5 Mitigation Formal Verification 106
6.6 Summary 109

CHAPTER 7 SHARING DIRECT COMMUNICATION CHANNELS 111
7.1 Introduction 111
7.2 Analysis 112
7.3 Mitigation Design 128
7.4 Mitigation Evaluation 141
7.5 Summary 144

CHAPTER 8 SHARING DEVICES IN IoT ENVIRONMENTS 146
8.1 Introduction 146
8.2 Analysis 149
8.3 Mitigation Design 153
8.4 Mitigation Evaluation 164
8.5 Discussion 170
8.6 Summary 172

CHAPTER 9 DESIGN PRINCIPLES 174
9.1 Contextual Threat Model 176
9.2 Granularity of Mechanism 177
9.3 Layered Responsibility 179

CHAPTER 10 SUMMARY AND FUTURE DIRECTIONS 182
10.1 Summary 182
10.2 Contributions 184
10.3 Moving Forward 186

REFERENCES 187

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION

Eleven years now, after the first iOS and Android enabled smartphones, the technology behemoths are now responsible for 90% [1] of total smartphone sales in 2018. These devices have revolutionized the way people communicate and manage personal and business tasks. Their unprecedented nature, which combines mobility, computational power and a model of easy to replace applications that can facilitate every facet of our everyday lives, constitute them an integral tool for people of any age. This very model, designed to leverage developers' creativity to provide users with a menagerie of smartphone applications (apps for short) of any perceived purpose, led to the release of an astounding number of apps in official application markets. Statista reports an almost exponential increase in the number of available smartphone apps on the official application store for Android devices, with a recent gnaw-dropping recorded number of 3.5 million apps [2]. These apps cover a broad spectrum of functionality: applications for entertainment purposes, like games for children and for adults; apps for educational purposes that can be used at schools and at home; apps that render managing financial investments trivial; apps that help people manage their time and tasks; office apps, data management apps; even apps for medical purposes, facilitating decision making for doctors, or helping patients manage their treatment or daily activities to improve quality of life; and recently apps to control and interface with Internet of Things (IoT) devices, such as connected motion sensors and cameras.

Increasingly, smarpthone apps incorporate third-party libraries for two main reasons: (a) for advertising; (b) for utility. For example, Grace et.al found that approximately 50% out of 100,000 collected apps use in-app advertising libraries (ad libraries for short) for monetization [3]. They further found that a third of them integrate one ad library while one of the apps they analyzed includes 20 such libraries. In our work, we further found that 33.3% out of 230,000 collected apps that request the camera permission, include at least one third-party library [4]. These third-party libraries are used for a variety of purposes such as, location services, character encoding, audio encoding/decoding, text recognition, credit card scanning and computer vision support among others. This highlighs the fact that including third-party libraries is a common practice on the smarpthone ecosystem. These libraries, interface with their host apps through APIs which the hosts utilize to pass information to the libraries for their tasks.

In turn, the host applications utilize operating system resources and capabilities for their own functionalities. Smartphone operating systems offer a rich API to application developers for accessing user data such as a user's contact list, incoming SMSs or collaborate with other applicaitons on the system. Smartphone APIs also allow processes to leverage a smartphone's advanced sensing and communication capabilities. These capabilities allow contemporary phones to receive and transmit information from and to accessories, remote servers and devices: Bluetooth is being utilized to allow smartphone users to manage their medical conditions, keep track of their fitness progress and communicate with other Bluetooth enabled phones; NFC made credit card payments fast and seamless and can automate repetitive tasks through the tap of the phone on an NFC tag; smartphone audio jacks can be used again for monetary transactions [5] or for receiving sensitive information from accessories regarding its user's body functions; SMS can be more than a message exchange between users as it can be used for sensitive tasks i.e 2-factor authentication; also the ability to connect through WiFi with remote domains allow app developers to offer mobile advertising for monetization or interface with IoT devices in a home area network or across the world.

Of particular interest, is the Android OS which dominates the smartphone marketshare [1]. Its open source nature led to the adoption of Google's proud green robot by the vast majority of hardware vendors, offering Android enabled devices for everyone, regardless of their financial capabilities. Android smartphones are available from \$40 to \$800 with a variety of different specifications. Flagship Android phones and tablets, now feature quad core processors, 4GB of RAM, in par with modern laptops and notebooks. The computational power of those devices, in tandem with their ubiquitous, always-present nature and its current penetration has dictated the use of Android smartphones for personal, business and medical purposes.

This vast adoption of Android, created an equally large attack surface for malicious applications aiming to infringe users' privacy. Unequivocally, investment in malware makes more sense when a security vulnerability or breach affects a wide user base and Android is the ideal candidate for doing just that. As malware targeting Android increases, we have witnessed a large scale of malicious attempts [6] exploiting the system's vulnerability to gain root access, or charging users money, by sending SMSs or calling premium numbers [7]. Furthermore the scientific community delineated another spectrum of the popular system's vulnerabilities, using more sophisticated attacks such as permission re-delegation [8] and capability leaks [9]. Lastly, in a data-driven world, multi-billion dollar industries such as analytics and advertising, rely on building detail user profiles, which in essence incentivizes them as well to follow aggressive data harvesting practices, which are not always in accord with users expectations.

Android marketshare, penetration, use in sensitive settings and the fact that is being targeted by the vast majority of smartphone malware, analytics and advertising networks, highlight the significance of an analytical approach for studying the security of the platform but also the need for designing backward compatible, efficient and effective defense mechanisms that detect questionable data harvesting behaviors and prevent malicious ones. For the rest of this thesis I will be using Android as a use case of a modern smartphone operating system.

1.2 PROBLEM STATEMENT

The Android OS uses various security strategies to control userspace process access to sensitive resources (Android apps run as userspace processes). Android leverages a Discretionary Access Control (DAC) scheme to isolate processes and their data. Moreover, it recently introduced a Mandatory Access Control (MAC) scheme to further strongly isolate system processes from third-party applications. Both are enforced at process level in the kernel layer. On top of the kernel, lies the Android middleware which leverages a permission model to govern process access to system and application resources. For example, when an app wants to utilize the sensitive system API to record audio, it needs to first get granted the *MICROPHONE* permission by the user of the system.

Unfortunately, these mechanisms are inadequate to guarantee the confidentiality of sensitive information. All strategies are applied at the process granularity and as such fail to identify threats from libraries which **share the same privileges** with their host processes [10, 4]. The DAC scheme which protects filesystem resources is adopted from a static environment. However, **filesystem resources** shared across processes in a stationary machine sometimes require different access control management when used on a mobile platform [11]. The MAC scheme only focuses on isolating system processes from third-party apps allowing attacks between apps of the latter kind. Moreover, the Android permission model does not distinguish between system permissions defined by the system (a privileged principal) and custom permissions defined by untrusted third-party apps (unprivileged principals) to protect their **shared application components**, which allows compromising the confidentiality of those components [12]. Moreover, this permission model mostly depends on user input when protecting access to communication channels and as such it needs to strike a balance between usability and security granularity. In essense, this model is both too coarse-grained and users are desensitized to permission prompts. This further allows third-party apps to gain unfetterd access to **shared direct communication resources** such as the Bluetooth, NFC and Audio channels [13, 14] and **shared indirect communication channels** such as WiFi devices on the same network [15]. This allows malicious apps to stealthily attack other devices that can connect to the smartphone.

Previous works have been taking ad-hoc and impractical approaches to mitigate these issues. Some works simply propose more permissions for controlling access to sensitive resources at a finer-granularity [16]. However these introduce a permission overbloat problem and exacerbate the desencitization of users to the permission model. Other works considered spliting libraries from their host apps so we can utilize process level isolation [17]; these are both inefficient and impractical as they break the business model of advertising networks, a role increasingly assumed by smartphone vendors as well. Other works focus only on code analysis for prevention which is not scalable and slow to utilize at runtime [3, 18, 19, 20, 21, 22, 23, 24, 25, 26] while other researchers propose ad-hoc approaches for quickly patching found vulnerabilities; these are hard to maintain and usually do not address the root causes of the security problems.

Instead, I postulate that we need an analytical approach to systematically reason about the security challenges on smartphones. Such an approach will allow us to better model the smartphone adversary and thus design unified, efficient, scalabe and robust systems to eradicate security problems.

1.3 APPROACH

There is no one-size-fits-all approach to securing smartphone systems. However, thinking about the resources we need to protect and the principals that try to—or have an incentive to—access those resources, greatly facilitates the process. Specifically, in my work I focus on how different kinds of resources are shared between tenants at different granularities on a smartphone platform to unearth security and privacy vulnerabilities. I then utilize these results to drive design desicions in building tools and systems for detection and prevention of information leakage through such shared resources. My solutions aim to satisfy the following important design properties: effectiveness; efficiency; backward compatibility and; maintainability.

In particular, in my work I systematically analyzed five classes of shared resources and found how tenants at different granularities can exploit them to compromise sensitive user information. These classes are: (a) shared process privileges; (b) shared filesystem resources; (c) shared system and application resources; (d) shared indirect communication channels

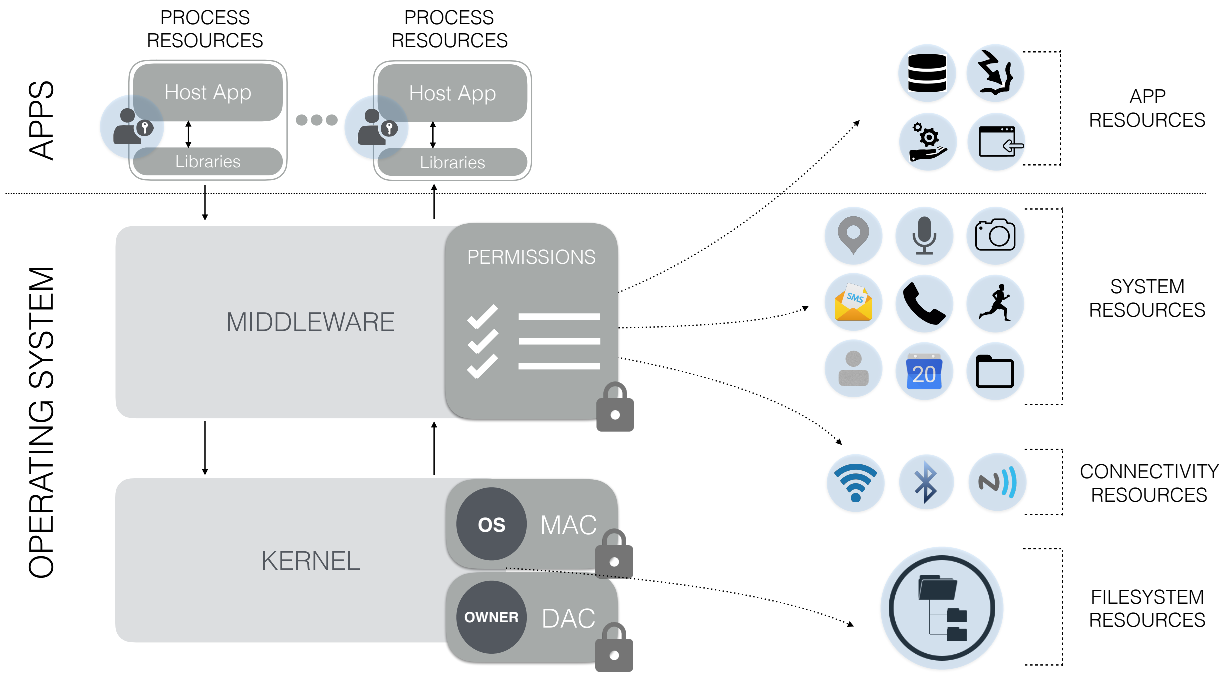


Figure 1.1: Smartphone shared resources.

and; (e) shared direct communication channels (d and e are depicted as connectivity resources). Figure 1.1 summarizes the approach at a conceptual level.

First, executable code from different untrusted sources, might run within the same process boundaries. Therefore they share privileges at the process level. For example, Android apps are commonly distributed for free. In turn they offer **advertising** which can result in monetizing user impressions and clicks among others. To achieve this return of investement the app developers include ad libraries into their source code which are compiled with the host application. As a result the advertising code runs within the same process as the host app. This symbiotic relationship comes with an intrinsic sharing of privileges: a third-party library can exploit that to access application data and platform resources. Other studies have focused on the ad libraries' current behaviors. Such approaches are limited since they cannot predict future behaviors. In contrast, we model all the different ways an ad library can access user data on the Android platform. Since in this case, the OS cannot make a decision whether advertising data collection constitutes a malicious behavior, I instead designed a detection system, called *Pluto* [10], which can be utilized by application markets to quantify the risk associated with embedding an ad library into an app. *Pluto* leverages Android OS domain knowledge combined with natural language processing and frequent pattern mining techniques to automatically detect sensitive user information that can be inferred by an ad library due to its vantage placement in a target app. Pluto performs a risk assessment and provides a privacy-leakage risk score associated with embedding an advertising library in a mobile app.

Second, I found that Android suffers from information leaks stemming from unprotected filesystem resources. The protection of such resources on Android, is delegated to the traditional Linux Discretionary Access Control, where a user or a group of users is granted a combination of the `read`, `write` and `execute` permissions. However, information seemingly innocuous on a stationary machine, that is made available to any process, can have grave privacy implications when used on a smartphone (or any multi-tenant mobile) platform. Therefore, if some of those resources are transferred from Linux to Android without the proper access control modifications, then private information leaks are a pragmatic and imminent threat. Indeed I found that an adversarial smartphone app can utilize such shared filesystem resources as side-channels to infer a smartphone user's identity, medical condition and financial preferences. Since then a lot of other works followed, utilizing other shared filesystem resources as side-channels. Google on version 6, introduced restrictions to third-party application access to such resources.

Third, smartphone applications can share their application components with other applications on the same platform. The Android OS provides application developers the a security mechanism to protect such components from unauthorized accesses. Specifically, third-party app developers can declare their own *custom* permissions in the system. However the OS does not distinguish at runtime between a permission declared by the system and a custom permission declared by an untrusted third-party app. Moreover, it does not have a way of tracking ownership of custom permissions. In essence custom permissions are identified by their developer-declared name which can be claimed on a first-come-first-served basis. This allows malicious third-party apps to manipulate their component-sharing security mechanism to elevate their privileges and gain unauthorized access to other tenants components but also system resources such as the calendar, camera, contacts, location, microphone, phone, sensors, SMS and storage. To mitigate this I introduced a new version of the Android permission model which separates management of system-defined and custom permissions, can identify at runtime whether a permission is system or custom and can further track ownership of custom permissions. The new permission model is designed to be backward compatible with third-party apps, is efficient, and formally verified to be correct with fundamental security properties.

Four, Bluetooth, NFC, Audio, and SMS constitute shared channels of direct communication between a smartphone and a remote or external source. Here we use external and remote interchangeably as remote sources are indeed external resources for the mobile OS on smartphones. Since these channels carry private information most of the times, Android OS developers correctly protected access to those channels with permissions. However, not only permissions are being neglected or granted without scrutiny from users [27] but even if users bestow the appropriate attention, this work argues that they are very coarse-grained to protect the resources they guard. Consider for example an app that requires the Bluetooth permission to supposedly connect to an accessory. Once the permission is granted, that app gains unfettered access to the Bluetooth channel irrespective of the accessory currently connected to the phone. Similarly, an app with the NFC permission can access any NFC device in vicinity. An app with the AUDIO permission cannot only be used to support a speaker but can read data transmitted to a cable-connected fitness accessory [28]. In my work, I extended the Android MAC scheme and introduced a flexible DAC scheme to allow both enterprise administrators (admins for short) and users to construct rules to control at application-level how these shared direct communication channels can be accessed. Such control allows a messaging app to read all SMSs except from those that are protected, such as an SMS from Chase which can be configured to be read only by the Chase Bank app. It will also allow an app to talk to its Bluetooth headset but restrict it from talking to a protected Bluetooth blood glucose meter and so on.

Lastly, smartphones are increasingly used to connect to WiFi smart-home devices. These IoT devices are typically located behind a home area network router and are controlled through smartphone apps. A lot of these systems tend to rely on the Wi-Fi router to authenticate other devices [15] or suffer from common vulnerabilities devices such as hardcoded credentials, weak or no authentication [29]. This treatment exposes them to attacks from malicious smartphone apps, particularly those running on authorized smartphones, which the router does not have information to control. Mitigating this threat cannot solely rely on IoT manufacturers, which may need to change the hardware on the devices to support encryption, increasing the cost of the device, or software developers who we need to trust to implement security correctly. We could tackle such attacks at the smartphone OS with stronger access control such as in the case of direct communication channels. However, this would entail assuming that not only the owners, but also all guests of the household, or an adversary that compromised the WiFi passphrase use our proposed smarpthone OS. Since these devices are shared by the router, a more practical approach would be to built our defense there. To tackle this problem I built a system which uses an approach inspired by software-defined networking (SDN) (see [30] for a survey) to offer fine-grained protection: each phone runs a non-system userspace Monitor app to identify the party that attempts to access the protected IoT device and inform the router through a control plane of its access decision; the router enforces the decision on the data plane after verifying whether the phone should be allowed to talk to the device.

Focusing on analyzing shared resources within a process, across processes and across devices, allows us to discover new adversarial capabilities on smartphones and in environments where smartphones are introduced (e.g. IoT). This facilitates better modeling of the smartphone adversary which leads to the design of robust systems for both detection and prevention of malicious behaviors by untrusted userspace smartphone programs.

1.4 THESIS CONTRIBUTIONS

This thesis makes significant contributions in the analysis of smartphone adversary models and the design of tools and system enhancements for detecting suspicious behaviors and preventing malicious behaviors on the smartphone ecosystem.

• *Provides a systematic analysis of information reach of advertising libraries embedded in smartphone apps.* Previous work has focused on past and current behaviors of advertising libraries, overlooking the fact that these behaviors can change opportunistically. This thesis focuses on the fact that such libraries share process space and privileges with their hosts and as such can eventually take advantage of those privileges. It systematically models all shared privileges a library has with its hosts which leverages for the design of an automatic open-source tool for estimating the risk of sensitive user information exposure by a host app to its advertising library.

• *Discovers new side-channels hidden in shared filesystem resources and demonstrates new adversarial inference techniques.* An analysis of Android shared filesystem resources led to the discovery of new side-channels which can be exploited by malicious applications with a suite of new inference techniques to bypass the process isolation boundaries and infer a user's identity, medical condition and financial preferences. This work led to Google introducing further restrictions on Android filesystem resource access by third-party apps.

• *Redesigns the Android Runtime Permission Model to tackle attacks on shared system and application components.* This thesis introduces a redesign of the Android permission model which (a) separates management of system-defined and third-party-defined permissions and (b) tracks ownership of custom permissions. This solves a perennial problem on Android where vulnerabilities kept arising because of this non separation of trust between permissions. The proposed model is backward compatible and formally verified to be correct with respect to fundamental security properties.

• *Unearths threats on Android's communication with external resources.* This work systematically studies Android's shared channels of communication with external resources such as Bluetooth and NFC devices, devices that connect through the Audio port, incoming SMSs and, WiFi smart-home devices. It defines a new threat, called the device mis-bonding (DMB) problem, to highlight the system's incapacity to create application-level bonds. It further demonstrates that Android's system permissions are too coarse-grained to support the utility of the apps while guaranteeing the confidentiality and intergrity of the data communicated through these channels. Furthermore it measures the prevalence of the problem in the Android ecosystem.

• *Introduces smartphone OS-level enhancements to safeguard the communication with Android external resources, using both MAC and DAC.* This is the first mechanism that provides comprehensive protection of different kinds of Android external resources over their channels in a uniform way. The enhancements are built on top of SELinux on Android and achieve both MAC and DAC in an integrated, highly efficient way, without undermining their security guarantees. These new techniques help both system administrators and ordinary Android users to specify their policies and safeguard their accessories and other external resources.

• *Introduces a novel distributed application-level access control system to safeguard vulnerable shared smart-home devices from malicious smartphone apps.* It shows how smartphones can collaborate with enforcing points in smart environments to enable fine-grained access control for WiFi smart-home devices. The design focuses on OS-level enahancement at the enforcing point (the router) which makes device-level decisions and utilizes trusted applications on smartphones for application-level decisions. The trusted applications utilize novel traffic monitoring techniques while the overall solution is independent from IoT device manufacturers.

• *Impact on real-world smartphone operating systems.* Threats revealed in this work were acknowledged by Google, who overhauls the development of Android (the most popular smartphone operating system) which is used by millions of users. Google introduced security enhancements to Android to address these issues.

• *New design principles for the security of systems.* Proposed three new principles for the design of secure systems.

1.5 THESIS ORGANISATION

In Chapter 2, I will present background knowledge on the Android operating system, its security mehcanisms and its shared resources. In Chapter 3, I will provide a discussion of the available literature on Android security. In Chapter 4 I will present my analysis on shared process privileges by libraries and their host apps and discuss how this analysis allowed me to build an automatic tool for assessign the potential exposure of sensitive user information to advertising libraries. In Chapter 5 I will introduce new side-channels on Android stemming from shared filesystem resources. I will also discuss new adversarial inference techniques which exploit those side-channels to compromise user's data confidentiality. In Chapter 6 I will analyze the Android runtime permission model, propose, implement and evaluate a re-design to tackle privilege escalation attacks. In Chapter 7 I will analyze the security of shared direct communication channels with external resources and propose security enhancements on the Android operating system to enable application-level access control to such resources. In Chapter 8 I will look into new attack surfaces introduced by smartphones sharing devices in smart-homes and propose a distributed fine-grained access control scheme to protect smart-home devices from malicious smartphone apps. In Chapter 9 I propose three new principles to guide the design of secure systems stemming from my analysis on smartphone operating systems. Lastly, in Chapter 10 I will conclude this treatise and discuss future directions. Lastly in Section 10 I will conclude this thesis and discuss its findings.

CHAPTER 2: BACKGROUND

In this Chapter I provide some background on the Android OS and its security features.

2.1 ANDROID OS

The advent of Android was announced on November 5th, 2007. This exquisite mobile platform was a result of a partnership of Google with OHA (Open Handset Alliance), a consortium of telecommunication, software and hardware companies and its source code is made publicly available. Android is an open source software stack encompassing a kernel layer, a middleware layer and basic applications.

Since the announcement of the first Android version, a number of new OS releases followed as depicted in table 2.1, with every version being playfully given a desert name [31]. Google ships its `Nexus` devices with the unmodified Android open source code while other hardware companies such as Samsung and HTC release their devices with appropriate modifications to satisfy their specific UI or hardware requirements.

Table 2.1: Android Versions [31]

No	**Release Number**	**Code Name**
1	1.0	Android Alpha
2	1.1	Android Beta
3	1.5	Cupcake
4	1.6	Doughnut
5	2.0-2.1	Eclair
6	2.2-2.2.3	Froyo
7	2.3-2.3.7	Gingerbread
8	3.0-3.2.6	Honeycomb
9	4.0-4.0.4	Ice Cream Sandwich
10	4.1-4.3.1	Jelly Bean
11	4.4-4.4.4	KitKat
12	5.0-5.1.1	Lollipop
13	6.0-6.0.1	Marshmallow
14	7.0-7.1.2	Nougat
15	8.0-8.1	Oreo

2.1.1 Architecture Overview

Android is usually depicted as a software stack featuring a Linux Kernel at the lower level. On top of that lies the Android middleware which integrates libraries written in C, the Android runtime, and the application framework written in Java. The Android software stack is displayed in figure 2.1.

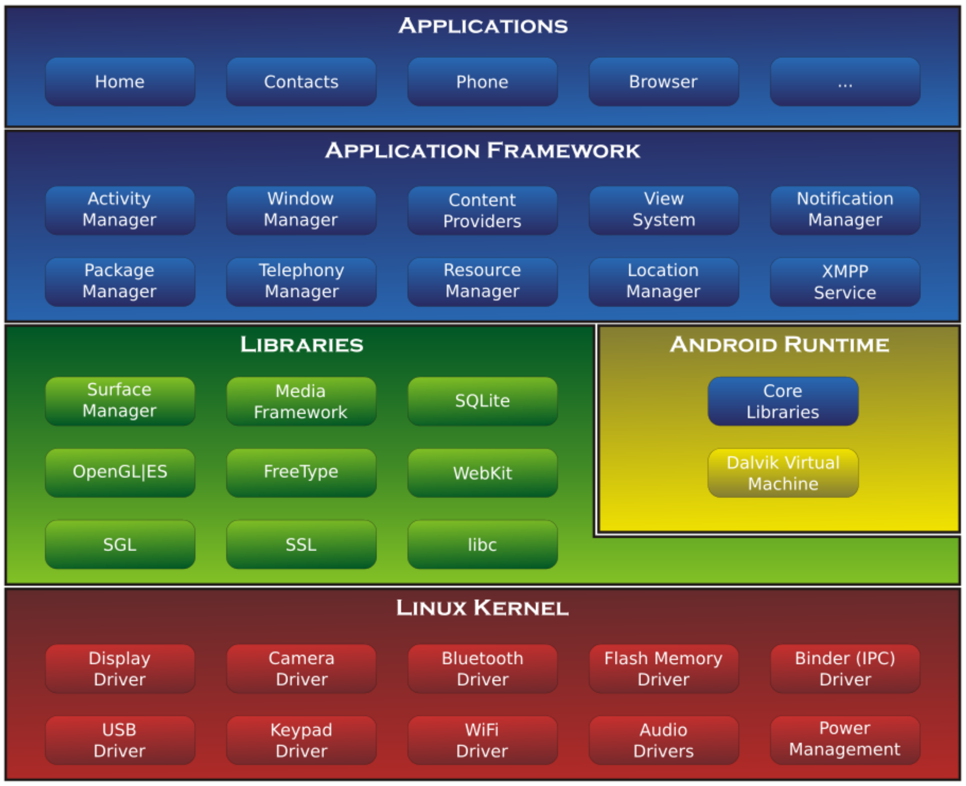
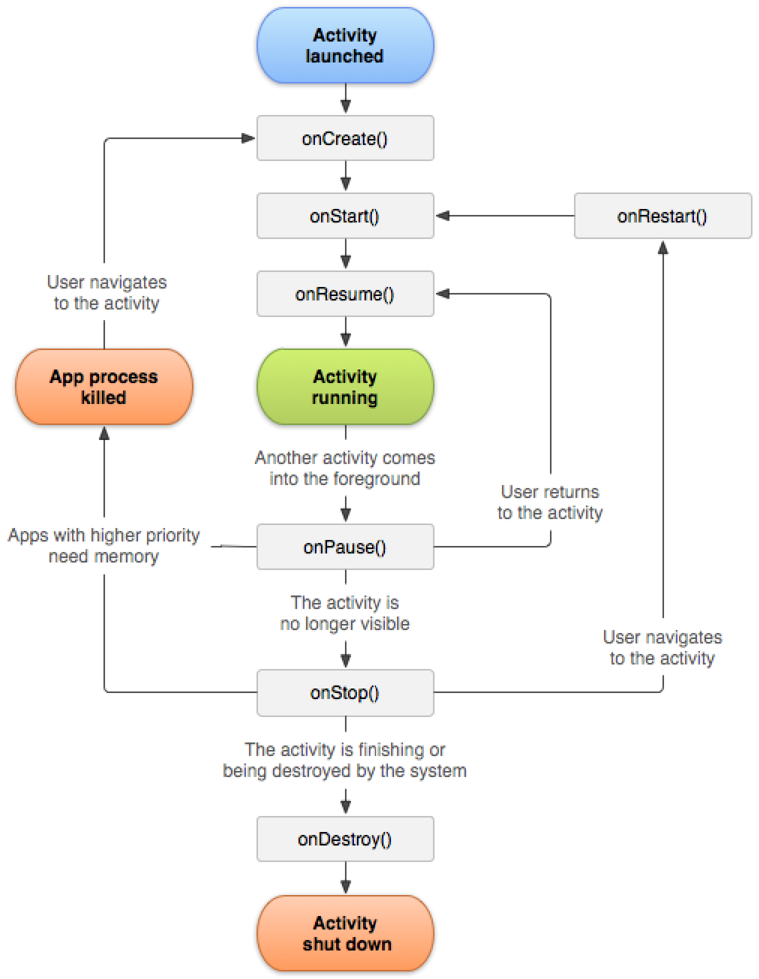


Figure 2.1: Android Software Stack [32]

Applications are also written in Java and can make use of a rich API provided by the Application Framework to access resources on the device such as the SMSs or contacts and perform actions such as place a phone call, handle an incoming phone call or SMS, access the GPS or accelerometer data and so on. Nevertheless, use of native code (C, C++) is not prohibited and apps can use it although they rarely do. An app can also use the JNI (Java Native Interface) that allows Java code to interact with native code when use of both is imperative. An application's major components are `Activities`, `Services`, `Content Providers`, `Intents`, `Broadcast Receivers`.

Activity: An Activity is usually correlated with a UI screen on the phone. An activity can display UI elements when in the foreground, invoke another activity (screen) or be invoked to be shown on the foreground. It must extend the Android Activity class and follow the Activity Lifecycle as shown in figure 2.2 given by the official Android documentation [33].



Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.2: Activity Lifecycle [33]

Service: A Service is an application component that does not need a UI to run. It is being used to perform tasks in the background and can continue running even if the parent app is not. They have high priority and they are the last being killed by the OS in the event that resources need to be freed. Even then they are immediately restarted once enough resources are made available.

Content Provider: A Content Provider is a convenient structure provided by the application framework to applications, to access databases on the device. For example if an app needs to access the SMSs, it can use the appropriate content provider which allows the app to query the SMS database.

Intents: Intents is a powerful inter-component communication tool for applications and userspace processes. An application (built-in or third-party) can notify other applications about an event, or even send data to interested applications through this mechanism. Interested applications can receive such broadcasted intents through *Broadcast Receivers*.

Broadcast Receiver: An application can register a broadcast receiver to receive specific intents. For example an app can register to receive the intent sent by a framework app notifying that the system has booted, or that a bluetooth device has just paired. Another example is the `Activity Manager` that can receive intents regarding the intention of an activity to launch a new activity. We will elaborate on how this works later on.

It is also important to understand that each Android application runs as a separate Linux process with its own instance of the Dalvik Virtual Machine (DVM) as shown in figure 2.3. Dalvik is an efficient process virtual machine with just-in-time (JIT) compilation specially designed for Android due to its constraints in memory and processor speed. Android programs are usually written in Java and then compiled to bytecode. Then they are converted from .class files compatible with the Java Virtual Machine, to Dalvik executable files (.dex). Subsequently these .dex files are compressed in an apk (Android Application Package) and installed on the Android device. In version 5.0, Android replaced Dalvik with Android Runtime (ART) which performs a more efficient and power preserving ahead-of-time (AOT) compilation, which compiles entire applications into machine code at installation time.

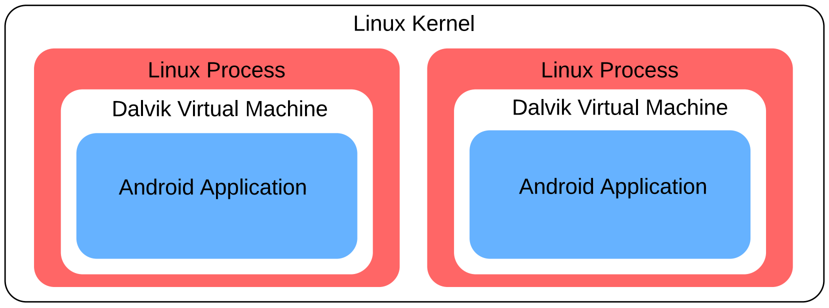


Figure 2.3: Application Isolation on Android

2.1.2 Android Boot Sequence and the Zygote Process

When an Android device boots, the bootloader runs first, which eventually starts the kernel. Once the Kernel is up and running it will mount the root filesystem and launch the `init` process. This process will look into a file called init.rc which dictates which system services will have to be launched next and set up filesystem and other system parameters. Init will start the `Service Manager` which is responsible for managing services' registration and requests for registered services. The init process will also start the `Zygote`. The Zygote is the parent process of every other process. For example since every application is essentially a process, that must be forked out from the Zygote and this exactly what the Activity Manager is doing. Next the Zygote initializes the Dalvik VM and forks the GUI process and the System Server process in their respective DVMs. The System Server process is responsible for starting the Android system services such as the Activity Manager, Telephony Manager, Package Manager (handles installation/uninstallation of applications), Bluetooth and so on.

When the System Server starts a Service, that action goes through the Service Manager which maintains an index of all started services. Now, if an app wants to access a system service, it has to go through an RPC (Remote Procedure Call) mechanism called `Binder` which in turn will deliver the request to the Service Manager. The Manager then will return again through the Binder, a handle to the application which will allow it to use the service. The Binder is implemented in the kernel and the app developers do not interact with it directly when requesting a Service access.

Having a basic understanding of the Android platform and important terms covered we will now scrutinize over the Android Security Model.

2.2 ANDROID SECURITY MODEL

Android employs a number of security features. We will focus on the inherent Linux security, the permission model to protect sensitive API calls and the latest integration of SELinux on Android which enables Mandatory Access Control on the kernel.

2.2.1 Application Sandbox

As stated before, Android features a Linux kernel. As a result it benefits from its `discretionary access control` (DAC) on the filesystem. This is an implementation of `access control lists` (ACLs), where for each object the system stores a list of users that can access it. In Unix and in extend Linux and Android, users can be grouped together to avoid long sparse lists. This is stored in the file's node and when a user requests access to it, the OS will check whether the requesting user is the owner of the resource. If that is not the case it will then check if the user belongs to a group that can access it. Lastly it checks whether the resource can be accessed by the `rest of the world` to decide if it will grant access. The actions that can be performed by a user on a Linux file are one of three: `Read`; `Write` or `Execute`.

On Android each application is considered a different user and runs in its own Linux process. This way it owns its own memory stack and can access its own resources taking advantage of Linux's user-based protection. The system bestows a unique User IDentifier called UID to every installed application and runs it in a newly forked process. Linux ensures that no process can access another process's resources and restrict communication between them through its secure IPC (interprocess communication) mechanism. This is known as the `Application Sandbox` and its implemented in the kernel. Thus it can protect applications from each other whether they use Java or native code. Consequently, application sandbox can be compromised only when the kernel itself is compromised.

However Android provides developers the capability to share resources among their own applications: Apps are signed with certificates whose private key are in the acquisition of their respective developers. Applications signed with the same certificate, can request to share UID and thus consider as a single Linux user and share the same resources. This request to the system, can be defined by the application developer in the app's manifest file, namely `AndroidManifext.xml`. The presence of that file in the app's root directory is non optional. It tells the system about the major components the app is using (Content Providers, Broadcast Receivers, Services, Activities e.t.c), lists libraries that the app must be linked against, requests permissions to access protected APIs, names the Java package for the app which can be used to uniquely identify it and contains other essential information about running the particular app.

2.2.2 Permission Model

Android offers applications a rich API to access resources on the system through its application framework shown in figure 2.1. The Android sanbox allows access to some basic resources. To protect access to resources that are considered sensitive, such as accessing services that might cost users money, or functions that can lead to private information leaks, Android employs a security mechanism called `Permissions`. According to this mechanism, a permission is mapped with one or more sensitive functions. An application must declare in its manifest all permissions required for it to run properly, according to the function call (or resource accesses) it makes.

Android Permissions can have different `protection levels`. A permission's protection level can have the value `normal`, `dangerous`, `signature` or `signatureOrSystem`. A normal permission is consider to be of minimal risk to the application, the system or the user. Such permissions can be granted automatically by the system without user interaction during installation unless their revision is explicitly requested by the user. A dangerous permission is of higher risk as it can provide access to private information or device features that can adversely impact the user. These kind of permissions must be presented to and accepted by the user. A signature permission, is granted automatically by the system only if the requesting application is signed with the same certificate as the application that declared the permission. Lastly a signatureOrSystem permission that the system automatically grants to the requesting application, if that application is either signed with the same certificate as the declaring application or the requesting application is built as part of the Android system image (i.e a system application). The first comprehensive study on Android Permissions was conducted by Felt et al. [27].

Before Android 6.0 (API level 23), all permissions were granted at application installation time. However, starting with version 6.0, Android adopted the runtime permission model, where dangerous permissions are granted to an app at runtime by the user the first time they are used by this app, and the user is given the ability to revoke these permissions to apps at any time. This ask-on-first-use model was first proposed by Wijesekera et al. [34]. Normal and signature permissions are still granted at installation and cannot be revoked by the user. Additionally, in version 6.0, Android introduced permission groups which cluster permissions based on their utility [35]. According to the runtime model, if a dangerous permission in a permission group is granted to an app, all the dangerous permissions in that group will also be granted (if explicitly requested by the app) in order to minimize user's effort. There are currently nine permission groups on Android: calendar, camera, contacts, location, microphone, phone, sensors, SMS and storage (Figure 2.4).

The Android OS checks system permissions in 2 ways as shown in Figure 2.5: Either at the framework level or at the kernel level. Most commonly, an application can request access to a sensitive API using the appropriate *Manager*. The Manager provides a convenient way to apps to query a `Service` for a resource. The request will go from the Manager, through the Binder to the Service, which will check whether the calling process has the permission to access the requested resource. If it does, access is granted, otherwise a Security Exception is thrown back to the application. Consider for example an application that wants to connect to a paired Bluetooth device. That app will use the BluetoothAdapter to find the BluetoothDevice it needs. Then it will obtain a BluetoothSocket handle calling `device.connectRFcommSocket` for serial data transfer with the RFCOMM protocol. The socket handle can be used to call `socket.connect` to actually establish the connection. The connect request will go through Binder RPC to the Bluetooth Manager Service which binds to `AdapterService`. The Adapter Service is responsible to establish the connection on

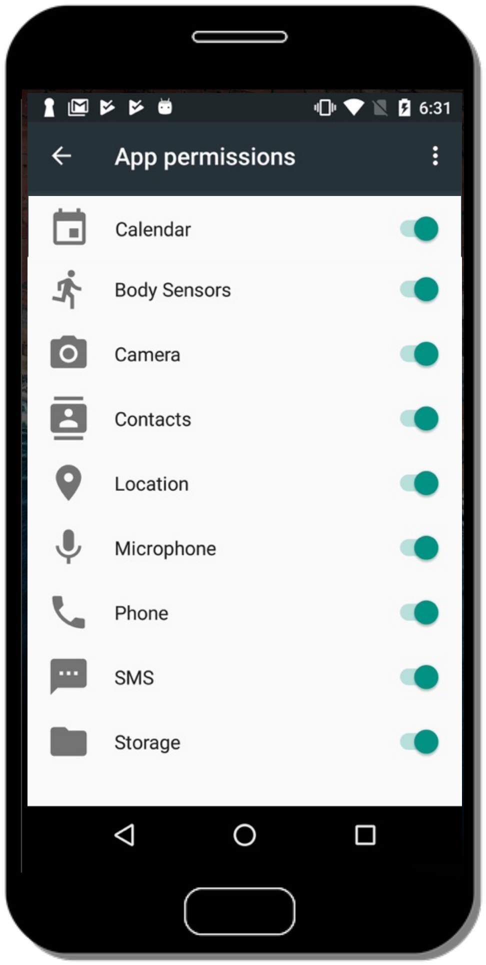


Figure 2.4: Permission groups on Android version 7.0 (Nougat).

behalf of the app. Before doing so, it checks whether the calling app has the *BLUETOOTH* permission.

Alternatively an app can directly request access to a hardware feature. This request can be checked for permission at the kernel layer. For example when an app is granted the *INTERNET* permission during installation, its assigned UID is mapped with the Internet Group's ID (GID), which corresponds to the number 3003 and referred to with the constant `AID_INET` in the kernel. Before an IPv4 or IPv6 socket is created, the kernel first checks whether the requesting process belongs to the group AID_INET. If it doesn't, it returns an access error.

Custom Permissions. Third-party apps are allowed to define new permissions on Android. These permissions, are called *custom permissions*, and are used to protect an app's own components from other apps. In order to define a new custom permission, an app must provide a permission name and can optionally include a permission group (>= version 6) to

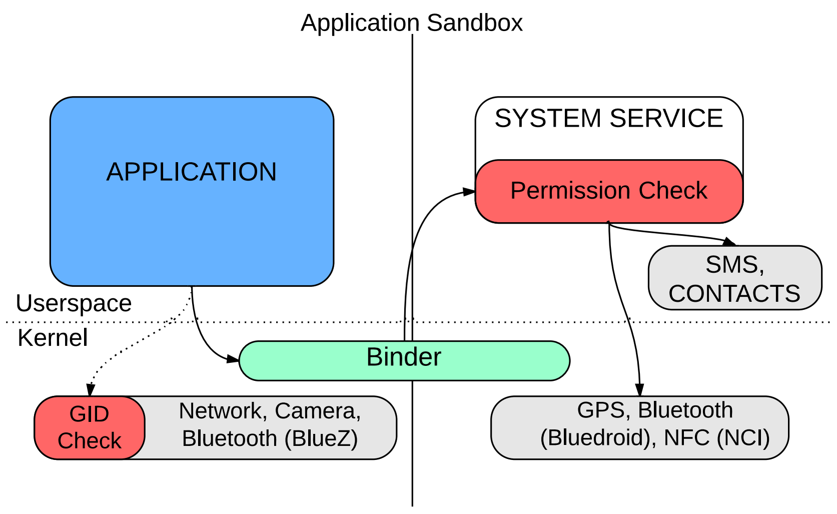


Figure 2.5: Android Permission Check

which this permission belongs and a description regarding the utility of the permission in its manifest. Additionally, in order to request a permission, an Android app needs to declare the use of the permission by referring to it with its name. Furthermore, Android allows applications to create custom permissions dynamically via the use of the `addPermission()` API method. In order for this method to work successfully, apps need to declare permission trees in their manifest file which state the domain name under which the dynamic permissions will be created.

Although it is suggested that reverse domain name notation should be used for custom permission names, there is currently no naming convention enforced by the system for custom permissions and apps can use any name they desire when creating new custom permissions. One exception is that Android does not allow two different permissions to coexist on the same device if they have the same name; hence, installation of an app which defines a permission with a name that belongs to an existing permission on the device will be denied by the system. Conventional use case for custom permissions is for apps to define custom permissions with the signature protection level so that only the apps that are signed with the same certificate (e.g., apps that belong to the same developer) can utilize the definer app's resources. For example, if an application's component is protected with a signature permission, only the apps that are signed with the same certificate as that of the component owner can access it.

2.2.3 SELinux on Android

SELinux is a Mandatory Access Control (MAC) security mechanism, designed by United States National Security Agency, and is integrated in various popular Linux distributions. Smalley et al. [36] published a detailed solution to port SELinux on Android, called Security Enhanced Android (SEAndroid).

Security-Enhanced Android is built on top of Android [36]. It is designed to mediate all interactions of an app with the Linux kernel and other system resources. Furthermore, SEAndroid confines even system daemons to limit the damage they can cause once they are compromised. It also provides a centralized policy configuration for system administrators and device manufacturers to specify their policies.

More specifically, SEAndroid [36] associates with each subject (e.g., process) and object (e.g., file) a `security context`, which is represented as a sequence `user: role: domain or type[: level]` and indexed by a `Security Identifier` (SID). The most important component here is `type`[1]. Under a `type enforcement` (TE) architecture, a security policy dictates whether a process running within a `domain` is allowed to access an object labeled with a certain `type`. Following is a policy specified for all third-party apps: `allow untrusted_app shell_data_file:file rw_file_perms`. This policy states that all the apps within the domain of `untrusted_app` are allowed to perform "`rw_file_perms`" operations on the objects with a type of `shell_data_ file` within a `class`[2] `file`.

SEAndroid appeared in Android in version 4.3, running in `permissive` mode. In this mode, the system allows a process to access a resource even if that violates the policy. However it records the violations and reports it in the system's logs. It is common practice to test SELinux policies in permissive mode, to identify policy inadequacies or unearth policy bugs that might result to system crashes. In version 4.4 we saw SEAndroid running in enforcing mode for several root daemon processes such as installd (responsible for installing apps), the zygote (responsible for forking new processes for newly launched apps), the vold process (volume daemon: manages device nodes) and the netd (network daemon: provides access to the Network). All other processes, including system and third-party apps and services still run in permissive mode.

The policy files are under `external/sepolicy` in AOSP's (Android Open Source Project) source code and are built with the system such that the resulting policy in binary code is read-only and unable to be modified without shipping a new binary and rebooting the phone. The most important files are `mac_permissions.xml`, `file_contexts`, .te files for each

[1]`role` is for role-based access and `level` for multi-level security.

[2]A `class` defines a set of operations that can be performed on an object.

domain that processes can be assigned to and `seapp_contexts`. In mac_permissions.xml, policy engineers can define a label to be assigned to an app, according to the certificate used to sign it. That label is called `seinfo`. In file_contexts, every Linux file is assigned a security `type`. In seapp_contexts, domains are defined for `seinfo` labels. Lastly a domain is defined by creating a "domain name".te file. Inside that file the rules dictating what a process that belongs to that domain can access are defined.

Consider the following example. Let's say that we want to assign an app called `TestApp` to a domain called `testdomain_app`. Then we want to allow that app to open the wallpaper file `/data/data/com.android.settings /files/wallpaper`. First we must assign a security context to the subject, i.e the file. Inside file_contexts we add the following line:

```
  /data/data/com.android.settings/files/wallpaper \
u:object_r:wallpaper_file:s0
```

This will assign the type `wallpaper_file` to our file in question. Next we must create the domain that will be allowed to access this file. For that we create under external/sepolicy a `testdomain_app.te` file. Inside this file we will place all the rules that will dictate what a process assigned to this domain can access. Thus we include a rule like below:

```
  allow testdomain_app wallpaper_file:file open;
```

The class file is defined in the file `external/sepolicy/access_vectors`. In that file the operation `open` is defined for subjects that will belong to the class file. Our rule will allow any subject in the testdomain_app domain, to perform the action open on the wallpaper_file object which is a file. We are still missing something though. We haven't told the system how to associate our TestApp app with the testdomain_app domain. For that we include the app's certificate (e.g testApp.x509.pem file) under built/target/product/security. Inside external/sepolicy/keys.conf we define a tag name (e.g TESTTAG) to refer to our app's certificate. To do that we use the following syntax:

```
  [@TESTTAG]

  ENG :testApp.x509.pem
```

Next in mac_permissions.xml we associate this certificate with an seinfo tag let's say testApp_seinfo. To do that we include the following lines of code:

```
<signer signature="@TESTTAG">

<seinfo value="testApp_seinfo" />

</signer>
```

Lastly we associate the seinfo tag assigned to our app with the testdomain_app domain in seapp_contexts by adding the following line:

```
user=_app seinfo=testApp_seinfo domain=testdomain_app
```

The SEAndroid module currently incorporated into the AOSP (Android Open-Source Project) 4.3 and 4.4 defines five domains within its policy files: `platform_app`; `shared_app`; `media_app`; `release_app` and `untrusted_app`. The **platform** domain is assigned to all apps signed with the platform key, i.e packages that are considered as part of the core platform such as System UI, Bluetooth, Settings e.t.c. The **shared** domain is assigned to the launcher and contacts related packages while the **media** platform is assigned to the gallery app and media related providers. The **release** domain is assigned typically to device's vendor apps and google apps. The last one, **untrusted** domain, is the domain assigned to all applications installed by the user.

As noted before, these policy files are ready-only and compiled into the Android kernel code. They are enforced by security hooks placed at different system functions at the kernel layer. For example, the function `open` we saw before, is instrumented to check the compliance of each call with the policies: it gets the type of the file to be opened and the domain of the caller, and then runs `avc_has_perm` with the SIDs of both the subject (testdomain_app) and object (wallpaper_file) to find out whether this operation is allowed by the policies. Here `avc_has_perm` first searches an Access Vector Cache (AVC) that caches the policies enforced recently and then the whole policy file. In addition to the components built into the kernel, SEAndroid also includes a separate middleware MAC (MMAC) that works on the application-framework/library layer. The current implementation of MMAC is limited to just assigning a security tag (testApp_seinfo) to a newly installed application (TestApp) (through mac_permissions.xml). When Zygote forks a process for an app to be launched, it uses that tag in tandem with a policy file (`seapp_contexts`) to decide which SELinux domain should be assigned to it.

SELinux integration on Android creates new possibilities for defending the system and the applications it supports and this work we will take advantage of this it and seamlessly extend it to protect against critical vulnerabilities that we will discuss on later chapters.

2.3 BACKROUND ON TECHNIQUES AND METHODOLOGIES USED

NLP Techniques: The NLP community has developed different approaches to analyze unstructured data. For example, NLP is used to parse user reviews online or user voice commands to digital personal assistants. Work focused on extracting grammatical information to understand what the user is trying to convey. Part-of-speech Tagging (POS Tagging), is a typical technique to achieve that. It is used to determine for each word in a sentence whether it is a noun, adjective, verb, adverb, proposition, and other part of speach. A common problem in NLP arises when one needs to perform word sense disambiguation. That is, to derive a given a word's semantic meaning. This can be challenging as a word might have multiple meanings and complex relationships with other words. To this end, Wordnet [37], an English semantic dictionary has been proposed, where the community tried to capture most of senses, of most of the English words. Wordnet also provides relationships between words, such as whether two words are synonyms, or connected with `is-a` relationship and so on. In essence, Wordnet is a graph with words as nodes and relationships as edges. To assist in better capturing the relationships between words, the community has developed multiple similarity metrics which are different ways to parse the Wordnet graph. For example, the LCH [38] metric, uses the shortest paths between two words to determine how similar the words are. To accurately determine which of the multiple senses of the word is the most appropriate, one needs to carefully select the right similarity metric or design a new similarity metric, and design her system in a way that incorporates domain knowledge. These are challenges we had to overcome in our work to enable extraction of targeted data from local files. Furthermore, our target files do not contain real words that can be used in an actual conversation but rather variable names. These are some of the challenges I had to overcome (see Chapter 4).

Formal Verification via Alloy: Alloy is a declarative specification language that is used to model the behavior and structural constraints of complex systems [39]. It provides a modeling tool called Alloy Analyzer that operates based on first-order (i.e., predicate) logic and can be used to analyze formal models created with the Alloy language. Statements in

Alloy can be interpreted both from object-oriented (OO) programming paradigm and from set theory perspectives. *Signature* is a declaration of a schema, which defines the vocabulary of the model. It is similar to the concept of a class in OO paradigm and to a set in set theory. It can consist of several *fields*, which are equivalent to fields in OO paradigm and to relations from a set theoretical perspective. *Facts* are global constraints to the model that are always supposed to hold. *Predicates* define parametrized constraints, which can be interpreted as operations that can be performed in the model. *Functions* are expressions with declaration parameters and they return a result based on the parameters. *Assertions* are assumptions made on the model and they can be validated via the Alloy analyzer. Additionally, Alloy allows using multiplicity keywords as quantifiers in quantified constraints: `all` (universal quantifier), `some` (existential quantifier), `lone` (zero or one), `one` (exactly one), `no` (zero). Also, it is possible to use `some` (or `set` interchangeably), `lone`, and `one` for field declarations in signatures to indicate the number of elements a field can take and also for signature declarations to indicate the number of elements that can belong to the set of the signature.

The Alloy Analyzer tool performs only finite scope checks on the models. The analysis is sound since it can never return false positives and is complete up to a scope as the tool will never miss any counterexamples that are equal or smaller than the specified scope. As in traditional model checking, Alloy models are infinite, that is, the specification dictates how the components of a system should behave without any restrictions on their quantity. The analyzer also provides automated analysis by allowing automatic generation of examples that satisfy a given model as well as counterexamples to claims (i.e., assertions) that are expected to hold in the model.

2.4 ANDROID'S SHARED RESOURCES

On a par with any multi-process operating system, modern smartphone operating systems manage process access to resources. This thesis performs a security analysis on such shared resources across smartphone applications. In particular it focuses on shared process privileges; shared filesystem resources; shared system and application resources; and shared connectivity resources.

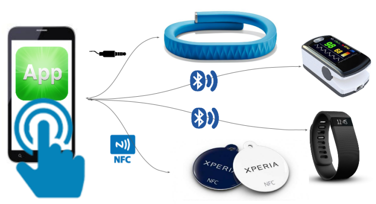
• *Shared Process Privileges.* Each app on Android is assigned a unique static UID when it is installed. This allows the operating system to differentiate between apps during their lifetime on the device, so it can run them in distinct Linux processes when launched. In this way Android leverages the traditional Linux process isolation to ensure that one app cannot access another app's resources.This is with the exception of apps signed with the same developer key. In that case, the apps can indicate in their manifests that they should be assigned the same UID. Developers also commonly utilize third-party libraries for either utility (avoid reinventing the wheel for common funcitonality) or advertising (allows for monetization of free apps). However, when developers include a third-party library, the library is treated as part of the host app. The operating system will assign one `UID` for the app as a whole, even though the library and host app have different package names. Every time an app is launched, the OS will assign a process identifier (PID) to the app and associate that with the app's UID. Again this PID is shared between the host app and its libraries that run within the same Linux process. As a result, the host app and the library components will also share privileges and resources, both in terms of Linux discretionary access control (DAC) permissions and in terms of Android permissions granted. The former allows the library to access all the local files the host app is generating. The latter allows it to use the granted permissions (*e.g.,* ACCESS_COARSE_LOCATION) to access other resources on the device (such as GPS), that can expose user information (such as her location).

This multifaceted ecosystem, where there are strong incentives for more data collection by all stakeholders, needs to be better understood. Of particular interest are advertising libraries offered by advertising networks which rely on building detailed user profiles to optimize their services. Studying the current practices of ad libraries is an important place to start. Indeed our community already found that ad libraries collect some types of data for themselves even without the cooperation (or with the implicit consent) of the host app developer. Such behaviors have been observed in the wild since 2012 [3] and as a routine practice today [40] for certain types of information. Nonetheless, to fully assess the privacy risk associated with embedding a library into an app, we need to take into account not only past and current behaviors, but also all allowed events that can lead to breaches of users' data confidentiality. My work aims to take the first step into the direction of modeling ad libraries, not based on previous behaviors but based on their allowed actions on the Android platform. I show how this can be leveraged to design a tool that can assess the targeted data exposure to ad libraries (Chapter 4).

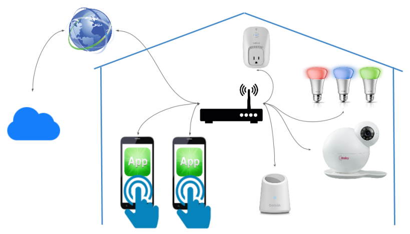
• *Shared Filesystem Resources.* Android is built on top of a stripped down version of a Linux kernel. Linux, historically makes available a large amount of resources considered harmless to normal users, to help them coordinate their activities. A prominent example is the process information displayed 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 a 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 [41] to keep track of individual apps' mobile data consumption.

• *Shared System and Application Components.* Android offers a rich API which allows third-party applications to request access to system resources. Felt et al [27] found 1259 API calls with permission checks in Android 2.2, while Au KW et al [42] found 17,218 permission protected APIs on a study spanning Android 2.2 until 4.1. As mentioned in Section 2.2.2 permissions can be of different protection levels. APIs protected by *dangerous* permissions (granted by the user) protect access to nine (9) main sensitive system resources: calendar, camera, contacts, location, microphone, phone, sensors, SMS and storage. Third-party apps can request permission from the user to access these. At the same time, Android allows application developers to share application components (content providers, services, broadcast receivers, activities) with other apps on the smartphone (see Section 2.1 for a description of app components). Application developers can declare their own permissions (*custom* permissions) to control access to these *exported* components. For example, by protecting a *content provider* with a *signature* permissions, developers are guaranteed by the OS that the component can only be accessed by other co-installed applications that are signed with the same developer key.

• *Shared Connectivity Resources.* Android and other mobile systems are routinely employed by their owners for managing their external resources. Particularly, almost every app running on these systems is supported by a remote service, which interacts with the app through the Internet or the telephone network (using short text messages). Such services are increasingly being utilized to store and process private user information, particularly the data related to online banking, social networking, investment, healthcare, etc. Moreover, the trend of leveraging smartphones to support the Internet of Things, brings in a whole new set of external devices, which carry much more sensitive data than conventional accessories (e.g., earpieces, game stations). In my work I categorize IoT devices into two classes: *Personal* devices are devices which connect to smartphones directly, through a proximity protocol such as Bluetooth, NFC, WiFi Direct e.t.c. (see Figure 2.6b). Examples include health and fitness systems (e.g., blood pressure monitors [43], Electrocardiography sensors [44], glucose meters [45]), and remote vehicle controllers (e.g., Viper SmartStart [46]) among others. Other devices belong to the *Shared* class (see Figure 2.6d). Such devices are typically home



(a) Smartphone apps compete for connectivity resources to access *personal* IoT devices.



(c) Smartphones and their apps compete for connectivity resources to access *shared* IoT devices.

Figure 2.6: Shared connectivity resources can be used for accessing external devices.

automation andvsecurity systems [47]. Examples are smart thermostats [48, 49], cameras for streaming surveillance video to a mobile phone [50]; the baby monitors [51], the smart refrigerators [52] and others.

CHAPTER 3: LITERATURE REVIEW

3.1 ADVERTISING LIBRARIES

Several efforts try to characterize the current mobile advertising libraries. MAdScope [40] and Ullah et al. [53] both found that ad libraries have not yet exploited the full potential of targeting. My work is driven by such observations and tries to assess the data exposure risk associated with embedding a library in an app.

Many studies describe alternative mobile advertising architectures. AdDroid [54] enforces privilege separation by hard-coding advertising functions as a system service into Android platform. AdSplit [17] achieves privilege separation via making ad libraries and their host apps run in separate processes. Leontiadis et al. [55] proposes a client-side library compiled with the host app to monitor the real-time communication between the host app and the ad libraries to control the exposed information. MobiAd [56] suggests local profiling instead of keeping the user profiles at the data brokers to protect users' privacy. Most of these alternative architectures envision a separation of ad libraries from their host apps. However, none of these solutions are deployed in practice as they all disrupt the business model of multiple players in this ecosystem. I take a different approach by analyzing and modeling the capabilities of ad libraries in order to proactively assess apps' data exposure risk.

There are a number of studies that aim to—or can be used to—detect and/or prevent current privacy-infringing behaviors in mobile ads. Those works mainly fall into three general categories: (1) static scanning [3, 18, 19, 20, 21], (2) dynamic monitoring [22, 23, 24, 25], and (3) hybrid techniques using both [26]. A combination of these techniques could detect and prevent some of the attack strategies of ad libraries we discussed in this work, if they are adopted in practice. However, such countermeasures can still fail to protect against all allowed behaviors. For example, TaintDroid [25] and FlowDroid [20] cannot evaluate the sensitivity of the data carried. Moreover, static code analysis will miss dynamically loaded code, and code analysis in general cannot estimate the potential reach of libraries. Further, by merely encrypting local files we cannot prevent libraries within the same process from using the key the host app uses to decrypt the files. In addition, there is no mechanism to address data exposure through app bundle information as we reveal in this work because (1) this is not considered as a sensitive API from AOSP and (2) even if marked as sensitive it is unclear how access to it by apps and/or libraries should be mediated, as there are legitimate uses of it. My focus is not on detecting and tackling current behaviors but assessing the data exposure given all allowed behaviors. This is critical when trying to assess the privacy risk of an asset.

SUPOR [57] and UIPicker [58] seek instances where apps exfiltrate sensitive data. These works also use NLP and machine learning techniques to find data of interest in user interfaces. However, their focus is on data like account credentials and financial records, whereas I focus on general targeted data with validation based on data of interest to advertisers. As with most of the other work in this area, SUPOR and UIPicker seek existing exfiltration instances rather than allowed instances, although some of their techniques can facilitate finding allowed instances.

3.2 INFORMATION LEAKS THROUGH FILESYSTEM RESOURCES

Information leaks have been studied for decades and new discoveries continue to be made in recent years [59, 60, 61]. Among them, most related to my work is the work on the information leaks from procfs, which includes using the ESP/EIP data to infer keystrokes [62] and leveraging memory usages to fingerprint visited websites [63]. However, it is less clear whether those attacks pose a credible threat to Android, due to the high non-determinism of its memory allocation [63] and the challenges in keystroke analysis [62]. In comparison, our work shows that the usage statistics under procfs can be practically exploited to infer an Android user's sensitive information. The adversarial inference technique I will introduce in this work is related to prior work on traffic analysis [64]. However, those approaches assume the presence of an adversary who sees encrypted packets. Also, their analysis techniques cannot be directly applied to smartphone. The attack I demonstrate 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, BSSID databases and Google Maps) that comes with the target app and the mobile OS.

Information leaks have been discovered on smartphone by both academia and the hacker community [8, 9, 65]. Most of known problems are caused by implementation errors, either in Android or within mobile apps. By comparison, the privacy risks which manifest due to shared resources in the presence of emerging background information have not been extensively studied on mobile devices. Up to my knowledge, all prior research on this subject focuses on the privacy implications of motion sensors or microphones [66, 67, 68, 69, 70]. What has never been done before is a systematic analysis 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 [25, 71, 9, 72, 73, 74, 8]. Different from such research, my work focuses on the new privacy risks emerging from the fast-evolving smartphone apps, which could render innocuous shared filesystem resources indicative of sensitive user information.

3.3 IPC, SYSTEM RESOURCES AND SHARED APPLICATION COMPONENTS

Previous work has shown ways of exploiting IPC on Android to acquire unauthorized access to resources. In [75], the authors discuss the permission re-delegation problem where an unprivileged app can access system resources through a privileged app via IPC. Additionally, [76] shows ways of exploiting the Intent mechanism to send or receive Intents in an unauthorized manner and get access to other app's private resources. Wei et al studied the evolution of permissions across Android versions and showed that the set of permissions on Android tends to grow with every release [77]. Stowaway tool aims to detect if apps follow the least privilege for permission requests [27]. Additionally, [78] presents a formal analysis of Android permissions for older Android versions (<6.0) in Alloy; whereas [79, 80] introduce similar models in Coq. One of the early works on the runtime permissions (¿= Android version 6.0) shows the necessity of having revocable, ask-on-first-use type permissions on Android, supported by user studies [27]. [81] provides an initial analysis on the runtime permission model and identifies several problems in this model that might open up ways for exploits. In [82], the authors analyze the undesirable side effects of switching to runtime permissions and introduce a tool called RevDroid that aims to identify these problems in apps. DP-transform provides a tool which helps developers adapt to the runtime model by automatically introducing the permission requests required by the model into the application code [83].

Although previous work has studied Android permissions which protect system resources, there is little work done specifically regarding Android custom permissions which are typically used by thirs-party developers to protect their shared application components.The blog post in [84] discusses how the "first one wins" approach for custom permission definitions can create problems. Shin et al presents a viable attack on custom permissions by exploiting the naming convention problem of custom permissions [85], to which Google responded with bug fixes. In [86], the authors discuss how permissions can stay dormant on the Android platform, later to be revived by the installation of a permission definer app, and demonstrate attacks on custom permissions via the exploitation of this undesirable property. In this word I present a systematic analysis of custom permissions on Android which reveals the root causes of those vulnerabilities and propose a new design of the permission model which eradicated these problems.

3.4 SHARED COMMUNICATION CHANNELS

Shared communication channels such as Bluetooth, NFC, Audio and SMS, rely on system permissions for delegating access to third-party apps. The Android permission system has been under scrutiny for years [87, 88, 89, 25, 90, 91, 89, 92, 93]. Much has been proposed to extend this security model, allowing the phone user to selectively grant permissions to apps [93], deny those with dangerous permission combinations [74], utilize app-defined fine grained access control [87] or leverage IPC provenances for security protection [8]. However, all these prior approaches are designed to guard a phone's local resources. In contrast little has been done on mobile OSes to protect the external devices that connect to smartphones. In particular, on Android, an app that acquires the permissions to use a channel (e.g., Bluetooth, NFC, etc.) is automatically granted the access to any device attached to this channel. There is nothing to bind a device to its authorized app.

Related to my study on how Android delegated access to SMSs, is Porscha [89], which controls the content an app can access on a phone for digital rights management. Porscha controls access to SMS messages through sending an IBE encrypted message to a Porscha proxy on the phone, which further dispatches the message to authorized apps according to a set of policies. While effective, this solution is ad-hoc and specific to SMSs. I follow an architectural approach which allow me to propose a unified, easily maintainable and extentible design for controlling access to all shared communication channels with external resources.

Prior works on the security issues of health devices are also closely related to our work. Rahman et al. [94] identified several vulnerabilities on Fitbit, a wireless wearable fitness device, which can be leveraged to inject data into the device and launch a denial of service attack against it. Li et al. [95] look into the security weaknesses of glucose monitoring and insulin delivery systems and proposed the technologies for protecting those devices' operations using rolling-code and body-coupled communication. Also, Marti et al. [96] lay out a few necessary requirements for building a secure mobile health care system. All such prior work focuses on the security problems of a specific health device or the communication protocol it uses, whereas my research aims at understanding the security implications of Android's shared communication channels to such external devices in proximity, in the presence of malicious apps running on the phone.

3.5 SHARED IoT DEVICES

IoT attacks. Recent works demonstrated attacks on IoT devices [97, 98, 99, 100, 101, 102]. Fernandes et.al. found vulnerabilities on SmartThings' applications [101]. Their work focuses on a specific IoT hub that can integrate third-party IoT devices, whereas I propose a design applicable to an infrastructure that exists in almost all households with IoT devices. [98, 99], revealed vulnerabilities on smart-home devices. However they consider an adversary on a separate device. [102] considers an intricate mobile adversary which colludes with a cloud. I illustrate that the mobile adversary can succeed with minimal effort. All reported attacks further motivate the need for practical smart-home defenses.

Android side-channels and network monitors. [103] used the VPN service on Android for passive monitoring of mobile apps to collect user traffic information for analysis. However, it redirects all packets to a server that further routes the packets. This raises privacy concerns which I show we can avoid by implementing the routing functionality locally.

Access control. There have been various works on home access control which we classify in three major areas: surveys [104, 105, 106]; access control systems [107, 108, 109, 110, 100, 111]; and user studies for usable policy specifications [112, 113]. More relevant to my study on shared devices is the second. Nonetheless, most of these systems assume a clean-slate design where the OSes of participating nodes can be modified. My proposed solution is backward compatible: it requires just a software upgrade on a home's router and downloading an app on the phone. Other work focused on access control enforced on the mobile phones [114, 115, 116]. In Chapter 7 I also illustrate the design of hybrid MAC and DAC approach on smartphone operating systems to guarantee applicaiton-level access control to devices. This works well for *personal* devices which are typically owned by the smartphone user. In contrast, smart-home devices are shared across a local area network where guest users might connect their own smartphones which we cannot trust to carry our improved OS version. In Chapter 8 I will show how we can build an access control scheme distributed across a home area network router and trusted smartphones to tackle this problem effectively.

IDS and Firewalls. Work on intrusion detection systems (IDS), personal and application firewalls [117, 118, 119, 120], focuses either solely at the host or at a network node, or only at the network layer. The system I propose (Chapter 8) is distributed, consolidating application level semantics from hosts, and network level information from the network node. Furthermore, we do not require experts to set up policies.

In all previous works, the solutions are either ad-hoc, or impractical. In this thesis I conduct a systematic analysis of the security of shared resources on smartphones which reveal new adversarial capabilities of third-party smartphone apps. To mitigate such adversaries, I design solutions focusing on both detection of information leakage and prevention at an the operating or a distributed system level. My solutions follow four important design properties: (a) effectiveness; (b) efficiency; (c) backward compatibility and, (d) maintainability.

CHAPTER 4: SHARING PROCESS PRIVILEGES

Android enforces access control decisions at the process/application boundaries. However, smartphone apps commonly utilize third-party libraries from other untrusted sources for advertising. Thus, these libraries share all the privileges their host process is granted. Since advertising networks depend on building detailed user profiles we expect them to follow aggressive data harvesting techniques. In this chapter, I analyze how advertising libraries can take advantage of the shared process privileges with their host apps. Then I utilize this analysis to design a tool for automatic detection of potential sensitive information leakage to advertising libraries [10].

4.1 INTRODUCTION

Advertisers aim to generate conversions for their ad impressions. Advertising networks assist them in matching ads to users, to efficiently turn impressions into conversions. I call the information that achieves this *targeted data*. Android smartphones contain rich information about users that enable advertising networks to gather targeted data. Moreover, there is considerable pressure on advertising networks to improve the number and quality of targeted data they are able to offer to advertisers. This raises many privacy concerns. Mobiles often contain sensitive information about user attributes which users might not comfortably share with advertising networks but could make valuable targeted data. This, in turn, led to a substantial line of research on privacy and advertising on mobiles in two general areas: (1) strategies for detection and prevention [121, 22, 18, 19, 26, 3, 122, 23, 123, 53, 24, 40], and (2) architectures and protocols that improve privacy protections [56, 54, 17, 55]. The first of these approaches primarily provides insights into the current practices of advertisers and advertising networks. The second examines a future in which a changed advertising platform provides better privacy. However, some of the studies show that the development and use of targeted data on mobiles is modest at present [53]. This is at least partially because most applications do not pass along information about users to the advertising network—through its ad library embedded in the app—unless the advertising network requires them to do so [40]. This leave open an important question: *what if advertising networks took full advantage of the information-sharing characteristics of the current architecture?*

In particular, when one wants to assess the privacy risk associated with an asset, she needs to take into account not only past and current hazardous behaviors but all allowed actions that can result in potential privacy loss [124]. In the case of opportunistic advertising libraries, a privacy loss is possible if such libraries have the ability to access private user information without the user's consent. Current app privacy risk assessment techniques [125, 126], try to detect when sensitive data leaks from an app. To achieve that, they employ static or dynamic analysis of apps and/or libraries. However, applying this sort of assessment is constrained by the apparent practices of the advertising libraries. For example, every time an ad library is updated, or a new ad library appears, such analysis must be performed again. To make things worse, some ad libraries load code dynamically, [3] which allow them to indirectly update their logic without dependency on the frequency of their host app's updates. In this way, any analysis dependent on current library behaviors is unreliable as the analysis can not predict the behavior of updated code or dynamically downloaded/loaded code. Thus, to assess such risks, we need to have a systematic way to analyze the potential data exposure to ad libraries independent of current or apparent practices. A privacy risk assessment should consider what an adversary is allowed by the system to do instead of only what she is currently doing. My work takes the first step in this direction by analyzing the shared intra-process resources available to libraries and modelling their data collection capabilities on an Android platform.

I model opportunistic ad networks based on their abilities to access targeted data on an Android platform through at least four major attack channels: protected APIs by inheriting the permissions granted to their host apps; reading files generated at runtime by their host apps and stored in the host apps' protected storage; observing user input into their host apps; and finally unprotected APIs, such as the `PackageManager.getInstalledApplications()` that allow the ad library to access platform-wide information. We further categorize these attack channels into two classes, namely the `in-app` and `out-app` exploitation class. The `in-app` class contains attack channels that are dependent on the ad library's host app. The protected API's, app local files and user input are examples of such channels. The `out-app` class contains attack channels that are independent of the host app. The public API's are an example of this. In particular, Grace et. al. [3] identified that the list of installed applications on a user's device—which can be derived from a public API on Android—raises privacy concerns. In this work I systematically explore how this information can be exploited by an adversary in practice. I demonstrate and evaluate how well such APIs can result in an adversary learning a user's targeted data. Based on my data exposure modeling, I have designed and developed a framework called `Pluto`. Pluto aims to facilitate assessment of the privacy risk associated with embedding an untrusted library into an app. I show that Pluto is able to reveal the potential data exposure of a given app to its ad libraries through the considered attack channels. Frameworks like Pluto are extremely useful to app developers who want to assess their app's potential data exposure, markets aiming to better inform their users about the privacy risk associated with downloading a free app, and users themselves. In addition, I hope that this will spur similar academic attempts to capture the capabilities of third-party libraries on smartphones and serve as a baseline for comparison.

4.2 ANALYSIS

4.2.1 Threat Model

A risk is the potential compromise of an *asset* as a result of an exploit of a *vulnerability* by a *threat*. In this case, I define assets to be user targeted data, the threat is an opportunistic ad library, and a vulnerability is what allows the ad library to access targeted data without the device user's consent or the consent of the library's host app. Here, we examine the capabilities of the ad libraries to collect such data on an Android platform.

Because libraries are compiled with their host apps, are in extend authorized to run as the same Linux process as their hosts on an Android OS. Thus the ad library code and the host app's code will share the same identifier as far as the system is concerned (both the static UID and the dynamic PID). In essence, this means that any given ad library runs with the same privileges as its host app. Consequently, the libraries inherit all the permissions granted by the user to the host app. There is no way for the user to distinguish whether that permission is used by her favorite app or the ad libraries embedded in the app. This permission inheritance empowers the ad libraries to make use of *permission-protected APIs* on the device. For example, if an app granted the `GET_ACCOUNTS` permission, its libraries can opportunistically use it to retrieve the user's registered accounts (e.g., the email used to login to Gmail, the email used to login to Facebook, the email used for Instagram, Twitter and so on).

Furthermore, during their lifetime on the device, apps create *local persistent files* where they store information necessary for their operations. These files are stored in app-specific directories isolated from other applications. This allows the apps to offer seamless personalized services to their users even when they are not connected to the Internet. In addition this practice enables the apps to avoid the latency of accessing their clouds, provided they have one. Android offers a convenient way through its `SharedPreferences` class to store and retrieve application and user specific data to an XML file in its UID-protected directory. In that directory, apps can also create their own files typically using standardized formats such as `XML`, `JSON`, or `SQLite`. In this way, they can utilize widely available libraries and Android APIs to swiftly and easily store and parse their data. The ad libraries, running as the same Linux user as their host apps, inherit both the Linux DAC privileges and the

SE Android MAC capabilities of their host apps. This allows them to access the app's locally stored files as their hosts would. Consequently, the ad libraries could read the user data stored in those files. Consider, for example, the app *My Ovulation Calculator* which provides women a platform to track ovulation and plan pregnancy. This app, listed under the *MEDICAL* category on Google Play, has been installed 1,000,000–5,000,000 times. By parsing the app's runtime generated local files, an ad library might learn whether its user suffers from headaches, whether she is currently pregnant, and, if so, the current trimester of her pregnancy. All these are targeted data which advertisers can monetize [127], making them a valuable addition to ad libraries.

Moreover, an aggressive ad library could utilize its vantage position to peak on *user input*. In particular, such a library could locate all the UI elements that correspond to targeted data related to user input [58, 57] and monitor them to capture the data as they become available. For example, by monitoring the user's input on `Text Me! Free Texting & Call`, a communication app with 10,000,000–50,000,000 downloads, an ad library would be able to capture the user's `gender`, `age` and `zip code`. Note that these data constitute the quasi identifiers [128] proven to be enough to uniquely identify a large percentage of registered voters in the US.

Nonetheless, an ad library can exploit both the inherited privileges of its host app and the position on a user's device. Irrespective of the host app, the ad libraries, can make use of *public APIs* to learn more about the user. Such APIs are considered harmless by the Android Open Source Project (AOSP) designers and are left unprotected. This means that the apps can use those APIs without the need to request permissions from either the system or the user. In this chapter, I show that by merely acquiring the list of installed applications through such APIs, one can learn targeted data such as a user's `marital status`, `age`, and `gender` among others.

To model these attack channels, I further categorize them them into two classes, namely the `in-app` and `out-app` exploitation class. The `in-app` class contains attack channels that are dependent on the ad library's host app. The protected API's, app local files and user input, are examples of such channels. The `out-app` class contains attack channels that are independent of the host app. The public API's are an example of this. Through the rest of this work, we assume that an ad library can gain access to targeted data through permission-protected APIs, runtime-generated app local files, user input, and unprotected APIs.

4.2.2 Data Exposure through In-App Shared Process Capabilities

Ad libraries can leverage their position within their host apps to access exposed targeted data. Some targeted data are dependent on what the host apps themselves collect from the users. An ad library can access such data by parsing the files its host app created at runtime to store such information locally, that is in its own UID-protected storage. Furthermore, it can inherit the permissions granted to its host app and leverage that privilege to collect targeted data through permission-protected APIs. Finally, it can peek on what the host app user inputs to the app. In this section, I explore what an ad library can learn through these in-app attack channels. We elaborate on our methodology and provide insights from real world examples. To gain insight on what an ad library can learn, I perform manual inspection of some real-world free apps. This way we can validate the assumptions about data exposure through in-app attack channels and further create ground truth for test data that we can use to do evaluations of the framework in subsequent sections.

I first cherry-pick a few free apps I selected for purposes of illustration. I downloaded the target apps from Google Play and used Apktool to decompile them. I located the packages corresponding to the Google AdMob advertising network library and located an entry point that is called every time an ad is about to be loaded. I injected our attack logic there to demonstrate how the ad library can *examine local files*. In particular, this logic dumps the database and xml files that the app has created at runtime. I then compiled the app and ran it on a physical device by manually providing it with some input. Here are some examples of what such an aggressive ad library could learn in this position (or what AdMob is, in principle, able to learn now).

`I'm Pregnant` helps women track their pregnancy progress and experience. It has 1,000,000–5,000,000 installations and is ranked with 4.4 stars [1] on Google Play. The code was able to read and extract the local files created by the host app. After manually reviewing the retrieved files, I found that the host app is storing the weight of the user, the height, current pregnancy month and day, symptoms such as headaches, backache and constipation. It also recorded events such as dates of intercourse (to establish the date of conception) and outcomes like miscarriage or date of birth.

`Diabetes Journal` helps users better manage their diabetes. It has 100,000–500,000 installations and ranked with 4.5 stars on Google Play. The code was able to extract the local files generated by the app. Manually reviewing these files, I found that it exposes the user's

[1] Applications on Google Play are being ranked by users. A 5-star application is an application of the highest quality.

birth date, gender, first-name and last name, weight and height, blood glucose levels, and workout activities.

`TalkLife` targets users that suffer from depression, self-harm, or suicidal thoughts. It has 10,000–50,000 installations on Google Play and ranked with 4.3 stars. In contrast with the other two apps above, TalkLife stores the user information in a user object which it serializes and then stores in a local file. In this case, some knowledge of the host app allows our code to deserialize the user object and get her email, date of birth, and first name. Deserializing the user object also provided the library the user password in plain text.

Thus, if an opportunistic advertising library is included in apps like these, then a careful manual review of the apps will reveal some pathways to targeted data. At this point it helps to have a little more terminology. Let us say that a *data point* is a category of targeted data point values. For example, gender is a data point, whereas knowing that Bob is a male is a data point value. What we would like to do, is examine a collection of apps to see what data points they expose to ad libraries.

To explore these ideas and their refinement I develop three datasets listed in the first three rows of Table 4.1. For the first, I make a list of the 100 most popular free apps in each of the 27 categories on Google Play to get 2700 apps. After removing duplicate apps, we are left with 2535 unique apps. We can call this the *Full Dataset, FD*. From these I randomly selected 300 apps for manual review. From these apps I removed the ones that crashed on our emulator or required the use of Google Play Services. We will refer to this as the *Level One Dataset (L1)*. On this dataset, I searched for data point exposure by two means. First, I inspected the manifest to see if the permissions themselves would suggest that certain types of data points would be present. For example, we can predict that the address attribute could be derived by the library if the host app is granted the `ACCESS_COARSE_LOCATION` or the `ACCESS_FINE_LOCATION` permission, the email attribute from the `GET_ACCOUNTS` permissions, the phone attribute from the `READ_PHONE_STATE` permission and the online search from the `READ_HISTORY_BOOKMARKS` permission. Second, I launched the app, looked to see what local files it produced, and looked into these files to see if they expose any particular data points.

The data points we consider must include user data that the ad libraries are likely interested in harvesting. To this end, I extract data points mostly based on a calculator provided by the Financial Times (FT) [127]. This calculator provides illustrative information sought by data brokers together with an estimate of its financial value in the U.S. based on analysis of industry pricing data at the time the calculator was created. For example, according to the FT calculator, basic demographic information like age and gender are worth about \$.007. If an opportunistic advertising network can learn that a user is (probably) an accountant, then the cumulative information is worth \$.079 (according to the calculator); if they

Table 4.1: Datasets

Name	Number	Description
Full Dataset (FD)	2535	Unique apps collected from the 27 Google Play categories.
Level One Dataset (L1)	262	Apps randomly selected from FD.
Level Two Dataset (L2)	35	Apps purposively selected from L1.
App Bundle Dataset (ABD)	243	App bundles collected through survey.

also know that this accountant is engaged to be married, this increases the value to $.179. Engaged individuals are valuable because they face a major life change, are likely to both spend more money and change their buying habits. An especially noteworthy data point is a pregnancy. This is well illustrated by events surrounding Target's successful program to use the habits of registered expecting shoppers to derive clues about unregistered ones in order to target them with advertising about baby care products [129]. The FT calculator provides us with a realistic way of exploring the relative value of an information gathering strategy. The precise figures are not important, and have probably changed significantly since the introduction of the calculator, but they give some ballpark idea of value and the system provides a benchmark for what a more accurate and detailed database of its kind might use.

I abstracted the questionnaire-like attributes from the FT calculator into keywords and used these as a guide to data points to find in the apps reviewed. For example, I transformed the question "Are you a fitness and exercise buff" into "workout". We refer to the overall attack technique that examines local files and uses protected APIs, as a *level one inspection* (L1-I). I found 29 categories of data points in L1 by this means, including 'gender', 'age', 'phone number', 'email address', 'home address', 'vehicle', 'online searches', interests like 'workout' and others. Table 4.2 depicts some popular apps and the data points they expose to ad libraries performing a *level one inspection.*

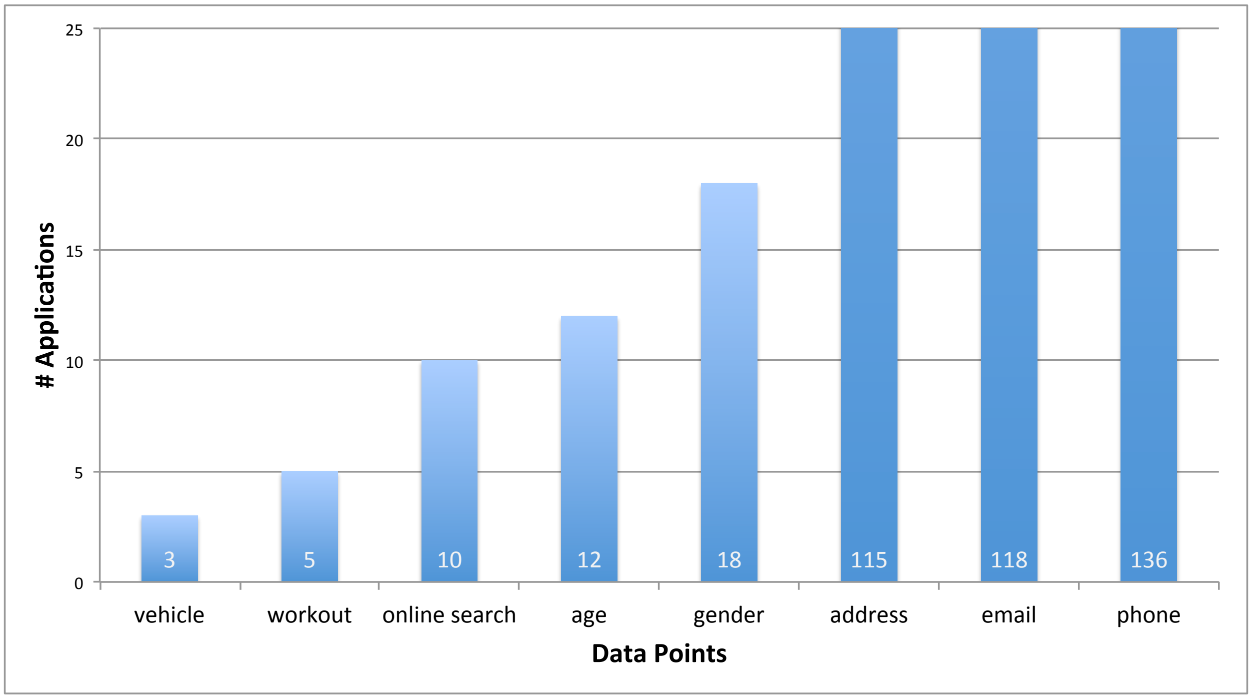
However, an ad library could also utilize the fact that it can *eavesdrop on user inputs* in its host app. This can be done on Android by exploring the resource files of packages.

Table 4.2: Data exposure from popular apps to ad libraries performing level-one (L1-I) and level-two (L2-I) inspection.

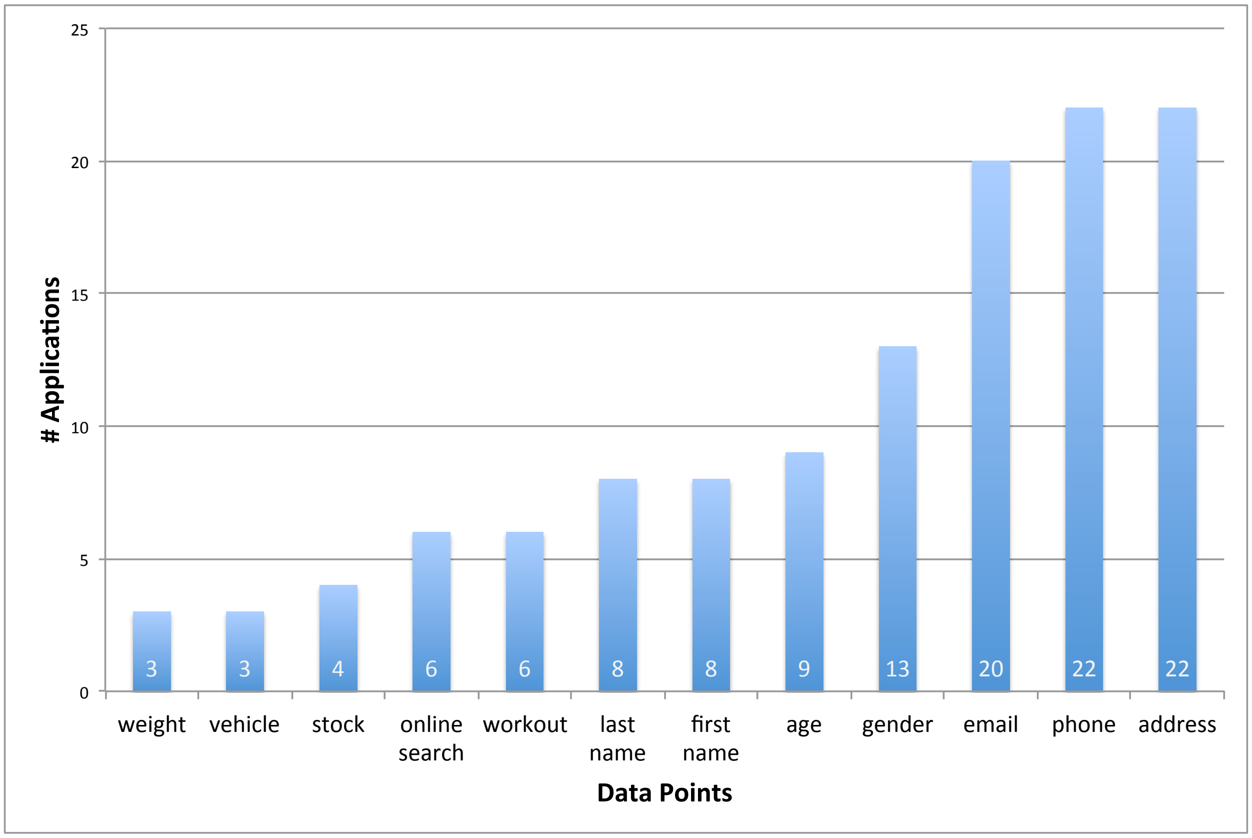
Attack Strategy	**Category**	**App Name**	**Num. of Installation**	**Exposed Data Points**
L1-I	MEDICAL	Menstrual Calendar	$1\times10^6 - 5\times10^6$	pregnancy, trimester, headache
L1-I	EDUCATION	myHomework Student Planner	$1\times10^6 - 5\times10^6$	gender, age, address
L2-I	HEALTH & FITNESS	Run with Map My Run	$5\times10^6 - 10\times10^6$	phone, email, first name, last name, age, gender, address, workout
L2-I	LIFESTYLE	BeNaughty - Online Dating App & Call	$5\times10^6 - 10\times10^6$	phone, email, age, gender, address, marital status, parent

Once an interesting layout file is found, an offensive library can inflate the layout from the library package and read from its UI elements. With this strategy, the ad library can find targeted data that are input by the user but not necessarily kept in local files. Let us call the attack strategy that utilizes not only local files and protected APIs, but also user input eavesdropping, a *level two inspection* (L2-I). To better understand what data points are exposed to an ad library performing a level two inspection, I selected 35 of the apps in the L1 dataset and reviewed them manually to find data points that level two inspection could reveal. Lets call this the *L2 dataset.* The 35 apps in question are ones that exposed one or more data points other than ones derived from the manifest. We make this restriction to assure that there was no straight-forward strategy for finding data points in these apps so we could better test the automated inference techniques we introduce later. Table 4.2 depicts some popular apps and the data points they expose to ad libraries performing a *level two inspection.* We observe that apps expose not only demographic information but also more sensitive data such as user health information. The complete list of apps and the data points they expose is omitted due to space limitations.

Figure 4.1a displays the number of apks in the level one inspection that were found to expose the basic data points we listed earlier. Figure 4.1b portrays a similar graph for the level two inspection. Here, I prune all data points with frequency less than three. We observe that data points that can be derived by exploiting the host app's permissions are more prevalent than other ones. This is because the permissions are coarse-grained and app developers are likely to use them for a number of reasons, whereas other data points would be present only if the host app is explicitly collecting that information. Overall, it is clear that targeted data is exposed by apps through in-app attack channels to ad libraries. Next I will examine exposure through out-app channels.



(a)



(b)

Figure 4.1: Number of apps with data points inferred by (a) level one inspection of L1, (b) level two inspection of L2.

4.2.3 Data Exposure through Out-App Shared Process Capabilities

Ad libraries can surreptitiously access targeted data not only through in-app attack channels but also from host-app-independent channels such as public APIs. Such APIs are considered to be harmless and thus made available to all applications on the platform without the need of special permissions. In particular, Android provides a pair of publicly available functions, which we will abbreviate as getIA and getIP, that return app bundles, the list of installed apps on a mobile.[2] They can be used by the calling app to find utilities, perform security checks, and other functions. They also have high potential for use in advertising. An illustration of this is the Twitter *app graph* program [130], which was announced in late 2014. Twitter asserted its plans to profile users by collecting their app bundles[3] to "provide a more personal Twitter experience for you." Reacting to Twitter's app graph announcement, the Guardian newspaper postulated [131] that Twitter "reported $320m of advertising revenues in the third quarter of 2014 alone, with 85% of that coming from mobile ads. The more it can refine how they are targeted, the more money it will make." This progression marks an important point about the impact of advertising on privacy. Both the Financial Times [127] and a book about the economics of the online advertising industry called The Daily You [132] emphasize the strong pressures on the advertising industry to deliver better quality information about users in a market place that is both increasingly competitive and increasingly capable. This is a key insight of this chapter: what may seem opportunistic now may be accepted business practice and industry standard in a few years, and what is viewed as malicious today may be viewed as opportunistic or adventurous tomorrow. Twitter provides warnings to the user that Twitter will collect app bundles and offers the user a chance to opt out of this. Other parties are less forth-coming about their use of this technique of user profiling.

Use of App Bundles

Getting app bundles is a great illustration of the trajectory of advertising on mobiles. In 2012 the AdRisk tool [3] showed that 3 of 50 representative ad libraries it studied would collect the list of all apps installed on the device. The authors viewed this as opportunistic at best at the time. But what about now? We did a study of the pervasiveness of the use of app bundles by advertising networks in Google Play. The functions getIA and getIP are

[2]Their formal names are `getInstalledApplications` and `getInstalledPackages`. The first returns the applications, the second returns the packages and, from these, one can learn the application names.

[3]We use the term app bundle rather than app graph because we do not develop a graph from the app lists.

built into the Android API and require no special permissions. We decompiled the 2700 apps we have collected from Google Play, into smali code [4] for analysis and parsed these files to look for the invocations of getAP and getIP in each app. This allowes us to narrow the set of apps for analysis to only those that actually collect a list of apps on the mobile, which we deem an app bundle. I then conducted a manual analysis of the invocation of these functions by ad libraries.

Of the 2700 apps selected for review, 165 apps were duplicates, narrowing our sample size down to 2535 distinct apps. Of these, 27.5% (679/2535) contained an invocation of either of the two functions. This total includes invocation of these functions for functional (utility and security) as well as advertising purposes. To better understand if an ad library invokes the function, analysis required a thorough examination of the location of the function call to see if it is called by an advertising or marketing library. I found that many apps pass information to advertisers and marketers. This analysis is conducted manually to best capture a thorough list of invocations within ad libraries. Ultimately 12.54% of the examined apps (318/2535) clearly incorporate ad libraries that invoke one of the functions that collects the app bundle of the user. I found 28 different ad libraries invoking either getIA or getIP. These results do not necessarily include those apps that collect app information themselves and pass it to data brokers, advertising or marketing companies, or have their own in-house advertising operation (like Twitter). These results demonstrate that many types of apps have ad libraries that collect app bundles, including medical apps and those targeted at children. Interestingly, I did not detect collection of app bundles by the three ad networks identified by AdRisk. However, a number of other interesting cases emerged.

`Radio Disney`, for example, uses Burstly, a mobile app ad network whose library [5] calls getIP. Disney's privacy policy makes no direct reference to the collection of app bundles for advertising purposes. Use of this technique in an app targeted at children is troubling because it might collect app bundle information from a child's device without notifying either the parent who assisted the download or an older child that this type of information is collected and used for advertising purposes. Disney does mention the collection of "Anonymous Information" but the broad language defining this does not give any indication that the Radio Disney app collects app bundles.[6]

[4]The smali format is a human-readable representation of the application's bytecode.

[5]`burstly/lib/apptracking/AppTrackingManager.smali`

[6]Formally, they define anonymous information as "information that does not directly or indirectly identify, and cannot reasonably be used to identify, an individual guest." App bundles are similar to movie play lists; it is debatable whether they indeed satisfy this definition.

`Looney Tunes Dash!` is a mobile app provided by Zynga that it explicitly states that they collect "Information about ... other third-party apps you have on your device."[7] In fact, this is the privacy policy for all Zynga apps.

Several medical apps (12) collect app bundles. Most surprisingly, `Doctor On Demand: MD & Therapy`, an app which facilitates a video visit with board-certified physicians and psychologists collects app bundles through the implementation of google/ads/ conversion tracking. However, their linked privacy policy makes no reference to passing any user information to advertisers. Other apps in the medical category with advertising libraries that collect app bundles include ones that track ovulation and fertility, pregnancy, and remind women to take their birth control pill.

Survey Study

Upon learning of the prevalence of the app bundle collection by advertisers, we need to better understand what type of information could be learned by advertisers based on the list of apps on a user's mobile device. To do this, we can devise a study that would allow us to collect our own set of app bundles to train a classifier.

The study consisted of a survey and an Android mobile app launched on the Google Play Store. The protocol for all the parts of the study was approved by the Institutional Research Board (IRB) for our institution. All participants gave their informed consent. We required informed consent during both parts of the study, and participants could leave the study at any time. Participants were informed that the information collected in the survey and the information collected by the mobile app would be associated with one another.

Participants included individuals over the age of 18 willing to participate in the survey and who owned an Android device. Crowdsourcing platforms such as Amazon's Mechanical Turk are proven to be an effective way to collect high quality data [133]. The survey was distributed over Microworkers.com a comparable crowdsourcing platform to Amazon's Mechanical Turk (MTurk). We chose Microworkers.com over Amazon Mechanical Turk because Amazon Mechanical Turk did not allow tasks that involve requiring a worker to download or install any type of software.

Moreover, I designed the mobile app, AppSurvey, to collect the installed packages on a participant's phone. The study directed the participant to the Google Play Store to download the mobile app. Upon launching AppSurvey, a pop-up screen provided participants information about the study, information to be collected, and reiterated that the participation in the study was anonymous and voluntary. If the participant declined the consent,

[7] `https://company.zynga.com/privacy/policy`

no information would be collected. If the participant consented, the app uploaded the app bundles from the participants phone and anonymously and securely transmit it to our server. AppSurvey also generated a unique User ID for each individual which participants were instructed to write down and provide in the survey part of the study. Finally, AppSurvey prompted participants to uninstall the mobile app.

The survey is designed based upon the FT calculator. Specifically, it consisted of 25 questions about basic demographic information, health conditions, and Internet browsing and spending habits. The survey also contained two control questions included to identify survey participants not paying sufficient attention while taking the survey. If either of these questions were answered incorrectly, we excluded the survey response. In addition, the workers were not compensated until after the finished tasks were reviewed and approved by the survey conductors. Before taking the survey, participants were required to give informed consent to the information collected in the survey. To link the app bundle information collected by AppSurvey to the responses provided by participants in the survey, participants were required to input the unique User ID generated by AppSurvey. The collection of this data allows us to establish a ground truth for users' app bundles.

The survey resulted in answers and app bundle information from 243 participants., and 1985 total distinct package names.

4.3 DETECTION DESIGN

The analysis in the previous section highlights the need for detecting information exposure to ad libraries through shared process privileges. To this end I have design Pluto. Pluto is a modular framework for estimating in-app and out-app targeted data exposure for a given app. In-app Pluto focuses on local files that the app generates, the app layout and string resource files, and the app's manifest file. Out-app Pluto utilizes information about app bundles to predict which apps will be installed together and employs techniques from machine learning to make inferences about users based on the apps they have on their mobile. I describe each of these in a pair of subsections.

4.3.1 In-app Pluto

In-app Pluto progresses in two steps as illustrated in Figure 4.2. First, the Dynamic Analysis Module (`DAM`) runs the given app on a device emulator and extracts the files the app creates. Then it decompiles the app and extracts its layout files, resource files, manifest file and runtime generated files. At the second step, the files produced by the DAM are

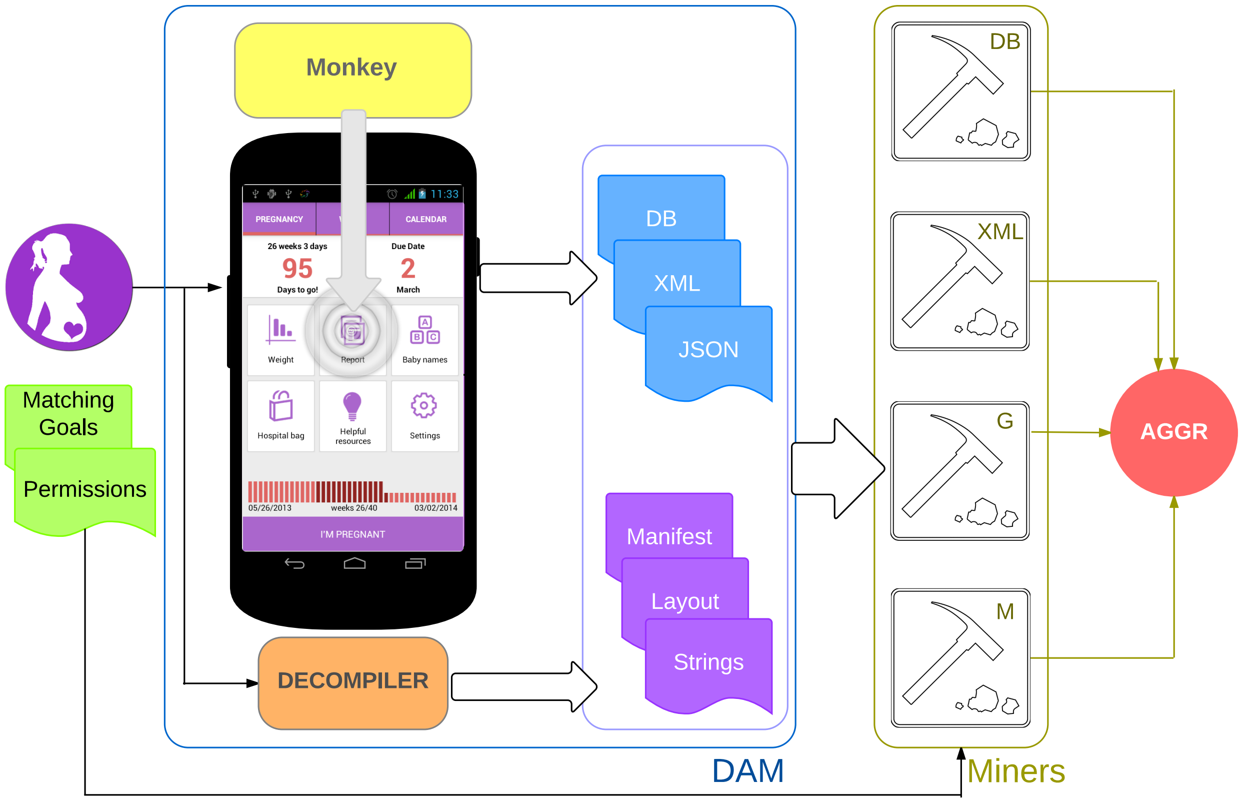


Figure 4.2: Design of In-app Pluto

fed to a set of file miners. The file miners utilize a set of user attributes and user interests, possibly associated with some domain knowledge, as a matching goal. A miner will reach a matching goal when it decides that a data point is present in a file. When all the app's files are explored, the Aggregator (AGGR) removes duplicates from the set of matching goals and the resulting set is presented to the analyst. Pluto's in-app component's goal is to estimate offline, the exposure of targeted data—or data points—to ad libraries at runtime. In-app Pluto can be configured to estimate data points for a level 1 aggressive library by looking only at the runtime generated files and available permissions. To perform exposure discovery for a level 2 of aggression, it mines targeted data also from the resource and layout files. In essence Pluto is trying to simulate what an ad library is allowed to do to estimate what is the potential data exposure from a given app. To perform in-app exposure discovery, Pluto employs dynamic analysis and natural language processing techniques to discover exposure of in-app data points. Here I report on a prototype implementation focusing on manifest, SQLite, XML, and JSON files.

Dynamic Analysis

To discover the files that an app is generating at runtime, Pluto runs the app on an emulator for 10 seconds and then uses a monkey tool to simulate user input. [8] This can generate pseudo-random streams of clicks, touches, and system-level events. I chose to use a monkey because some apps might require user stimulation before generating some of their local files. To validate this assumption, we performed two experiments. First, I configured Pluto's DAM module to run all 2535 apps in the FD dataset for 10 seconds each. I repeat the experiment, this time configuring DAM to issue 500 pseudo-random events to each app after its 10 second interval is consumed. As we see on Table 4.3, Pluto explores approximately 5% more apps in the second case. [9] More importantly, DAM_Monkey generates 1196 more files than DAM which results in 100 apps with 'interesting' files more. Android's Monkey was previously found to achieve approximately 25.27% LOC coverage [135]. However, Pluto's components can be easily replaced, and advances in dynamic analysis can be leveraged in the future. For example, PUMA [136] is a very promising dynamic analysis tool introduced recently. If new levels of library aggression are introduced in the future, PUMA could be used instead of Android's monkey to better simulate behaviors that can allow libraries to access user attributes at runtime.

Table 4.3: DAM's coverage. * denotes interesting files (SQLite, XML, JSON)

DA Strategy	**% successful experiments**	**#files**	**# *files**	**#of apps w/ *files**
DAM	0.718	14556	9083	1911
DAM Monkey	0.763	15752	10171	2021

Once the execution completes, DAM extracts all the 'runtime' generated files. Subsequently, it decompiles the input android app package (apk) and extracts the Android layout files, Android String resources and the app's manifest file.

File Miners empowered by Natural Language Processing

Once the DAM module generates 'runtime' files, Pluto's enabled file miners commence their exploration. I have implemented four types of file miners in the prototype: `MMiner`; `GMiner`; `DBMiner`; `XMLMiner`. The MMiner is designed to parse manifest files, the DBMiner for SQlite database files, the XMLMiner for runtime generated XML files and the GMiner

[8] In our implementation we used the Android SDK-provided `UI/Application Exerciser Monkey` [134].
[9] An unsuccessful experiment includes apps that failed to launch or crashed during the experiment.

is a generic miner well suited for resource and layout files. The miners take as input, a set of data points, [10] in the form of noun words and a mapping between permissions and data points that can be derived given that permission.

Input processing: Pluto utilizes Wordnet's English semantic dictionary [37] to derive a set of synonyms for each data point. However, a word with multiple meanings will result in synonyms not relevant to Pluto's matching goal. Consider for example the word `gender`. In Wordnet, gender has two different meanings: one referring to grammar rules and the one referring to reproductive roles of organisms. In our case it is clear that we are interested in the latter instead of the former. In this prototype, the analyst must provide Pluto with the right meaning. While it is trivial to make this selection, for other data points it might not be as trivial. For example, `age` has 5 different meanings in Wordnet. Other data points which we have not explored, might have even more complex relationships. *Visuwords.com* is a helpful tool which can be used to visualize such relationships and immensely facilitated such selections. For example, the list of data points in the FT calculator, is indeed feasible to analyze manually. However, Pluto does not require this from an analyst. If the meaning is not provided, Pluto will take all synonym groups into account with an apparent effect on precision.

NLP in Pluto: The NLP community developed different approaches to parse sentences and phrases such as `Parts of Speech` (POS) Tagging and `Phrase and Clause Parsing`. The former can identify parts of a sentence or phrase (i.e., which words correspond to nouns, verbs, adjectives or prepositions), and the latter can identify phrases. However, these cannot be directly applied in our case because we are not dealing with well written and mostly grammatically correct sentences. In contrast, Pluto parses structured data written in a technically correct way (e.g., .sqlite, .xml files). Thus in our case we can take advantage of the well-defined structure of these files and extract only the meaningful words. For the database files, potentially meaningful words will constitute the table name and the columns names. Unfortunately, words we extract might not be real words. A software engineer can choose anything for the table name (or filename), from `userProfile`, `user_profile`, `uProfil`, to `up`. We take advantage of the fact that most software engineers do follow best practices and name their variables using the first two conventions, the `camelCase` (e.g. userProfile) and the `snake_case` structure (e.g. user_profile). The processed extracted words are checked against Wordnet's English semantic dictionary. If the word exists in the dictionary, Pluto derives its synonyms and performs a matching test against the data points and their synonyms. [11]

[10]We derived most of the data points from the FT calculator [127].

[11]In our prototype we used the JWI [137] interface to Wordnet, to derive sets of synonyms.

If a match is determined, then a disambiguation layer decides whether to accept or reject the match. Next, I elaborate on the functions of the disambiguation layer.

Context Disambiguation Layer: Words that reach a matching goal, could be irrelevant with the actual user attribute. Consider for example the word `exercise`. If a Miner unearths that word, it will be matched with the homonymous synonym of the matching goal `workout`. However, if this word is found in the Strings resource file that doesn't necessarily mean that the user is interested in fitness activities. It could be the case that the app in question is an educational app that has exercises for students. On the other hand, if this word is mined from an app in the `Health and Fitness` Google Play category, then it is more likely this is referring to a fitness activity. Pluto employs a disambiguation layer that aims to determine whether the match is valid. It attaches to every user interest the input app's Google Play category name. We call that a `disambiguation term`. For user attributes, the disambiguation term is currently assigned by the analyst [12]. In addition, Pluto assigns some `domain knowledge` to data points. For attributes, it treats the file name or table name as the domain knowledge, and for interests it uses the matching goal itself. The prototype's context disambiguation layer calculates the similarity between the `disambiguation term` and the `domain knowledge`. If the similarity value is found to surpass a specific threshold, then the match is accepted.

The NLP community already proposed numerous metrics for comparing how similar or related two concepts are. The prototype can be configured to use the following existing similarity metrics to disambiguate attribute matches: `PATH [138]`; `LIN [139]`; `LCH [38]`; `LESK [140]`. Unlike the first three metrics which are focused on measuring an `is-a` similarity between two words, LESK is a definition-based metric of relatedness. Intuitively this would work better with user interests where the disambiguation term is the app's category name. The other metrics are used to capture `is-a` relationships which cannot hold in most of the user-interests cases. For example, there is no strong is-a relationship connecting the user interest `vehicle` with the category transportation. [13] LESK seems well fit to address this as it depends on the descriptions of the two words. Indeed, LESK scores the (vehicle, transportation) pair with 132 with (vehicle, travel and local) coming second with 103.

However, in this study I found that LESK might not always work that well when applied in this domain. Studying the scoring of LESK with respect to one of our most popular user interests in our L1 dataset we found it to be problematic. When comparing the matching goal `workout` with the category `Health and Fitness`, LESK assigns it one of the lowest

[12] We used the word `Person`.

[13] We found that similarity metrics that find these relationships do not assign the best score to the pair(vehicle, transportation) when compared with other (vehicle, *) pairs.

Table 4.4: Comparison between rankings of (interest, category name) pairs from LESK and droidLESK. TF denotes the data point term frequency in local files created by apps in a category.

DATA POINT	RANK	LESK	TF	TF*LESK
VEHICLE	1	TRANSPORTATION	FINANCE	TRANSPORTATION
VEHICLE	2	BOOKS AND REFERENCES	TRANSPORTATION	FINANCE
VEHICLE	3	TRAVEL AND LOCAL	LIFESTYLE	LIFESTYLE
WORKOUT	1	BOOKS AND REFERENCES	HEALTH AND FITNESS	HEALTH AND FITNESS
WORKOUT	2	TRAVEL AND LOCAL	APP WIDGET	NEWS AND MAGAZINE
WORKOUT	3	MUSIC AND AUDIO	NEWS AND MAGAZINE	APP WIDGET

scores (33), with the maximum score assigned to the (workout, books and references) pair (113).

Here I present a new improved similarity metric that can address LESK's shortcomings when applied to our problem. I call our similarity metric `droidLESK`. The intuition behind droidLESK is that the more frequently a word is used in a category, the higher the weight of the (word, category) pair should be. droidLESK is then a normalization of $freq(w, c) \times LESK(w, c)$. In other words, droidLESK is the weighted LESK were the weights are assigned based on term frequencies. To evaluate droidLESK, I create pairs of the matching goal `workout` with every Google Play category name and assign a score to each pair as derived from droidLESK and other state of the art similarity metrics. To properly weight LESK and derive droidLESK, I perform a term frequency analysis of the `workout` word in all 'runtime' generated files of the L1 dataset. I repeat the experiment for the word `vehicle`. droidLESK's scoring was compared with the scores assigned to the pairs by the following similarity metrics: `WUP` [141]; `JCN` [142]; `LCH` [38]; `LIN` [139]; `RES` [143]; `PATH` [138]; `LESK` [140] and `HSO` [144].

The results are very promising—even though preliminary—as shown in table 4.4. [14] We observe that the proposed technique correctly assigns the highest score to the pair (workout, health and fitness) than any other pair (workout,*). The same is true for the pair (vehicle, transportation). droidLesk was evaluated on the two most prevalent user interests in our dataset. Since this approach might suffer from over-fitting, in future work I plan to try this new metric with more words and take into account the number of apps contributing to the term frequency. I further discuss the effects of using droidLESK in Pluto's in-app targeted data discovery in the evaluation Subsection 6.4.

[14]Due to space limitations, I omit uninformative comparisons.

4.3.2 Out-app Pluto

Out-app Pluto aims to estimate what is the potential data exposure to an ad library that uses the unprotected public gIA and gIP APIs. That is, given the fact that the ad library can learn the list of installed applications on a device, it aims to explore what data points, if any, can be learned from that list. Intuitively, if an ad library knows that a user installed a pregnancy app and local public transportation app, it would be able to infer the user's gender and coarse location. However, the list of installed applications derived from gIA and gIP is dependent on the device the ad library's host app is installed, which renders estimation of the exposure challenging. To explore what an ad library can learn through this out-app attack channel, I derive a set of co-installation patterns that reveals which apps are usually installed together. This way we can simulate what the runtime call to gIA or gIP will result in given invocation from an ad library incorporated into a particular host app. I then feed the list of co-installed applications into a set of classifiers we trained to discover the potential data exposure through the out-app channel.

The Pluto out-app exposure discovery system runs machine learning techniques on a corpus of app bundles to achieve two goals. First, it provides a Co-Installation Pattern module (CIP) which can be updated dynamically as new records of installed apps are received. The CIP module runs state-of-the-art frequent pattern mining (FPM) algorithms on such records to discover associations between apps. For example, such an analysis can yield an association in the form of a conditional probability, stating that if app A is present on a device then app B can be found on that device with $x\%$ confidence. When an analyst sets Pluto to discover out-app targeted data regarding an app offline, Pluto utilizes the CIP module to get a good estimation of a vector of co-installed apps with the target app. The resulting vector is passed to the classifiers which in turn present the analyst with a set of learned attributes. Second, it provides a suite of supervised machine learning techniques that take a corpus of app bundles paired with a list of user targeted data and creates classifiers that predict whether an app bundle is indicative of a user attribute or interest.

Co-Installation Patterns

The CIP module uses frequent pattern mining to find application co-installation patterns. This can assist Pluto in predicting what will an ad library learn at runtime if it invokes gIA or gIP. We call a co-installation pattern, the likelihood to find a set of apps installed on a device in correlation with another app installed on that device. In FPM, every transaction in a database is identified by an id and an `itemset`. The itemset is the collection of one or more items that appear together in the same transaction. For example, this could be the items bought together by a customer at a grocery store. `Support` indicates the frequency of an itemset in the database. An FPM algorithm will consider an itemset to be frequent if its support is no less than a minimum support threshold. Itemsets that are not frequent are pruned. Such an algorithm will mine `association rules` including frequent itemsets in the form of conditional probabilities that indicate the likelihood that an itemset can occur together with another itemset in a transaction. The algorithm will select rules that satisfy a measure (e.g., a minimum confidence level). An association rule has the form N:N, where N is the number of unique items in the database. An association rule is presented as $X \Rightarrow Y$ where the itemset X is termed the `precedent` and Y the `consequent`. Such analysis is common when stores want to find relationships between products frequently bought together.

Pluto's CIP uses the same techniques to model the installations of apps on mobile devices, as itemsets bought together at a grocery store. Our implementation of Pluto's CIP module uses the FPGrowth [145] algorithm, a state of the art frequent pattern matching algorithm for finding association rules. I have chosen FPGrowth because it is significantly faster than its competitor Apriori [146]. CIP runs on a set of app bundles collected periodically from a database containing user profiles that include the device's app bundles and derives a set of association rules, indicating the likelihood that apps can be found co-installed on a device. Our CIP association rule will have the form 1:N because Pluto is interested in finding relationships between a given app and a set of other apps.

CIP uses `confidence` and `lift` as the measures to decide whether an association rule is strong enough to be presented to the analyst. Confidence is defined as $conf(X \Rightarrow Y) = \frac{supp(X \cup Y)}{supp(X)}$, where $supp(X)$ is the support of the itemset in the database. A confidence of 100% for an association rule means that for 100% of the times that X appears in a transaction, Y appears as well in the same transaction. Thus an association rule $facebook \Rightarrow skype, viber$ with 70% confidence will mean that for 70% of the devices having `Facebook` installed, `Viber` and `Skype` are also installed.

Another measure CIP supports is Lift. Lift is defined as: $lift(X \Rightarrow Y) = \frac{supp(X \cup Y)}{supp(X) \times supp(Y)}$. Lift indicates how independent the two itemsets are in the rule. A Lift of one will indicate that the probability of occurrence of the `precedent` and `consequent` are independent of each other. The higher the Lift between the two itemsets, the stronger the dependency between them and the strongest the rule is.

Learning Targeted Data from App Bundles

Pluto uses supervised learning models to infer user attributes from the CIP-estimated app bundles. Pluto aims to resolve two challenges in training models based on app bundles: 1) skewed distribution of values of attributes; 2) high dimensionality and highly sparse nature of the app bundles.

Balancing distributions of training sets: Based on the empirical data collected, some attributes have a more skewed distribution in their values. To orient the reader using a concrete example, consider an example where 1 of 100 users has an allergy. In predicting whether a user has an allergy in this dataset, one classifier can achieve an accuracy of 0.99 by trivially classifying each user as having an allergy. In view of this, for the attribute "has an allergy" the value "yes" can be assigned a higher weight, such as 99, while the value "no" has a weight of 1. After assigning weights, the *weighted accuracy* for predicting an attribute now becomes the weighted average of accuracy for each user; the weight for a user is the ratio of the user's attribute value weight to the total attribute value weights of all users. Therefore, in this example, the weighted accuracy becomes 0.5, which is fair, even when trivially guessing that each user has the same attribute value. In order to train an effective model for Pluto, the distribution of training sets is balanced following the aforementioned idea. To balance we adjust the weights of existing data entries to ensure that the total weights of each attribute value are equal. In this way, the final model would not trivially classify each user to be associated with any same attribute value. Accordingly, I will adopt measures *weighted precision* and *weighted recall* in the evaluation where the total weights of each attribute value are equal; this is to penalize trivial classification to the same attribute value [147].

Dimension reduction of app-bundle data: Another challenge we face in this context is the high dimensionality and highly sparse nature of the app bundles. There are over 1.4 million apps [148] on Google Play at this moment, and it is both impractical and undesirable for the users to download and install more than a small fraction of those on their devices. A recent study from Yahoo [149] states that users install on average 97 apps on a device. To make this problem more tractable I used a technique borrowed from the Machine Learning community which allows us to reduce the considered dimensions. The prototype employs three classifiers, namely K-Nearest Neighbors (KNN), Random Forests, and SVM.

To apply these classifiers to our data, each user u_i in the set of users U is mapped to an app installation vectors $\mathbf{a}_{u_i} = \{a_1, \ldots, a_k\}$, where $a_j = 1$ ($j = 1, \ldots, k$) if u_i installs a_j on the mobile device, otherwise $a_j = 0$. Note that the app installation vector is k-dimensional and k can be a large value (1985 in our study). Thus, classifiers may suffer from the "curse of dimension" such that the computation could be dominated by less relevant installed apps when the dimension of space goes higher. To mitigate this problem, we can use principal component analysis (PCA) by selecting a small number of the principal components to perform dimension reduction before applying a classifier.

4.4 DETECTION EVALUATION

In this section I evaluate Pluto's components in estimating data exposure. We will first evaluate Pluto's performance to discover Level-1 and Level-2 in-app data points. Next we will apply Pluto's CIP module and classifiers on real world data app bundles and the collected ground truth, and evaluate their performance.

4.4.1 Evaluation of Pluto's in-app exposure discovery

In this section I present empirical findings on applying Pluto on real world apps.

Experimental setup: I provided Pluto with a set of data points to look for, enhanced with the meaning—sense id of the data point in Wordnet's dictionary—and the class of the data point (i.e., user attribute or user interest). I also provide Pluto with a mapping between permissions and data points and we configured it to use the `LCH` similarity metric at the disambiguation layer for user attributes and our `droidLESK` metric for user interests. I found that setting the LCH threshold to 2.8 and the droidLESK threshold to 0.4 provides the best performance. To tune the thresholds, I parameterized them and ran Pluto multiple times on the L1 dataset. A similar approach can be used to tune the thresholds on any available - ideally larger - dataset [15], and data point set. In all experiments, all Miners were enabled unless otherwise stated. The MMiner mined in manifest files, the DBMiner in runtime-generated database files, the XMLMiner in runtime-generated XML files and the GMiner in String resource files and layout files. I compared Pluto to the level-1 and level-2 ground truth we manually constructed as described in Section 4.2.

In-app exposure estimation: I ran Pluto on the set of 262 apps (Pluto L1) and the full set of 2535 apps (Pluto FD). Figure 4.3 plots the distribution of apps with respect to data points found within those apps. I observed that the number of data points found in apps remains consistent as we increased the number of apps. I repeated the experiment for the level-1 dataset that consists of 35 apps. Figure 4.4. depicts Pluto's data point discovery.

[15]Note that it requires little effort to get Android app packages.

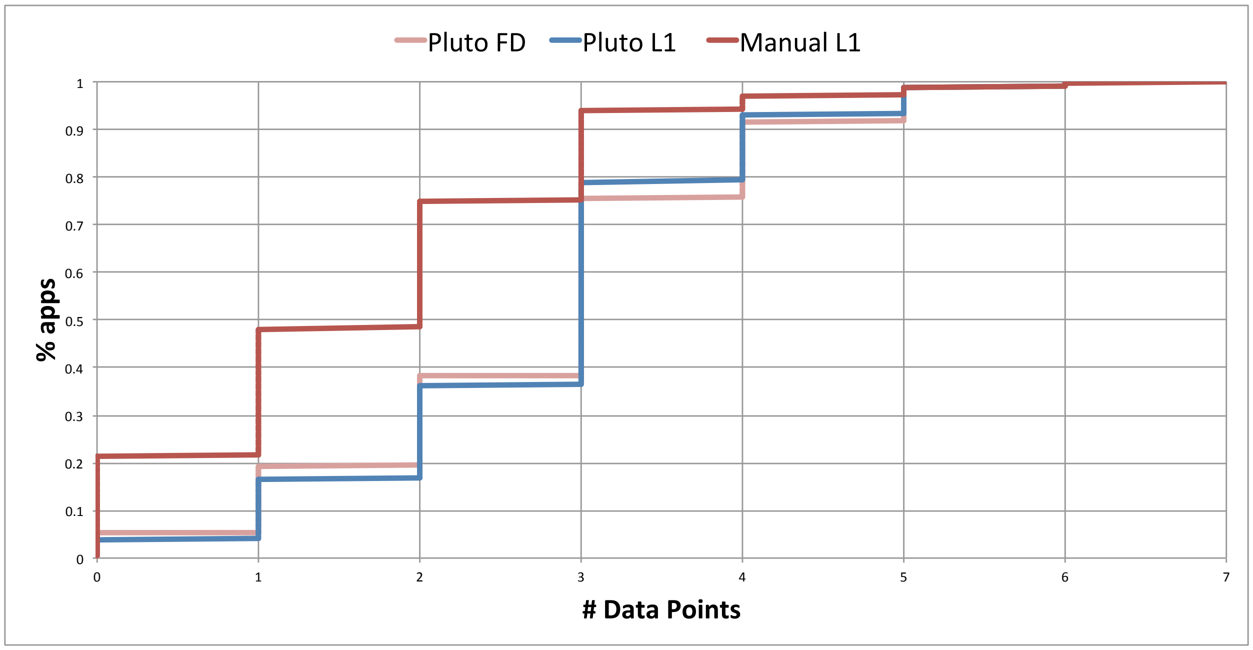


Figure 4.3: CDF of apps and number of data points (level-1)

I compared Pluto's data point prediction with the respective level-1 and level-2 manual analysis 4.2.

Evidently, Pluto is optimistic in estimating in-app data points. In other words, Pluto's in-app discovery component can flag apps as potentially exposing data points, even though these are not actually there. A large number of Pluto's false positives stem from parsing the String constants file. Parsing these files increases coverage by complementing our dynamic analysis challenge in generating files that host apps created after the user logged in. It also addresses the layer-2 aggressive libraries can read from the user input. However, this results in considering a lot of extra keywords that might match a data point or its synonyms. Their location in the Strings.xml makes it harder for Pluto to disambiguate the context for certain data point classes. In this study, I makes the first attempt towards mitigating this pathology by proposing droidLESK.

Pluto is designed to find user attributes, user interests, and data points stemming from the host app's granted permissions. Next, I present the performance of Pluto's prototype implementation with respect to the above categories.

Finding user-attributes: Figure 4.5 depicts the performance of Pluto in finding the data point `gender` when compared to the level-1 and level-2 datasets and Figure 4.6 shows the same for the user attribute `age`. Gender had absolute support of 13 in the level-1 dataset and 18 in the level-2 and `age` had 12 and 9 respectively. We observe that Pluto is doing better in discovering data points available to the more aggressive libraries. For example,

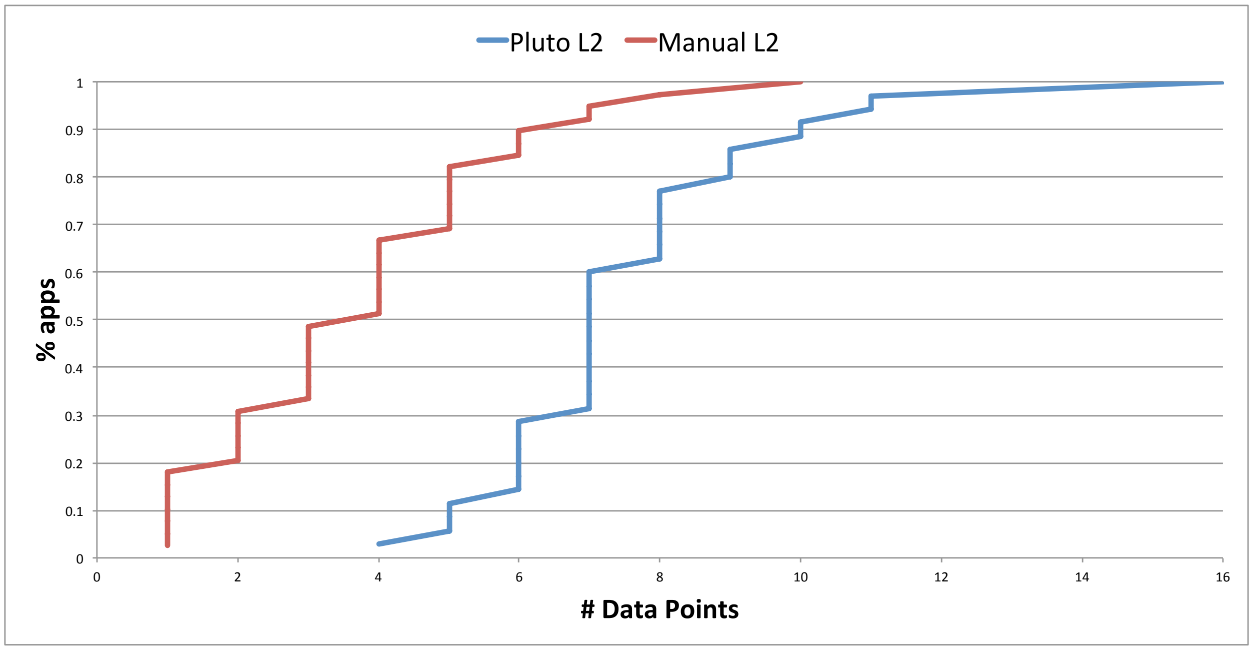


Figure 4.4: CDF of apps and number of data points (level-2)

the word age, was found in a lot of layout files and Strings.xml files while the same was not present in the runtime generated files. Comparing age with the level-1 ground truth, results in a high number of false positives, since the analyst has constructed the ground truth for a level-1 aggressive library. When Pluto is compared with the ground truth for a level-2 aggressive library, its performance is significantly improved.

Finding interests: Next, I evaluated Pluto's performance in discovering user interests. Figure 4.7 illustrates the user interest `workout` when Pluto is compared against the level-1 ground truth and the level-2 ground truth. Workout had absolute support of 5 in the level-1 dataset and 6 in the level-2. Again, Pluto does much better in the latter case for the same reasons stated before.

Preliminary results for droidLESK: In the experiments I used droidLESK as the most appropriate similarity metric on Pluto's context disambiguation layer for user interests. I compared that with an implementation of Pluto with no disambiguation layer and an implementation that uses the LESK metric. `droidLESK` achieved an astonishing 103.3% increase in Pluto's precision whereas LESK achieved an improvement of 11.37%. This is a good indication that droidLESK is a promising way of introducing domain knowledge when comparing the similarity between words in the Android app context. I plan to further explore droidLESK's potential in future work.

Finding data point exposure through permission inheritance: Pluto's MMiner scrapes through application manifest files to look for permissions that would allow a level-1

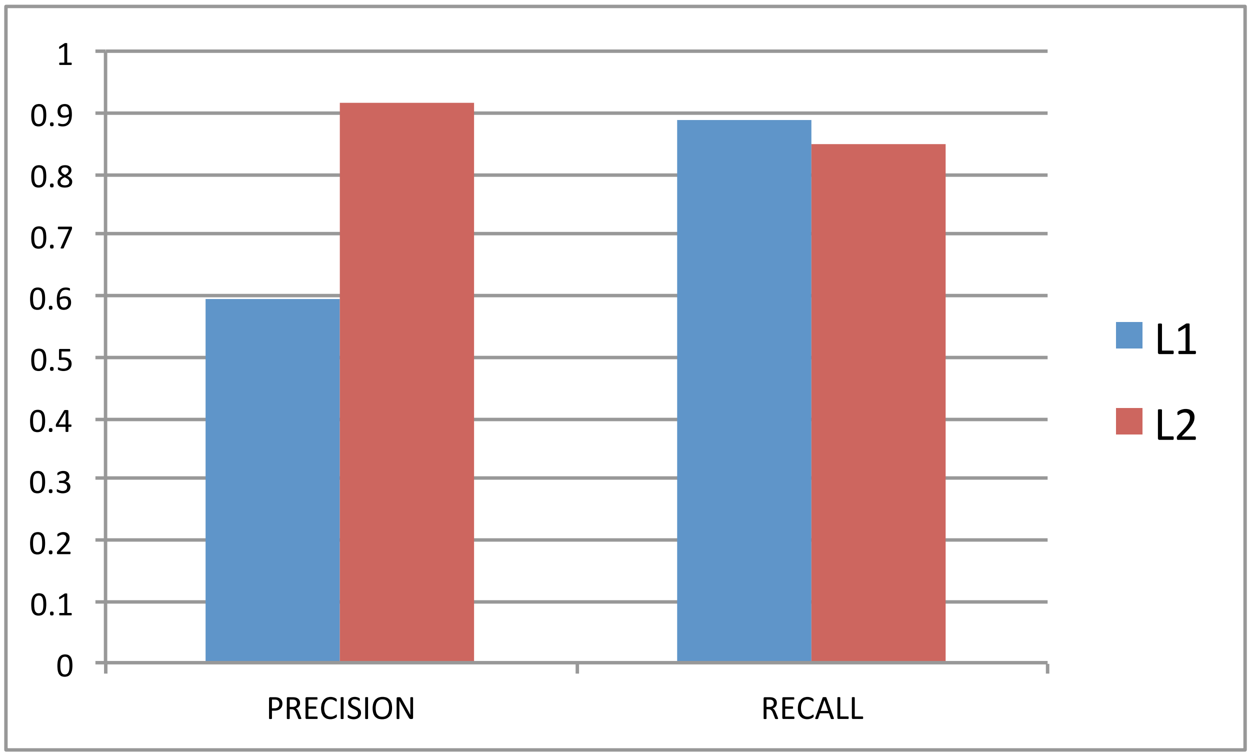


Figure 4.5: **gender** prediction performance given the L1 and L2 ground truth.

or level-2 aggressive library to get access to user attributes or interests. I compared Pluto's performance in two different configurations. In configuration 1 (L1 or L2), Pluto is set to look for a data point using all of its Miners whilst in configuration 2 (L1:MMiner and L2:MMiner) Pluto is set to look for a data point only using the MMiner, if the data point can be derived from the host app permissions. We performed the experiment on the larger level-1 dataset, providing as input the mapping between the permissions ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION with the data point `address`. Figure 4.8 depicts Pluto's performance in predicting the presence of `address` given the above two configurations for both the L1 and L2 datasets and ground truths. As expected, Pluto's prediction is much more accurate when only the MMiner is used. It is clear that in the cases where an data point can be derived through a permission, the best way to predict that data point exposure would be to merely look through the target app's manifest file.

The main reason for the false negatives we observe in all previous experiments was because some data points that the analyst has discovered were in runtime files generated after the user has logged in the app, or after a specific input was provided. Pluto's DAM implementation cannot automatically log in the app. We leave this challenge open for future work.

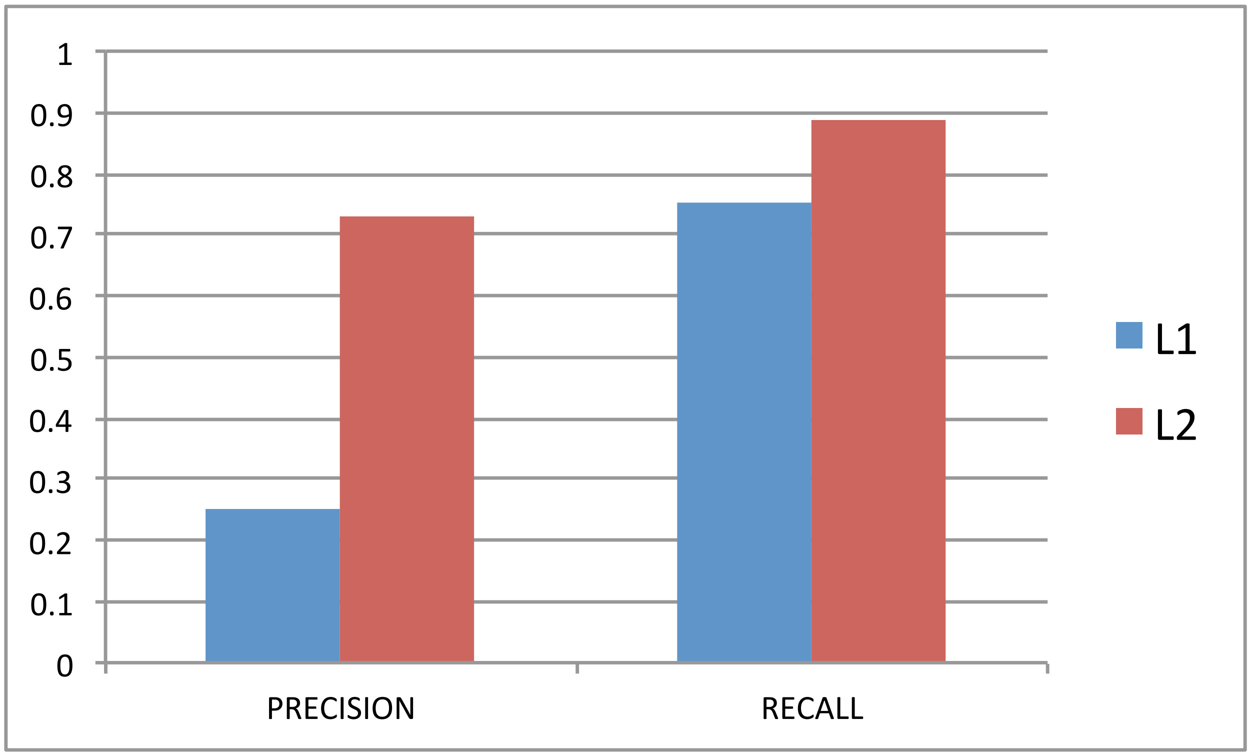


Figure 4.6: **age** prediction performance given the L1 and L2 ground truth.

4.4.2 Evaluation of Pluto's out-app exposure discovery

Next, weneed to evaluate Pluto's ability to construct co-installation patterns and predict user attributes and interests based on information that can be collected through the out-app channel. I ran Pluto's CIP module and classifiers on the ABD dataset we collect from real users (see Section 4.2).

Mining application co-installation patterns: Pluto's CIP module implementation uses FPGrowth [145], the state of the art frequent pattern matching (FPM) algorithm for finding association rules. I chose FPGrowth because it is significantly faster than its competitor Apriori [146]. I applied Pluto's CIP module on the app bundles we collected through our survey. I set FPGrowth to find co-installation patterns in the form 1:N and prune events with `support` less than 10%. Table 4.5 lists the 5 strongest—in terms of `confidence`—association rules that CIP found when run on the survey dataset.

We observe that `Facebook` is likely to be installed together with the `Facebook Messenger` app. This is likely because Facebook asks their users to install the `Facebook Messenger` app when using the Facebook app. Our survey dataset reflects this as well. The strong relationship between the Facebook app and Facebook Messenger app revealed by FPM illustrates its effectiveness for this application. Such rules are critical for Pluto to estimate co-installation

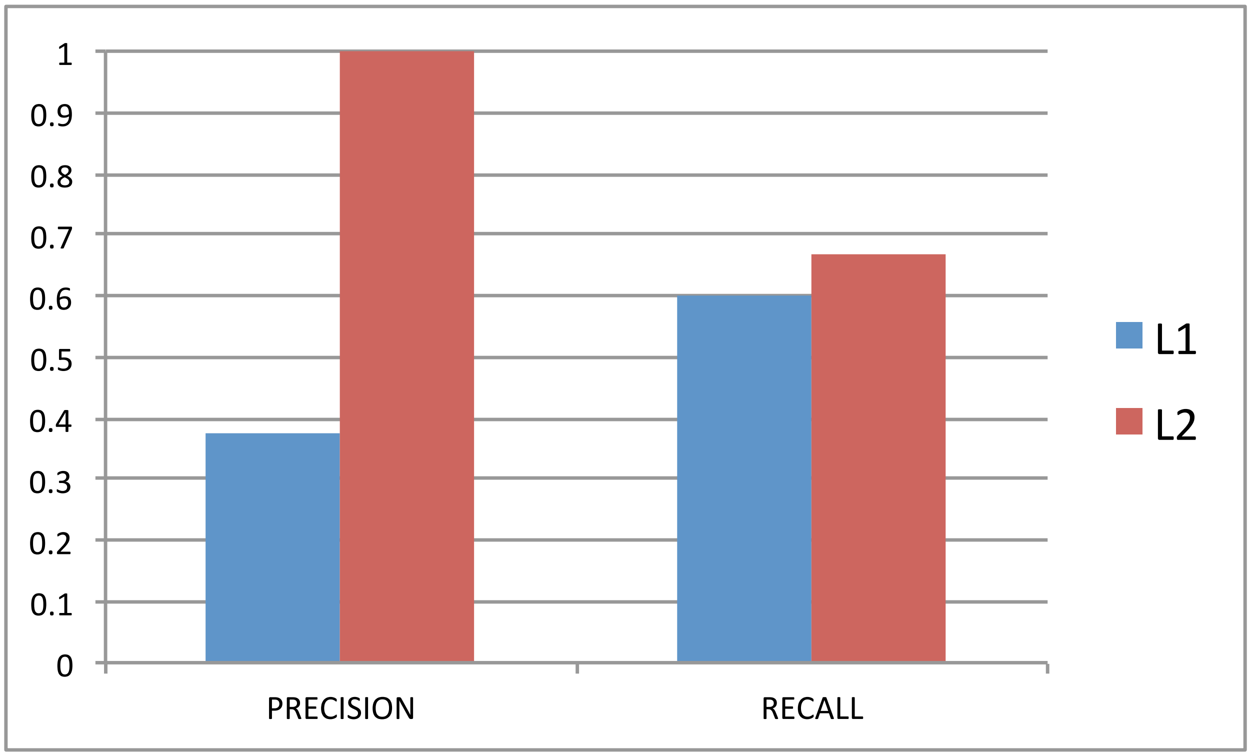


Figure 4.7: **workout** prediction performance given the L1 and L2 ground truth.

patterns between the input application and other applications. Pluto leverages such patterns to provide an estimation of what user attributes can be potentially derived from the app bundles of users that have the input app. Co-installation patterns can also be used to reduce redundancy when combining the in-app data exposure of multiple applications. For example, one might want to estimate what are the in-app data points exposed by app A and app B. However, if these applications are installed on the same device, then the total amount of information the adversarial library will get will be the union of both removing duplicates.

Table 4.5: The strongest co-installation patterns found by the CIP module when run on the survey app bundles.

Precedent	**Consequence**	**Conf**	**Lift**
com.facebook.katana	com.facebook.orca	0.79	2.10
com.lenovo.anyshare.gps	com.facebook.orca	0.75	2.01
com.viber.voip	com.facebook.orca	0.74	1.98
com.skype.raider	com.facebook.orca	0.71	1.88
com.skype.raider	com.viber.voip	0.70	2.32

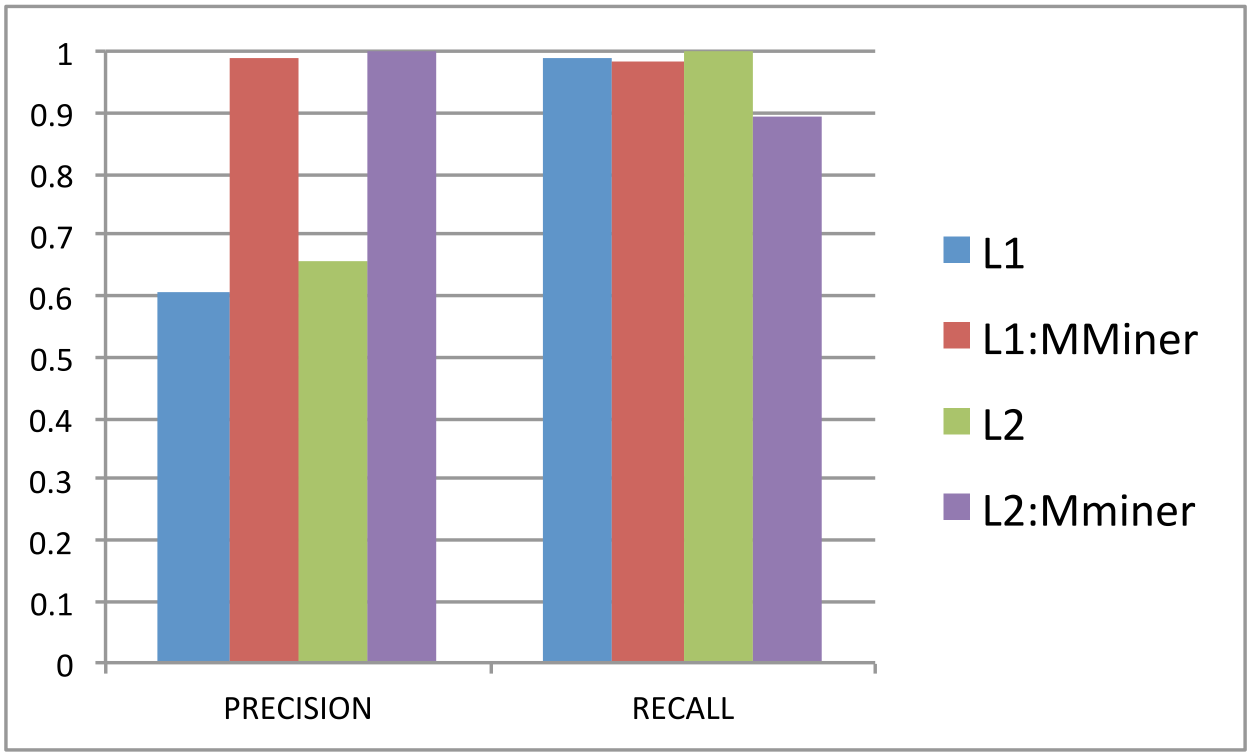


Figure 4.8: **address** prediction performance in different configurations, given the L1 and L2 ground truth.

Performance of Pluto's classifiers: Pluto's classifiers can be used to estimate user attributes derived from CIP app bundles or real-time app bundles from user profiles. I evaluated the performance of Pluto's classifiers on real app bundles we collected from our survey (see Section 4.2). I used the users' answers to the questionnaire in the survey as the ground truth to evaluate the classification results. To justify our use of dimension reduction technique, we evaluated the classifier on both dataset before dimension reduction and dataset after dimension reduction. The results on representative attributes are shown in Table 4.6 and Table 4.7 respectively.

Based on the results shown in both tables, Random Forest performs best across all prediction tasks. The superiority of Random Forest in our evaluation agrees with the existing knowledge [150]. Specifically, because our dataset has a relatively smaller number of instances, the pattern variance is more likely to be high. The ensemble technique (voting by many different trees) employed by Random Forest could reduce such variance in its prediction and thus achieve a better performance.

Comparison of Table 4.6 and Table 4.7 show dimension reduction can effectively improve the performance of Random Forest and KNN. However, the performance of SVM becomes poorer after dimension reduction. One possible reason is that SVM can handle

Table 4.6: Performance of classifiers before dimension reduction

Classifier	Age		Marital Status		Sex	
	P(%)	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**
Random Forest	64.1	66.3	89.8	83.6	91.5	89.6
SVM	65.5	63.6	89.0	82.1	87.4	83.1
KNN	62.7	60.0	86.3	77.7	83.4	74.8

P = Weighted Precision, R = Weighted Recall

Table 4.7: Performance of classifiers after dimension reduction

Classifier	Age		Marital Status		Sex	
	P(%)	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**
Random Forest	88.6	88.6	95.0	93.8	93.8	92.9
SVM	44.8	35.4	66.9	50.5	80.9	70.1
KNN	85.7	83.6	92.5	91.2	91.6	89.9

P = Weighted Precision, R = Weighted Recall

high-dimension data such as our original dataset. The model complexity of SVM is determined by the number of support vectors instead of dimensions.

4.5 UTILITY AND LIMITATIONS

Utility of Pluto: In this chapter, I propose an approach that can be leveraged to assess potential data exposure through in-app and out-app channels to a third-party library given its access to shared intra-process privileges. Note that even though I use ad libraries in free apps as a motivating example, this approach can be adapted to assess data exposure by any app to any third-party library. I chose ad libraries because they are quintessential examples of third-party libraries with strong business incentives for aggressive data harvesting. Motivated by rising privacy concerns related to mobile advertising, users can exert pressure on markets to integrate data exposure assessment into their system and present the results in a usable way to users when downloading an app. In light of this information, users would be able to make more informed decisions when choosing an app. Furthermore, government agencies, such as the Food and Drug Administration (FDA), could benefit from this approach to facilitate their efforts in regulating mobile medical device apps [151] and the Federal Trade Commission (FTC) could leverage Pluto to discover apps that potentially violate user privacy.

Next, I describe a simple way for markets (and in extend other interested parties) to utilize Pluto's results and rank apps based on their data exposure. Intuitively, the harder

Table 4.8: Most risky apps based on their in-app data exposure. M = MEDICAL, HF = HEALTH & FITNESS

CATEGORY	PACKAGE	DESCRIPTION	AVG #INSTALL	SCORE [0 - 10]
M	com.excelatlife.depression	Depression management	$100 \times 10^3 - 500 \times 10^3$	8.14
M	com.medicaljoyworks.prognosis	Clinical case simulator for physicians	$500 \times 10^3 - 1 \times 10^6$	6.31
HF	com.workoutroutines. greatbodylite	Workout management	$100 \times 10^3 - 500 \times 10^3$	7.33
HF	com.cigna.mobile.mycigna	Personal health information management	$100 \times 10^3 - 500 \times 10^3$	5.62

it is for an adversary to get a data point of a user, the more valuable that data point might be for the adversary. Also, the more sensitive a data point is, the harder it will be to get it. Thus sensitive data points should be more valuable for adversaries. Consequently, a market could use a cost model, such as the one offered by the FT calculator, to assign the proposed values acting as weights to data points. In fact, Google, which acts as a data broker itself, would probably have more accurate values and a larger set of data points. They could then normalize the set of exposed data points and present the data exposure score for each app. For example, let D be the set of data points in the cost model and X the set of data point weights in the cost model, where $|D| = |X| = n$. We include the `null` data point in D with a corresponding zero value in X. Also, let α be the app under analysis. Then the new ranked value of α would be $z_\alpha = \frac{x_\alpha - min(X)}{\sum_{i=1}^{n} x_i - min(X)}$ where x_α is the sum of all weights of the data points found to be exposed by app α. Here, $min(X)$ corresponds to an app having only the least expensive data point in D. $\sum_{i=1}^{n} x_i$ corresponds to an app exposing all data points in D. z_α would result in a value from 0 to 1 for each app α under analysis. The higher the value the more the data exposure. This can be presented in the applications download site in application markets along with other existing information for that app. For better presentation, markets could use a number from 0 to 10, stars, or color spectrum with red corresponding to the maximum data exposure.

To provide the reader with a better perspective on the result of this approach, I applied Pluto and performed the proposed ranking technique on the collected apps from the `MEDICAL` and `HEALTH & FITNESS` Google Play categories respectively. In the absence of co-installation patterns for all target apps, I do not take into account the effect of having an app on the same device with another data exposing app [16]. We found that most apps have a low risk score. In particular 97% of `MEDICAL` and also 97% of `HEALTH & FITNESS` apps had scores

[16]Note that to perform the out-app Pluto analysis one needs co-installation patterns for all ranked apps. Markets can easily derive those using the FPM approach described earlier. In that case, one should take into account the UNION of in-app and out-app exposed attributes.

below 5.0. Those apps either expose a very small amount of highly sensitive targeted data, targeted data of low sensitivity, or both. For example, we found *net.epsilonzero.hearingtest*, a hearing testing app, exposed two attributes, the user's phone number and age, and scored 0.02. This ranking technique ensures that only a few apps stand out in the rankings. These are apps with a fairly large number of exposed data points including highly sensitive ones. For example, the highest scored medical app `com.excelatlife.depression` with a score of 8.14, exposes 16 data points including "depression," "headache," and "pregnancy," which have some of the highest values in the FT calculator. Table 4.8 depicts the two most risky apps per category. Pluto in conjunction with the proposed ranking approach can help a user/analyst to focus on those high risk cases.

These ranking results also depict the prevalence of targeted data exposure. As we observe on Table 4.8 the highest ranked apps were installed in the order of hundreds of thousands of devices. Consequently, highly sensitive data of hundreds of thousands of users are exposed to opportunistic third-party libraries. I defer to future work the study of practical approaches to mitigate the data exposure by apps to third-party libraries.

App Bundles: The collection of app bundle information by app developers, advertising companies, and marketing companies is troubling. Currently, the ability of apps to use gIP or gIA with no special permissions provides an opportunity for abuse by both app developers and advertisers. My research demonstrates that this abuse is occurring. I further demonstrate that such information can be reliably leveraged to infer users' attributes. Unfortunately, companies fail to notify consumers that they are allowing the collection of app bundles. With this, they have also failed to notify users as to what entity collects the information, how it is used, or steps to mitigate or prevent the collection of this data. The failure of the Android API to require permissions for the gIP or gIA removes from the users the possibility to have choice and consent to this type of information gathering. To prevent abuse of gIP or gIA, app providers should notify users, both in the privacy policy and in the application, that app bundles are collected. Additionally, applications should provide the user the opportunity to deny the collection of this information for advertising or marketing purposes. Potentially, the Android API could require special permissions for gIP or gIA. However, the all-or-nothing permissions scheme might not add any additional value besides notice to the user and the warning may not be necessary for an app that is using these two functions for utility and functional purposes.

Limitations of the proposed approach: The estimation of data exposure to libraries is constrained by the specific attack channels we consider. The prototype employs specific examples for each channel and performs data exposure assessment based on those. Nevertheless, the cases we considered are not the only ones. For example, someone could include the `CAMERA` permission or the `RECORD_AUDIO` in the protected APIs. The camera could be used opportunistically to get pictures of the user in order to infer her gender or location. The microphone could be used to capture what the user is saying and, by converting speech to text and employing POS tagging, infer additional targeted data. More channels can also be discovered such as new side channels or covert channels. These can be used to extend Pluto for a more complete assessment. The current prototype and results can serve as a baseline for comparison.

4.6 SUMMARY

In this Chapter I performed an analysis on the security of shared intra-process privileges. I showed how an untrusted incentivized third-party library can exploit those shared privileges to aggressively harvest sensitive user information on a smartphone. In my analysis I detailed the adversary model in this setting, and introduced previously unknown inference techniques for such advertisers. I then utilized these observations to built a tool for automatically detecting potential information leakage to such adversaries. Figure 4.9 visualizes this contribution on the smartphone ecosystem. The tool combines techniques from code analysis, natural language processing and machine learning to emulate the information reach of third-party libraries on smartphones. This study reveals a pressing need for considering such intra-process adversaries when designing resource isolation mechanisms on smartphones. Next I will look into the security of another shared resource, in particular shared filesystem resources across userspace applications/processes on smartphones.

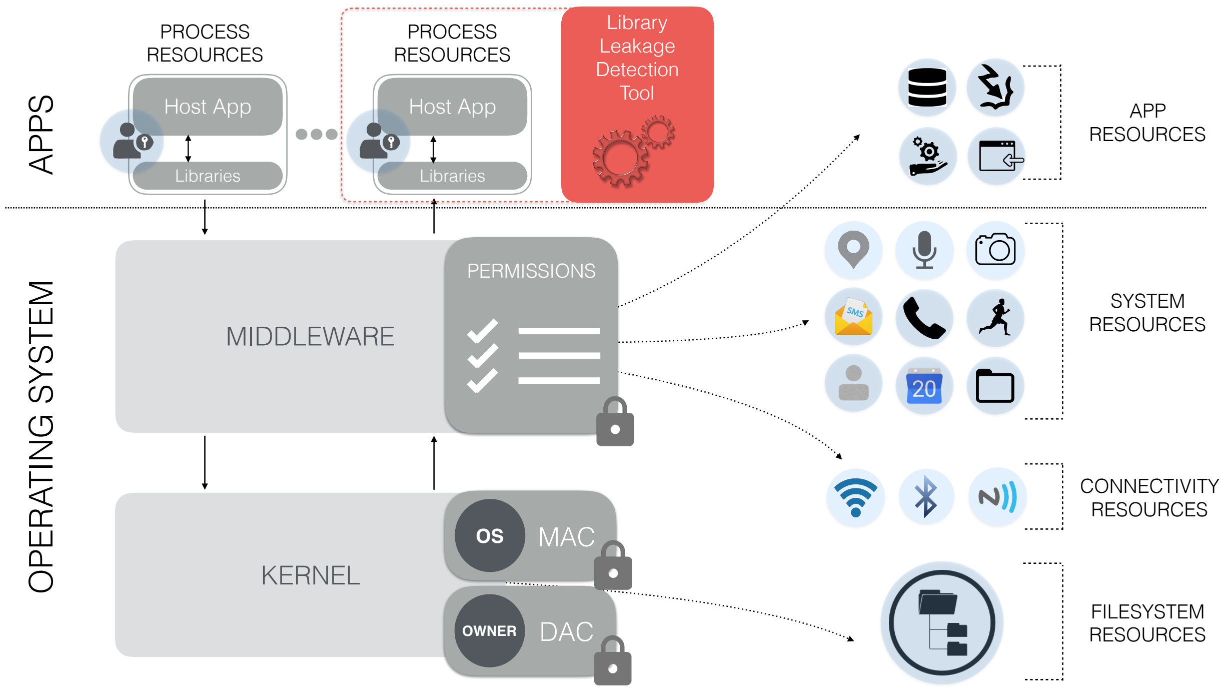


Figure 4.9: Detecting information leakage to third-party libraries.

CHAPTER 5: SHARING FILESYSTEM RESOURCES

Mobile applications can access filesystem resources made available by the host operating system. Android in particular is built on top of a stripped down version of the Linux kernel, which is optimized for mobile devices. The Linux kernel uses a virtual process filesystem (procfs) for efficiently mirroring some of the kernel data structures to userspace programs. Files in this filesystem are protected using a traditional discretionary access control scheme. However, files which hold seemingly harmless information on a static platform (e.g. desktop machines) can be hazardous when accessed on a mobile platform. In Chapter I will present my analysis on the Android shared filesystem resources. I will show that malicious mobile applications can exploit unprotected files as side-channels to perform inference attacks and compromise user confidentiality [11]. .

5.1 INTRODUCTION

Android provides unprivileged applications with access to basic local filesystem resources. 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 the `ps` command and among the apps using audio resources they access through he API call `AudioManager.requestAudioFocus`. However, those old design assumptions on the public local resources are becoming increasingly irrelevant in front of the fast-evolving ways to use smartphones. In [11] I identified two fundamental design/use gaps that are swiftly widening, affecting the Android ecosystem:

Firstly, I found that there is a gap between Linux's design and the 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 virtual process filesystem (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*).

Secondly, I observed the manifestation of a gap between the assumptions on Android public resources and evolving app design, functionalities and background information throughout this study. For example, an app is often dedicated to a specific website. Therefore, the 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.

Specifically, in this study I carefully analyzed the ways filesystem resources are utilized by the OS and popular apps on Android, together with the public online information related to their operations. My study discovered two confirmed new sources of information leaks:

- *App network-data usage* (Section 5.2.2). I found 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, I developed a suite of inference techniques that can reveal a phone user's disease conditions she is interested in from the network-data consumption of WebMD app, her identity from that of Twitter app, and the stock she is looking at from Yahoo! Finance app.
- *Public ARP information* (Section 5.2.3). I further discovered that the public ARP data released by Android (under its Linux public directory) contains the BSSID of the WAP a phone is connected to, and demonstrate how to practically utilize such information to locate a phone user through BSSID databases.

I built a zero-permission app that stealthily collects information for these attacks. This chapter elaborates on side-channel attacks designed and executed based on these newly found information leaks throufh shared filesystem resources and discusses mitigation strategies. Firstly we will see the capabilities that adversary (5.2.1) possess to be able to deploy such attacks.

5.2 ANALYSIS

5.2.1 Adversary Model

The adversary considered in this study 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, searching Google Maps, etc. Such activities can be performed by ordinary Internet users.

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 and 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.

A malicious app should be able to share the surreptitiously stolen data with the adversary's remote location. Leviathan's blog describes a zero-permission technique to smuggle out data across the Internet [65]. 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. I re-implemented this technique in my research and further made it stealthy. Leviathan's approach does not work when the screen is off because the browser is `paused` when the screen is off. I improved this method to smuggle data right before the screen is off or the screen is being unlocked. Specifically, the adversarial 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 the app to send out data through the browser without being noticed by the user. After the data transmission is done, the app can redirect the browser to Google and also set the phone to its home screen to cover this operation.

A malicious app should also be aware of the system's situation or state. The designed zero permission app defines a list of target applications such as stock, health, location applications and monitors their activities. It first checks whether those packages are installed on the victim's system (`getInstalled Applications()`) 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.

5.2.2 Side-Channel 1: per-App Network Traffic

Usage Monitoring and Analysis

Mobile data usages of Android are made public under `/proc/uid_stat/` (per app) and `/sys/class/net/[interface] /statistics/` (per interface). The former was 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 numbers of bytes received and sent by a specific app respectively. I found 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 [41]. However, my research shows that their updates can also be leveraged to fingerprint an app's network operations, such as sending HTTP POST or GET messages.

To catch the updates of those statistics in real time, I built 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 5.1, in which I compare the packet payloads observed by Shark for Root (a network traffic sniffer for 3G and WiFi) [152], when the user is using Yahoo! Finance, with the cumulative outbound data usage detected by our usage monitor.

From the figure 5.1 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 I will discuss later.

In terms of performance, the 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 about 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 5.1 measured using AnTuTu [153] with a sampling rate of 10Hz for network usage 5.2.2 and 50Hz for audio logging. To make this data collection stealthier, I adopted a strategy that samples intensively only when the target app is being executed,

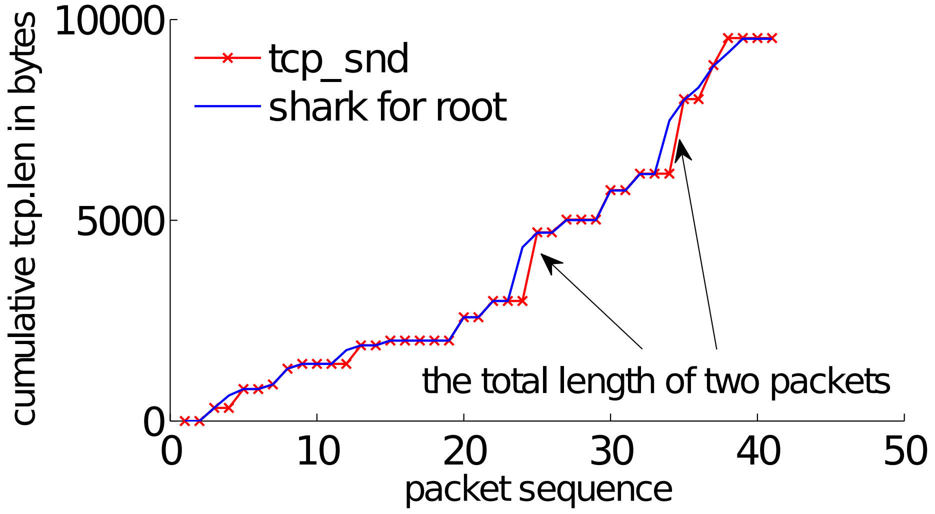


Figure 5.1: Monitor tool precision

which is identified through `ps` (Section 5.2.1). The UI of the monitor tool is shown in Figure 5.2.

Table 5.1: Performance overhead of the monitor tool: there the baseline is measured by AnTuTu [153]

	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%

However, 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 designed a methodology to bridge the gap between the real sequence and what the monitor sees.

My 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 its behavior. Once the monitor collects a sequence of usage increments from the app's runtime on the victim's Android phone, I 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

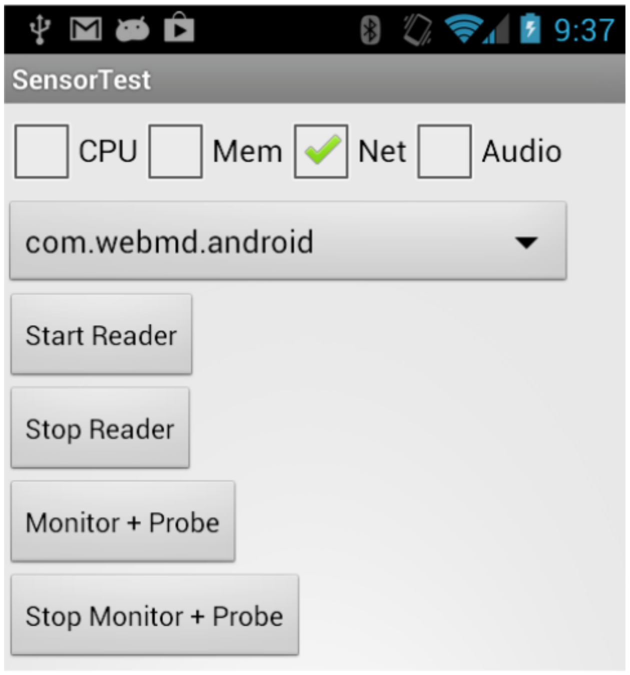


Figure 5.2: Monitor tool UI

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.

For example, consider that the signature for requesting the information about a disease condition *ABSCESS* by WebMD is $(458, 478, 492 \rightarrow)$, where "$\rightarrow$" indicates outbound traffic. Usage sequences matching the signature can be $(458, 478, 492 \rightarrow)$, $(936, 492 \rightarrow)$ or $(1428 \rightarrow)$.

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. The User-Agent is related to the phone's type, brand and Android OS version. For example, the User-Agent of the Yahoo! Finance app on a Nexus S phone is:

```
User-Agent:  YahooMobile/1.0 (finance; 1.1.8.1187014079); (Linux; U; Android 4.1.1;
sojus
Build/JELLY_BEAN);
```

Given that the format of this field is known, all the adversary needs, is a set of parameters (type, brand, OS version etc.) for building up the field, which is important for estimating the length of the field and the payload that carries the field. Such information can be easily obtained by a zero-permission app, without any permission, from `android.os.Build` and `System.getProperty("http agent")`.

Health Data

Next I will show that the data-usage statistics a zero-permission app collects through shared filesystem resources, leak out apps' sensitive inputs, e.g., disease conditions a user selects on WebMD mobile [154]. This has been achieved by fingerprinting her actions with data-usage sequences they produce. The same attack technique also works on Twitter 5.2.2 and Yahoo! Finance 5.2.2.

WebMD mobile is an extremely popular Android health and fitness app, which has been installed $1 \sim 5$ million times in the past 30 days [154]. To use the app, one first clicks to select 1 out of 6 sections, such as "Symptom Checker", "Conditions" and others as seen in Figure 5.3. In my research, I analyzed the data under the "Conditions" section, which includes a list of disease conditions (e.g., Asthma, Diabetes, etc.). Each condition, once clicked on, leads to a new screen that displays the overview of the disease, its symptoms and related articles. As we can see from Figure 5.4, all such information is provided through a simple, fixed user interface running on the phone, while the data there is downloaded from the web. I found that the changes of network usage statistics during this process can be reliably linked to the user's selections on the interface, revealing the disease she is interested in.

Attack Methodology. I first analyzed the app offline (i.e. in a controlled context) using Shark for Root, and built a detailed finite state machine (FSM) for it based on the payload lengths of TCP packets sent and received when the app moves from one screen (a state of the FSM) to another. The FSM is illustrated in Figure 5.5. Specifically, the user's selection of a section is characterized by a sequence of bytes, which is completely different from those of other sections. Each disease under the "Conditions" section is also associated with a distinctive payload sequence.

In particular, every time a user clicks on a condition she is interested in, there are a number of requests being generated: 3 `POST` $\{p_1, p_2, p_3\}$ requests which correspond to *Overview, Symptoms* and *Related Articles* and 4 `GET` requests for ads and tracking. The 4 `GET`s can be readily filtered out due to their fixed packet sized with small variations, e.g., the `GET ads/dcfc.gif` is always 174 bytes and the size of `GET event.ng/type=...` is always 391-415 bytes. Interestingly, different from what has been observed from the browser-based web applications [64], whose information leaks typically happen through the responses, for the simple app studied here, even the sizes of its request payloads give away enough information for a first order classification of all 204 conditions into 32 categories with 4 conditions being already uniquely identified (see Figure 5.6). Table 5.2 shows an example of distinct transmission traffic patterns between "Anemia. iron deficiency" and "Vulvodynia".

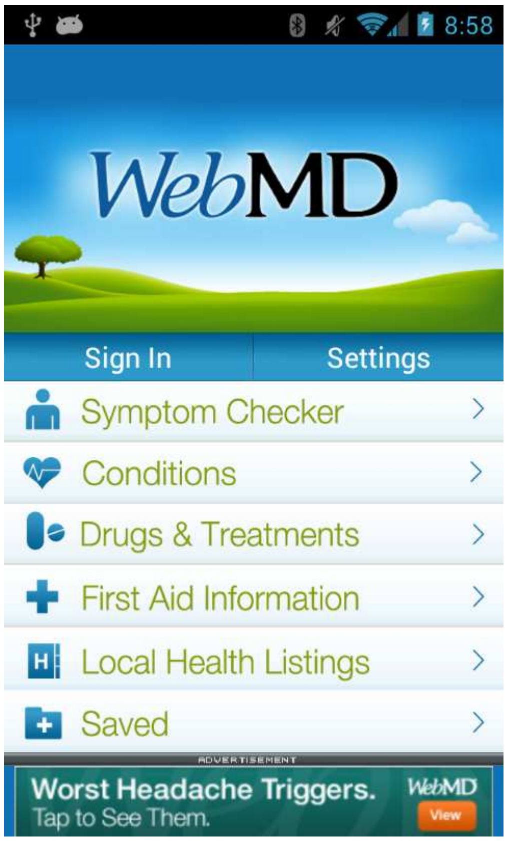
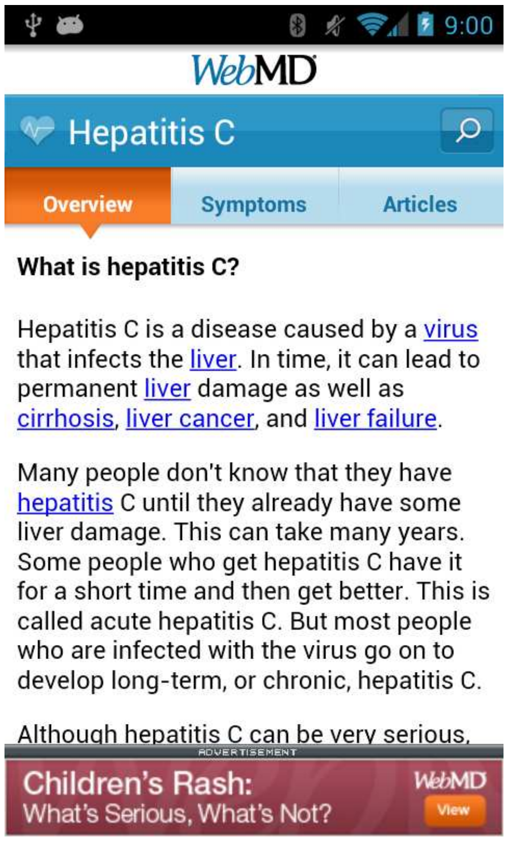


Figure 5.3: WebMD: First Screen



What is hepatitis C?

Hepatitis C is a disease caused by a virus that infects the liver. In time, it can lead to permanent liver damage as well as cirrhosis, liver cancer, and liver failure.

Many people don't know that they have hepatitis C until they already have some liver damage. This can take many years. Some people who get hepatitis C have it for a short time and then get better. This is called acute hepatitis C. But most people who are infected with the virus go on to develop long-term, or chronic, hepatitis C.

Although hepatitis C can be very serious,

Figure 5.4: WebMD: A Condition's Screen

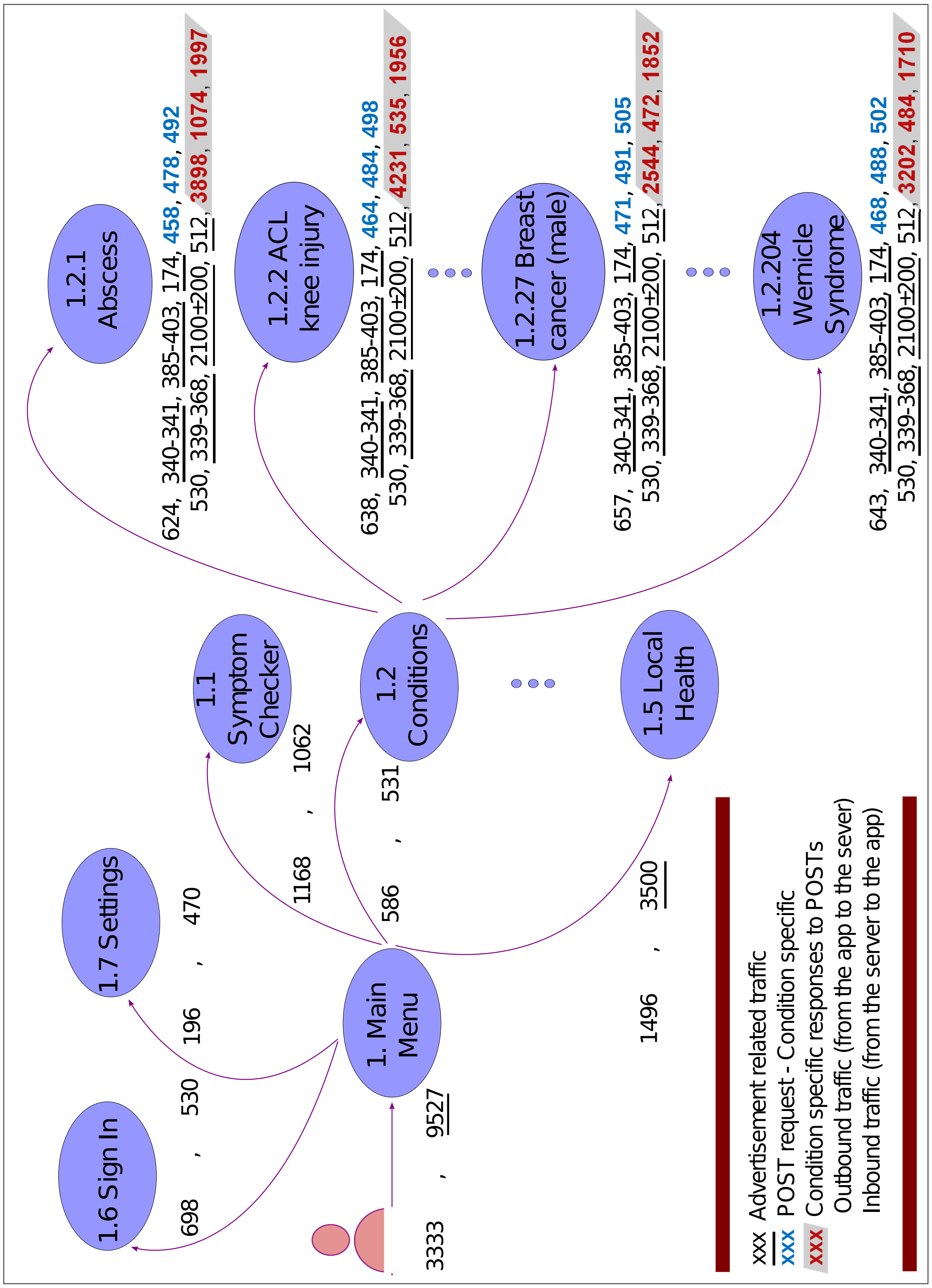


Figure 5.5: WebMD Finite State Machine

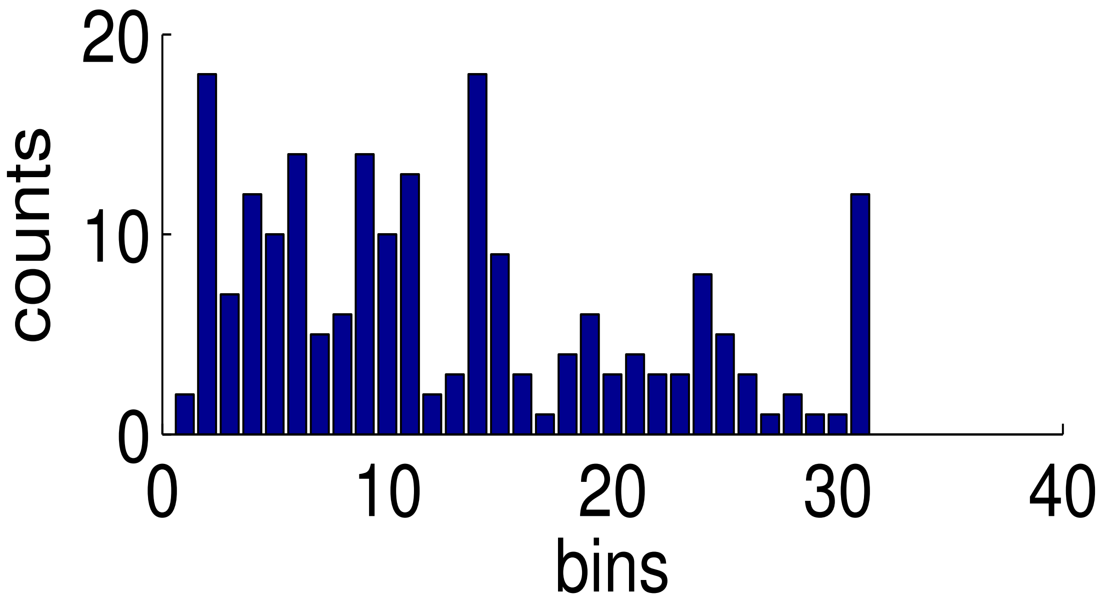


Figure 5.6: First Order Traffic Classification of WebMD's conditions

Furthermore, lets denote the corresponding response pattern with $\{r_1, r_2, r_3\}$ excluding the ads traffic. The latter gives us some trouble but can be removed from the analysis also due to its predictable packets pattern, for example it always contains a 450 ± 100 bytes `GIF` image and a packet of 2100 ± 200 bytes payload. From the signature $\{p_1, p_2, p_3 \rightarrow; r_1, r_2, r_3 \leftarrow\}$, we first utilize $\{p_1, p_2, p_3\}$ to classify all 204 conditions into 32 categories using the technology in 5.2.2. Subsequently we can use the information from $\{r_1, r_2, r_3\}$ to further differentiate between conditions of the same category.

In a real attack, however, the zero-permission app cannot see the traffic. The usage increments it collects could come from the combination of two packets. For the requests, this problem can be easily addressed using the technique described in Section 5.2.2, as their payload lengths are fixed and we can compare an observed increment to the cumulative length of multiple packets. The approach becomes less effective when we work on the responses, due to the non-determinism of payload lengths. Fortunately, inter-packet duration of the inbound traffic is reasonably long, allowing the usage monitor to accurately identify different payloads most of the time.

Another fact that the adversary must address in a real context is that when a request is being made from the application to the server, the device's user agent is also being sent. This can affect the matching of the offline created signatures with the data the malicious app collects when the corresponding devices used differ in model, especially when the attack relies on accuracy of byte granularity. To compensate for that the malevolent app can readily

Table 5.2: WebMD. Comparison of Bytes Transmitted between two Conditions of different Categories

Anemia, iron deficiency	
Request Description	**Bytes TX**
...	...
Get Overview (POST)	474
Get Symptoms (POST)	494
Get Related Articles (POST)	508
Vulvodynia	
Request Description	**Bytes TX**
...	...
Get Overview (POST)	461
Get Symptoms (POST)	481
Get Related Articles (POST)	495

acquire the device's user agent and sent it out to the attacker's remote server before it starts emitting any of the previous metrics it records. To be consistent we integrate this piece of functionality to our prototype despite its trivial nature.

To collect the data the adversary needs to complete her attack I used the following methodology: As stated before, using Shark for Root we can create a detailed map of the states the application can be at any possible time. We can refer to states as screens being displayed to the user as denoted by the simplified state diagram on Figure 5.5 . For each state of the application we can record the length of the bytes (TCP payload) that were sent and received for that screen to be displayed. The recordings are at the granularity of HTTP requests/responses. This technique would allow us to distinguish the user's navigation on the device. To achieve that we used the outbound traffic because of the requests' consistency among different iterations of the same experiment. The inbound traffic contained advertisement data that change as the advertisement being fetched is different every time. Furthermore this issue is aggravated when a user is visiting disease conditions: For each Condition screen three pieces of disease specific information are being received. Firstly the application receives the "Overview" of the disease, then the "Symptoms" that appear to a suffering patient and lastly some links to disease "Related Articles" redirecting the user for further reading as shown in Figure 5.4. However some other information relative to the app or advertisements is being retrieved from different ports of the responding server or even different servers. If these information responses happen faster than our tool's sampling speed then the tool will report multiple response readings in one record. This makes the break-

Table 5.3: WebMD. Traffic Analysis for the *ACUTE SINUSITIS* condition navigation

ACUTE SINUSITIS

No	**HTTP Request**	**Bytes TX**	**HTTP Response**	**Bytes RX**
1	GET /b/ss/webmdplglobal...	638	HTTP 1/1 200 OK	512
2	GET html.ng/transactionID=..	341	$< ad > </.. >$	~ 2202
3	GET event.ng/type=..	415	HTTP/1.1 302 FOUND	349
4	GET ads/dcfc.gif	174	HTTP1/1 200 OK (GIF87a)	401
5	POST GetOverview	464	$< Overview > </.. >$	9308
6	POST GetSymptoms	484	$< Symptoms > </.. >$	3334
7	POST GetRelatedArticles	498	$< Related > </.. >$	4857

down of that record to the individual responses hard especially when multiple conditions receive information that vary less than the advertisement variation range.

WebMD has 204 available conditions for user perusal (at the time of writing). Using the payload of the outbound requests we classified them into 32 Categories (Figure 5.6). The request on row 1 of Table 5.3 is specific to the condition but can vary sometimes: For every such request the condition's name is passed as a parameter which results in collisions when the titles of two different conditions have the same number of HTTP characters. A specific id is also used for every condition but in most cases is of the same number of digits. Lastly whether the request was made on a day of the month that can be described with 1 digit or 2 affects the request. For the classification I have used the requests made for the three aforementioned condition specific information, which I mark at the fifth, sixth and seventh row of Table 5.3. Those requests are always identical when visiting the same Condition. The other requests are common for all conditions. Nevertheless, some Categories result in a high number of collisions (many counts per bin on Figure 5.6). To address that we used the inbound traffic for a second order Classification. With much less possible candidates - the category's members - to match our tool's inbound traffic recording and based on the fact that our tool's high sampling rate can help us distinguish at least a fraction of the responses, we managed to identify all the Condition visits.

To collect the data and construct tables with inbound and outbound traffic (see Table 5.3) generated with each condition click and also understand the application protocol in place, I ran a set of experiments. For those experiments I have used a Google Nexus S 4G device running Android 4.1.1 with root access to the Operating System, available. On the device I installed Shark for Root which can capture the traffic and generate pcap files that we can analyze using an appropriate tool such as Wireshark. I have also installed WebMD and the monitor tool on the device. Before every experiment, I launch the tool, set to monitor

WebMD's traffic, and Shark which captures all network traffic on the device. Then I launch WebMD and navigate to a particular condition. Subsequently I stop the tool and Shark, and analyze the results matching the tool's recordings with the measurements from Shark. Based on this analysis I generate tables for each condition that hold the Number of Bytes TX and Number of Bytes RX for each HTTP response and request of WebMD. For example, the data collected for ”ABSCESS” is shown on Table 5.3.

Attack evaluation. To evaluate the effectiveness of the attack on WebMD, I repeated the experiments. This time, I didn't mark the tool's output with the Condition being visited on the device by the user. Conversely I perform experiments visiting all available Conditions on WebMD and then use a script that shuffled the results. Shuffling the results eliminates the possibility that the analyst remembers the order of condition visiting. By the end of this process I have performed 221 experiments for 204 available Conditions. The shuffling tool rejected 2 outputs which leaves us with 219 results to analyze. I manually scrutinized the experiments' outputs and tried to match the recorded measurements with our data collected offline. According to the bytes received we can locate the Category of Conditions that particular output corresponds to. Then I further analyze the inbound traffic to identify the precise condition in the Category that has similar traffic with the observed one. The tool's sampling rate has been proven instrumental to this effort as in most cases, a single reading of it could disclose one exact match with one of the 3 total Condition relevant responses. Conditions on the same Category rarely have identical such responses as the information received is very specific to the Condition they describe.

Out of the 219 available experiments' outputs I was able to uniquely identify all 204 Conditions. In 5 cases a Condition was matched twice. This can be attributed to the fact that network connectivity in some cases rendered the application unable of retrieving the Condition's information. In those cases I had repeated the experiment. Even if the experiment failed in the sense that it didn't simulate a normal navigation to a condition, we were able from the fraction of information received by WebMD and recorded by our tool, to identify the Condition clicked. Finally, 11 outputs failed to be identified as a condition and were the result of erroneous clicks by the user, that inadvertently followed a different path on the application (i.e. a Condition was not visited).

Identity

Next I will show that the data-usage statistics collected by the zero-permission app through shared filesystem resources, also leak out an Android user's identity. A person's identity, such as name, email address, etc., is always considered to be highly sensitive [155, 156, 157, 158] 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 I show how 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 there 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 [159]. In [11] we performed 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 discovered 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 I describe below poses a realistic threat to Android users' identity.

Attack Methodology. In this 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 sends tweets to the Twitter server, the app detects this event and stealthily sends its timestamp to the malicious server. This results in a vector of timestamps for the user's tweets, which we can then be used 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 identify 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 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. My solution is using less protected data such as the coarse location (e.g, city) of the person who tweets, to narrow down the search range (see Section 5.2.3 for an attack that allows an adversary to gain such information).

To fingerprint the tweeting event from the Twitter app, I 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 the zero-permission app collects *online* from the victim's phone to identify the moment she tweets.

Specifically, during the offline analysis, I observed the following TCP payload sequence produced by the Twitter app: $(420|150, 314, 580\text{–}720)$. 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, we want to use the signature to find out the timestamp when the user tweets. The problem 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. The offline analysis shows that the payload for Client Hello can be reliably detected by the monitor. However, the time interval between the 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 (Section 5.2.2) 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 $(420|150, 314, 580\text{–}720)$ or $(420|150, 894\text{–}1034)$.

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 [160]: 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 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. However, I found that the user's fine location could also be inferred from shared filesystem resources, once the victim user connects her phone to a Wi-Fi hotspot (see Section 5.2.3). Getting her coarse location in this case is much easier: the 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 [161] to find her city. This

Table 5.4: 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

allows the adversary to set the query parameters using Google Maps. Note that smartphone users tend to use Wi-Fi whenever possible to conserve their mobile data (see Section 5.2.3), which gives the adversarial app chances to get the victims' coarse locations. Note that the adversary does not require the user to geo-tag each tweet. The twitter search results include the tweets in a area as long as the user specified her geo-location in her profile.

As discussed before, the adversarial 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 $t_i \in T$, the adversary can 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 big 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 the app reports a timestamp, the adversary can search it in the database to find those who tweeted at that moment.

Attack Evaluation. In [11] we evaluated the effectiveness of this attack at three cities, Urbana, Bloomington and Chicago. Table 5.4 describes these cities' information.

We first studied the lengths of the time intervals the 1500 tweets returned by a Twitter query can cover in these individual cities. To this end, we examined 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 5.4. 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 the experiment, we ran a script that repeatedly called the Twitter Search API, at a rate of 135 queries per hour. All the results without duplicates were stored in a local SQL database. Then, we posted tweets through the Twitter app on a smartphone, under the surveillance of the zero-permission app. After obvious robot Twitter accounts were eliminated from the query results, our Twitter ID were recovered by merely 3 timestamps at Urbana, 5 timestamps at Bloomington and 9 timestamps in Chicago, which is aligned with the city size and population.

Investment Data

A person's investment information is private and highly sensitive. Here I 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. I show that Yahoo! Finance discloses a unique network data signature when the user is adding or clicking on a stock.

Attack Methodology. Similar to all aforementioned attacks, I assume that a zero-permission app which irunning 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. 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.

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 studied the signature when a stock code is clicked upon. We found 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 this study, we ran the 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 was being used to add 10 random stocks onto the stock watch list, we managed to narrow down the candidate list to 85 possible stocks that matched the data-usage features of these 10 stocks. Further analyzing the increment sequence when the user clicked on a particular stock code, which downloaded related news to the phone, we were able to uniquely identify each of the ten stocks the user selected among the 85 candidates.

5.2.3 Side-Channel 2: ARP Info

This Section elaborates on how Android unprotexted local resources can leak a user's location. As with all the side-channel attacks, this is work conducted with Zhou et al. [11].

The precise location of a smartphone user is widely considered to be private and should not be leaked out without the user's explicit consent. Android guards such information with a permission `ACCESS_FINE_LOCATION`. The information is further protected from the websites that attempt to get it through a mobile browser (using `navigator.geolocation.getCurrent Position`), which is designed to ask for user's permission when this happens. In this section, we show that despite all such protections, our zero-permission app can still access location-related data, which enables accurate identification of the user's whereabouts, whenever her phone connects to a Wi-Fi hotspot.

As discussed before, Wi-Fi has been extensively utilized by smartphone users to save their mobile data. In particular, many users' phones are in an auto-connect mode. Therefore, the threat posed by our attack is very realistic. In the presence of a Wi-Fi connection, we show in Section 5.2.2 that a phone's coarse location can be obtained through the gateway's IP address. Here, we elaborate how to retrieve its fine location using the link layer information Android discloses.

Location Inference

My analysis further revealed that the BSSID of a Wi-Fi hotspot and signal levels perceived by the phone are disclosed by Android through another shared procfs entry. Such information is location-sensitive because hotspots' BSSIDs have been extensively collected by companies (e.g., Google, Skyhook, Navizon, etc.) for location-based services in the absence of GPS.

However, their databases are proprietary, not open to the public. In this section, we show how we address this challenge and come up with an end-to-end attack.

Interestingly, in proc files `/proc/net/arp` and `/proc/net/wireless`, Android documents the parameters of Address Resolution Protocol (ARP) it uses to talk to a network gateway (a hotspot in the case of Wi-Fi connections) and other wireless activities. Of particular interest to us is the BSSID (in the arp file), which is essentially the gateway's MAC address, and wireless signal levels (in the wireless file). Both files are accessible to a zero-permission app. The app I implemented periodically reads from procfs once every a few seconds to detect the existence of the files, which indicates the presence of a Wi-Fi connection.

The arp file is inherited from Linux, on which its content is considered to be harmless: an internal gateway's MAC address does not seem to give away much sensitive user information. For smartphone, however, such an assumption no longer holds. More and more companies like Google, Skyhook and Navizon are aggressively collecting the BSSIDs of public Wi-Fi hotspots to find out where the user is, so as to provide location-based services (e.g., restaurant recommendations) when GPS signals are weak or even not available. Such information has been gathered in different ways. Some companies like Skyhook wireless and Google have literally driven through different cities and mapped all the BSSID's they detected to their corresponding GPS locations. Others like Navizon distribute an app with both GPS and wireless permissions. Such an app continuously gleans the coordinates of a phone's geo-locations together with the BSSIDs it sees there, and uploads such information to a server that maintains a BSSID location database.

All such databases are proprietary, not open to the public. In the contxt of this study we communicated with Skyhook in an attempt to purchase a license for querying their database with the BSSID collected by our zero-permission app. They were not willing to do that due to their concerns that our analysis could impact people's perceptions about the privacy implications of BSSID collection.

Nevertheless, an adversary can exploit commercial location services that are being used by their respective apps: Many of those commercial apps that offer location-based services, need a permission `ACCESS_WIFI_STATE`, so they can collect the BSSIDs of all the surrounding hotspots for geo-locating their users. In our case, however, our zero-permission app can only get a *single* BSSID, the one for the hotspot the phone is currently in connection with. We need to understand whether this is still enough for finding out the user's location. Since we cannot directly use those proprietary databases, we have to leverage these existing apps to get the location. The idea is to understand the protocol these apps run with their servers to generate the right query that can give us the expected response.

Table 5.5: Geo-location with a Single BSSID

Location	**Total BSSIDs Collected**	**Working BSSIDs**	**Error**
Home	5	4	0ft
Hospital1	74	2	59ft
Hospital2	57	4	528ft
Subway	6	4	3ft
Starbucks	43	3	6ft
Train/Bus Station	14	10	0ft
Church	82	3	150ft
Bookstore	34	2	289ft

Specifically, we utilized the Navizon app to develop such an indirect query mechanism. Like Google and Skyhook, Navizon also has a BSSID database with a wide coverage [162], particularly in US. In our research, we reverse-engineered the app's protocol by using a proxy, and found that there is no authentication in the protocol and its request is a list of BSSIDs and signal levels encoded in Base64. Based upon such information, we built a "querier" server that uses the information our app sneaks out to construct a valid Navizon request for querying its database for the location of the target phone.

Attack Evaluation

To understand the seriousness of this information leak, we ran our zero-permission app to collect BSSID data from the Wi-Fi connections made at places in Urbana and Chicago, including home, hospital, church, bookstore, train/bus station and others. The results are illustrated in Table 5.5.

In particular, our app easily detected the presence of Wi-Fi connections and stealthily sent out the BSSIDs associated with these connections. Running our query mechanism, we successfully identified all these locations from Navizon. On the other hand, we found that not every hotspot can be used for this purpose: after all, the Navizon database is still far from complete. Table 5.5 describes the numbers of the hotspots good for geo-locations at different spots and their accuracy.

5.3 MITIGATION DESIGN

Given the various unprotected filesystem resources on Android, the information leaks the analysis revealed 5 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, lets first take a closer look at the attacks revealed in the analysis. The ARP data (see 5.2.3) has not been extensively utilized by apps and can therefore be kept away from unauthorized parties by changing the related file's access privilege to `system`. A simple solution to control the audio channel can be to restrict the access to its related APIs, such as `isMusicActive`, only to system processes whenever sensitive apps (e.g. navigation related) are running in the foreground. The most challenging facet of such a mitigation venture is to address the availability mechanism of the data usage statistics (see 5.2.2), 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, I report our approach on mitigating the threat deriving from the statistics availability, while maintaining their utility.

5.3.1 Mitigation Strategies

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

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 [64, 163]. 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 α an integer given to our enforcement framework implemented on Android (Section 5.3.2). 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. We call this strategy `Round up and round down`.

A limitation of the simple rounding strategy (`Round up and round down`) 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 hide packets with exceedingly large payloads. To address this issue, we can accumulate the data usage information of multiple queries, for example, conditions on WebMD 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. We will refer to this technique as `Aggregation`.

5.3.2 Enforcement Framework

A naive idea to address the leakage of information from Android public local resources, 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 [27]. 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 special "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 the 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 we added to Android, by holding back the answer, adding noise to it (as described in Section 5.3.1) or periodically updating the information.

To enable the enforcement of the aforementioned policies in our framework, public resources on the Linux layer, such as the data usage counters, are set to be accessible only by system or root users. Specifically, for the `/proc/uid_stat/` resources, we modified the `create_stat` file in `drivers/mis/uid_stat.c` of the Android Linux kernel and changed the mode of `entry` to disable direct access to the proc entries by any app. With direct access turned off, the app will have to call the APIs exposed in `TrafficStats.java` and `NetworkStats.java` such as `getUidTxBytes()` to gain access to that information. In our research, we modified these APIs so that whenever they are invoked by a query app that

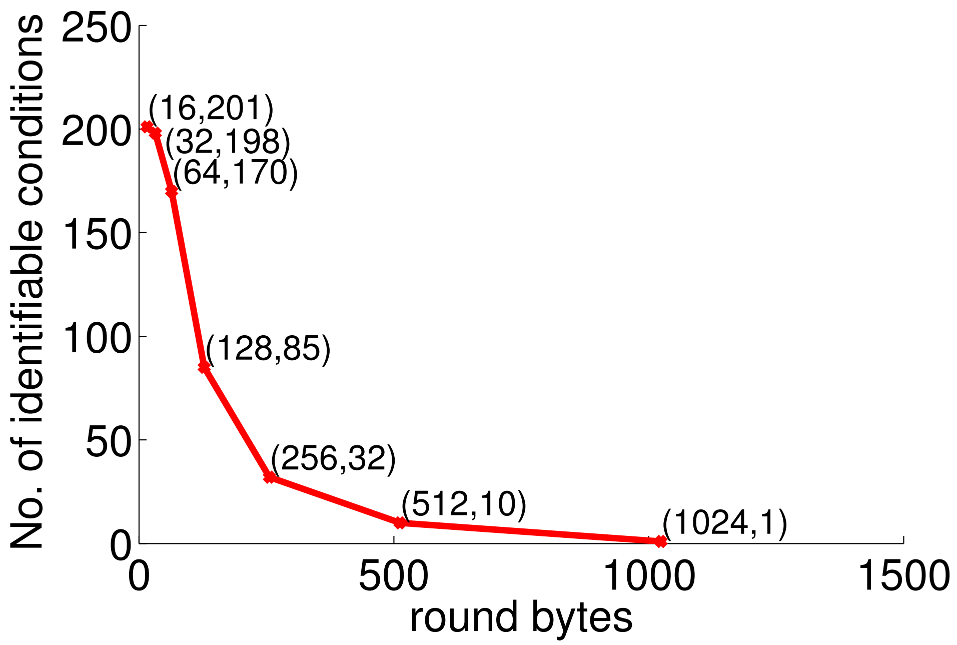


Figure 5.7: Effectiveness of round up/down mitigation technique

requests a target app's statistics, they pass the parameters such as the target's `uid` through IPC to the `UsageService`, which checks how the target app (`uid`) wants to release its data before responding to the query app with the data (which can be perturbed according to the target's policy). In our implementation, we deliberately kept the API interface unchanged so existing data monitor apps can still run.

5.4 MITIGATION EVALUATION

To understand the effectiveness of this technique, we first evaluated the round up and round down scheme using the WebMD app. Figure 5.7 illustrates the results: with α increasing from 16 to 1024, the corresponding number of conditions that can be uniquely identified drops from 201 to 1. In other words, except a peculiar condition *DEMENTIA IN HEAD INJURY* whose total reply payload has 13513 bytes with its condition overview of 11106 bytes (a huge deviation from the average case), all other conditions can no longer be determined from the usage statistics when the counter value is rounded to a multiple of 1024 bytes. Note that the error incurred by this rounding strategy is no more than 512 bytes, which is low, considering the fact that the total data usage of the app can be several megabytes. Therefore its impact on the utility of data consumption monitoring apps is very small (below 0.05%).

We further measured the delay caused by the modified APIs and the new `UsageService` on a Galaxy Nexus, which comes from permission checking and IPC, to evaluate the overhead incurred by the enforcement mechanism we implemented. On average, this mechanism brought in a 22.4ms delay, which is negligible.

Our defense mechanism is demonstrably efficient and effective when applies on the traffic usage information. Nevertheless, it is challenging to come up with a bullet proof defense against all those information leaks from unprotected local resources for the following reasons. a) Shared resources are present all over the Linux's file system from `/proc/[pid]/`, `/proc/uid_stat/[uid]`, network protocols like `/proc/net/arp` or `/proc /net/wireless` and even some Android OS APIs. b) Public (rest-of-the-world accessible) resources are different across different devices. Some of this information is leaked by third party drivers like the LCD backlit status which is mounted in different places in the `/sys` file system on different phones. c) Traffic usage is also application related. For the round up and round down defense strategy to be applied successfully, the OS must be provided with the traffic patterns of the apps it has to protect before calculating an appropriate round size capable of both securing them from malicious apps and introducing sufficiently small noise to the data legitimate traffic monitoring apps collect. A more systematic study is needed here to better understand the problem.

5.5 SUMMARY

In this Chapter we saw how Android shared filesystem resources can leak private information that an adversary can utilize to infer a user's health and financial information, her identity, and her location. The side-channel attacks I described, leverage the system's erroneous security design at both the Linux (shared files) and the framework layer (shared APIs). I performed an analysis to model the adversary which can exploit those shared resources and introduced new strategies such adversaries can employ. To mitigate these problems I proposed an alternative design of sharing filesystem resources, where information is released in coarse-grained manner to tackle such nuance adversaries. These results are published at a security conference [11]. Figure 5.8 visualizes this proposed addition to the smarpthone operating system.

After this work, a lot of other researchers have also focused on exploiting other shared filesystem resources. Zhang et al. [97] introduced another technique for mitigating such adversaries from a trusted userspace app which kills processes responsible for aggressive file monitoring. Finally, in version 6, Google has introduced further restrictions to third-party processes in accessing other processes information through shared files.

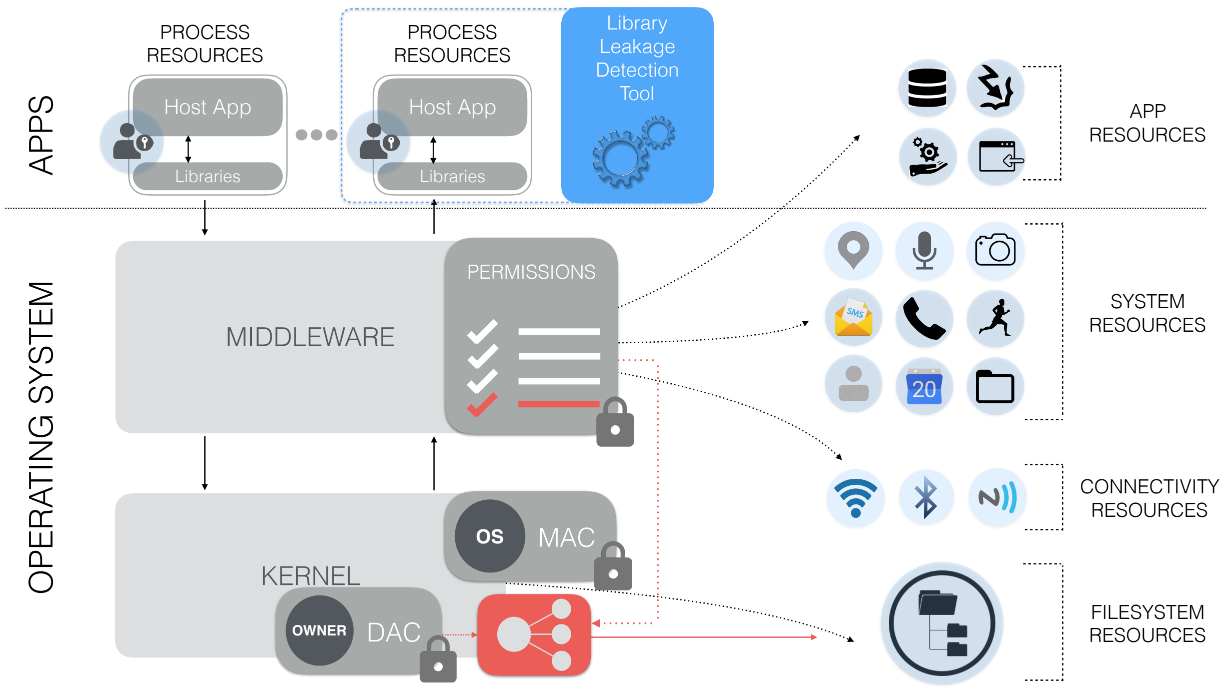


Figure 5.8: Alternative sharing of filesystem resources.

So far we analyzed adversaries exploiting shared intra-process privileges and shared filesysem resources and introduced designs of strategies for detection and prevention. Next I will look into the security of another shared resource on Android. In particular I will examine how Android allows third-party application developers to share their application components and access system resources.

CHAPTER 6: SHARING SYSTEM AND APPLICATION RESOURCES

Android uses a permission model to govern third-party application access to system resources. Moreover, it allows third-party application developers to share their own application components with other apps. Android provides developers the ability to declare their own*custom* permissions to protect their shared application components. In this Chapter I will present my security analysis on the Android permission model. I will show that this model suffers from grave design issues which allows untrusted third-party apps to escalate their privileges and gain unfettered access to system resources and permission-protected application components of other apps. I will also demonstrate, how we can re-design this permission model to eradicate this issues [12].

6.1 INTRODUCTION

As discussed in Chapter 2, in order to protect shared platform resources (e.g., microphone, Internet etc.), Android uses *system permissions*, which are a predefined set of permissions introduced by the platform itself. The permission model also provides the platform with finer-grained security as a means to protect Inter-Process Communication (IPC) between different shared app or system components. Specifically for this purpose, Android introduces *custom permissions*: these are application-defined permissions which allow developers to regulate access to their app components by other apps. In fact, the use of custom permissions is very common among third-party applications. According to our study on the top free apps on the Google Play Store, 65% of the apps define custom permissions, while 70% request them for their operation.

Unfortunately, design flaws and vulnerabilities in custom permissions can completely compromise the security of IPC, inevitably leading to exploits on third-party apps and the platform itself. Previous work has consistently found custom permissions to be problematic [85, 86], and as a response to these studies, Google made an effort to address the identified problems by releasing bug fixes. However, similar vulnerabilities still exist even after Google patched this initial wave of vulnerabilities. In this chapter, we present an analysis on the security provided by the Android permission model for shared platform resources and application components. In particular, the analysis results in one critical observation: *there is no separation of trust between system and custom permissions in the Android framework*, which leads to the manifestation of permission vulnerabilities.

First, system and custom permissions are currently insufficiently isolated and they receive the same kind of treatment from Android, which opens up opportunities for malicious apps to utilize custom permissions to obtain unauthorized access to platform resources. Second, there is currently no enforced naming convention for when declaring custom permissions—apps are allowed to declare custom permissions with *any* name they desire. This creates a confused deputy problem where a privileged app's protected resources can be utilized by unauthorized apps that possess different custom permissions declared with the same name as of the ones used by the privileged app to protect its resources. In order to systematically address these problems, I propose a design and corresponding implementation which we call CUSPER. CUSPER decouples the handling of custom permissions from system permissions to prevent an adversary from escalating their privileges and stealthily acquiring system resources. Additionally, CUSPER implements an OS-level naming convention for custom permissions to prevent custom permission spoofing. This is backward-compatible with existing apps and enables the system to properly identify custom permissions according to the developer signature of their definer apps. CUSPER is not only empirically tested but also proven to be formally correct with respect to two fundamental security properties: 1) there should be no unauthorized component access, and 2) there should be no access to high-risk ('dangerous') platform resources without user's consent.

6.2 ANALYSIS

Custom permissions provide security to IPC that apps harness for their operations. They are utilized by app developers to restrict access to shared app components as per the sensitivity of the protected resource. In this section, we investigate the prevalence of custom permissions among the top free apps on Google Play and showcase two high-profile apps that we selected to launch attacks on by exploiting the vulnerabilities in custom permissions.

6.2.1 Prevalence

Custom permissions provide security to IPC that apps harness for their operation. They are utilized by app developers to restrict access to shared app components as per the sensitivity of the protected resource. Here, I investigate the prevalence of custom permissions among the top free apps on Google Play. Towards this end we collected 50 top free apps from each of the Google Play Store categories (with some failures in collection) and in total analyzed 1308 apps to identify statistics regarding the use of custom permissions. We found that 65% of the apps in the dataset declare custom permissions (statically or dynamically) and 70% of them request custom permissions for their operations. This highlights that custom permissions are widely used by third-party application developers for sharing access to resources. It also highlights that any potential vulnerabilities in their use create risks for the security of both shared platform resources and shared app components.

6.2.2 Adversary Model

Here I consider an adversary that has the ability to crawl app markets (e.g., Google Play Store) to download victim apps of interest, reverse engineer them by utilizing several tools [164, 165, 166], and analyze the Android manifest files and source code of these apps to observe the cases where custom permissions are used to protect app components. The adversary can build and distribute on app markets a set of malicious apps that exploit potential vulnearabilities on the security of shared platform and application resources on Android.

After a close inspection of the Android runtime permission model we identified serious vulnerabilities. By exploiting these vulnerabilities, an app can bypass user consent screens for granting/denying permissions to obtain high-risk system resources and can also gain unauthorized access to protected components of other apps. We reported these to Google which acknowledged both of them as severe vulnerabilities. Next I will provide a high level overview of these attacks. I refer an interested reader to Gliz et al. [12] for more details.

6.2.3 Attack on Shared System Resources

Android runtime permission model (supported by Android 6.0 and onward) requires user's approval for granting apps permissions of protection level `dangerous`. This attack enables a malicious app to completely bypass the user consent screen and automatically obtain *any* `dangerous` system permissions [167].

First, the adversary creates an app that includes in its manifest file a custom permission declaration with the protection level `normal` or `signature` and sets this custom permission to be a part of a system permission group (e.g., storage, camera etc.). Then, they update the definition of this custom permission so that the protection level is changed to `dangerous` and proceed to push an update to their app on the respective app market. Here, this update can be pushed to all the app users after the app reaches a target user base. In addition, specific user groups can be targeted via the use of push services (e.g., Google Cloud Messaging (GCM) [168], which is used by 94% of the apps in our database that utilize custom permissions) that allow sending update notifications, and via enterprise app stores (e.g., Appaloosa [169]) that enable enforced targeted updates. The expectation is that since the custom permission is of level `dangerous`, the user will be prompted at runtime to make a decision on whether to grant or deny this permission in the runtime permission model. However, the malicious app automatically gets granted the permission. In addition, since the runtime permission model grants dangerous permissions on a group basis, the app also automatically obtains all the other requested dangerous permissions of the system permission group that the original permission belongs to. Same procedure can be followed to attack *any* system permissions group; hence, the adversary can silently obtain *all* system permissions simultaneously. Requesting dangerous permissions in the Android manifest constitutes no problems for the adversary, as permission requirements of an app are not directly presented to users at installation since Android 6.0. Hence, the user will be completely unaware that all these system permissions are granted to the app.

6.2.4 Attack on Shared Application Components

In this attack, the adversary exploits the lack of naming conventions for custom permissions on Android to launch an attack on a victim app that utilizes custom permissions to protect its shared components [170]. To do this, the adversary counterfeits the custom permissions of the victim app by reusing their names in her own permission declarations and takes advantage of the system's inability to track the true origin of permissions to access protected components of the victim app. The adversary's goal is to get the operating system to grant their apps a signature custom permission of a victim app that is signed by a different key than that of the adversary and therefore obtain unauthorized access to the components protected by this signature permission.

In order to achieve this, the adversary develops two applications: 1) a definer attack app which spoofs the custom permission of the the victim app by reusing the same permission name but changing the protection level to `dangerous`, 2) a user attack app which only requests this permission in its manifest file. The reason adversary needs two apps to carry out this particular attack is that Android currently does not allow two applications that declare a custom permission with the same name to coexist on the same device. Hence, the adversary's app cannot simultaneously exist on the device along with the victim app if it declares a permission with the same name to the one used by the victim. However, the adversary can divide their attack into two different apps, one that spoofs the custom permission as long as the victim app is not installed on the device, and a second one that only requests this permission and is able to coexist with the victim. The definer attack app needs to be installed first by the user, and this should be followed by the installation of the user attack app. After the spoofed permission of the definer attack app is granted to the user attack app at runtime, the definer attack app can be uninstalled by the user or updated by the app developer (for all users or targeted to a specific group by using services like GCM or Appaloosa) to remove the custom permission definition so that the victim can be installed afterwards. After the installation of the victim app, the user attack app is able to launch an attack on the victim to freely access victim's signature-protected components even though it is not signed with the same app certificate as the victim. Google acknowledged this as a high-severity attack since it bypasses operating system protections that isolate application data from other applications.

Note that there can be many ways for an adversary to get the user to install two applications on their device. For instance, the app developer can use in-app advertisements and links to direct the user to app stores to download their other app (e.g., Facebook and Messenger). Another effective way would be to utilize a common Android app development practice called plug-in architectures [171, 172], which on demand unravel new features to the user in the form of new apps in order to foster re-usability and save storage space by unlocking features only if they are necessary. An example to apps using this architectures is Yoga Guru [173], which unlocks users new yoga exercises—as part of new apps—only after making progress with the set of exercises they currently have.

This a serious problem which can affect very popular applications and lead to severe information leakage. For example *Skype* is an Android app by Microsoft that allows users to make voice and video calls over the Internet. It has 500,000,000+ downloads and a 4+ rating. Skype has an activity which can be invoked to start the call functionality to any telephone number. This activity is protected using a signature permission which, once bypassed, would allow the adversary to invoke calls to a specified person or number. This could have many implications. For example, it could be used as part of a suite of other spying capabilities. This attack is able to spoof the original permission and launch Skype calls without the user's knowledge through this protected activity.

Also, *CareZone* is a medical Android app produced by a company with the same name. It has 1,000,000+ downloads and has a 4+ rating on the Google Play Store. The app allows users to store medical-related information such as health background (e.g., blood type, medical conditions, allergies etc.), medication lists, medical contacts and their addresses, calendar events, insurance information, and photos or health files in an organized manner. It also features calendars to track appointments as well as to-do and notification lists for tracking tasks. All of this medical data and meta-data is stored in a single content provider which has been exported and is protected by a signature permission. There are no other dynamic checks on accessing the content provider. Our attack is able to bypass the signature requirements and read the entire content provider, which gives us access to the aforementioned sensitive data without the user's explicit consent or knowledge. The fact that all this data was stored in a single content provider seems to reflect the developer's implicit reliance on the security guarantees provided by the platform.

6.3 MITIGATION DESIGN

System permissions are defined by the platform—a privileged principal—whereas custom permissions are defined by apps—less privileged principals. The former kind typically protects shared system resources while the latter is utilized to protect shared application resources. The fact that the system treats them the same, results in severe security vulnerabilities as the ones we discovered (Section 7.2). Note that other vulnerabilities might also exist or might manifest in the future because of this non-separation between the two classes of permissions. Ideally, we need a new design which will allow us to achieve a clean separation of trust between the system and custom permissions. This way, the system will have to handle the two cases differently avoiding logic errors and at the same time, any potential vulnerabilities in third party app custom permissions will not allow privilege escalation, which can enable exploits of system permissions and platform resources. However, such a new design needs to be carefully constructed to be practical. In fact, it needs to be as simple as possible to be adopted in practice, and backward compatible. A complete redesign of the Android permission model would require non-trivial modifications to the Android framework while thousands of apps relying on custom permission would be immediately affected. Instead, in our work, we introduce two main design principles which can easily be incorporated into the current design of Android permissions, require no changes to the existing apps, and can guarantee a separation of trust eliminating the threat of privilege escalation in permissions, without breaking the operation of system and third-party components that rely on permissions. These design principles are: (a) decoupling of system and custom permissions; (b) new naming scheme for custom permissions. I implement these in our system that we call Cusper.

6.3.1 Isolating System from Custom Permissions

Currently, Android does not maintain distinct representations for system and custom permissions, that is, the system does not track whether a permission originated from the system or from a third-party app. Due to this reason, both types of permissions are also granted and enforced in the same fashion. As as shown in Section 7.2, this is problematic as it allows apps to use custom permissions to gain unauthorized access to system permissions. For example, a malicious app can declare a custom permission and assign it to a system permission group. This behavior is allowed by Android since it does not differentiate between the two permission types. Thus, when the custom permission is granted, the app automatically gains access to the system permissions in the same group, essentially elevating its privileges from a permission defined by a low trust principal to permissions defined by the platform. Cusper *never allows custom permissions to share groups with system permissions.* Additionally, the fact that Android internally treats all permissions the same way is an important limitation with security repercussions: platform developers tend to overlook the existence of custom permissions when handling permissions. The *custom permission upgrade attack* is an example of that. To overcome this, *system and custom permissions have distinct representations* in Cusper. By doing this, we can now differentiate between the two types of permissions during granting as well as enforcement and apply different strategies depending on the type of permissions.

Implementation. In order to decouple the two permission kinds, one could create separate object representations and data structures. This would require a complete redesign of the Android permission implementation throughout the Android framework which we think is impractical. Alternatively one could use existing fields in the current permission representation in Android which can give us information on the source of a permission. `BasePermission` class has a `sourcePackage` field that indicates the originating package of a permission. For system permissions defined in the platform manifest, this field is set to `android`, for system permissions defined in system packages, it *usually* starts with `com.android`, and for custom permissions it is the package name of the defining third-party app. However, the package name itself cannot be used to identify whether a package is system or third-party, as there are already system apps with package names not starting with `com.android` (e.g., browser) and even third-party apps can have package names starting with the system prefixes (`com.android` etc.). Hence, `sourcePackage` is not a reliable identifier of whether a permission is custom or system.

Instead, a both practical and robust approach, would be to extend the object representation of a permission with an additional member variable, indicating whether this permission is a custom permission. In Cusper, I implement this by augmenting the `BasePermission` and the `PackageParser. Permission` classes. The value of the new variable is assigned when an app's manifest is parsed (`PackageParser.java`) during installation or upgrade. If the app under investigation is untrusted (as indicated by its non-platform signature), we mark its permissions as custom. When parsing an untrusted app's manifest, Cusper further checks whether the app developer assigned a custom permission to a system permission group. In this case, Cusper ignores the assignment, which results in the permission having no group. Moreover, if the app declares a custom permission group, Cusper ensures it does not use a system permission group prefix (`android.permission-group`). In essence, we thwart the vulnerability while ensuring that even if future vulnerabilities manifest, there will be no escalation to system permissions.

After doing this, we can now track the creation of custom permissions by third-party apps. In order to particularly thwart the *Custom Permission Upgrade*, when a custom permission—which we can now effectively and efficiently differentiate from system permissions—is created with the protection level normal or signature (i.e., install permission), Cusper ensures that the permission will not be granted automatically if it is later updated to be a dangerous (runtime) permission.

6.3.2 Naming Conventions for Custom Permissions

Android allows third-party apps from different developers to declare permissions with the same name. The current solution is to never allow two permission declarations with the same name to exist on the device. While this sounds effective, it is unfortunately unable to stop the second attack we demonstrated: a definer app A declares a permission and another app B gets the permission granted. When the first app A is uninstalled and a victim app C comes in declaring and using the same permission to protect its shared components, it is vulnerable to confused deputy attacks from app B. To solve this problem I introduced a new *internal* naming convention in Cusper: Cusper enforces that *all custom permission names are internally prefixed with the source id of the app that declares it.* Note that Cusper does not expect app developers to change their practices. Custom permissions are still declared with their original names in the manifest files of apps to allow backward compatibility. However, in Cusper, the custom permission names are *translated* to `source_id` : `permission_name`. Thus, even if permission revocation such as in the above attack scenario fails, the attack will be rendered ineffective. This is because, as far as Cusper is concerned, the granted permission to app B will be an entirely different permission than the one app C uses to protect its components.

Choosing the appropriate source id is not straightforward. Consider for example using an app's package name as the `source_id`. This introduces two main problems. First, repackaged apps distributed on third-party application markets could use the package name of an app distributed on Google Play. Thus, the repackaged app could take the role of the *definer attack app* (see Section 7.2) and instigate a confused deputy attack. This is possible since the repackaged app and the victim app share the same package name and a permission created by the repackaged app cannot be distinguished from the one created by the victim if they share the same permission name. Second, using the package name as the `source_id` might break the utility of `signature` custom permissions for some use cases. For example, developers that have a set of applications which utilize each other's components, commonly use signature permissions to protect the components of their apps from others. Since the installation order cannot be determined in advance, each app in the set has to declare the same permission (i.e., same name and protection level) in their manifest to make sure this permission will be created in the system. If permissions are prefixed with their declarer app's package name, then the system will treat them as different permissions. Therefore, any attempted interaction will be wrongfully blocked.

Cusper instead uses the app's signature as the source id to prefix permission names. In the case of a repackaged app, assuming the malicious developer does not possess the private keys of the victim app developer, the declared permission will be a different permission in the system than the victim's declared permission. Moreover, utility is preserved since custom permissions with the signature level will be treated as the same permission as long as they come from the same developer, which is exactly the purpose. Note that the same scheme can also be utilized for permission tree names.

Lastly, the official suggestion to Android app developers which declare custom permissions, is to use names that follow the reverse domain name paradigm (similar to the one for package names). However, Android does not enforce this naming convention. Even though it will ignore a permission declaration with the exact same name as an existing permission, it allows third-party apps to use a system permission name prefix (e.g., `android.permission`) in their custom permission declarations. Since permission names and groups are currently the only information the system has regarding the intention and source of the permission, this treatment is at the very least hazardous. Cusper, we addresses this naturally as it adds prefixes to permission names and *never allows a custom permission to use a name prefix reserved for system permissions.* Since we decouple the two types, we can now identify the type and origin of permissions, and readily enforce this rule. To maintain backward compatibility and ensure that the custom and system permission names are distinct, we also ignore system permission names for custom permissions (as the original system currently does).

Implementation. To thwart custom permission spoofing attacks of any sort (including our *Confused Deputy* attack), apart from distinguishing between custom and system permissions, we further need a way to track the origin of custom permissions and uniquely identify them in the system. Towards this end, I implement a naming convention for custom permissions in Cusper. The implementation consists primarily of a permission name translation operation to prefix the permission names with their source id to ensure uniqueness in the system. This translation happens during installation and update for the names of the declared custom permissions and requested install time permissions, and at runtime for dangerous permissions and the permissions used to protect components (guards).

At the time of installation, Cusper allows the system to parse declared custom permission names from an untrusted app's manifest; however, it translates their names to be prefixed with the hash of their app's signature before the actual permission is created in the system. In the case an app is signed with multiple keys, Cusper sorts the hashes of the keys and concatenate them. Note that one could attempt to perform the translation in place. For example, it could perform the translation while parsing a permission name from the manifest. However, at that point, the app's certificates are not yet collected. Doing so would incur non-negligible overhead since it involves a number of file opening and reading operations (`PackageParser.collectCertificates()`). Instead, Cusper keeps the parsed data unaltered until after the certificate collection normally happens. Then, it scans the package's meta-data to perform the necessary translations. This approach resulted in great performance savings which keep Cusper's performance comparable to the original system (see Section 6.4).

Similarly, Cusper first proceeds to translate the names of the requested permissions during installation or update. This is done to correctly grant install time permissions (i.e., normal and signature). Note that a requested permission might not necessarily exist in the system at this time and therefore the permission name translation cannot happen. For example, an app that declares the permission might be installed at a later point in time. Since the declared permission will be translated, it will essentially be treated as a different permission than the one requested, violating application developers' expectations. This is not a problem with install time permissions: the permission correctly will not be granted as its definition does not exist on the system at the time of installation, which is on a par with the behavior of the original Android OS. In the case of dangerous permissions which are granted by the user at runtime, we need to dynamically check for existing declared permissions. Therefore, Cusper performs a requested permission translation at runtime. In particular, when a dangerous permission is to be displayed to the user, Cusper performs a scan on all declared permissions to find a custom permission with the same suffix as the requested permission. In Cusper's implementation, the system does not allow declaration of custom permissions with the same name which ensures that the scan will result in only one possible permission. This is also the current design of Android which does not allow two apps to declare the same permission.

Note, however, that since we prefix custom permissions, one could extend Cusper to allow multiple apps to use the same custom permission names. In case of an app requesting that permission, we could readily resolve the conflict if one of the declarers has the same signature. If all declarer apps come from different developers, a mechanism similar to Intent filters could be utilized to allow the user to select the appropriate declarer app.

It is worth noting that one could alternatively create a separate hash map for custom permissions (e.g., key-value pairs of (suffix, prefix)) to avoid the linear scan for suffix lookup. However, this hash map would need to be kept consistent with the original hash map for all declared permissions in the system (e.g., tracking addition/removal of permissions), which is hard to achieve since there are multiple places throughout the Android source code where this in-memory data structure is updated or sometimes even constructed from scratch from files in persistent storage. Hence, for the sake of consistency and not breaking utility, we prefer the linear scan method and do not change the structure of the in-memory data types for permissions. As I will show in our evaluation in section 6.4, this method does not result in any significant overhead.

Finally, as for permissions that are used to protect app components (guards), their name translation takes place at runtime during enforcement since a guard might not necessarily exist in the system at the time of installation.

6.4 MITIGATION EVALUATION

Here I provide evidence regarding the practicality of Cusper's system implementation. Toward this end, I empirically evaluate Cusper's implementaiton on Android (version 6), with respect to (a) its ability to thwart the specific attacks we presented and (b) its performance overhead incurred in the affected Android operations.

Effectiveness: empricical evaluation. To evaluate the effectiveness of Cusper, I carried out the two attacks mentioned in Section 7.2 on Cusper-augmented Android and showed that both attacks fail.

First, I attempted the *Custom Permission Upgrade* attack on Cusper-augmented Android and verified that the attack could no longer succeed. The user is correctly being consulted to grant the pemission by the system once a permission declaration changes from a normal protection level to dangerous. Moreover, I verified that a third-party app can neither assign a custom permission in a system permission group, nor declare a custom permission group using the system permission group naming convention. At the same time, normal operations of benign third-party and system apps are preserved.

With respect to the *Confused Deputy* attack, using the the apps mentioned in Section 7.2 (i.e., Skype, CareZone) as well as other real-world apps, I verified that the attack can no longer succeed while again utility is preserved with Cusper. I further tested that permission revocation happens correctly when the declarer app is uninstalled. Moreover, I verified that declared custom permissions are prefixed by a hash of the app developer's signature, and the same happens for the custom permissions used to protect app components. Finally, I tested that granting normal and signature permissions at installation time, granting dangerous permissions at runtime, and using the permissions to access protected app or system components, happen correctly; hence, Cusper does not break any utility.

Efficiency. In evaluating the performance of our system, we focused on the operations affected by Cusper modifications on the original Android operating system. These include the app install operation, the app uninstall operation, runtime (dangerous) permission granting, and permission enforcement. Here I omit the evaluation for the app update operation as its performance is similar to that of app installation. A Nexus 5 phone running Android 6.0 (android-6.0.1_r77) was usedfor all experiments. According to a previous study, Android users have on average 95 apps [174] installed on their devices. In addition, during the analysis stage, apps were found to create one custom permission on average. In order to evaluate Cusper under realistic conditions, I mimic this average case in the experiments and create 100 custom permissions along with all of the system permissions.

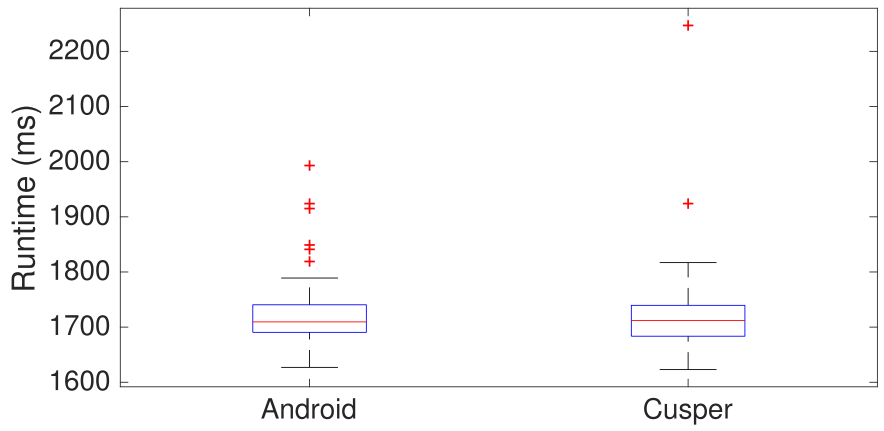


Figure 6.1: Performance evaluation of Cusper for app installation

In the *app install* and *app uninstall* experiments, the *Android Debug Bridge* (adb) was used to install and uninstall an app of size 1.2 MB 100 times. The app declares a custom permission, with `protection-level` dangerous, uses the permission, and declares a service which is protected by that permission. For the experiments the `installPackageAsUser()` method in the `PackageManagerService` class was instrumented to get the start time of

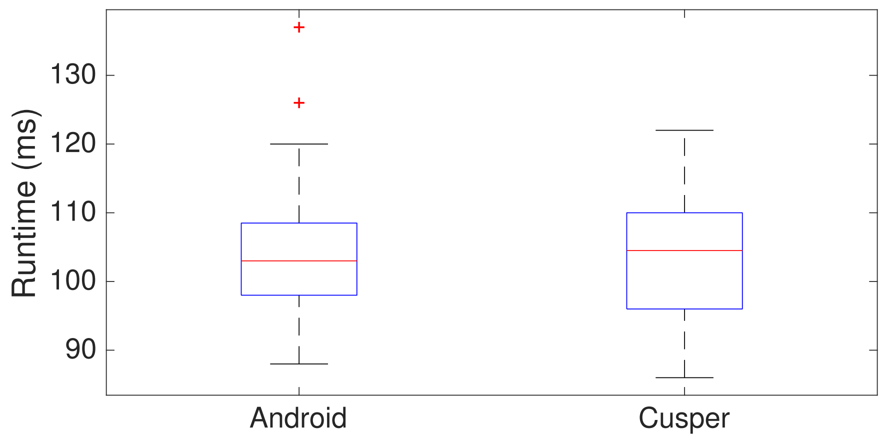


Figure 6.2: Performance evaluation of Cusper for app uninstallation.

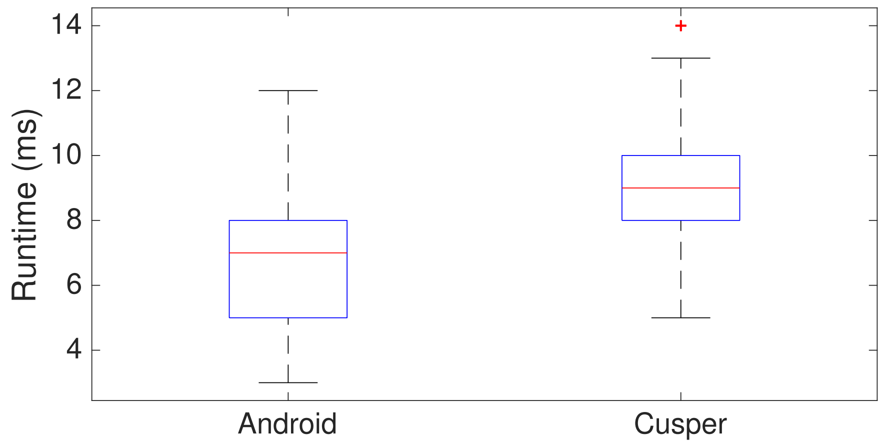


Figure 6.3: Performance evaluation of Cusper for runtime (dangerous) permission granting.

app installation. The end time is captured at the point before the system broadcasts the `ACTION_PACKAGE_ADDED` intent indicating the completion of the package installation. For app uninstallation, the methods `deletePackage()` and `deletePackageX()` were instrumented, to get the start time and end time respectively. Figure 6.1 and Figure 6.2 illustrate the experiments' results.

Next Cusper is compared with the unmodified Android version (*Android*). During installation, Cusper performs checks during parsing, performs the permission translation, and handles the permission revocation. While parsing, it checks and stores whether a permission definition is for a custom permission and it enforces the permission group checks. Then, it parses the in-memory meta-data of an app to perform a custom permission translation. Nonetheless, as shown in the evaluation, the performance overheads are indeed negligible: there is no statistically significant deviation between Cusper and the original version.

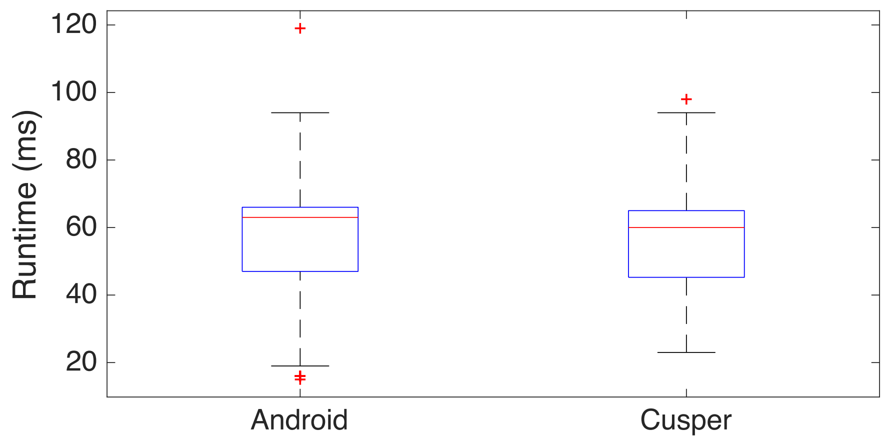


Figure 6.4: Performance evaluation of Cusper for component access (Activity).

In addition, I evaluated the operation of granting a dangerous permission at runtime. For this we can use an app which requests a custom permission previously defined in the system. Note that this is a process which involves user interaction: the system pops up a dialog box asking the user to grant or deny the permission request. This process was automated and ran this experiment 100 times. However, to avoid the unpredictable temporal variable of user interaction, the time between the display of the dialog box and the time the dialog box is removed is not counter. The evaluation instrumentation is deployed in the `GrantPermissionsActivity` class. Figure 6.3 summarizes the results. Evidently, Cusper does not incur any distinctive overhead.

Finally, I present the evaluation of the performance of permission enforcement for custom permissions. For this case, I show performance results for accessing permission-protected app components of all kinds: activity (Figure 6.4), service (Figure 6.5), broadcast receiver (Figure 6.6), and content provider (Figure 6.7). As can be seen, Cusper indeed incurs negligible overhead for all types of component invocation operations that require permission checks.

In summary, the modifications to the Android system are shown to have no perceivable performance overhead while they greatly strengthen the security of the Android OS.

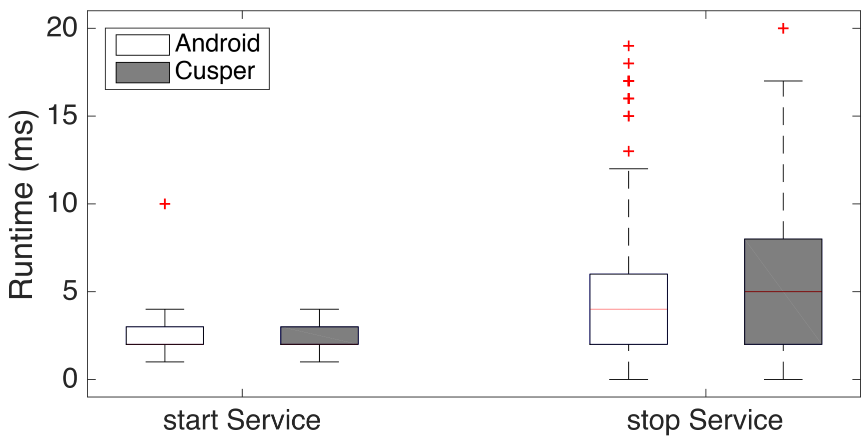


Figure 6.5: Performance evaluation of Cusper for component access (Service).

6.5 MITIGATION FORMAL VERIFICATION

As a part of the software development process, to verify that a piece of software meets the requirements, it is common practice in industry to rely *only* on software testing and not provide formal proofs of program correctness for the underlying model as formal verification is highly time consuming, difficult and expensive. However, fundamental components like a permission system are naturally worth more effort as any failure in such components can make way for critical security vulnerabilities or even render the security of the whole system ineffective. Additionally, numerous security bug reports on similar issues present further proof that the current testing methodologies for Android permissions are not completely effective and a better way of proving program correctness is necessary.

Formal verification allows us to systematically reason about the design of Cusper by covering many cases that would otherwise be difficult to investigate with static analysis or testing. This is not to say software testing is unnecessary when a formal correctness proof is provided. In fact, we still need software testing to verify that our implementation conforms to our proposed model (which is formally verified to be correct). On the other hand, "formal verification reduces the problem of confidence in program correctness to the problem of confidence in specification correctness" [175]. In other words, verification is performed

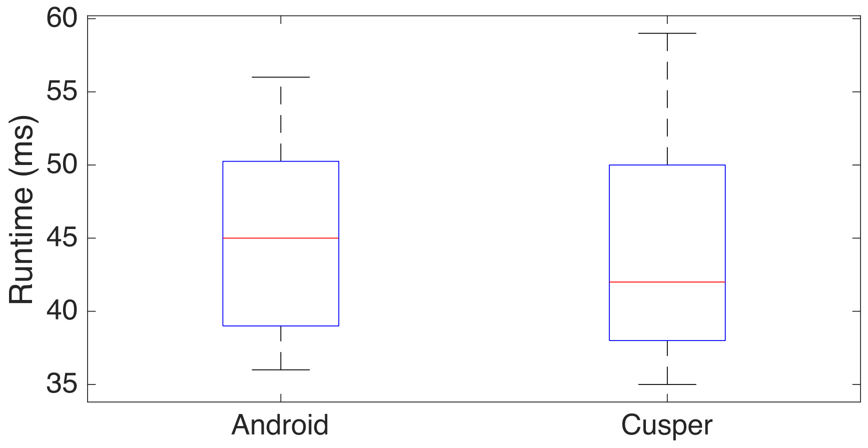


Figure 6.6: Performance evaluation of Cusper for component access (Broadcast Receiver).

not on the actual implementation but on a representation that is as close to the original implementation as possible. This is because it is challenging to perform formal verification at a scale required by source code, especially at the huge scale of the Android source code. Progress in the area does exist towards this for other programming languages [176], but such approaches are typically employed at the time of development, where the developer is required to annotate the code. This would be infeasible in our case where a large portion of the Android source code is already written. Additionally, correctness is proved *only* with respect to a set of fundamental properties that were defined based on the specification. There is *no* guarantee the system will behave correctly under any condition that was not a part of the defined properties or in case of redesigns of the system that might invalidate the model assumptions. Hence, the state of the art formal verification is not a silver bullet but still a best effort technique for proving correctness.

To analyze the security of Android permissions, previous work proposed formal models that correspond to the older Android versions which supported only install-time permissions [78, 79, 80]. Unfortunately, no such model exists for Android's currently-adopted runtime permissions. In [12], we build the first formal model of the Android runtime permissions and use it to verify the correctness of Cusper. This allows us to investigate Cusper under many

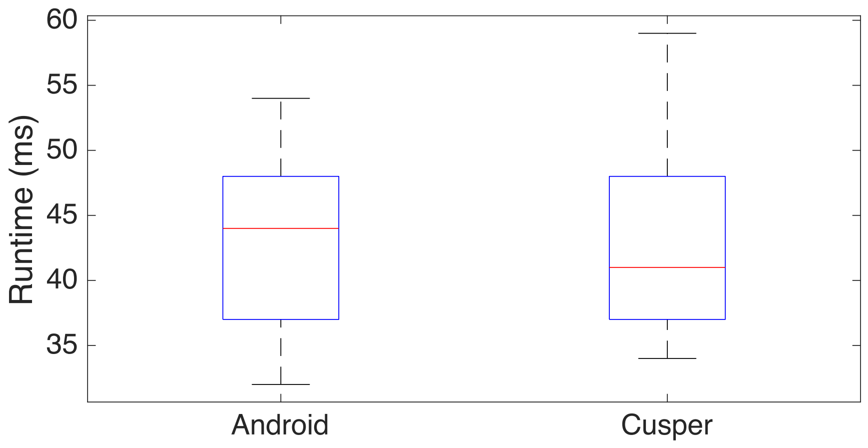


Figure 6.7: Performance evaluation of Cusper for component access (Content Provider).

cases such as all possible installation orders and app declarations. Note that having such a formal model has other benefits; for example, security researchers can use it to verify other properties of their interest on the runtime permission model. We based our model on the Alloy implementation of [78] as Alloy is a high-level specification language that is easy to interpret. However, we spend a significant amount of effort to extend this model to conform to the official specification for the new runtime permissions [35].

We analyze the security of the model through an automated analysis and show that when it is augmented with the design of Cusper, two fundamental security properties that were violated in the original Android runtime permission model are satisfied in Cusper. The properties are: 1) there should be no unauthorized component access, and 2) there should be no access to high-risk ('dangerous') platform resources without user's consent. This greatly imprives our confidence regarding the security offered by the proposed system. I refer an interested reader to Guliz et al. [12] for the implementation of the formal model.

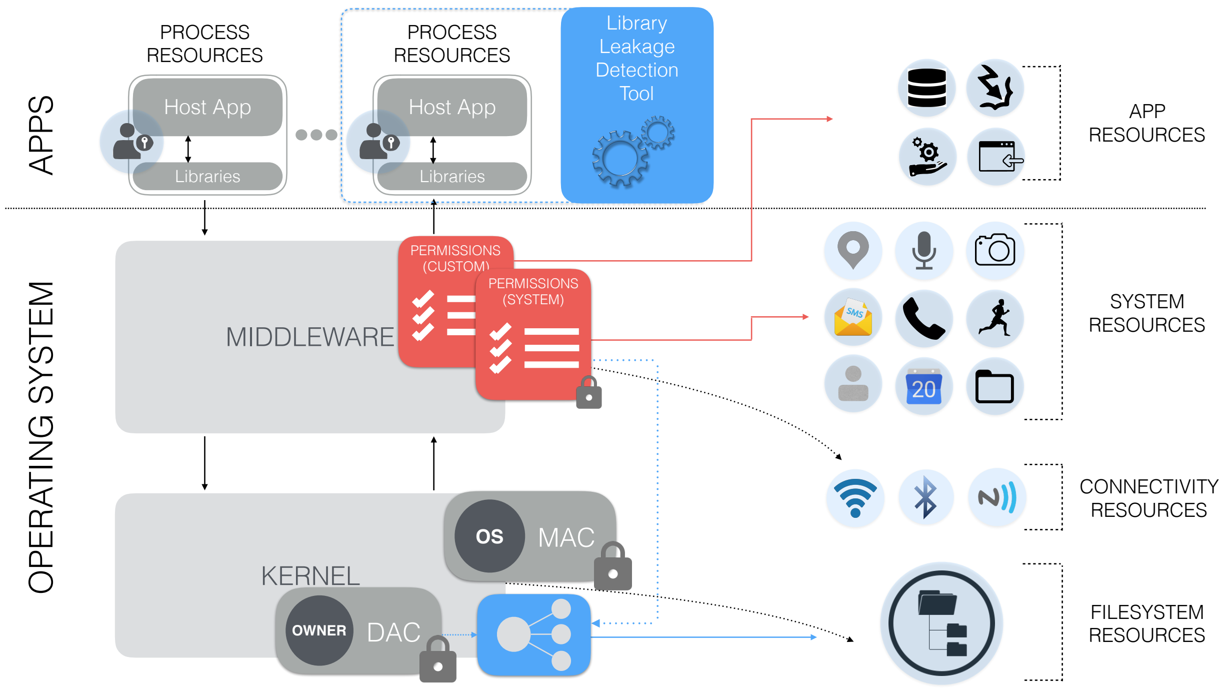


Figure 6.8: Alternative permission model which splits management of custom and system permissions.

6.6 SUMMARY

In this chapter I analyzed the security of shared platform and application resources. I found that the permission model used to protect access to these resources sufers from serious security vulnerabilities which allow a malicious application to ain unfettered access to all sensitive system resources and protected application components. My analysis identified the root causes of those vulnerabilities which stem primarily from (a) a lack of distinction between system permissions declared by a privileged principal (the system) and custom permissions declared by untrusted principals (third-party applications) and, (b) the incapacity of the system to track custom permission ownership. To mitigate these problems I proposed a redesign of the Android permission model which systematically eradicates those problems. Figure 6.8 visualizes this change on the smartphone operating system. The proposed system (Cusper), is empirically found to be both effective and efficient. Moreover, Cusper is formally verified to be correct with fundamental security properties grealy improving our confidence regarding the security offered by the proposed mechanisms. The vulnerabilities discovered during the analysis of shared system resources and application components, were reported to Google which acknowledged them as serious security vulnerabilities. Google will be addressing these issues in future releases of Android, starting from version 8.

So far we analyzed adversaries exploiting shared intra-process privileges, shared filesysem resources and shared system resources and application components. In response I introduced effective and efficient designs of strategies for detecting and preventing these problems. On Android, third-party apps also share communication channels to connect with devices in proximity and other external resources such as incoming SMSs. Next I will present my analysis on the security of such shared communication channels.

CHAPTER 7: SHARING DIRECT COMMUNICATION CHANNELS

In this Chapter I will present my analysis on the communication channels Android offers to third-party applications for accessing external resources. I will demonstrate that yet another shared resource is subject to attacks from a mobile adversary; I will also illustrate the design of a robust and flexible defense mechanism within the smartphone OS which can achieve fine-grained access control to such shared resources [13, 14].

7.1 INTRODUCTION

As a mobile platform Android is equipped with communication channels to enable connections with an assortment of external resources. As we seen on Chapter 2, the Android Security Model only controls the access right on the channel used for communicating with such external resources, such as Bluetooth, NFC and Audio devices, SMSs. As long as an app acquires a system permission for such a channel, it automatically gains access to any information communicated through it. Specifically, all apps with the same permission are either affiliated with the same Linux group (GID) in which case the kernel enforces the access control or being checked whether they owned the appropriate permission by the framework right before the appropriate system Service decides to return or not the requested data. Nevertheless, Android does not have the capability to overhaul any semantics of the data being requested. For example it will either allow reading all SMSs or deny reading any of them. For this I argue that the security model is too coarse-grain to satisfy the utility of the apps while preserving the confidentiality of the data originating from external resources.

In this chapter I will elaborate on my studies on the risks associated with this coarse granularity of the Android security model when it comes to share communication channels. On Section 7.2.1 I will take a closer look at the Bluetooth channel and elaborate on attacks stemming from the fact that a Bluetooth device is paired with the phone instead of pairing with the app that actually wants to use it. I will refer to this as the `Device Mis-Bonding Threat` or simply `DMB`. After that I will discuss (see Section 7.2.4) other risks rising from the coarse granularity of the OS's security model and its inherent inability to safeguard the SMS, Audio, and NFC channel. Lastly, I will present the design of a system which offers strong security guarantees for shared communication channels 7.3

7.2 ANALYSIS

7.2.1 Bluetooth Mis-Bonding Problem

The fundamental cause of the DMB problem is the inadequacy of the Android security model in protecting communication channels with external devices. As an example, consider a medical device that communicates with its Android app using Bluetooth. To make this happen, the smartphone hosting the app first needs to *pair* with the device, which forms a *bond* between the phone and the device. This pairing process yields a set of bonding information, which allows these two devices to connect to each other automatically in the future. The bonding information includes the external device's MAC address and its Universal Unique Identifier (UUID), together with a secret link key for authentication and encrypted communication (when the devices decide to do so). Note that such a bond relation is only established on the device level; there is nothing to prevent an unauthorized app (with Bluetooth permissions) on an authorized phone from connecting to the device. This permission also makes the app a member in the `net_bt_admin` group. As a result, the unauthorized app is given the privilege to break the bonding with an authorized medical device and pair the phone with a malicious one configured with the former's bonding information so as to feed fake medical data into a patient's medical record [13].

Given the limitations of the Android security model, device manufacturers are on their own to address this security risk. One thing they can do is to design a way to secure the communication between the device and its official app. An instance we are aware about is the Square credit card reader [5], which connects to a smartphone through its audio port. Its early version is vulnerable because every app with audio permission can read from it. The later one comes with an encryption capability: the reader encrypts the data (using AES) collected from a credit card using a hard-coded key and transmits the ciphertext through the phone to the web. Most other devices, however, do not provide any app-device level protection, as confirmed in our measurement study (Section 7.2.3), possibly due to the fact that most of them are simple sensors, without sufficient computing resources to support cryptographic operations. These devices can upload the data to the online service through the smartphone, which also provides an interface for the user to see and analyze their data. Encrypting this data in the device and just using the phone as a communication relay would severely affect the usability of the device, as the user would not be able to use her phone to see her data. All the devices we analyzed have apps that display the user data on the smartphone. Hence, the treatment adopted by Square does not seem to be suitable for these devices.

Adversary Model and Targeted Bluetooth Devices

In my research, I have conducted a study on this under-researched yet critical security problem. As the first step, the study focuses on Bluetooth healthcare devices, which are becoming increasingly popular in recent years, especially with the advent of wearables. The security risks discovered and the new adversarial technique introduced, are extended to other types of external resources, as we will see on Section 7.2.4.

I assume that a malicious app is present on the victim's Android phone with both the Bluetooth and Bluetooth Adminstration permissions. These two permissions are claimed by almost all the Bluetooth-capable apps. For a data-injection attack, in which a malicious party clones the target device, we also assume that the fake device can be placed close to the victim's phone (within 100 meters).

As mentioned before, I will first focus on Bluetooth devices. Specifically, I analyzed four popular healthcare devices. All of them except one (iThermometer) are FDA approved Class II medical devices [177], in the category of X-ray machines, infusion pumps, etc., which are used to deal with real patient care and life critical information. The first three devices either have their online services available or are capable of synchronizing the information they collect with other cloud based health-services. Here is more detailed information about these devices:

- *Bodymedia Wireless LINK Armband* [178] is one of the most popular activity monitoring systems, which has been used in over 120 clinical studies [179]. It utilizes four different sensors to collect data about the user's motion, temperature, perspiration, etc., for accurate calculation of calories burned and monitoring of sleep patterns. The output of the device can be displayed by a mobile app running on Android or iOS, and further synchronized to an activity manager website. Disclosure of the data can leak out the user's health status and daily activities.

- *Nonin Onyx II 9560 Pulse Oximeter* [180] is one of the best wireless finger pulse oximeters. Along with a smartphone app, it enables clinicians to remotely monitor blood-oxygen saturation levels and pulse rates of the patients with chronic diseases such as Chronic Obstructive Pulmonary Disease (COPD) or asthma [177]. The device uses Bluetooth to connect to the smartphone, which can deliver the data to the health provider, online health services or stored locally for later analysis. The data collected here is also critical for understanding the patient's status and choosing an effective treatment. This device is Microsoft HealthVault[1] certified [177].

[1]Microsoft HealthVault is a free online service for personal health information management.

- *Entra Health System MyGlucoHealth Blood-Glucose Meter* [181] is one of the most popular glucose monitoring devices. It comes with a complete diabetes management system (including testing at home) uploading data to the online account through its Android app, which helps a patient manage her disease and share this data with her health provider. Glucose levels determine the amount of insulin to be injected into the patient's body, which is private and also life-critical: a wrong amount of injection can have severe implications, including death [182]. Along with FDA, this device is also approved by CE[2] and is fully HL7[3] compliant [181].

- *iThemometer* [183] is an electronic thermometer that works with Android through Bluetooth for personal health or long-distance monitoring of elderly persons or babies. The body temperature is an indicator for life-threatening conditions like infection.

All these devices involve user's critical data, whose confidentiality and integrity is important to her health and well-being. In the presence of the malicious insider app, however, I will show that such data becomes extremely vulnerable to the DMB threat.

7.2.2 Data-stealing Attack

In my research, I investigated the feasibility of data-stealing attacks on Bluetooth devices, in which a malicious app running on the victim's phone attempts to steal sensitive data collected by the target device. The attack turns out to be more complicated than it appears to be: particularly, depending on the nature of a device, the malicious app needs to capture a small time window during which the device is on and in proximity, under the competition of the official app that also wants to make a connection to the device. Here I describe how an adversary can overcome such technical challenges to design end-to-end attacks on real devices.

Attack Strategies

Given the `BLUETOOTH` and `BLUETOOTH _ADMIN` permissions, a malicious app appears to have all it needs to steal data from these healthcare devices, and merely because Android does not mediate which app is supposed to connect to the devices. Any app with access to the channel immediately gets access to all data communicated through it. In practice, however,

[2]CE Mark is medical device approval mechanism in Europe.

[3]HL7 – Health Level Seven International – is a globally interoperable standard for health information exchange.

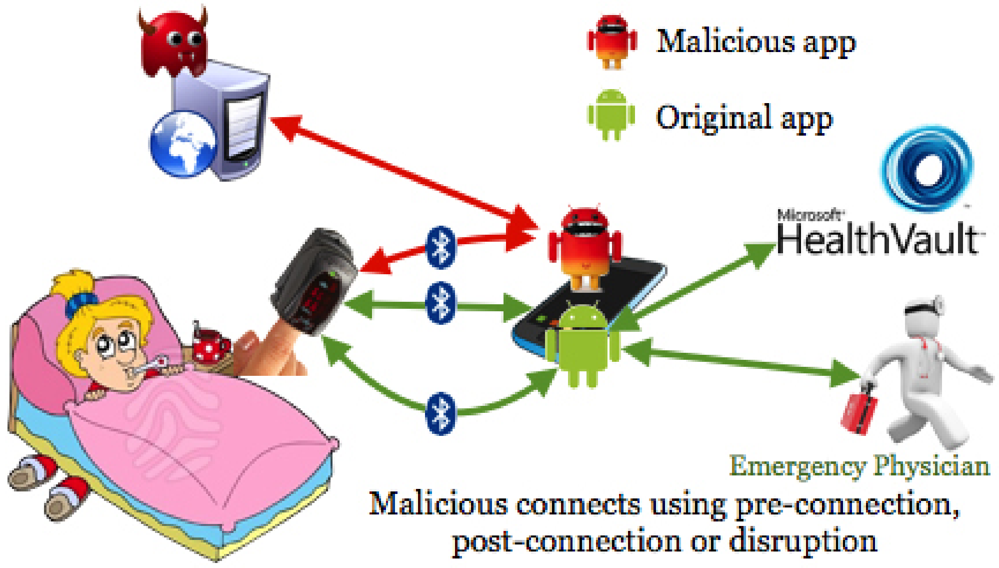

Malicious app
Original app
Microsoft HealthVault
Emergency Physician
Malicious connects using pre-connection, post-connection or disruption

Figure 7.1: Data-stealing Attack

the situation is much more subtle than it appears to be at a first look: a malicious app must not be oblivious to the fact that the target device could or could not be in proximity and even when they are, for some of them one needs to push a button or take some actions to activate their Bluetooth services. Specifically, the Bodymedia armband is activated a few seconds after it is put on one's arm; the iThemometer has such a button on it; the Nonin pulse oximeter turns on when one inserts her finger into the device and turns off once she takes out her finger; and the MyGlucoHealth meter has a button for activating the Bluetooth and the meter turns off automatically after sending data to the phone. Also complicating the attack is the presence of the official app. Once the official app establishes a socket connection with the target device, the malicious app cannot directly talk to the device before this connection is torn down and vice versa.

A straightforward solution is an opportunistic strategy in which the malicious app either periodically invokes the service discovery protocol to find out whether the target device is in its vicinity or blindly makes repeated connection probes, hoping to get to the device as soon as it shows up. However, neither of these approaches works well in practice due to alarmingly increasing power usage of the Bluetooth radio, a power-consuming practice that is usually suggested against [184]. For instance, a user may keep the Bluetooth communication off to save power. Then, when she wants to use it, she runs a Bluetooth-capable app that automatically turns on Bluetooth. A malicious app using this strategy must repeatedly enable Bluetooth to discover the target device; this consumes more battery power than expected and could also be noticed by the user, given the presence of the Bluetooth icon on the top notification bar of the Android phone.

In my research, I found that an adversary can use a lightweight and stealthy strategy to perform the surveillance. Simply put, the execution of the device's official app is a strong indication that the device is in action and also within the connection range of the target device. Based on this observation, the malicious app can keep checking when any of the target apps launches, an event that can be used to trigger an attempt to catch the window of opportunity. Specifically, our app, which works as a service in the background, periodically runs the Android API `getRunningTasks()` to get the app running in the foreground in constant time $O(1)$. This needs an additional permission `GET_TASKS`. Alternatively, we can use `getRunningAppProcesses()`, which does not need any permission, but returns a list of running processes in an unspecified order that the malicious app needs to traverse in search for the target app, which takes $O(n)$ running time, where n is the number of concurrently-running processes on the phone. The same result can be achieved by executing the Linux command `ps`. After the malicious app determines that one of the target apps is in the foreground, it attempts to establish a Bluetooth connection with its respective device.

A catch here is that, when the official app is in communication with the target device, the malicious app cannot connect to it. To get the data, the malicious app needs to connect to the device right before this legitimate connection is established, right after it completes, or during some disruption of the connection. Below I summarize these options:

- *Pre-connection.* The official apps of these devices, once executed, often need the user's intervention to start the communication with their devices. For example, all the apps for the MyGlucoHealth, iThermometer and the Bodymedia armband have a soft button that needs to be pushed to initiate the connection. These apps can also be configured to attempt automatic connections to their respective devices as soon as they are launched. Therefore, in order to capture data from the target device, the malicious app should be in position to exploit the time gap between the moment it discovers that the target app is running and the moment when the legitimate connection is established (after the soft button is pushed or the automatic connection goes through). The likelihood of this succeeding is contingent on how frequently the malicious app checks currently-running processes, i.e., its *sampling rate* for monitoring the official app.

- *Post-connection.* After discovering the running official app, the malicious app can simply wait until its connection ends and then immediately connect to the device. This strategy avoids aggressive monitoring of the official app: the malicious app can keep a slow sampling rate, as long as it can still detect the target during its execution.

There is a risk, however, that the user turns off the target device *before* exiting its app. When this happens, the adversary loses the chance to get data at that specific point.

- *Disruption.* The malicious app can disrupt the legitimate app's communication by deactivating Bluetooth on the phone. It can then reactivate the channel and immediately make a connection to the target device. During this attack, the user might observe the disruption and have to manually click the button on the app again to resume data collection. The approach makes the attack less stealthy but more reliable in getting the data from the target device.

Here I elaborate how an adversary can utilize these techniques to launch data-stealing attacks on the healthcare devices.

Attack Implementation

I demonstrated the data-stealing attacks on all four healthcare devices. To prepare for the attacks, the adversary analyzes the code of these devices' official apps and their Bluetooth traffic captured using `hcidump` [185] to facilitate her understanding of their protocols (for talking to the devices), and further built these protocols into the malicious app. During its operation, the malicious app calls the `getBondedDevices()` API to get a list of external devices already paired with the phone and their bonding information, including the name, the MAC address and the UUID of the device of interest. Using such information, the malicious app makes RFCOMM connections to the device to download sensitive user data.

The attack strategy I implement includes a surveillance component that periodically calls the API `getRunningTasks()` to monitor the execution of the device's official app twice per second. With this implementation, the adversarial app can keep a low profile incurring, on average, around only 3mW of extra power consumption. In the meantime, given that human interventions (clicking on a button after the app is activated) can take seconds, the app stands a good chance of capturing the time window before the official app establishes a connection to its device. In case, automatic connection is configured on the target apps, there is a race condition on the socket establishment. To make sure that we do not miss the opportunity to capture data when a target app is launched, the adversarial design incorporates both the pre-connection and the post-connection strategies: as soon as the malicious app finds that the target app is running, it first makes a connection attempt; if not successful, the app listens for the asynchronous `ACTION_ACL_DISCONNECTED` event broadcasted by the OS, which notifies the app once a low level –(Asynchronous Connection-Less (ACL)– connection with a remote device ends, and then tries to connect to the target device again if the device is the one disconnected and the disconnection is not caused by the malicious app itself. If either the pre-connection or post-connection attempt succeeds, the malicious app requests and captures the data from the appropriate external Bluetooth device, sends them to the adversary and closes the connection, to make it available to the legitimate app.

It is particularly tricky when the official app is configured to automatic connections: once the pre-connection attack succeeds, the malicious app rapidly finishes its operations and releases the socket that is almost instantly captured by the legitimate app. This causes the OS to miss reporting the DISCONNECT event and the consecutive CONNECT. Hence, when the legitimate app releases the socket, the malicious app believes that the disconnection is initiated by itself. As a result, it skips the post-connection opportunity and thus misses the new data the device collects during the period of the legitimate app's connection. To address this issue, the malicious app checks whether enough time elapses from the moment it sends out a disconnection request to when it receives a disconnection event from the OS; if so, the app believes that the event it gets is about another app and then goes ahead to make another connection attempt.

Effectiveness Evaluation

I run the malicious app on a Nexus 4 development phone running JellyBean (4.2), together with all target devices' apps. I evaluated the effectiveness of the data-stealing attack by observing the success rate when the apps were configured to initiate automatic connections to their respective devices once launched. This is the worst case scenario as this operation is much faster than its alternative where the user must click a button to initiate such connections, hence the window of opportunity is smaller for the pre-connection attempt. I found that the malicious app is often successful in capturing this window. The experimental results are presented in Table 7.1.

For the Bodymedia armband device, I found that in 100 pre-connection trials, the malicious app managed to connect 99 times to the device, get the sensitive data and send them to a remote server. The case that the connection failed was attributed to a device de-synchronization issue that rendered even the official app unable to connect to it. The adversary achieves this high success rate because the Bodymedia Link Armband mobile app does some pre-processing operations before attempting to connect to its device, which gives enough time for the malicious app to perform its operations and release the socket. The success rates were also high for other apps, except for iThermometer (e.g., 42 out of 100 trials) due to its app's prompt response in establishing Bluetooth socket connections. When the malicious app won the race, the authorized app failed to connect but it automatically retried after 10 seconds, and succeeded as this interval was often enough for the malicious app to finish its task and release the socket. The post-connection attacks succeeded most of the time except for the glucose meter, MyGlucoHealth, as long as the devices were switched off after the official apps stopped. MyGluooHealth automatically turns itself off after sending data to its app to save its battery power, so none of the post-connection attacks on it succeeded. I also tested the disruption strategy, which also worked, allowing the adversarial app to discontinue the official app's connection and get the health data. A problem with this attack strategy, is that the legitimate connection needs to be interrupted, which could be noticed by the user.

Table 7.1: Success rate of data-stealing attack. This table depicts the successful connections made by the malicious app on 100 trials.

Target Device	**Pre-connection**	**Post-connection**
Bodymedia LINK Armband	99/100	100/100
iThermometer	42/100	100/100
Nonin Pulseoximeter	99/100	92/100
myGlucoHealth	100/100	0/100*

*the device turns off few seconds after sending data to the phone.

Power Consumption Evaluation

A rough estimation of the power consumption of different surveillance strategies is important for understanding the stealthiness of the malicious app, because this activity dominates all of its operations in terms of the time interval that it has to run. We tested the different options we had. I evaluated `getRunningTasks` (the strategy implemented in the malicious app) and its alternatives including calling `getRunningAppProcesses()` and making repeated attempts to connect to or check the existence of the target Bluetooth device. I ran the app using each strategy independently for 10 minutes. The average power consumption of the strategy under scrutiny is illustrated in Table 7.2. As depicted, the adopted strategy (`getRunningTasks`) turned out to be both much more efficient and stealthier (as the Bluetooth sign appears on the screen only when it is supposed to be, i.e., when the the official app is running). I further compared the power consumption of this strategy with that of two popular apps, as described in Table 7.3. As we can see here, the surveillance strategy has a comparable power-consumption level (3mW) as those apps (1 to 18mW). Accurate power consumption measurement is not required to do this evaluation, we only need rough *relative* power measurement. The software used for the power consumption evaluation pro-

Table 7.2: Average power consumption over 10 minutes per surveillance technique using PowerTutor[184].

Technique	**Avg Power Consumption**	**Sampling Rate**
getRunningAppProcesses()	8mW	2 samples/s
getRunningTasks()	3mW	2 samples/s
connect()	17mW	0.18 samples/s
startDiscovery()	15mW	0.054 samples/s

Table 7.3: Average power consumption over an hour. Comparison between our surveillance technique and 2 popular applications using PowerTutor[184].

Technique	**Avg Power Consumption**
Facebook	18mW
getRunningTasks()	3mW
Gmail	1mW

vides accurate measurement for a very limited number of phones and rough measurements for all Android phones. This rough measurement suffices for this evaluation.

7.2.3 Measuring the DMB threat

As discussed before, the DMB problem in the shared bluetooth communication channel stems from the lack of a bonding between an Android external device and its official app which allows another app with access to the same channel to steal data from the external resource, or a spurious external resource to inject data into a victim app. In the absence of OS-level protection, this threat can only be addressed by the app-device authentication developed by individual device manufacturers. The design and implementation of such an authentication mechanism, however, can be non-trivial, which could raise the cost of the devices. To find out whether such a security measure has already been taken in practice, we performed a measurement study that analyzed a relevant set of apps from Google Play. The study reveals that *all* of the selected apps are actually vulnerable, indicating that the DMB problem is indeed realistic and serious. Given the pervasiveness of vulnerable devices and challenges in fixing them (which could require modifying their hardware), an OS-level solution becomes inevitable (see 7.3).

To perform our study we collected relevant official apps for different Bluetooth devices. Our methodology for collecting those apps is as follows: we first searched Google Play for those apps compatible with Google NEXUS 4, using the following terms: “Bluetooth Door Lock”, “Bluetooth Health”, “Bluetooth Medical Devices” and “Bluetooth Meter”. All

Table 7.4: Sampled apps

Total apps	90
Apps not using Bluetooth (eliminated)	2
Device apps with sensitive information	68
Device apps with insensitive information	20

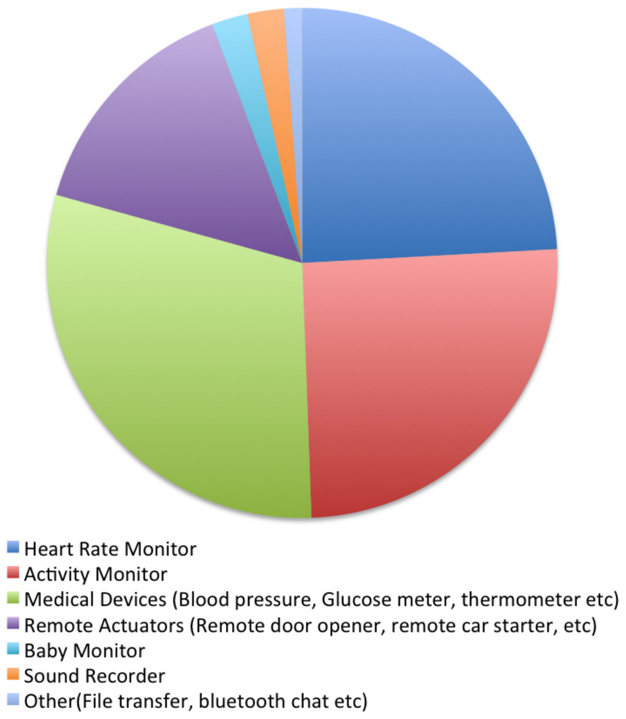


Figure 7.2: Classifications of the sampled apps. Some of them collect information in multiple categories.

together, these queries gave us 90 apps. For each of these apps, we manually inspected its descriptions to determine whether it received sensitive user data from its device. Among these 90 apps, 68 involved some private user information, such as the heart rate, blood pressure, body temperature, glucose level, daily activities, and so on as summarized in Figure 7.2.

Application Analysis

To avoid purchasing all 68 devices that can be used with our sampled apps, we analyzed the apps' code to find out whether they included any app-device authentication. This analysis was done both automatically and manually, as follows.

We first decompiled all the 68 apps and searched for authentication-related programming structures. Authentication should be based upon a secret, which was not hard-coded into any of those apps, given the fact that from two independent downloads of the same app, we always got the same code and data. Therefore, such a secret should either come from some external inputs of the app, particularly its user interfaces, web communication or internal memory files, or is generated by cryptographic operations. In our study, we inspected all such potential sources of authentication secrets (Table 7.5) to determine whether their outputs affected the inputs of the app's Bluetooth communication, particularly that of `BluetoothSocket.write`, which transmits data to the device through a Bluetooth socket connection.

We ran a script that used `grep` to locate the APIs related to those sources and identified the apps where such APIs only appeared within public libraries. For example, we found that, for most apps, their cryptographic APIs (provided by Java JCE, Bouncy castle and spongycastle [186, 187, 188]) were all included in shared libraries such as Google ads, Twitter authentication, OAuth, etc. Those libraries are used for specific purposes, getting ads or performing web-based authentication, for example. It is unlikely that they be used for authenticating the app to its Bluetooth device. Therefore, we removed all the apps that did not have any of those APIs outside the public libraries. There were 48 such apps among all we collected.

For the remaining 20 apps, we manually inspected all the occurrences of these "suspicious" APIs (Table 7.5) in their code. We looked at the functions where the calls to the APIs were made. It turned out that they were all used for the purposes having nothing to do with app-device authentication. For example, most reads from memory files appeared in the crash-handling mechanisms and most cryptographic operations were performed on the SQL queries on web databases. We also found that `HttpClient` was used in the functions for sharing tweets or getting the user's workout data from the web. None of these API outputs were propagated to the inputs of the app's Bluetooth communication.

We further installed all the 68 apps and manually inspected their user interfaces. None of them asked for passwords, PINs, etc. for authenticating themselves to their corresponding devices.

Study Results

As discussed above, we found no evidence that any of these 68 apps, which were relevant apps in Google Play, performed any app-device authentication. Table 7.5 summarizes our findings. The 48 apps we removed automatically either did not have any suspicious APIs or

Table 7.5: Manual analysis on 20 apps. The other 48 apps were automatically filtered out by the locations of their suspicious APIs.

Authentication Methods	**Libraries/ Functions used**	**Total**	**Apps with app-device authentication**
Crypto	e.g., `javax.crypto`, `bouncycastle`	9	0
Internal storage	e.g., `openFileInput()`	15	0
Web communication	e.g., `HttpClient`	5	0
UI for app-device authentication	`Manual`	0	0

had such APIs in their shared libraries, including those for advertising, web authentication, crash analysis, etc. For the 20 apps we manually analyzed, 9 called cryptographic APIs in their own code, 5 invoked web APIs and 15 read from memory files. Also, for all the 68 apps we studied, none had user inputs for app-device authentication. Again, none of these apps generated any data flow that affected the inputs of Bluetooth communication functions.

This study also shows that most of these apps supported secure Bluetooth communication: 42 apps utilized secure socket only; 12 worked under both secure and insecure communications and the rest utilized insecure communication only. This indicates that most of the devices processing sensitive user data do take privacy protection seriously. However, the presence of malicious apps with the Bluetooth permissions on Android (which since version 6 are granted automatically) renders such device-device authentication insufficient for protecting access to shared communication channels resulting in leakage of private user information.

This study on Bluetooth suggests that indeed the Android Security Model is too coarse-grained to both support the utility of the apps and protect the confidentiality of the information communicated through that shared channel. These findings inspired the study of more such channels of communication with external resources which I report on the next Section (7.2.4).

7.2.4 Other External-Resources Attacks

To understand the significance and the applicability of the problem incurred to Android due to to unprotected external resources, a study was carried on other external resources namely NFC, SMS, Audio. For this study we analyzed a set of prominent accessories and online services that utilize popular channels, including SMS, Audio and NFC. Our findings echo our previous findings on on Bluetooth 7.2.1 and related studies on the Internet (local socket connections) [189] channels. The latter study found that all no-root third-party screenshot services can be exploited by a malicious app connecting to them through the Internet channel. This Section demonstrates that the SMS, Audio and NFC channels are equally under-protected, exposing private user information like bank account balances, password reset links etc. These findings point to the security challenges posed by the widening gap between the coarse-grained Android protection and the current way of sharing external resources on smartphones.

Methodology

To further study channels of communication with external resources, we collected apps from Google Play, choosing those that may access private user data or perform sensitive operations through Audio or NFC. Specifically, we searched the Play store for popular apps using these channels and then went down the list to pick out 13 Audio and 17 NFC apps that could perform some security-related operations. For SMS, we looked into 14 popular online services, including those provided by leading financial institutes (Bank of America, Chase, Wells Fargo, PayPal) and social networks (Facebook, Twitter, WhatsApp, WeChat, Naver Line, etc.), and a web mail (Gmail). Those services communicate with `com.android.sms` and sometimes, their own apps using short text messages.

Table 7.6 provides examples for the apps and services used in our study. All the services we analyzed clearly involve private user data, so do 6 fitness, credit-card related Audio apps. Some payment related apps using the Audio jack, are heavily obfuscated and we were not able to decompile them using popular de-compilation tools (dex2jar, apktool). Most of the other apps in the Audio category are remote controllers or sensors that work with a dongle attached to the phone's Audio jack. Although those devices do not appear to be particularly sensitive (e.g., the camera that can be commanded remotely to take pictures), such functionalities (e.g., remote control) could have security implications when they are applied to control more sensitive devices. Our study also reveals that Most NFC apps are for reading and writing NFC tags (tags with microchips for short-range radio communication), which can be used

Table 7.6: Critical Examples

Channel	**App**	**Usage**	**# of downloads**	**Details**
AUDIO	EMS+	Credit card reader	5,000 - 10,000	Decrypt : Creates a private key of RSA with hardcoded modulus and private exponent. Uses it to load session key which is used in AES to process messages from credit card dongle.
AUDIO	UP	Tracks sleep, physical activity and nutritional info	100,000 - 500,000	Doesn't include any authentication features. A repackaged app with different credential is able to read existing data from the band.
SMS	All bank services	Alert messages and Text banking	NA	Both SMS can be read by any app with SMS permission.Alert messages: sensitive financial activity and amount info. Text banking: receive, send money and check balance.
SMS	Chat and SNS	Authentication	100,000,000 - 1,000,000,000	2 step authentication; verification code sent via SMS.
NFC	SquareLess	Credit card reader	10,000 - 50,000	Reads credit card information. Malicious apps may also read credit card data as this app does.
NFC	Electronic Pickpocket RFID	Credit card reader	10,000 - 50,000	Reads credit card information. Malicious apps may also read credit card data as this app does.

to keep sensitive user data (e.g., a password for connecting to one's Wi-Fi access point) or trigger operations (e.g., Wi-Fi connection). A more sensitive application of NFC is payment through a digital wallet. However, related NFC equipment is hard to come by.

Over those apps and services, we conducted both dynamic and static analyses to determine whether there is any protection in place when they use those channels. For SMS, we simply built an app with the `SMS` permission to find out what it can get. All NFC apps were studied using NFC tags, in the presence of an unauthorized app with the `NFC` permission. For those in the Audio category, we analyzed a Jawbone UP wristband, a popular fitness device whose app (`com.jawbone.up`) has 100,000 to 500,000 downloads on Google Play, to understand its security weakness. In the absence of other Audio dongles, relevant apps were decompiled for a static code inspection to find out whether there is any authentication and encryption protection during those apps' communication with their external devices. Specifically, we looked for standard or homegrown cryptographic libraries (e.g., javax.crypto, BouncyCastle, SpongyCastle) within the code, which are needed for establishing a secret with the dongles. Also, the apps are expected to process the data collected from their dongles locally, instead of just relaying it to online servers, as a few payment apps do. This forces them to decrypt the data if it has been encrypted. Finally, we ran those apps to check whether a password or other secrets need to enter for connecting to their dongles. Our analysis was performed on a Nexus 4 with Android 4.4.

Study Results

This analysis shows that most external resources studied have not been protected by apps and service providers. The consequences here can be very serious, as elaborated below.

Firstly we examine the SMS-based services. As expected, all short messages leading online services delivered to our Nexus 4 phone were fully exposed to the unauthorized app with the `SMS` permission. Note that such messages should only be received by `com.android.sms` to display their content to the owner of the phone, as well as those services' official apps: for example, Facebook, Naver Line, WeChat and WhatsApp, directly extract a verification code from their servers' messages to complete a two-step authentication on the owner's behalf.

Information leaks through this under-regulated channel are serious and in some cases, catastrophic. A malicious app can easily get such sensitive information as account balances, incoming/outgoing wire transfers, debit card transactions, ATM withdrawals, a transaction's history, etc. from Chase, Bank of America and Wells Fargo, authorized amount for a transaction, available credit, etc. from Chase Credit Card and Wells Fargo Visa, and notifications for receiving money and others from PayPal. It can also receive authentication secrets from Facebook, Gmail, WhatsApp, WeChat, Naver Line and KakaoTalk, and even locations of family members from Life360, the most prominent family safety online service. An adversary who controls the app can also readily get into the device owner's Facebook and Twitter accounts: all she needs to do is to generate an account reset request, which will cause those services to send the owner a message with a reset link and confirmation code. With such information, even the app itself can automatically reset the owner's passwords, by simply sending requests through the link using the mobile browser. A video demo of those attacks is posted online [190]. Note that almost all banks provide mobile banking, which allows enrolled customers to check their account and transaction status through SMS messages.

Secondly we inspect the risks associated with the Audio channel. To do that, we analyzed the Jawbone UP wristband [28], one of the most popular fitness devices that utilize the low-cost Audio channel. The device tracks its user's daily activities, when she moves, sleeps and eats, and provides summary information to help the user manage her lifestyle. Such information can be private. However, we found that it is completely unprotected. We ran an unauthorized app that dumped such data from the device when it was connected to the phone's Audio jack.

For all other apps in the Audio category, we did not have their hardware pieces and therefore could only analyze their code statically. Specifically, among all 5 credit-card reading apps, PayPal, Square and Intuit are all heavily obfuscated, which prevented us from decompiling them. Those devices are known to have cryptographic protection and designed to send encrypted credit-card information from their card readers directly to the corresponding web services [191, 192]. The other two apps, EMS+ and Payment Jack, were decompiled in our research. Our analysis shows that both of them also receive ciphertext from their card-reader dongles. However, they decrypt the data on the phone using a hard-coded secret key. Since all the instances of these apps share the same key, an adversary can easily extract it and use it to decrypt a user's credit-card information downloaded from the app's payment dongle. Furthermore, all other apps, which either support sensors (e.g, wind meter) or remote controllers (e.g., remote picture taking), are unprotected, without authentication and encryption at all.

Lastly lets take a look at NFC. 5 out of 17 popular NFC apps (e.g., NFC Tools) we found are used to read and write NFC tags. They allow users to store any data on tags, including sensitive information (e.g., a password for one-touch connection to a Wi-Fi access point). However, there is no authentication and encryption protection at all[4]. We ran an unauthorized app with the `NFC` permission to collect the data on the tag whenever our Nexus phone touched the tag. Note that in the presence of the authorized app, Android will ask the user to choose the right one each time the tag is detected[5]. Although this mechanism does offer some protection, it completely relies on the user's judgment during every tap on an NFC device and cannot be used by system administrators to enforce their mandatory policies.

Among the rest of apps, NFC ReTag FREE utilizes the serial number of an NFC tag to trigger operations. Again, since the communication through the NFC channel is unprotected, a malicious app can also acquire the serial number, which leaks out the operation that the legitimate app is about to perform. The only NFC app with protection is the NFC Passport Reader. What it does is to use one's birth date, passport number and expiration date to generate a secret key for encrypting other passport information. The problem is, once those parameters are exposed, the adversary can recover the key to decrypt the data collected from the NFC channel.

[4]There are more expensive tags such as MIFARE that support encryption and authentication. The app using those tags needs the user to manually enter a secret. Clearly, they are not used for protecting the information like Wi-Fi passwords, which should be passed to one's device conveniently.

[5]More specifically, this happens when both the authorized app and the malicious app register with the same priority to receive the notification for device discovery.

7.3 MITIGATION DESIGN

In the previous section, I have discussed how the Android Security Model is incapable of protecting access to shared communication channels for interfacing with external devices and sources of information. In par with my previous studies, the natural next step after understanding an adversary's capabilities is to try finding ways to protect against possible attacks. In this Section I will address this challenge, and discuss the design, implementation and evaluation of a solution to this problem.

The analysis on Bluetooth, SMS, Audio and NFC 7.2, and prior findings on Internet [189] emphasize the urgent need to enhance Android access control, to better protect shared communication channels for direct interaction with external resources. In this section, I present the first uniform, backward compatible, and easily maintainable design of a system for this purpose. The system, called *SEACAT* (Security-Enhanced Android Channel Control), extends SEAndroid's MAC 2.2.3 to cover SMS, NFC, Bluetooth and Internet, and also adds in a DAC module to allow the user and app developers to specify rules for all these channels, in addition with Audio. I implemented SEACAT on Android 4.4 with an SEAndroid enhanced kernel 3.4.0.

7.3.1 Design Overview

The objective is to develop a simple security mechanism that supports flexible, fine-grained mandatory and discretionary protection of various external resources through controlling their channels of communication with smartphones. However, achieving this goal is by no means a smooth sail. Here are a few technical challenges that need to be overcome in the design and implementation of such a system.

- *Limitations of SEAndroid.* Today's SEAndroid does not model external resources. Even after it is extended to describe them, new enforcement hooks need to be added to system functions scattered across the framework/library layer and the Linux kernel. For example, the Bluetooth channel on Android 4.4 (Bluedroid stack) is better protected on the framework layer, which has more semantic information, while the control on the Internet should still happen within the kernel. Supporting these hooks requires a well though-out design that organizes them cross-layer under a unified policy engine and management mechanism for both MAC and DAC.

- *Complexity in integration.* Current Android already has the permission-based DAC and SEAndroid-based MAC 2.2. An additional layer of DAC protection for external resources could complicate the system and affect its performance[6]. How to integrate SEACAT into the current Android in the most efficient way is challenging.

To address these challenges, I will introduce a design that integrates policy compliance checks from both the framework and the kernel layer, and enforces MAC and DAC policies within the same security hooks (Figure 7.3). More specifically, the architecture of SEACAT includes a policy module, a policy enforcement mechanism and a DAC policy management service. At the center of the design is the policy module, which stores security policies and provides an efficient compliance-check service to both the framework and the kernel layers. It maintains two policy bases, one for MAC and the other for DAC. The MAC base is static, which has been compiled into the Linux kernel in the current SEAndroid implementation on AOSP. The DAC base can be dynamically updated during the system's runtime. Both of them are operated by a policy engine that performs compliance checks. The engine is further supported by two Access Vector Caches (AVCs), one for the kernel and the other for the framework layer. Each AVC caches the policies recently enforced using a hash map. Due to the locality of policy queries, this approach can improve the performance of compliance checks. Since DAC policies are in the same format as MAC rules, they are all served by the same AVC and policy engine.

The enforcement mechanism comprises a set of security hooks and two pairs of mapping tables. These hooks are placed within the system functions responsible for the operations on different channels over the framework layer and the kernel layer. Whenever a call is made to such a function, its hook first looks for the security contexts of the caller (i.e., app) and the object (e.g., a Bluetooth address, the Sender ID for a text message) by searching a MAC mapping table first and then a DAC table. The contexts retrieved thereby, together with the operation being performed, are used to query the AVC and the policy engine. Based upon the outcome, the hook decides whether to let the call go through. Just like the AVC, each mapping table has two copies, one for the framework layer and the other for the kernel. Also, the MAC table is made read-only while the DAC table can be updated during runtime.

Both the DAC policy base and DAC mapping table are maintained by the policy management service, which provides the user an interface to identify important external resources (from their addresses, IDs, etc.) and the apps allowed to access them. Also it can check manifest files of newly installed apps to extract rules embedded there by the developer (e.g., only the official Chase app can get the text message from Chase) to ask for the user's ap-

[6]Note that this new DAC cannot be easily integrated into the permission mechanism, since the objects there (different Bluetooth devices, web services, etc.) can be added into the system during runtime.

proval. Those policies and the security contexts of the labeled resources are uploaded to the DAC base and the mapping tables respectively.

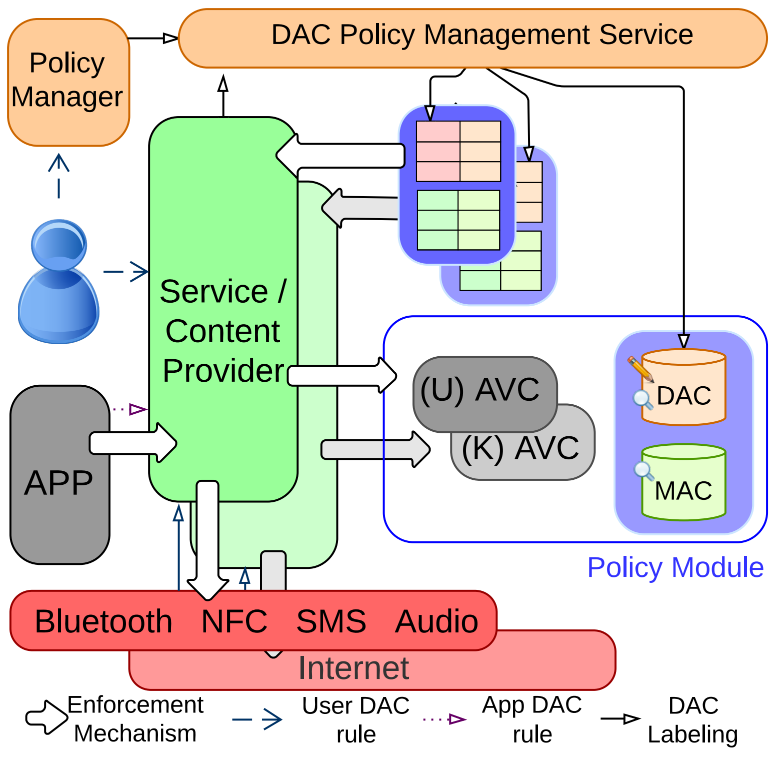


Figure 7.3: *SEACAT* architecture

7.3.2 Trusted Compute Base and Adversary Model

Like SEAndroid, the security guarantee of SEACAT depends on the integrity of the kernel. We have to assume that the adversary has not compromised the kernel to make the approach work. In the meantime, SEACAT can tolerate corrupted system apps, as long as they are confined by SEAndroid. Furthermore, the DAC mechanism is configured by the user and therefore could become vulnerable. However, the proposed design makes sure that even when it is misconfigured, the adversary still cannot bypass the MAC protection in place. Finally, I assume the presence of malicious apps on the user's device, with proper permissions to access all aforementioned channels.

7.3.3 Policy Specification and Management

To control access to external resources through shared communication channels, we first need to specify the right policies and identify the subjects (i.e., apps) and objects (e.g., a

Bluetooth glucose meter, the Chase bank which sends a sensitive SMS, etc.) to apply them. This is done within the policy module and the policy management service.

SEACAT has to provide a convenient way of specifying policies. Remember from the Background Chapter (Section 2.2.3), an SEAndroid rule determines which domain is allowed to access which resources, and how this access should happen. To specify such a rule for external resources, both relevant domains (for apps) and types (for external resources) need to be defined. The `domain` part has already been taken care of by SEAndroid: we can directly declare ones for any new apps whose access rights, with regard to external resources, need to be clarified. When it comes to `type`s, those within the AOSP Android have been marked as `file_type`, `node_type` (for sockets and further used to specify IP range), `dev_type`, etc. In my research, I further specified new categories of types (or `attributes`), including `BT_type` for MAC addresses of Bluetooth devices, `NFC_type` for NFC serial numbers and `SMS_type` for SMS Sender ID (originating addresses). Here is an example policy based upon these domains and types:

```
allow trusted_app bt_dev:btacc rw_perms
```

where `bt_dev` is a type for Bluetooth devices (identified by their MAC addresses) and `btacc` includes all the operations that can be performed on the type. This policy allows the apps in the domain `trusted_app` to read from and write to the MAC addresses in the type `bt_dev`. Later I describe how to associate such a domain with authorized apps, and the type with external resources.

The DAC policies used in SEACAT are specified in the same way, using the same format, which enables them to be processed by the policy engine and AVC also serving MAC policies. The DAC policy base, includes a set of types defined for the Audio channel. Audio has not been included in the MAC policies since the device attached to it cannot be uniquely identified: all we know is just whether the device is an input (headset) or output (speaker) device or the one with both capabilities. For user-defined DAC policies, SEACAT provides a mechanism to lock the whole audio channel when necessary, a process elaborated later. Moreover, although the DAC base is supposed to be updated at runtime, to avoid the overheads that come with such updates, SEACAT uses a predefined a set of "template" policies that connect a set of domains to a set of types in different categories (Bluetooth, NFC, SMS, Internet and Audio) with read and write permissions. The domains and types of those policies are dynamically attached to the apps and resources specified by the user during runtime. In this way, SEACAT only needs to maintain a mapping table from resources to their security contexts (`user_seres_contexts`) before the template rules run out.

Next, SEACAT must provide a mechanism for assigning domains to apps. For the domains defined for MAC, how they are assigned to apps can also be specified in the policies. SEACAT allows the administrator to grant trusted apps, permissions to use restrictive external resources. Such apps are identified from the parties who sign them. Specifically, when an app is being installed, SEAndroid assigns it an `seinfo` tag according to its signature. The mapping between this tag and the app's domain is maintained in the file `seapp_contexts`, which *Zygote* (see Section 2.1), the Android core process that spawns other processes, reads when determining the app's security context during its runtime.

Labeling apps for DAC is handled by SEACAT's policy management service, which includes a set of hooks within the `PackageManager` and `installd`. Before an app is installed, these hooks present to the user a "dialogue box", alongside the app's permission information. This allows the user to indicate whether the app should be given a domain associated with certain channels (Bluetooth, NFC, SMS, Internet and Audio), so that it can later be given the privilege to access protected external resources. For an app assigned a domain, the `PackageManager` labels it with an `seinfo` tag different from the default one (for untrusted, unprivileged apps) and stores the tag alongside its related domain within a dynamic mapping file `user_seapp_contexts`. Note that this action will only be taken, in the absence of a MAC rule already dictating the domain assignment for this app.

SEACAT furthers requires modification of `libselinux`, which is used by Zygote, to assign the appropriate security context to the process forked for an app. An instrumentation within `libselinux` enables loading `user_seapp_contexts` for retrieving the security context associated with a user-defined policy. Note that again, when an `seinfo` tag is found within both `seapp_contexts` and
`user_seapp_contexts`, its context is always determined by the former, as the MAC policies always take precedence. In fact the system will never create a DAC policy for an external resource that conflicts with a MAC policy. Nevertheless, if a compromised system app manages to inject erroneous DAC policies, they will never affect or overwrite MAC policies.

The design of SEACAT also allows the app developer to declare within an app's manifest the external resource the app needs exclusive access to. With the user's consent, the app will get a domain and the resource will be assigned a type to protect their interactions through a DAC rule. This approach makes declaration of DAC policies convenient: for example, the official app of Chase can state that only itself and Android system apps are allowed to receive the text messages from Chase; a screenshot app using an ADB service can make the IP address of the local socket together with the port number of the service off limit to other third-party apps.

Labelling apps is of course not enough. We need a way to label external resources as well, or in SEAndroid terms, we need to assign types to those objects in par with the labelling of local resources by SEAndroid. For standard local resources, such as files, SEAndroid includes policies that guide the OS to find them and label them properly. For example, the administrator can associate a directory path name with a type, so that every file stored under the directory is assigned that type. The security context of each file (which includes its type) is always kept within its extension, making it convenient to retrieve the context during policy enforcement. When it comes to external resources, however, we need to find a new way to label their identifiers and store their tags. This is done in our research using a new MAC policy file `seres_contexts`, which links each resource (the MAC address for Bluetooth, the serial number for NFC, the Sender ID for SMS and the IP/port pair of a service) to its security context. The content of the file is pre-specified by the system administrator and is maintained as read-only throughout the system's runtime. It is loaded into memory buffers within the framework layer and the Linux kernel respectively, and utilized by the security hooks there for policy compliance checks (Section 5.3.2).

Labeling external resources for the DAC policies is much more complicated, as new resources come and go, and the user should be able to dynamically enable protection on them during the system's runtime. SEACAT provides three mechanisms for this purpose: 1) connection-time labeling, 2) app declaration and 3) manual setting. Specifically, connection-time labeling happens the first time an external resource is discovered by the OS, for example, when a new Bluetooth device is paired with the phone. Also, as discussed before, an app can define the external resource that should not be exposed to the public (e.g., only system apps and the official Facebook app can get messages from the Sender ID "FACEBOOK"). Finally, the user is always able to manually enter new DAC policies or edit existing ones through an interface provided by the system.

For different channels, some labeling mechanisms work better than others. Bluetooth and NFC resources are marked mainly when they are connected to the phone: whenever there are apps assigned domains but not associated with any Bluetooth or NFC resources, SEACAT notifies the user once a new Bluetooth device is paired with the phone or an NFC device is detected; if such a new device has not been protected by the MAC policies, the user is asked to select, through an interface, all apps (those assigned domains) that should be allowed to access it (while other third-party apps' access requests should be denied). After this is done, a DAC rule is in place to mediate the use of the device. Note that once all such apps have been linked to external resources, SEACAT will no longer interrupt the user for device labeling, though she can still use the policy manager to manually add or modify security rules.

In SEACAT's implementation, a few system apps and services are modified to accommodate this mechanism. For Bluetooth, we need to change `Settings`, the Bluetooth system app and the Bluetooth service. When the `Settings` app helps the user connect to a newly discovered Bluetooth device, it checks the device's MAC address against a list of mandatory rules. If the address is not on the list, the Bluetooth service pops an interface to let the user choose from the existing apps assigned domains but not paired with any resources. This is done through extending the `RemoteDevices` class. The MAC address labeled is kept in the file `user_seres_contexts`, together with its security context. This file is uploaded into memory buffers (for both the kernel and the framework layer) for compliance checks. For NFC, whenever a new device is found, Android sends an Intent to the app that registers with the channel. SEACAT's implementation instruments the NFC Intent dispatcher to let the user label the device and specify the apps allowed to use it when the dispatcher is working on such an Intent. This is important when the NFC device is security critical, as now the control is taken away from the potentially untrusted apps and delegated to the user (if no MAC mechanism is in place) during runtime. Furthermore, by providing this mechanism, the system can protect itself, and it is deprived of any dependency on end-to-end authentication between apps and external devices. Lastly, by utilizing the association of apps with resources specified in MAC and DAC policies, the user can read already labeled tags directly, avoiding going through the app selection mechanism every time, which immensely improves the usability of the reading-an-NFC-device task. Again, the result of the DAC labeling is kept in `user_seres_contexts`. The syntax of the MAC policy file `seres_contexts` and the DAC policy file `user_seres_contexts` is demonstrated below:

```
resource_id=xx:xx:xx:xx:xx:xx channel=BLUETOOTH type=bt_dev2
resource_id=XXXXXXXX channel=NFC type=nfc_dev1
resource_id=24273 channel=SMS type=sms_dev3
resource_id=AUDIO channel=AUDIO type=audio_dev
```

External resources associated with SMS and Internet are more convenient to label through app declaration and manual setting. As discussed before, an app can request exclusive access to the text messages from a certain SMS ID. The user can also identify within the interface of our policy manager a set of SMS IDs ("GOOGLE", 32665 for "FACEBOOK", 24273 for Chase, etc.) to make sure that only `com.android.sms` can get their messages[7]. Use of Internet resources should be specified by the app. For example, those using ADB-level

[7]The SMS IDs for services are public. It is easy to provide a list of well-known financial, social-networking services to let the user choose from.

services [189] can state the local IP address and their services' port numbers to let our system label them.

Pertaining Audio, we need to label the whole channel at the right moment. Specifically, the DAC rule for the channel is expected to come with the app requiring it or set manually by the user through the policy manager. Whenever the system observes the Audio jack is connected to a device that fits the profile (input, output or mixed), SEACAT just pops up a "dialogue box" asking the user whether the device needs protection, if a DAC rule has already been required by either an app or the user. We can avoid this window popup when the app (the one expected to have exclusive access to the Audio channel) is found to run in the foreground. In either case, the whole Audio channel is labeled with a type, which can only be utilized by that app, system apps and services. This information is again stored in `user_seres_contexts` for policy enforcement. Notably, as soon as the device is detached from the Audio jack, the type is dropped from the file, which releases the entire channel for other third-party apps. To completely remove the pop-ups, the user can set the system to an "auto" mode in which the Audio is only labeled (automatically) when the authorized app is running. In this case, the user needs to follow a procedure to first start the app and then plug in the device to avoid any information leak.

7.3.4 Policy Compliance Check and Enforcement

To perform a compliance check, a hook needs to obtain the security contexts of the subject (the app), the object (MAC address, NFC serial number, etc.) and the operation to be performed (e.g., read, write, etc.) to construct a query for the policy engine (see Figure 7.4). Here the subject's context can be easily found out: on the framework layer, this is done through the SEAndroid function `getPidContext`, which utilizes the PID of a process to return its context information. Although the same approach also works within the Linux kernel, a shortcut is used in controlling Internet connections through sockets. Specifically, within the socket's structure, SEAndroid already adds a field `sk_security` to keep the security context of the process creating the socket. The field is used by the existing hooks to mediate the access to IP/port types. SEACAT's enforcement of DAC policies is palced there, which involves finding the security contexts of an IP-port pair from a DAC table within the kernel.

The object's context is kept within the MAC policy file `seres_contexts` and the DAC policy file `user_seres_contexts`. To avoid frequently reading from those files during the system's runtime, SEACAT uploads their content to a pair of buffers in the memory both in the framework layer and the kernel. These buffers are organized as hash maps, serving as the mapping tables to help a security hook retrieve objects' security contexts. Specifically, SEACTA uses a new function for searching the mapping tables within `libselinux`, and exposed this interface to the framework so that the security hooks can access it through Java or native code. Within the kernel, SEACAT employs another mapping table for the DAC policy[8]. This table is synchronized automatically with the one for the framework layer to make sure that the same set of DAC policies are enforced on both layers. The set of operations SEACAT introduces for manipulation and retrieval of information from the memory buffers and exposed through libselinux to the rest of the system, are listed within Table 7.7.

Table 7.7: *SEACAT* API

FUNCTION	**DESCRIPTION**
loadPDPolicy	Loads the MAC (seres_res_contexts) and DAC (user_seres_contexts) policy bases containing the resource with security context associations, into the SEACAT memory buffers.
getResourceSecContext	Performs a lookup in the SEACAT memory buffers for a security type assigned to a resource.
getResourceChannel	Performs a lookup in the SEACAT memory buffers for the channel that a resource belongs to.
isResourceMAC	Returns 1 if the resource is present in SEACAT memory buffers and was loaded from the MAC policy base, 0 if it was loaded from the DAC policy base, or NULL otherwise.
insertDACRes	Stores the security context of a resource in the appropriate memory buffer and the corresponding policy base.
getDomain	Returns the security context assigned to a third-party app.

Given the security contexts for a subject (the app) and an object (e.g., an SMS ID), a security hook is ready to query the AVC and policy engine to find out whether an operation (i.e., system call) is allowed to proceed. On the framework layer, this policy compliance check can be done through `selinux_check_access`. SEACAT wrapps this SEAndroid function, adding program logic for retrieving an object's security context from the mapping table.

[8]Note that we do not need to build the table for MAC here, since SELinux already has a table for enforcing MAC policies on IPs. Also, all other channels are enforced on the framework layer.

The new function `seacat_check_access` takes as its input a resource's identifier (Bluetooth MAC, SMS ID, etc.), the caller's security context and the action to be performed, and further identifies the resource's security context before running the AVC and the policy engine on those parameters. Note that for the resource appearing within both MAC and DAC tables, its security context is only determined by the MAC policy. Also, the resource not within either table is considered to be public and can be accessed by any app. Again, this new function is made available to both Java and native code. The same mechanism was also implemented within the kernel, through wrapping the compliance check function `avc_has_perm`. The AVC and the policy engine are largely intact here, as our system was carefully designed to make sure that the DAC rules are in the same format as their MAC counterparts and therefore can be directly processed by SEAndroid.

To be able to enforce these policies SEACAT needs to interject the security hooks in the appropriate functions of the framework or the kernel. This instrumentation allows the system to perform policy compliance checks before a requesting app accesses the information we want to safeguard. Since the external channels we are considering consist of Bluetooth, NFC, Internet, SMS and Audio, SEACAT has to introduce the hooks at the appropriate place for each channel such as to minimize both the risk that an adversary can bypass the protection from a lower level in the software stack and its implementation complexity.

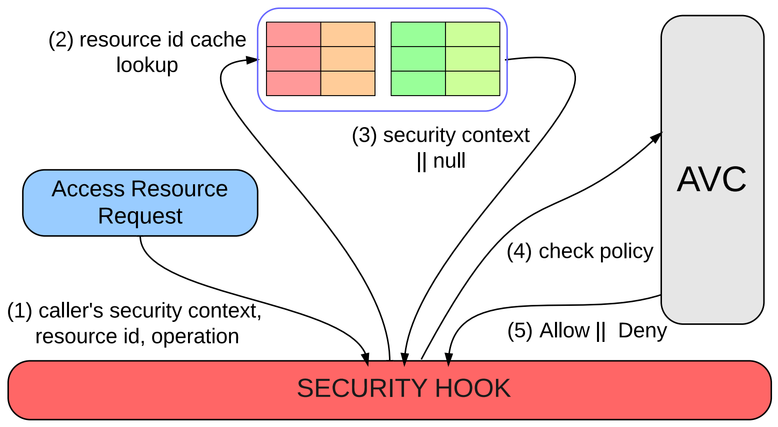

(2) resource id cache lookup
(3) security context || null
AVC
Access Resource Request
(4) check policy
(1) caller's security context, resource id, operation
(5) Allow || Deny
SECURITY HOOK

Figure 7.4: *SEACAT* Policy Compliance Check

To fully control the Bluetooth channel, all its functions need to be instrumented. A prominent example here is `Bluetooth Socket.connect` within the Bluetooth service, which needs to be invoked for establishing a connection with an external device. SEACAT requires a security hook at the beginning of the function to mediate when it can be properly executed. A problem is how to get the process ID (PID) of the caller process for retrieving its security context through `getPidContext`. Certainly we cannot use the PID of the party that directly invokes the function, which is actually the Bluetooth service. For this we can turn to Binder, which proxies the inter-process call (IPC) from the real caller app. Specifically, SEACAT's hook calls `getCallingPid` (provided by Binder) to find out the app's PID and then its security context, and passes the information to the Bluetooth stack. Inside the stack the actual connection attempt is instrumented to use the app's security context, the Bluetooth MAC address to be connected and the "`open`" operation as inputs to query `seacat_check_access`. What is returned by the function causes the connection attempt to either proceed or immediately stop. The Bluetooth service is notified accordingly regarding the success or failure of the connection attempt. In the same manner, we can instrument other functions in the Bluetooth stack.

We also need to effectively control access to the NFC channel. For the broadcomm chip on Google Nexus 4 devices, the NFC stack has been implemented on the framework/library layer through `libnfc-nci`. Thus, all SEACAT security hooks are placed on this layer, within major NFC functions `readNdef`,
`writeNdef` and `connect`, for mediating a caller process's operations on an NFC device with a particular serial number (which is treated as the device's identifier). A tricky part is that when a new NFC device is found to be in proximity, NFC runs a dispatcher to identify which apps have registered for that device through Intent-filters. The dispatcher will deliver an Intent exposing the content of the device to such an app. In cases where multiple apps request access to that NFC device, an "Activity Chooser" box will be presented to the user so she can choose which activity should be launched. Unequivocally, this operation will cause information leaks if the target app is malicious and therefore needs to be controlled. SEACAT's implementation instruments the dispatcher to execute the MAC and DAC policy compliance check against all such registered apps with regards to a specific device serial number. For those that fail the check, the dispatcher simply ignores them and therefore the Intent with the NFC device's contents will never reach them.

The Internet channel differs from both Bluetooth and NFC. Internet has been controlled inside the kernel, with security hooks placed within the functions for different socket operations. As discussed before, SEAndroid has already hooked those functions for enforcing mandatory policies on IP addresses, port numbers and others. In our research, we extended those existing hooks to add enforcement mechanisms for DAC policies. Specifically, we changed `selinux_inet_sys_rcv_skb` and `selinux_sock_rcv_skb_compat` to enable those wrapper functions to search the DAC mapping table within the kernel for the security contexts of IP-port pairs specified by the user and use such information to call `avc_has_perm`. Note that this enforcement happens to the objects (IP and port numbers) that have already passed the MAC compliance check: that is, those IP and port numbers are considered to be public by the administrator, while the user can still add her additional constraints on which party should be allowed to access them.

The SMS channel turns out to be more intricate. Whenever the `Telephony` service on the phone receives a text message from the radio layer, `InboundSmsHandler` put it in an Intent, and then calls `SMSDispatcher` to broadcasts it to all the apps that register with the event (`SMS_RECEIVED_ACTION` or `SMS_DELIVER_ACTION`). Also the `InboundSmsHandler` stores the message to the content provider of SMS. Such a message is limited to the text content with up to 160 characters. To overcome this constraint, the message delivered today mainly goes through *Multimedia Messaging Service* (MMS), which supports larger message length and non-text content such as pictures. What really happens when sending such a message (which can include multimedia content) is that a simple text message is first constructed and transmitted through SMS to the MMS on the phone, which provides a URI for downloading the actual message. Then, MMS broadcasts the message through the Intent to recipients and also saves the message locally through its content provider.

To mediate this complicated channel, we instrumented both SMS and MMS to track the entire work flow and enforce MAC and DAC policies right before a message being handed over to apps (Figure 7.5 in Appendix). Specifically, we hooked the function `processMessagePart` within `SMSDispatcher` to get the ID of the message sender (i.e., the originating address) through `SmsMessageBase.getOriginatingAddress()`. This sender ID serves as an input for searching the mapping tables. The security context identified this way is then attached to the Intent delivered to MMS as an extra attribute `SEC_CON`. On the MMS front, a security hook inspects the attribute and further propagates the security context to another attribute within a new Intent used to transmit the real message once it is downloaded. We also modified the function `deliverToRegisteredReceiverLocked` within `BroadcastQueue` to obtain the security context of each app recipient involved in the broadcast and runs `seacat_check_access` to check whether the app should be allowed to get the message before adding the message to its process message queue.

Besides getting SMS message from Intent receiver for `SMS_RECEIVED_ACTION` or `SMS_DELIVER _ACTION`[9], an app can also directly read from the SMS or MMS content provider given the `SMS_READ` permission. To mediate such accesses, we further instrumented the content provider of `SMSProvider` and `MMSProvider` to perform the policy compliance check whenever an app attempts to read from its database: based on the app's security context and each message's address, our hooks sanitize the cursor returned to the app, removing the message it is not allowed to read.

[9]On Android 4.4, only the default sms app gets this Intent

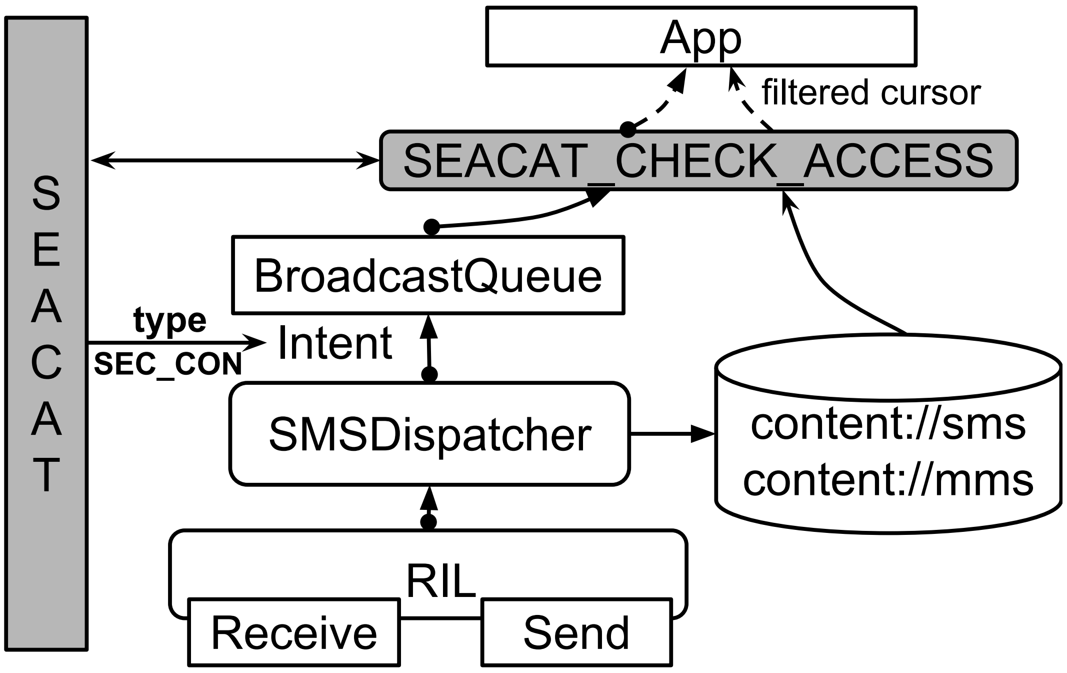


Figure 7.5: *SEACAT*'s enforcement on SMS: *SEACAT* labels each sms message intent and checks if an app can access the message before delivering the intent to the app. Also `SEACAT` filters the sms content provider query results according to the security context of the app

Like SMS, the Audio channel is also mediated on the framework layer. Whenever a device is connected to the Audio jack, `WiredAccessory Manager` detects the device and calls `setDeviceStateLocked`. Within the function, we placed a hook that identifies the type of the device (input/output
/mixed) and checks the presence of a policy that controls the access to such a device. If so, it directly calls the SEACAT function `SensChannel.assignType` to assign the object type in the policy to the Audio channel (which prevents the channel from being used by unauthorized third-party apps) when an authorized app is running in the foreground. Otherwise, it pops up a "dialogue box" to let the user decide whether the device is the object within the policy and therefore needs to be protected. In either case, as soon as the device is unplugged from the Audio jack, the hook immediately removes from the DAC mapping table the entry for the Audio channel, thereby releasing it to other third-party apps.

Policy enforcement happens within the functions for collecting data from the Audio channel. Particularly, SEACAT has a hook inside the `start Recording` method of `android.media .AudioRecord`. Once the method is invoked, it looks for the security contexts for the calling process (through `getContext`) and the Audio channel (using `getResourceSecContext`) to check polices and determine whether the call can go through.

7.4 MITIGATION EVALUATION

As with any security system design we need to evaluate its effectiveness and efficiency. The effectiveness of SEACAT was evaluated against all existing threats to Android shared communication channels and the performance overhead it introduces was recorded. The evaluation was performed on a pair of Nexus 4 phones with Android 4.4 (android-4.4_r12), kernel KRT16S, with the 3.4 kernel (androidmsmmako3.4kitkatmr0): one installed with an unmodified OS to serve as a baseline, and the other with the SEACAT-enhanced kernel. Following I report the evaluation findings. Video demos for this study can be found online [190].

Firstly we want to make sure that SEACAT actually solves the problem and can successfully safeguard all known external resources. Table 7.8 presents 5 known threats to external resources used in our research, which include collection of data from iThermometer through Bluetooth misbonding (see Section 7.2.1), unauthorized use of an ADB proxy based screenshot service through local socket connections [189], as well as attacks on SMS (stealing text messages from Chase and Facebook), Audio (gathering activity data from the UP wristband) and NFC (reading sensitive information from NFC tags) 7.2.4. In our study, we ran those attacks on the unprotected Nexus 4, which turned out to be all successful: the malicious app acquired sensitive information from the external resources through the channels (Bluetooth, SMS, Internet, Audio and NFC), exactly as reported in prior research [189] and Sections 7.2.1 and 7.2.4.

Table 7.8: Threats to Android external resources

No	**KNOWN THREATS**
1	Bluetooth misbonding attack
2	unauthorized adb-based screenshots
3	unauthorized read of an SMS message
4	unauthorized access to audio device
5	unauthorized read of an NFC device's contents

All such attacks, however, stopped working on the SEACAT-enhanced Nexus 4. Specifically, after assigning a type to the MAC address of the iThermometer device through SEACAT's policy management service, only the official app of iThermometer, which was assigned to an authorized domain, was able to get data from the device [190]. The malicious app running in the `untrusted_app` domain could no longer obtain body temperature readings from the thermometer. For SMS, once we labeled the Sender IDs of Chase and Facebook with a type that can only be accessed by the apps within the system domain, the third-party app could not find out when messages from those services came, nor was it able to read them from the SMS content provider `content://sms`. On the other hand, the user could still see the messages from `com.android.sms` [190]. Similarly, the screenshot attack reported in prior research [189] was completely thwarted when the local IP address and port number was labeled. Also the security type given to the serial number of an NFC tag successfully prevented the malicious app from reading its content. In the presence of both authorized and unauthorized apps, the protected Nexus directly ran the authorized app, without even asking the user to make a choice, as the unprotected one did. For Audio, after the user identified the presence of the Jawbone wristband or the official app of the device was triggered, the channel could not be accessed by the malicious app. It was released only after the wristband was unplugged from the Audio jack.

The effectiveness of our protection was evaluated under both MAC and DAC policies for all those attack cases, except the one on the Audio channel, which we only implemented the DAC protection (Section 7.3.3). Also, I tried to assign a resource specified by a MAC policy to a DAC type using our policy manager and found that the attempt could not go through. Even after I manually injected such a policy into SEACAT's DAC database and mapping table (which cannot happen in practice without compromising the policy manager), all the security hooks ignored the conflicting policy and protected the resources in accordance with the MAC rules.

After making sure SEACAT is efffective, we must study its overhead to determine whether it can be practically deployed. To evaluate the performance impact of SEACAT, I measured the execution time for the operations that involve our instrumentations, and compared it with the delay observed from the baseline (i.e., the unprotected Nexus 4). Table 7.9 shows examples of the operations used in this research. In the experiments, I conducted 10 trials for each operation to compute its average duration.

Table 7.9: A list of operations affected by *SEACAT* enhancements

No	OPERATION
1	install app
2	Bluetooth pairing
3	BluetoothSocket.connect
4	dispatchTag
5	dispatchTag (foreground)
6	Ndef.writeNdefMessage
7	Audio device connection
8	AudioRecord.startRecording

Specifically, I recorded the installation time for a new app, which involves assignment of domains. The time interval measured in our experiment is that between the moment the

PackageManager identifies the user's "install" click and when the BackupManagerService gets the Intent for the completion of installing an app with 3.06 MB. For Bluetooth, both the pairing and connection operations were timed. Among them, the pairing operation recorded starts from the moment it was triggered manually and ends when the OnBondStateChanged callback was invoked by the OS. For connection, I just looked at the execution time of BluetoothSocket.connect. Regarding SMS, we can measure the time from when a SMS message is received (processMessagePart) to when the message is delivered to all the interested receivers and the process of querying the SMS content provider. The Internet-related overhead was simply found out from the network connection time.

The amount of time it takes to dispatch an NFC message is related to the status of the target app: when it was in the foreground, we measured the interval between dispatchTag and the completion of the NfcRootActivity; otherwise, our timer was stopped when setForeground Dispatch was called. For the Audio channel, we can record the time for the call AudioRecord. startRecording to go through.

The results of this evaluation are presented in Table 7.10. As we can see from the table, the delays introduced by SEACAT are mostly negligible. Specifically, the overhead in the installation process caused by assigning domains to an app was found to be as low as 49.52 ms. Policy enforcement within different security hooks (with policy checks) happened almost instantly, with a delay sometimes even indistinguishable from the baseline. In particular, in the case of NFC, even when the unauthorized app with the NFC permission was running in the foreground, our implementation almost instantly found out its security context and denied its access request. The only operation that brings in a relatively high overhead is labeling an external device. It involves assigning a type to the resource, saving the label to user_seres_contexts, updating the DAC mapping table accordingly and even changing the DAC policy base to enable authorized apps' access to the resource when necessary. On average, those operations took 189.44 ms. Note that this is just a one-time cost, as long as the user does not change the type given to a resource. An exception is Audio, whose type is assigned whenever the dongle under protection is attached to the Audio jack. Note that the user only experiences this sub-second delay once per use of the accessory, which we believe is completely tolerable.

All the results presented here do not include the delay caused by human interventions: for example, the time the user takes to determine the domain of an app and the type of a resource. Such a delay depends on human reaction and therefore is hard to measure. Also they only bring in a one-time cost, as subjects and objects only need to be labeled once. Actually, for NFC, our implementation could even remove the need for human intervention during policy enforcement: in the presence of two apps with the NFC permissions, the user

Table 7.10: Detailed Performance Measurements in milliseconds (ms)

AOSP (A)			**SEACAT (S)**			A-S
Operation	**mean**	**stdev**	**Operation**	**mean**	**stdev**	**overhead (ms)**
install app	1415.6	40.61	install app (label)	1465.2	76.07	49.52
Bluetooth pairing	1136.5	351.65	Bluetooth pairing (label)	1434.4	237.60	279.9
BluetoothSocket.connect	1699.1	770.22	BluetoothSocket.connect BluetoothSocket.connect (block)	1616 6	306.83 3	-83.1 -1693.1
dispatchTag	87.3	4.32	dispatchTag (MAC:allow) dispatchTag (MAC:block) dispatchTag (label+allow)	96.9 113.1 358.28	4.63 3.57 40.47	9.6 25.8 270.98
dispatchTag (foreground)	272	26.33	dispatchTag (allow foreground) dispatchTag (deny foreground)	269 132.5	41.53 21.76	-3 -139.5
Ndef.writeNdefMessage (app A)	197.1	6.17	Ndef. writeNdefMessage (DAC/MAC allow)	190.89	14.61	-6.21
Ndef.writeNdefMessage (app B)	112.4	12.45	Ndef. writeNdefMessage (unlabeled)	117.5	16.36	5.1
SMS process message	94	7.3	SMS process message(allow) SMS process message (redirect)	106.5 154	8.11 12.11	12.5 60
SMS query()	2.7	1.1	SMS query() filter	6.39	2.4	3.69
Audio device connection	14.9	5.11	Audio device connection (label+ connect)	177.6	21.92	162.7
AudioRecord. startRecording (allow)	85.9	6.84	AudioRecord. startRecording (allow) AudioRecord. startRecording (block)	95.6 7.2	16.75 3.58	9.7 -78.7

could be asked to choose one of them to handle an NFC event whenever it happens, while under SEACAT, this interaction is avoided if one of the apps is within the domain authorized to access the related NFC device and the other is not.

7.5 SUMMARY

In this chapter I presented my study on Android's shared communication channels with external resources. This is inceasingly important in the context of IoT where smartphones connect to smart devices in proximity using Bluetooth and NFS among other protocols. My analysis revealed that Android's shared communication channels are coarsely protected, allowing malicious third-party apps to compromise the confidentiality fo the data transmitted through these shared channels. In response I proposed a uniform system design taking

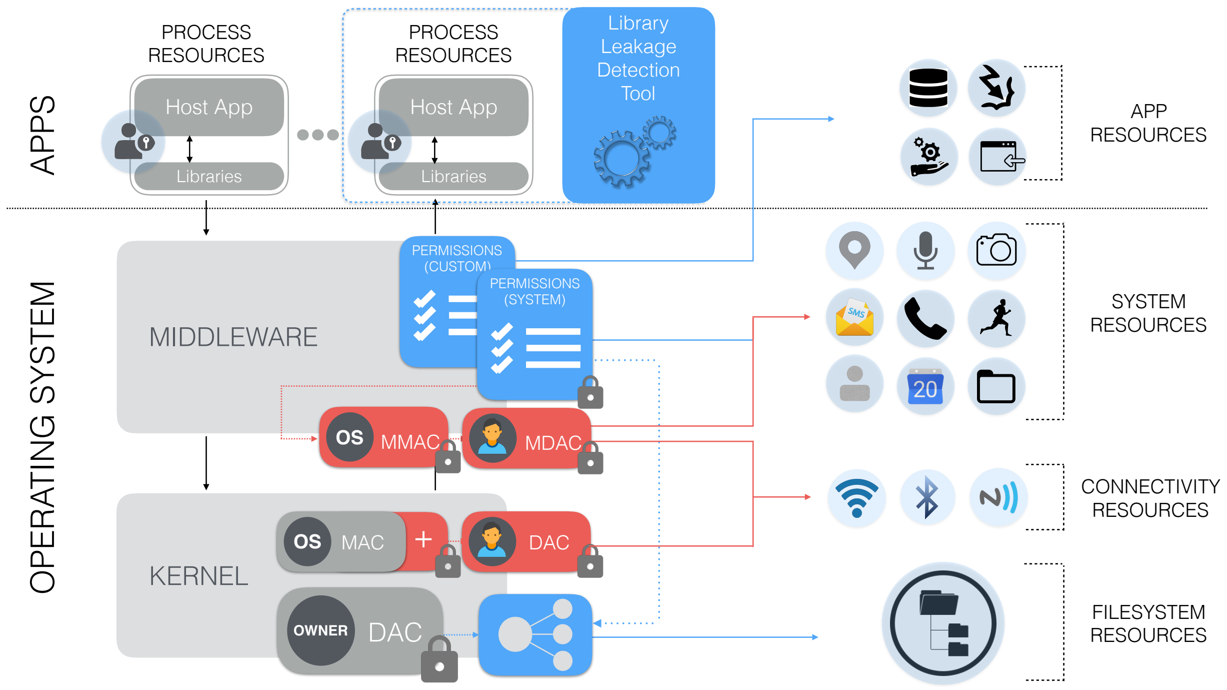


Figure 7.6: Hybrid MAC and DAC for enterprise admins and users to dictate access to personal IoT devices.

advantage of the SELinux infrastructure incorporated to AOSP since version 4.3, to protect all known channels of communication with external Android resources at the operating system level. Figure 7.6 visualizes this addition to the smartphone ecosystem.

So far we analyzed adversaries exploiting shared intra-process privileges, shared filesysem resources, shared system resources and application components, and shared communication channels with external resources. In response I introduced effective and efficient designs of strategies for detecting and preventing these problems. However, sometimes external resources can be hidden behind a mediating point such a router. This is increasingly realistic with the Internet of Things and the advent of smart-home technologies. Such smart-devices are increasingly controlled through smartphones and thus shared between smartphone apps. In the next Chapter I will describe my security analysis on such shared IoT devices.

CHAPTER 8: SHARING DEVICES IN IoT ENVIRONMENTS

In this Chapter I will describe my analysis on the security of smartphone communications with shared IoT devices. I will focus on the security of WiFi smart-home IoT devices against malicious applications running on authorized smarpthones within a home area network. In particular I will show that WiFi smart-home devices tend to rely on WiFi authentication for protection which leaves them vulnerable to attacks from compromised local systems. To prevent such attacks I will illustrate how we can design a practical, fine-grained access control mechanism within a home area network router endpoint which depends neither on IoT device manufacturers nor smartphone vendors [15].

8.1 INTRODUCTION

The pervasiveness of Internet of Things (IoT) devices has brought in a new wave of technological advances in home automation. According to Gartner [193], 6.4 billion IoT devices will be online in 2016, among which a significant portion are *smart-home* systems like smart thermostats [48, 49], fitness trackers, refrigerators, etc., and the number is expected to go above 20 billion by 2020. Examples of such devices include: the Belkin NetCam [50], a camera for streaming surveillance video to a mobile phone; the iBaby monitor [51], a device for remote babysitting; the Family Hub refrigerator [52], which enables online checking of the fridge's contents. Increasingly, these devices are designed to communicate not only with their servers in the cloud but also with other IoT devices and the user's phone over the Home Area Network (HAN), which is typically built around a Wi-Fi router. For example, Nest Protect Fire sensors [194] are capable of propagating an alarm across multiple sensors installed in different rooms of a house. For the convenience of management, such interconnected IoT equipment often relies on the secure connections of HAN (Wi-Fi authentication) for protection and trusts all the computing systems on the same network. This treatment, however, completely exposes the device to the attacks from compromised local systems, a threat becoming increasingly realistic.

Menace of local threats. Indeed, it has been reported that high-profile WiFi-enabled smart home devices, including the WeMo Switch and motion sensor [195, 196, 197, 198, 199], Belkin NetCam [200], baby monitoring devices [201, 202, 98] and smart light bulbs [203], are all vulnerable to a local attack: an adversary within the same HAN is shown to be able to control those devices or steal sensitive user information, e.g., live video streams [200], from them. Several other studies further reveal that this is possible since such devices have poor—

or no–authentication mechanisms [99, 204, 205, 206, 207, 208, 209, 210, 211] and therefore easily fall prey to a local attacker. This is also validated from the analysis on Chapter 7 where I show that most Bluetooth devices perform no authentication.

Defending against such attacks becomes particularly challenging when the IoT devices are controlled by smartphones: once the same phone also carries malware (even when the app has nothing but the network privilege), protecting the device it controls becomes impossible at the network level, as the phone is completely legitimate to access the device though the malicious app running on it is not. Given the high smartphone penetration rates [212], the millions of available mobile applications on both official and third-party markets [213], and the ease of distribution of such applications [1], devices that can be reached through mobile apps can also become an easy target to adversaries. Unfortunately, such adversaries are not only realistic; they are on the rise [214, 215, 216]. Because of that they become the main subject of study of many other academic works [217, 7, 218, 219] while concerns are also raised on public communication channels [220, 221, 222]. In this study I verify that IoT vendors tend to trust the local network (Section 7.2). This makes them vulnerable to a mobile adversary as we illustrate with attacks on real-world IoT devices, including the *WeMo Switch*, *WeMo Motion*, *WeMo in.sight.AC1* and *My N3rd*. The demos of these attacks can be found on this study's website [223].

Addressing the issue here cannot solely rely on device manufacturers: business factors such as time to market and keeping the cost of the device low but also operational factors such as low power consumption, lead to the production of devices without encryption capabilities [5]. In such cases, response to threats can only be reactive and it would entail manufacturing a new version of the device which would still leave users with the old version susceptible to attacks. To make things worse, device manufacturers can be slow in responding [224, 225] to security and privacy threats. Router vendors have already identified this threat. New hubs and routers pushed onto the market are increasingly armed with various IoT protections (e.g. Microsoft Azure IoT hub [226], Google's OnHub router [227]. Integrating protection and management capabilities in the router has significant benefits as the infrastructure is already in place in most households and it enables unified policy management. However, as mentioned above, security control at the router level cannot succeed without knowledge of the OS-level situation within an authorized mobile phone, particularly whether a request to a target device comes from its official app or an unauthorized party. Fundamentally, a practical solution to the problem needs to bridge the gap between the OS-level observation (apps making network connections on a phone) and the network-layer view (requests from the

[1] Android applications can be self-signed.

phone for accessing an IoT device), with minimum modifications on the HAN infrastructure and all the systems involved.

Situation-aware device access protection. A simple solution to the problem is just inferring the identity of the app communicating with an IoT device according to its traffic fingerprint. This approach, however, is unreliable and can be easily defeated by, for example, a repackaged app that closely mimics the authorized program's communication patterns. Also, individual apps' fingerprints need to be reliably generated, deployed and continuously updated, and further to be checked on the router against each communication flow it observes, which adds cost to both the router developer and the user. In this chapter, I present a different approach, a new technique that achieves fine-grained, situation-aware access control of IoT devices over a home area network. The proposed system, called Hanguard, distributes its protection logic across smartphones and the Wi-Fi router for jointly constructing the full picture of an IoT access attempt during runtime, which is then utilized to control the access on the network layer. More specifically, on the phone side, the information about the app making network connections is collected and passed to the router; on the router side, security policies are enforced to ensure that only an authorized app can touch a set of functionalities the device provides. In this way, malware on network-authenticated phones can no longer endanger the operations of the IoT devices, even when the IoT devices are not equipped with proper authentication and encryption protection.

Hanguard is designed to directly work on the existing HAN infrastructure, without modifying mobile operating systems or IoT devices. To deploy the system, one only needs to install a *Monitor* app with non-system privileges on mobile phones and update the firmware of the Wi-Fi router with a security patch. A key technical challenge here is how to gather situation information (processes making network connections) on mobile phones, which is not given to a third-party userspace app on both Android and iOS. Although all these systems provide VPN support, the app using the service still cannot observe the process generating traffic and will significantly slow down the network communication of the whole system (Section 8.3.2). To address the issue, Hanguard leverages side channel information for lightweight discovery of runtime situation on Android smartphones and utilizes the VPN to only mark out authorized apps' traffic on iOS smartphones (Section 8.3.2). Such information is then delivered to the router through a separate control channel, which is synchronized with the traffic generated by the app (over a data channel) and used by the router to determine whether the communication should be allowed to proceed.

I implemented this design over both Android and iOS which cover more than 95% of the mobile OS marketshare [228], and a TP-Link WDR4300v1 Wi-Fi router. The system evaluation shows that Hanguard identifies and blocks all unauthorized attempts to access shared IoT devices with negligible overhead in the common case (see Section 6.4).

8.2 ANALYSIS

I performed an analysis on shared WiFi IoT devices and demonstrate the security implications stemming from a mobile adversary. These findings informed Hanguard's design decisions.

Methodology. One approach for the analysis in thic case would be to investigate the IoT devices' firmware. That would entail—after identifying such devices–finding images of their firmware or, for each device, buying the device and extracting its firmware. Subsequently, each firmware needs to be analyzed, which is a non trivial task [229]. However, most of these devices are now controlled by mobile apps. Thus their control mechanisms can be examined by analyzing the apps instead of the firmware. Note that, the latter approach has multiple benefits over analyzing firmware: (1) we can easily acquire Android apps, (2) there is no monetary cost, (3) it is generally easier and faster to analyze mobile apps than an embedded device's firmware.

To discover Android apps for IoT devices, I searched for them at Google Play using keywords such as "home automation" and "internet of things". This, turned out to be not very effective: through manual inspections of search outcomes, we found that many apps identified this way were not related to any IoT systems and in the meantime, popular IoT apps fell through cracks. The solution is to crawl `iotlist.co`, a popular site for discovering IoT products. From the list, the crawler collected the meta-data of 353 products (all listed products at the time of the study). The meta-data include "Title", "Description", "Product Url", "Purchase Url" and others. Such data was further manually checked to identify a list of package names for the official apps of these devices. The crawler then used this list to download the apps and their meta-data from the Play store. Out of the 353 products, I found that 63% (223) of them have apps on Google Play, 2% (7) are iOS only and the rest are mostly unfinished products (listed on `kickstarter.com` and `indiegogo.com`) or are no longer available. This indicates that indeed most IoT devices today are controlled by smartphones. In this study I have also further used popular reverse engineering tools (e.g. apktool [164], dex2jar [166]) to facilitate manual inspection of their source code.

Focus on home automation IoT. To better understand the operations of smart home devices, I manually went through (1) the meta-data of the collected products, (2) their online documentations and websites, and (3) through their apps' source code when available.

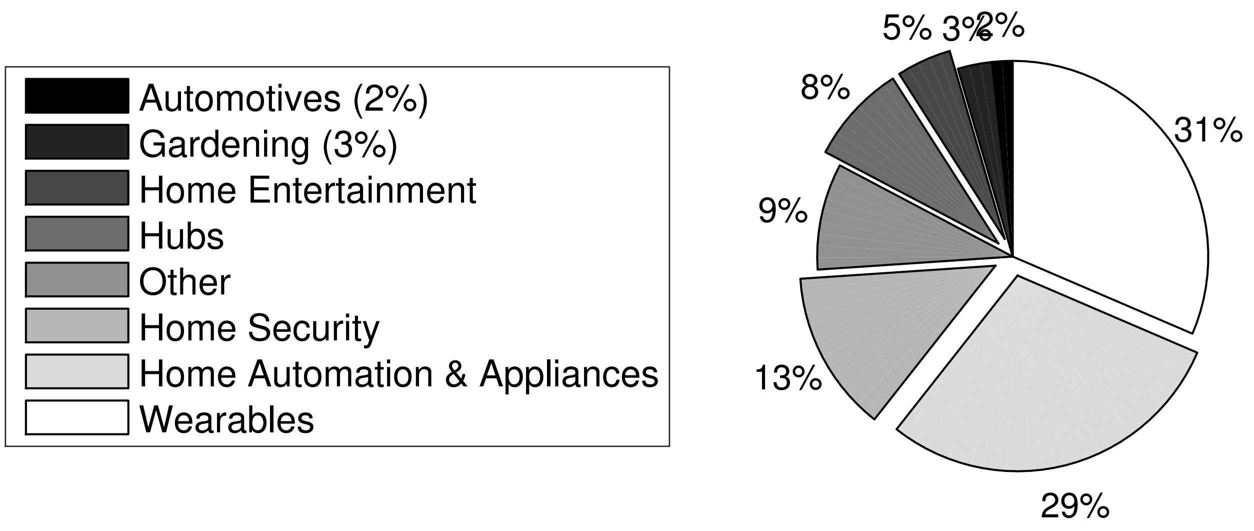


Figure 8.1: IoT product functionality categorization.

Figure 8.1 illustrates the manual categorization of the IoT products based on their functionality. Note that the *Wearables* category (31%) embodies mostly fitness and location trackers, smartwatches and personal medical devices. We call such devices *personal* devices; these commonly use Bluetooth to connect to a smartphone app. In the previous chapter 7 I have already presented my studies on the security of such personal devices, where I revealed problems with encryption and authentication and proposed a systematic solution. From the figure, we can also see that most of the listed IoT devices (55%) are smart home automation/entertainment/security/hub systems, which are the focus of this study. I call these *shared* devices. Such devices could directly benefit from an access control scheme built within the HAN. Previous work on shared devices, was focused on a single IoT integration platform (hub) [101, 111].

Focus on local connections. Prior research already demonstrated that the interaction between smartphone apps and the cloud is alarmingly unguarded [230, 231]. On the other hand, the local communication between the apps and the devices is not as well understood. In fact *it is unclear whether app developers and IoT device manufacturers treat the local network and everybody connected to it as trusted entities* and whether such treatments leave the devices susceptible to attacks from both local adversaries and remote adversaries that gain access to the HAN. Moreover, even though it has been reported that IoT devices come with serious problems [204, 205, 206, 207, 208, 209, 210, 211], little has been done to understand the security risks stemming from malicious mobile apps. This is particularly important since, IoT devices are controlled by apps which send commands either through the cloud or the local network. Here, we aim to bridge these gaps in knowledge. Our findings

Table 8.1: Contingency table of HAN IoT apps.

	w/ authentication	**w/o authentication**
w/ local WiFi connections	13	9
w/o local WiFi connections	25	0

build on to the existing evidence which collectively support the need for a unified security and management system built within the HAN to safeguard today's smart-home devices.

In particular, our IoT application study aims to achieve the following goals: (a) Find out whether vendors and developers of WiFi smart home devices/apps erroneously treat the home area network as a trusted environment; (b) Find out whether a mobile adversary can take advantage of such a problematic trust model to attack local smart home devices in practice.

(a) HAN Trust Model. I performed a statistical significance test focused on the following *null* hypothesis (N_0): *HAN apps with only remote connections are equally likely to perform authentication compared to HAN apps with only local connections.* To answer this question I separated our collected IoT apps into two groups. Apps with only remote connections and apps with only local WiFi connections. I used 55 unique Android applications with WiFi/Internet only connections to HAN IoT devices.

To separate the apps into the two groups, I manually went through (1) their online documentations and websites, (2) public forums, and (3) their Java Android code. I found that 22 (40%) do perform some internet socket connection with local discovered devices or fixed local IPs. 25 (45%) were found without local WiFi connections, 5 (9%) could not be determined, for 2(4%) decompilation failed, and 1 (2%) was by that time removed from Google Play. For each of the 2 sets (local; no local) I analyzed them further for any authentication practices. For the ones that perform only remote connections, I used a parsing tool that searches for password requests in the layout files of the apps. I then manually verified the existence/absence of a password request. I found that all these apps do perform authentication.

For the 22 apps with local WiFi connections I could not simply use the above tool since it would reveal little to no information on whether a password is used for a connection with the IoT device or the cloud. Thus I manually went through their code looking for network API calls responsible for local connections (e.g. creation of sockets connecting to local IPs, or UPnP discovery). I examined the calls to such APIs and found that 9 of the apps do not authenticate to the IoT device. The results are summarized in Table 8.1.

To determine whether apps with local connections are less likely to perform authentication one could perform a χ^2-test of independence. In our case this is not suitable due to the small absolute number of relevant available apps derived from `iotlist.co`. Instead I used the Fisher's exact test [232]—a common approach to derive statistically significant results when the sample size is small. I performed the test [233] on our *null* hypothesis (N_0). A 2-sided P value less than 0.05 was considered significant. The test yielded a 2-sided P-value of $0.00036 < 0.05$ and thus we can reject N_0. Therefore, we can confidently say that *HAN apps with local connections are less likely to get authenticated by IoT devices.* This validates an important intuition that IoT vendors consider the HAN to be a trusted environment. However, given the fact that smartphones are an integral part of such a network and that smartphones can carry self-signed apps from third-party markets, this treatment becomes detrimental to the security of shared HAN IoT devices.

(b) The mobile adversary threat. The previous finding is particularly alarming. Next I validated that a weak mobile adversary can take advantage of this problematic trust model and trivially compromise smart home devices. Towards this end, I cherry-picked four devices with local connections and authentication issues and performed real-world, practical attacks. The selected devices are listed on Table 8.2. The targets include the *WeMo Switch* and *WeMo Motion* [195], the *WeMo in.sight.AC1* [234], and *My N3rd* [235]. The *WeMo* devices are examples of popular plug-and-play devices. Just on Android, the official app of the WeMo devices was downloaded 100,000–500,000 times [2]. Note that all the WeMo devices are manufactured by a single vendor. By focusing on three WeMo devices I want to showcase how an erroneous trust model by a vendor can spread across various of its devices. This suggests that trusting the local network was a design decision and not an implementation issue manifesting in an isolated device. My N3rd, while not yet popular, it is chosen to showcase a new category of do-it-yourself (DIY) devices. It allows one to connect it to any other device enabling turning on/off that device from the My N3rd mobile app. Increasingly more such projects appear on the market with Arduino-based projects taking the lead. While exciting for users, such devices tend to inherit the problematic trust model and allow an adversary to take full control of ones devices. In our experiments we consider a mobile adversary that tries to get unauthorized access to the IoT devices. The mobile adversary can perform an attack from an unauthorized phone, or from an unauthorized app on an authorized phone [3]. To test the above cases, I used 2 Nexus phones. The first one is assumed to be untrusted and the second one is assumed to belong to one of the HAN users. I then tried to access the target IoT devices using both phones. Unfortunately I found that

[2]This is a conservative number as people can download the app from alternative Android app markets or from iTunes for iOS devices.

[3]Note that the case of an unauthorized app on an unauthorized phone trivially reduces to the first case we consider.

Table 8.2: Devices used in real-world attack demonstrations.

Target Device	**Description**	**# App Installations**
WeMo Switch	Actuator	100K - 500K
WeMo Motion	Sensor	100K - 500K
WeMo Insight Switch	Actuator	100K - 500K
My N3rd	Actuator	100 - 500

the adversary can trivially connect and control all WiFi shared devices. The video demos of the attacks can be found online [223].

8.3 MITIGATION DESIGN

My previous findings (Section 7.2) highlight the need for an access control system that can be integrated in home area networks with minimal changes to the existing infrastructure, that is backward compatible, independent of vendor and developer practices and which allows the users the flexibility to manage and control who should communicate to which device. In this section, I elaborate on the design of such a system called Hanguard and its implementation over the HAN and mobile platforms.

8.3.1 Design Overview

Adversary model. As shown in Section 7.2, IoT devices are controlled through smartphone apps. These devices are designed to act blindly on the commands from authorized phones (based upon their authentication with the HAN router). This treatment becomes increasingly problematic: while the smartphone may indeed belong to a rightful user, the applications that it runs can come from less known places (e.g., third-party app stores) and less trustworthy developers (e.g., malware authors). Given smartphone penetration [212], prevalence and ease of distribution of mobile applications [213], adversaries can now find their way to the HAN through a legitimate phone with minimal effort. Moreover—as demonstrated in Section 8.2—given the erroneous threat model of today's IoT devices, which trusts all the requests issued from a trusted source (a router or phone), such malicious applications can easily gain unauthorized control of IoT devices (e.g. turning on/off an actuator, or reading the collected data of a sensor).

Thwarting such attacks is inherently hard. For example, we could employ the solution (SEACAT) presented in Chapter 7. Smartphones with SEACAT will indeed protect attacks from malicious applications running on them. Nonetheless, WiFi smart-home devices are not only shared between apps on trusted users smartphones but also with any device on the network. Thus, a more robust solution for smart-home environments, is to implement a unified security logic in the router, since traffic from applications to IoT devices goes through it. However, the router alone does not have enough information to make any *application level access control* decision. One could resort to traffic fingerprinting techniques to infer the application generating the traffic. The approach can (1) be easily evaded by a malware repackaged from an authorized app, (2) bring in false alarms and (3) impacts the performance of the router.

Hanguard is designed to address this issue through bridging network and application level semantics, associating an app's identity to its traffic to enable a fine-grained access control on IoT devices. In the meantime, it does not modify both software and hardware of these devices, the operating systems of smartphones, and does not make assumptions about the router hardware. For this purpose, our adversary model is focused on the situation where a malicious app is installed on a smartphone device authenticated to the HAN. The adversary is considered to already know the communication protocol used by the victim IoT device. We further assume that the smartphone hosting the app has not been compromised at the OS or hardware level, which limits the adversary to the user land, at the app level. Note that though outside our adversary model, Hanguard can also provide coarser-grained protection against guest phones and compromised phones, remote adversaries and more traditional WiFi attacks. To avoid confusion, I refer an interested reader to our paper [15].

Idea and architecture. Figure 8.2 illustrates the architecture of Hanguard. Hanguard's design is partially inspired by software defined networking (SDN) (see [30] for a survey), which separates the network traffic (*data*) from its management (*control*). In the meantime, Hanguard is meant to be easily deployed to today's HAN. Serving this purpose is a distributed security control architecture that includes a *Controller* on a HAN router for policy enforcement and a *Monitor* on the user's phone for collecting its runtime situation and making access decisions (which are enforced by the router). To avoid changing the mobile OS, the Monitor is in the form of a user-space app. It detects the app making network communication and its compliance with security policies, and then pushes the access permit to the router's Controller through a secure control channel (Section 8.3.2). The router utilizes that information to enforce the policy (Section 8.3.3): only the traffic with a permit from the Monitor is allowed to reach IoT devices.

In essence, this design preserves the data channel within which unmodified information from smartphone apps is propagated to the router, and creates an independent control channel for security decisions. Such a separation, comes with obvious performance benefits: no

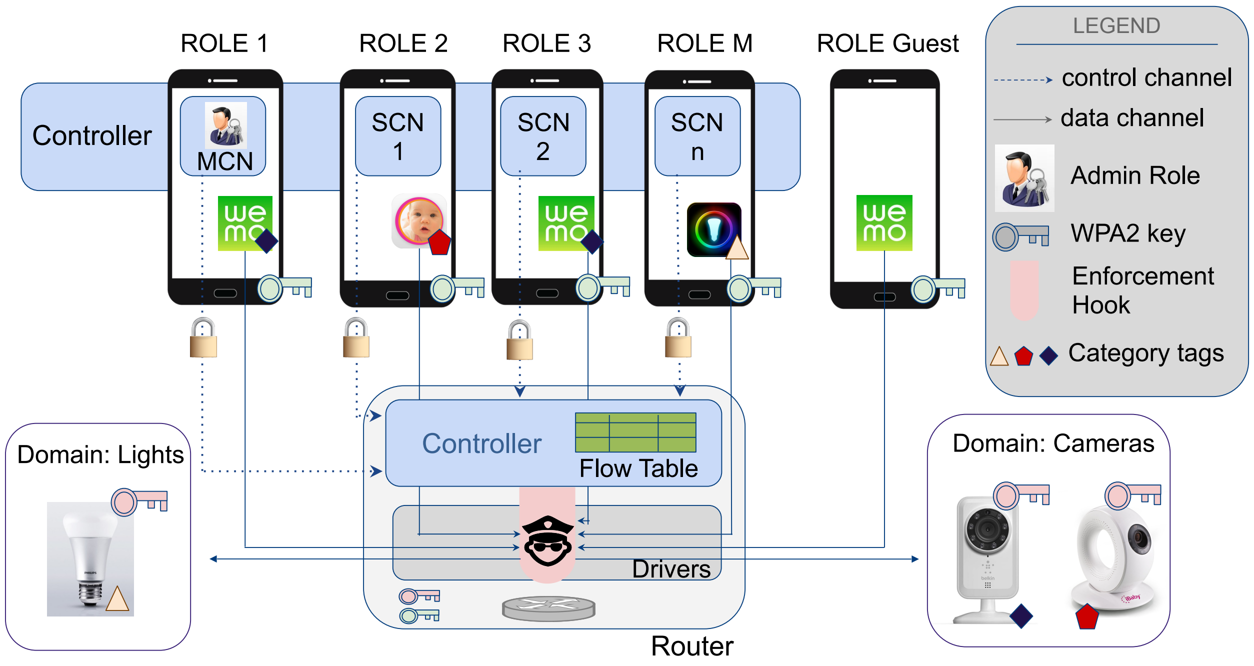


Figure 8.2: Hanguard high level architecture.

extra headers to be processed by the router on a per packet basis in the data channel. It can also guarantee that control information is always transmitted through a secure channel, and allows the router to further enforce policies and ensure, even in periods of heavy congestion, that security decisions are delivered in a reliable manner. In addition, our design allows for a clear separation of tasks: the security policies can be easily managed by the user through a mobile app interface; the router reduces to simply enforcing the flow decisions. This keeps the router as simple as possible and allows for readily updating the security logic with a mere application upgrade.

Policy Model. Hanguard implements an RBAC (*role-based access control*) policy model which leverages *type-enforcement* and *multi-category security* primitives. It uses them in a unique way to create SELinux-like policy rules, to protect smart-home devices. However, Hanguard does not need security experts to create the policies; policies are generated at runtime and transparently to the user. In particular, the user is only expected to perform simple mappings between a finite set of IoT apps, IoT devices and HAN users. Default policies are automatically created during setup to further reduce users' burden. Hanguard's access control model parses such mappings and assigns a *category* tag to each app and its respective IoT device. Further, each IoT device is labeled with a *type*. *Types* can be organized in overlapping groups called *domains*. Each mobile phone is assigned a *role* and each role can be configured to access a number of domains. For example, the iBaby camera can be labeled with the *type* “babyMonitor_t”. A *domain* “cameras_d” can be created to encompass the “babyMonitor_t” *type* device among others. Lastly, the *role* of a HAN user’s phone (e.g. “Adult”) that is supposed to be able to access the cameras, can be configured as eligible to access the “camera_d” *domain* and in extend the “babyMonitor_t” *type* device. The relation between the *role* and the *domain* ensures that an untrusted phone (e.g., a visitor’s phone) cannot touch protected devices and even an authorized phone, once compromised, cannot communicate with the IoT devices it is not supposed to talk to. At the same time, and orthogonally to the type-enforcement scheme, the iBaby camera and its official app, can be assigned the *category* “iBaby”. The *category* here binds a specific app on a phone to the device the phone is authorized to access. For example, the role “Adult” can be configured to access the domain “cameras_d”; while that stipulates that the adult’s phone can control the baby cameras, access is not granted unless the app on her phone and the actual baby camera that it tries to reach are tagged with the same category. Note that more than one category tags can be associated with a domain. This enables the generation of a policy rule which allows an app to access multiple devices of the same type.

By default, a phone registered with the HAN is assigned the role “HAN user”, which is allowed to access the “Home” domain. The latter encompasses every newly installed IoT device (which is assigned a unique *type*). However, the access can only succeed when the app on the phone is given the same category tag as the device it attempts to reach. Such an app-device binding is established when the app is used to configure the device, which is established through a special device, a phone or a PC, that takes the role of an *Admin*. This role can configure the router, register other user phones, access all domains and update security policies. During a policy update, new domains, roles and access relations between them can be generated. The policy model also handles unregistered phones (e.g., those belonging to visitors), which connect to the Han as a “Guest”, a *role* not allowed to interact with the devices in the “Home” *domain*. A security policy is shared by the phone side and the router side. Although its enforcement happens on the router, its compliance check is performed jointly by the router and the phone. The former ensures that only the authorized phone, as indicated by its *role*, can access the *domain* involving the device. The latter runs the *Monitor* to inspect the app and the target device’s *category* tags and asks the router to let their communication flows go through only when the category tags are the same. Next, I describe how individual components of the system work.

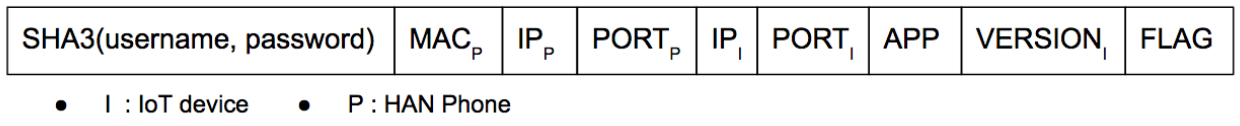


Figure 8.3: Hanguard Control Message delivered over TLS.

8.3.2 Phone-side Situation Monitoring

In this distributed access-control system, the *Monitors* are deployed as user-space apps. They are aiming at identifying the subject (app) trying to access an IoT device across the HAN, and determine whether it is authorized. Such information is delivered through a control message to the *Controller module* running on the router, informing it the context of the access attempt (since the router cannot see the app initiating the communication), which helps the router enforce appropriate security policies. Note that Hanguard is designed in a way that the workload on the router is minimized, which is important in maintaining the performance level needed for serving the whole local network. More specifically, the *Monitor* launches at boot time to establish an ongoing secure connection with the *Controller module* on the router. Through the channel, the situation on the phone is either *pushed* to, or *pulled* by the router (Section 8.3.3), enabling it to perform a *per-flow* (instead of *per-packet*) access control. Further, the security policies (Section 8.3.1) are broken into two parts: the *Monitor* checks whether an app is authorized to access a device and asks the router to enforce its decision, while the router implements a *phone-level* policy check as a second line of defense, which protects the smart-home devices even when a phone is fully compromised.

The communication between the *Monitor* and the router goes through a TLS control channel. The control message delivered through the channel is in the format illustrated in Figure 8.3. For example, it includes a hash of the user credentials (username, password), the sender phone's MAC address, an identifier for the detected flow (IP/port), an identifier for the app making the request, the policy's version number and a flag indicating whether this flow should be allowed or not. The negative flag is used to mark suspicious behavior (detection). Flow termination is handled by the router (Section 8.3.3).

Every registered phone on the HAN, can be assigned *roles* instantiating an RBAC (Role-Based Access Control) scheme. Furthermore, the phone used to configure the router is by default designated as the *Master Controller Node* (MCN) and every other phone is designated as the *Slave Controller Node* (SCN). A HAN user can update the policy through the *Policy Update Manager* running in her phone's *Monitor*. A *Monitor* accepts policy updates only when it is running on a master node and after verifying its user's credentials. A distributed *Policy Update Service* intermediates policy synchronization and replication in the system.

Every connected (reachable) node gets the latest policy replica as soon as it connects to the network or when there is an update. Unregistered devices are automatically assigned the "Guest" role as soon as they join the network. Each *Monitor* has a local in-memory replica of the policy base, that allows it to make decisions for its own traffic efficiently, alleviating the router from further processing. Having the policy also at the phone side is critical in SDN-like systems since it allows for efficient decision making by the Monitors, reduces the bandwidth on the control channel and keeps the routers simple and fast [30]. This way, Monitors send only their per-flow decision to the router instead of continuously sending all the mobile OS-situation measurements. In the last case, the number of control messages in the HAN would exponentially increase while the router would need to process all the measurements before making a decision, with severe performance degradation.

Situation awareness on iOS. As mentioned earlier, the *Monitor* is designed to find out which app is talking to an IoT device under protection. Such information, however, is not directly given to a non-system app on both iOS and Android. To tackle this Hanguard's iOS Monitor utilizes a new iOS capability that allows developers to proxy network traffic. Once this functionality is enabled by an app and approved by the user, all network packets from all apps will traverse the network stack and instead of being sent through the physical interface to the remote destination, they end up in a virtual interface (tunnel). The tunnel will redirect those packets to the proxy app running the VPN functionality.

iOS offers developers the capability to proxy network traffic with the `NEVPNManager` APIs). However, blindly tunneling apps' traffic through the VPN is very expensive, often slowing down the mobile system's network performance by an order of magnitude. This workflow is illustrated in Figure 8.4: when an app makes a network call this would entail, for every packet, a userspace-kernel context switch, traversing the network stack, trapping the traffic through the tunnel interface and context-switching to userspace again to deliver the network packets to the proxying app. Then the proxying app needs to process the network headers (essentially performing layer 3-4 translations) and then resending the packet.

The solution is to utilize the VPN in a unique way: instead of running the iOS Monitor to proxy the traffic of *all* apps (through the `NEVPNManager` APIs), which is expensive, requires a remote VPN server and gives little information about the identity of the app generating traffic, our iOS *Monitor* uses the `NEPacketTunnel Provider` APIs with a per-app VPN configuration, to tunnel the traffic *only* from *authorized apps* (the official apps of the IoT devices), while leaving all other traffic outside the tunnel to avoid unnecessary delays. Furthermore, over the tunnel, our iOS *Monitor* does not change the data: it merely acquires packet header information and forwards the packet to its original destination. After authen-

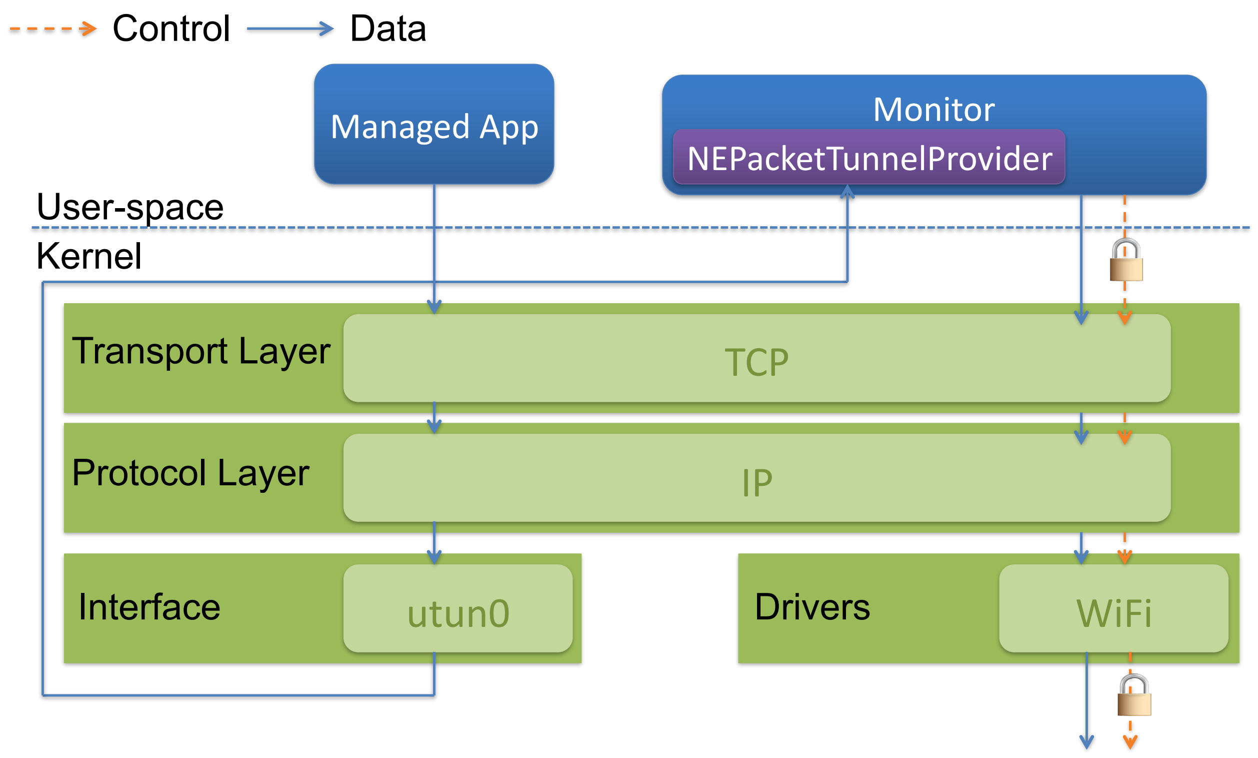


Figure 8.4: Traffic monitoring by a Hanguard iOS *Monitor*.

ticating itself to the *Controller module* on the router through TLS and its credentials, the *Monitor* informs the router that the flow in the tunnel is authorized. Other flows towards the IoT devices from the phone are by default considered illegitimate and will all be dropped at the router. In this way, we can strike a balance between the protection of legitimate IoT management traffic and the performance impact of the security control.

Situation awareness on Android. A straightforward way to capture traffic from other apps on Android is to follow a similar process with iOS and utilize the closely equivalent `VPNService` [236] API, introduced in Android 4.0. However, the implementation of VPN on Android is similar to the one in iOS and would entail similar overheads. To collect the situation information in a more lightweight manner, Hanguard leverages side channels on Android an approach which results in astounding performance benefits.

The Android *Monitor* continuously looks at the `procfs` file system (see Figure 8.5). `procfs` is a virtual file system which exposes the current status of an Android phone's kernel internal data structures. Particularly the files `proc/net/tcp`, `proc/net/tcp6`, `proc/net/udp` and `proc/net/udp6` disclose the ongoing TCP and UDP connections between the phone and a remote destination, including the source/destination IP addresses of the ongoing connection and its port numbers, the status of the connection etc [4]. The addresses here can be either IPv4 and IPv6 (with the suffix "6"). These connections are also associated with a specific UID that the *Monitor* can map to an installed app. To minimize operation overheads, the *Monitor* does not open and parse a file for each access. Instead it just checks the file's metadata (i.e. the last modified time or `mtime` in UNIX terms) to determine whether the file has been changed since the last visit. A complication here is that Android often fits an IPv4 address into the IPv6 format before reporting it to the user. Such an address is automatically captured by the *Monitor* and converted back to the IPv4 form. As an example, consider an app on a phone with an IPv4 address *192.168.1.189* that connects to an IoT device with the address *192.168.1.32*. During the app's runtime, the connection may not show up in `proc/net/tcp` but appears inside `proc/net/tcp6` instead with `0000000000000000FFFF 0000BD01A8C0` for the source IP and `0000000000000000 FFFF0000200 1A8C0` for the destination. It is clear that the IPv4 address is enclosed in the 32 least significant bits [5] and the 96 remaining bits are fixed. The *Monitor* detects the address from its fixed part and converts the rest to an IPv4 format before communicating the app's identity to the router through a control message. Note that Android suffers from the repackaged apps problem [219]. To address this the Android Monitor uses a package's signature to verify apps claiming the identity of policy-controlled apps.

8.3.3 Router-side Policy Enforcement

The design of the controller module mainly focuses on synchronizing security policies across all the systems within the HAN and enforcing these policies on the router, as illustrated in Figure 8.6. More specifically, the module maintains a *Master Policy Replica*, and runs a *Policy Update Service* responsible for updating the policies and distributing them across registered *Monitors*. Further, the *Controller module* introduces a *Per-Flow Decision Cache* (PFDC) for keeping the access decisions (on the app level) pushed by (or pulled from) the *Monitors*, and a *Garbage Collection Service* (GCS) for maintaining the cache. It also hooks on the router's packet flow for the policy enforcement. Due to space limitations, I omit discussion of the policy synchronization and focus on the policy enforcement.

Receiving decisions. By default the router blocks all flows to IoT devices. As mentioned earlier, app-level access control on the router relies on decisions made by the *Monitor* and

[4]Note that iOS does not reveal to an app the information about other processes through its `procfs` file system. Before iOS 9, one could use the system call `sysctl` to access such information. This channel has been closed since then.

[5]in little-endian order, presented using four-byte hexadecimals

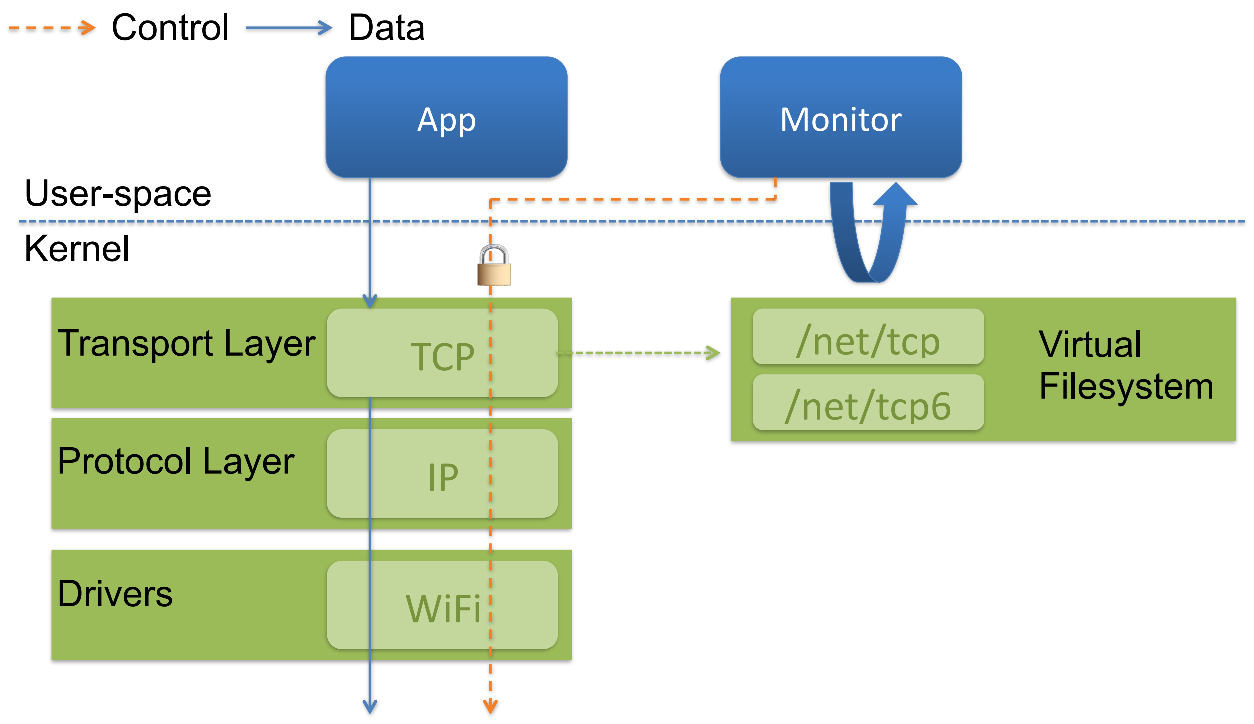


Figure 8.5: Traffic monitoring by a Hanguard Android *Monitor*.

delivered to the router through the control channel. To effectively enforce such decisions on a traffic flow, the *Controller module* is designed to efficiently authenticate and process the control messages to avoid holding up the legitimate interactions with the target IoT device. Specifically, the *Controller module* maintains TLS connections with the *Monitors* through a userspace program. When a decision from a *Monitor* arrives, after the successful TLS Monitor certificate validation, the router checks the policy version and the sender user's credentials, and once these are also validated, it passes the decision's *flow ID* (source IP and port, destination IP and port) to the kernel that updates the *PFDC* using the flow ID as the key to record the validation/invalidation decision on the flow, which is then enforced by the router. We highlight that data flows are first checked against a phone-level policy which ensures that the flow comes from a valid HAN phone.

Supporting this decision-making process requires an efficient userspace to kernel communication mechanism (for the router). Although this can be achieved through `system calls`, `ioctl` calls or `procfs` files, these approaches are either complicated to implement or unable to handle asynchronous interactions. Our solution employs the `netlink` socket IPC mechanism for the user-kernel communication, which can be easily built (without changing the kernel) and are asynchronous in nature: it queues incoming messages and notifies the receiver through a handler callback. In our implementation, the callback spawns a *worker*

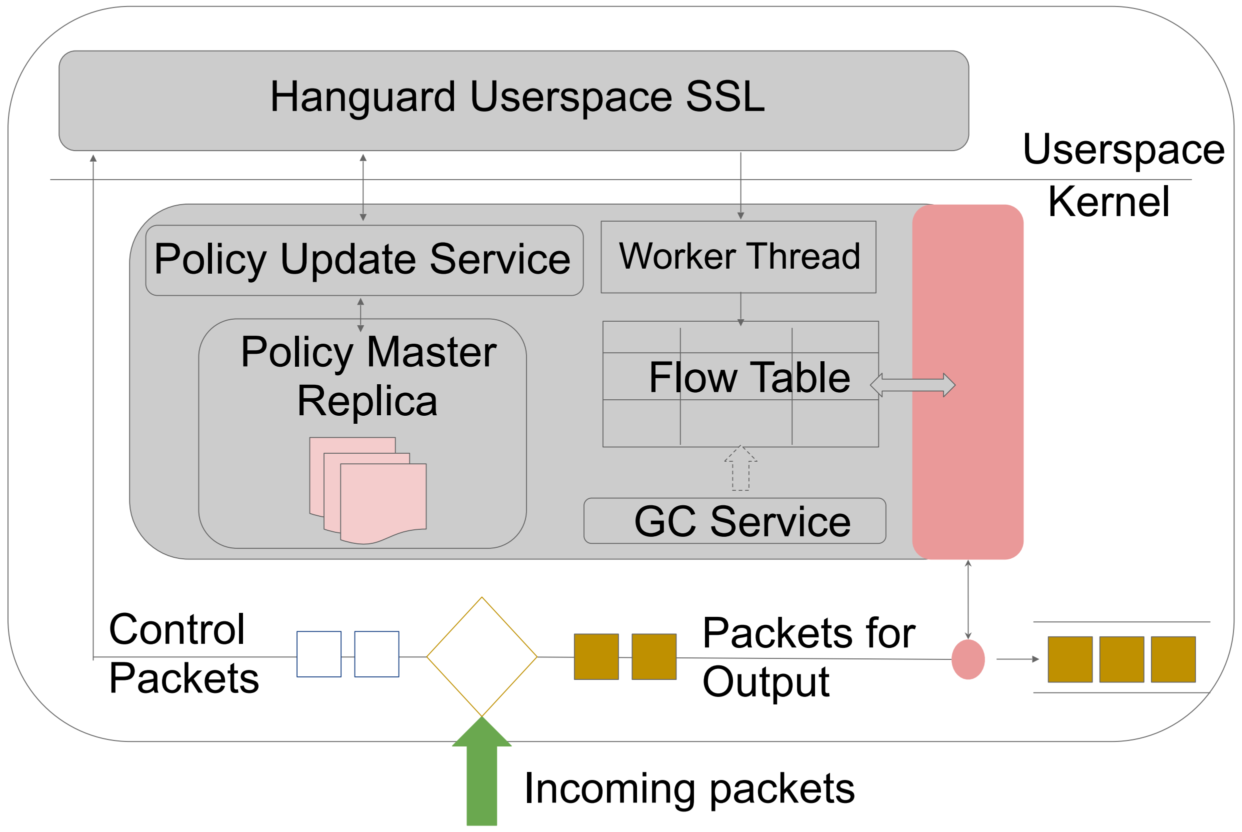


Figure 8.6: Hanguard Router Controller Module

thread that processes the message and updates the *PFDC*, either by inserting a valid flow or removing an invalid flow.

The *PFDC* is loaded at the router's boot time from its persistent storage. It holds the following information per-flow: the *flow ID*, the *flow validation/invalidation flag*, the *requesting app* and the *data last seen time*. This cache is used for enforcing app-level policies (whether a specific app is allowed to access a device), for the purpose of enhancing the existing flow-control capability of the router, which cannot differentiate two flows from the same IP and port but produced by different apps. By searching the cache, the router can apply the app-level access decision upon the whole flow, instead for every individual packet, an advantage over deep packet inspection and traffic fingerprinting techniques. To limit the amount of the resources the cache uses, a *Garbage Collector Service* (GCS) is run to remove the obsolete records with the oldest data last seen time. To prevent DoS attacks where a *Monitor* attempts to flood the cache, a per-phone limit is applied.

Enforcement. The router enforces *phone-level* and *app-level* policies. For the former, it checks every packet to determine whether it originates from a phone which is allowed to access a particular IoT device. Phones and IoT devices are identified based on their

MAC addresses. MAC-IP associations are statically defined during a new device enrollment. For *app-level* policies, the router checks the *PFDC* cache to determine whether the flow is generated from a valid app. A technical challenge in implementing the protection is where to place the security control within the existing router infrastructure. On a Linux-enabled system used by the router, once a packet is received, it is put by the link layer into a backlog queue from which the IP layer pulls packets for checksum checking and routing decisions. If the packet is destined for the current machine, it is passed to the transport layer. If not the packet is forwarded. Apparently, the security control should happen on the IP layer (e.g. in the `ip_forward()` function). However, a packet might follow a different path within the kernel depending on whether the current system is configured to run as a bridge or a router. For example, in a bridge mode, no layer 3 operation is involved and as a result the aforementioned function will never operate on the packet. To overcome this, Hanguard places the Controller hook in `dev_queue_xmit()`, a generic driver function, ensuring that no packet bypasses the check.

To minimize the impact on flows unrelated to the smart-home devices, the Hanguard-enhanced router quickly inspects each packet it receives to determine whether further attention is needed. Specifically, a TCP flow is considered interesting if its destination MAC address is associated with an enrolled IoT device. Packets not fitting this description are forwarded without a delay, and others are first handled according to the phone-level policy (whether the phone can access the IoT device) stored at the router, and then the app-level policy (whether the app can do that) which is based upon the validation flag set by the Monitor. For the packet allowed to go through, its flow's last seen time is updated to the packet's arrival time. Hanguard helps its users detect and react to spurious access attempts with its *notification mechanism*: Hanguard (1) keeps a log, and (2) sends out-of-band notifications to the admin user when a violation or tampering of the policy is attempted.

Flow Termination. A determined adversary could attempt to exploit the fact that a flow from a co-located app is allowed. For example, it could wait for the benign app to release its port and attempt to send a packet before the Monitor informs the router to invalidate the flow. For TCP flows, the router prevents such attacks: it proactively invalidates a validated TCP flow, when it sees its corresponding TCP FIN packet and then handles the session termination. For UDP, the situation is more complex. UDP is an unreliable protocol with no clear indications of a session establishment/termination. Hanguard can be configured to handle UDP flows in two ways: (a) in a *STOP-AND-WAIT* mode, for every packet, it *pulls* a decision from its Monitor. If between the time the packet is received by the router and the decision request is received by the Monitor, no other app on the same device attempted to send a UDP packet to the same target IoT device, then and only then the packet is allowed. This is a security stringent policy that prevents the attack. However, it comes with performance penalties since every packet is delayed approximately by one RTT. (b) In the *DETECTION* mode, the Monitor pushes a flow invalidation decision when the benign app releases the port. In this case, a malicious UDP packet from an Android app could make it through before the decision is enforced. However, the Monitor will (a) detect the malicious attempt; (b) can determine the offending app and; (c) can determine the affected device (destination). Once a violation is detected the user is notified to verify the status of the affected device and uninstall the offending app which is also blacklisted in the policy. On iOS such race attacks are always prevented: offending tunelled traffic is blocked on the phone whereas non-tunneled traffic to IoT devices is blocked at the router.

8.4 MITIGATION EVALUATION

I implemented a prototype of Hanguard—in *DETECT* mode for UDP (Section 8.3.3)—on top of a TP-Link WDR4300v1 router with a Gb NIC and a wireless network at the 2.4 GHz band (300Mbps) running OpenWRT Chaos Chalmer with a Linux 4.1.16 kernel, and also Nexus phones running Android 5 (Lollipop) and an iPhone 4S running iOS 9. The evaluation answers the following research questions:

- *RQ1:* Is Hanguard effective in thwarting attacks from malicious applications?
- *RQ2:* What is the performance impact and resource consumption of the *Monitors* on the phone side?
- *RQ3:* What is the overall overhead of Hanguard?

8.4.1 Effectiveness

To answer RQ1 and verify Hanguard's backward compatibility and practicality, I repeated the attacks I demonstrated on real world smart-home devices (listed in Table 8.2). I performed the following two experiments: (A) first I set up the target IoT devices over the "Vanilla" system (without Hanguard's components), and further installed a repackaged version of their legitimate app on the phone to mimic the adversary; (B) next, I updated the router with Hanguard-enhanced firmware, and also put our *Monitor* app on the same phone with a policy that allows the phone and the benign app to access the target IoT device. Under this protected setting, I repeated experiment (A), using the phone with the *Monitor* app to set up the IoT devices. As expected, both the original app and the repackaged one

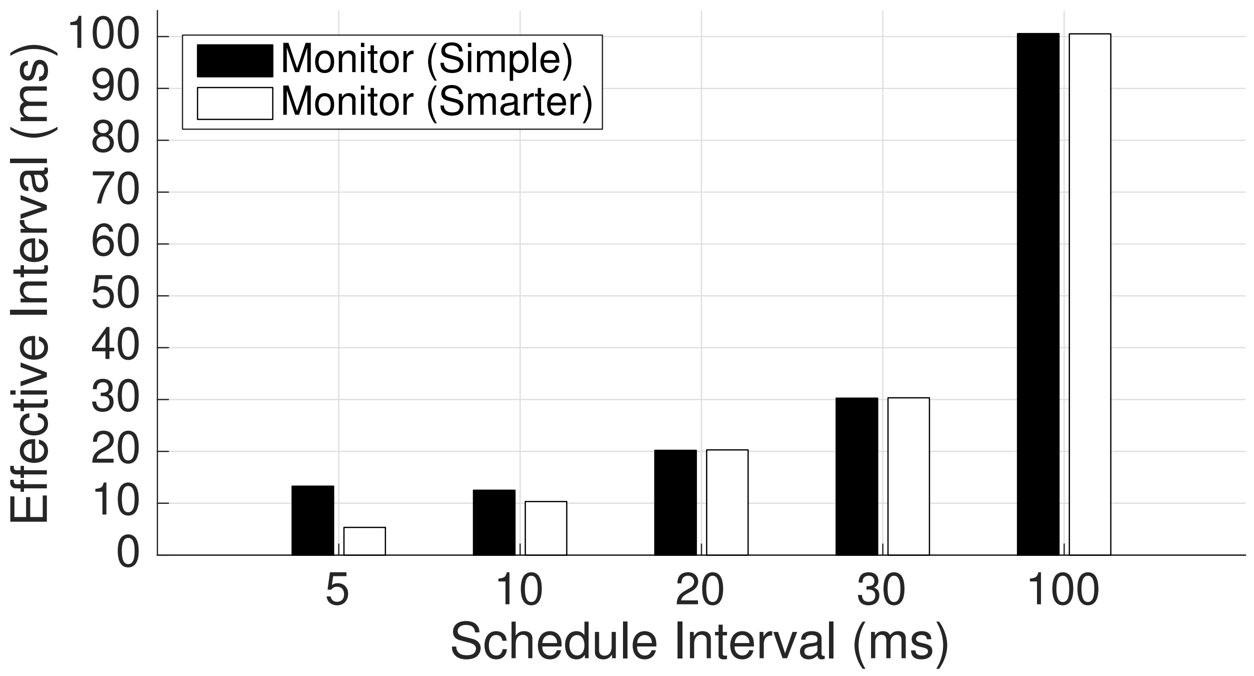


Figure 8.7: Android Monitor polling scheduled frequency vs Actual polling frequency.

could access the devices in the Vanilla system. With Hanguard enabled, only the official apps on the phone running the *Monitor* app could communicate with their respective IoT devices , which confirms the effectiveness of the access control enforced by Hanguard and its backward compatibility (see [223] for demos).

8.4.2 Phone-side Performance

Monitoring cost on Android. On Android, the *Monitor* continuously polls the `procfs` file system to detect ongoing network connections. Here I report a study on two monitoring strategies and their performance impacts. Specifically, I configured the Android *Monitor* on a Nexus phone to inspect the `procfs` file system in different granularity (every 5ms, 10ms, 20ms, 30ms, 100ms). After running for 30 seconds, the *Monitor* went through every single file line to check the presence of interesting network connections, a strategy called the *Naive* mode. The approach was compared with another strategy I also introduce, called the *Smarter* mode, which first looks at the last modified time of a file before accessing its content. The outcomes of the study are illustrated in Figure 8.7. Clearly, the *Smarter* strategy clearly can poll at a finer granularity (5 ms), given that it reads much fewer file lines compared with the *Naive* approach (Figure 8.8), which is translated to less work per iteration in the common case.

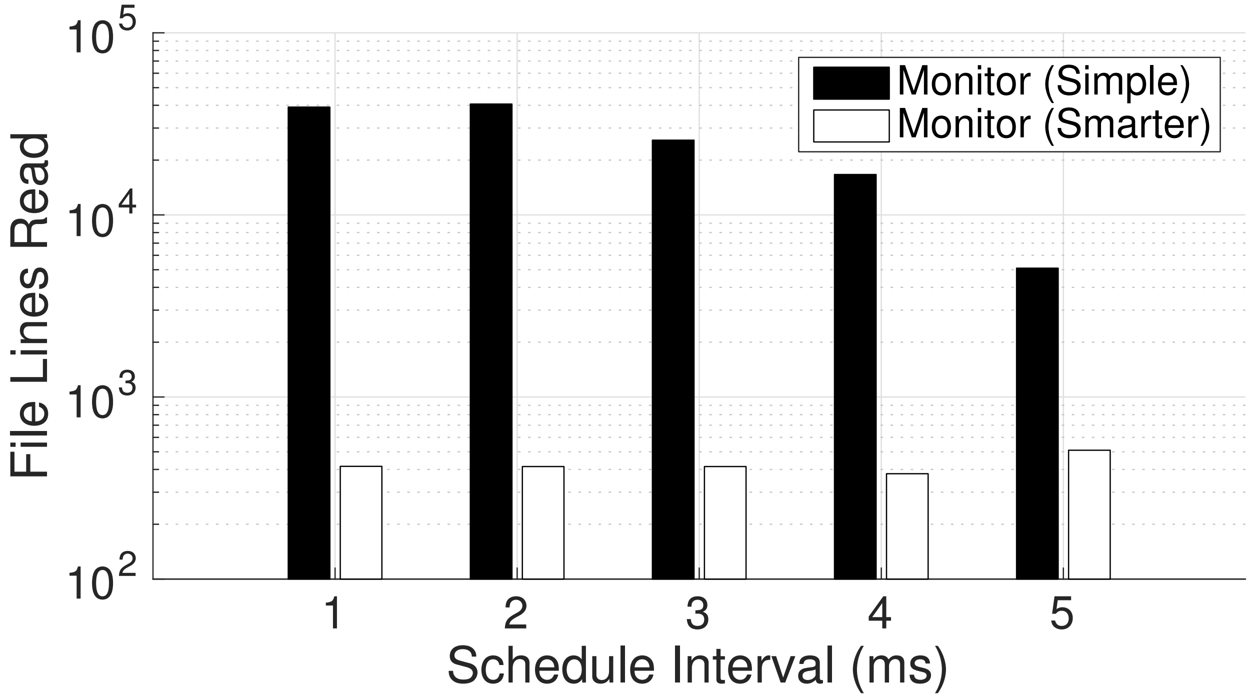


Figure 8.8: Number of file lines parsed for different Android Monitor configurations.

I further looked into the resource consumption of the *Monitor*. For this purpose, I configured the *Monitor* to poll at 10 ms and recorded its CPU and battery consumption for both the Naive and Smarter mode. On the same Nexus 5 phone, I also ran Trepn [237] by Qualcomm to collect the baseline power profile of the phone for 30 seconds before running our *Monitor* app for 2 minutes. Figure 8.9 illustrates the average battery consumption that can be attributed to the *Monitor*, and Figure 8.10 shows the average CPU usage (first 4 bars). To put things into perspective, I compared the *Monitor* with a popular Antivirus app in scanning mode and the de facto mailing app on Android (Gmail). As it is evident from the figures, the power consumption of the naive approach is comparable to an antivirus app performing an expensive operation while the smarter mode's is comparable with Gmail which is optimized to always run in the background.

Monitoring cost on iOS. To evaluate the iOS *Monitor*'s resource consumption, I used *Instruments* [238], a performance analysis and testing tool which is part of the official Apple IDE (Xcode [239]). Figure 8.10 depicts the % CPU utilization that can be attributed to a runtime process, where measurements on iOS are indicated with * (last 3 bars): the *Monitor* when proxying a TCP app that sends 500 messages with payload size equal to one character; the *Monitor* when proxying an equivalent UDP app; and YouTube while streaming a video configured to *auto-select* its quality. The figure reflects the fact that the iOS *Monitor* does

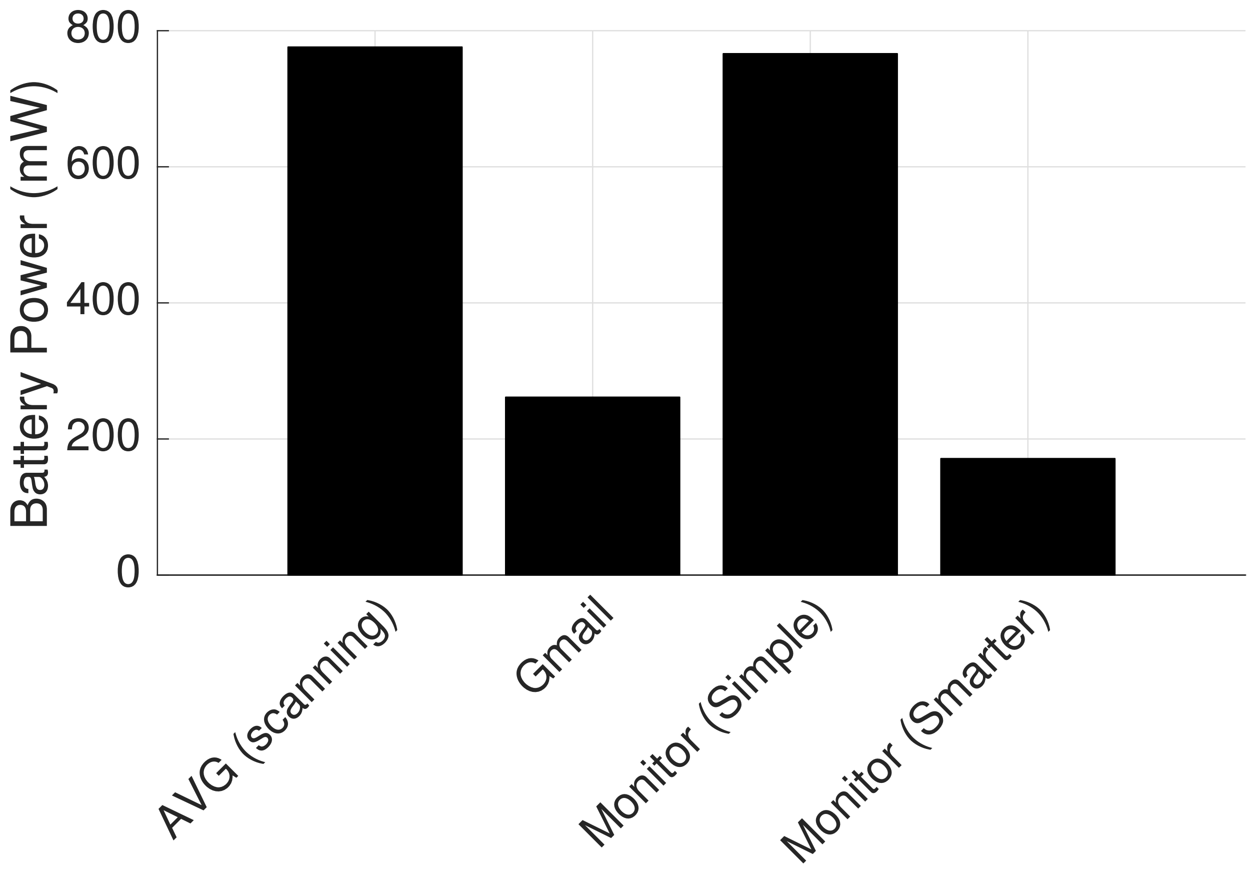


Figure 8.9: Battery Power on Android.

a lot of work when proxying TCP traffic: this is expected as TCP is a connection oriented protocol and the *Monitor* needs to guarantee reliable delivery of the packets. For UDP the *Monitor* does very little work. In idle mode (not proxying), the *Monitor* incurred no CPU overhead. *Instruments* can also report the *Energy Use Level* of an app at runtime as a value from 0 to 20. In all experiments the reported value was consistently 0/20.

Detection accuracy. The *Monitor*'s goal is to detect an interesting flow generated on the phone. For iOS the detection accuracy is 100% since all packets from interesting apps are routed through the *Monitor*'s VPN. For the Android *Monitor* though, the situation is more complicated. For example, an interesting app might quickly set up a socket, send a packet and then close the connection. The Android *Monitor*'s detection accuracy depends on whether it can catch such events given its polling interval. To answer this question I created a micro-benchmark that includes a TCP and a UDP app connecting to a TCP and UDP echo server respectively. They both stop the communication once the server response is received. Again, we ran the *Monitor* in the Smarter mode 10 times for each of the following polling configurations: 150ms, 100ms, 30ms and 10ms. My empirical results

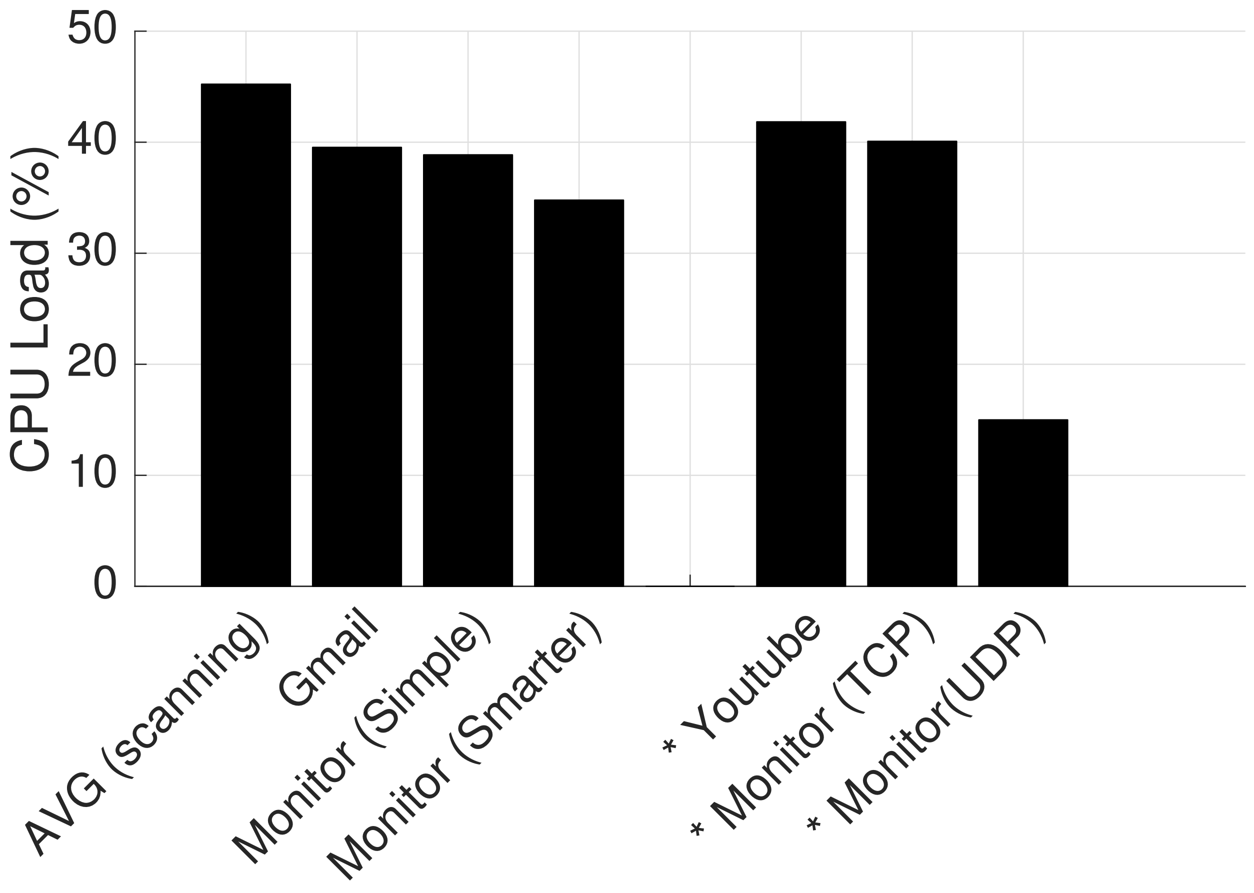


Figure 8.10: CPU Load on Android and iOS(*).

indicate (see Figure 8.11) that the 10ms configuration could always detect outgoing TCP and UDP connections.

8.4.3 Communication Overhead

To answer RQ3, I assessed the overall performance overhead of Hanguard, as this can be observed from a mobile app. I created a baseline by performing the experiments below on the unmodified system (Vanilla). To evaluate Hanguard communication overheads we repeated the experiments on Hanguard with the respective benchmark app being either policy-protected (Managed) or not protected (Unmanaged).

Application latency. I ran the TCP and UDP apps individually, configured to send 100 messages each. Figure 8.12 depicts the mean latency in milliseconds (ms) for a TCP message and a UDP message for Android. The latency is measured as round trip time (RTT) on the mobile app. In particular I measured the time interval between the API call to send the message and the time that the message is returned by the server and delivered to the

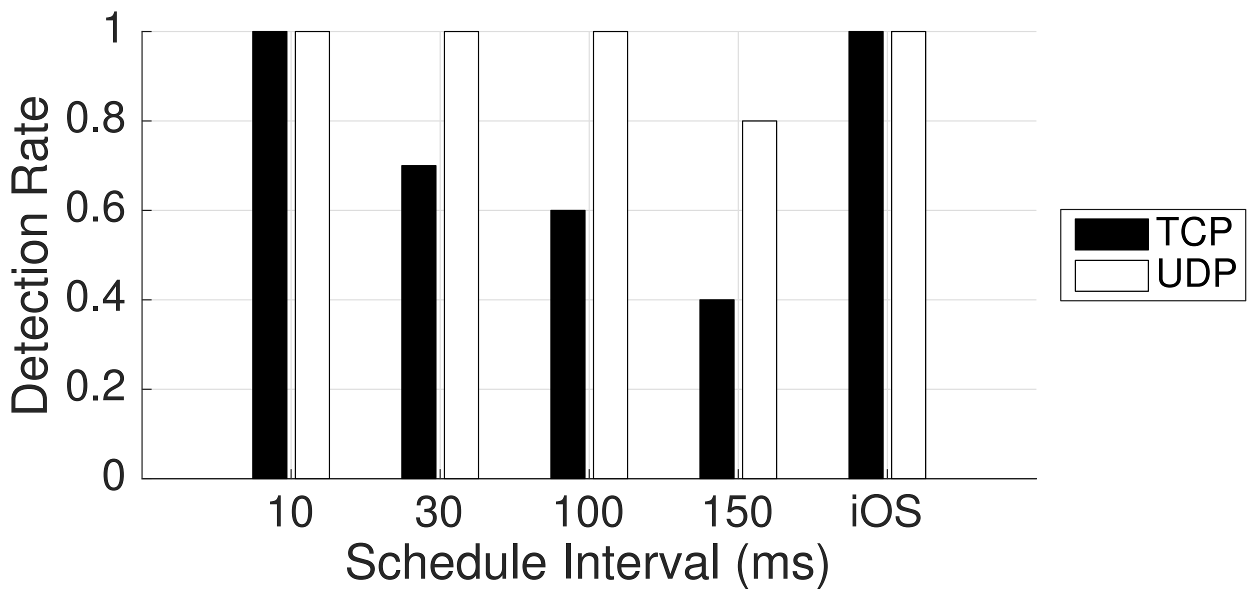


Figure 8.11: TCP and UDP flow detection accuracy for different Android *Monitor* configurations and for the iOS *Monitor*.

application layer. As we can observe, Hanguard introduces negligible latency for Managed apps on Android.

In Figure 8.13 we can see that there is a big increase on TCP packet latency for the Managed apps on iOS. Nevertheless, in practice this is often tolerable, since most devices are actuators and sensors that create mice flows [6] delivering a small amount of information: for example, it is completely imperceptible when the delay for switching a light grows from a few milliseconds to tens of milliseconds. This Figure also reveals an important benefit of our design: *the security controls have negligible impact on Unmanaged apps, on both Android and iOS devices, for both UDP and TCP.*

Application throughput. To measure Hanguard's throughput overhead, I used the benchmark apps to transmit a file of 20MB to their server counterparts. I repeated the experiment 10 times. Figure 8.14 and Figure 8.15 plot the throughput CDF for Android and iOS respectively. Evidently, Hanguard has negligible impact on throughput for all Android apps and iOS unmanaged apps. Our evaluation also reveals an interesting case: throughput drops significantly—but only—for the iOS Managed apps [7]. This happens because the iOS Monitor implementation uses the built in VPN utility of the OS. Thus, it has to inspect every

[6]A mouse flow is a flow with a short number of total bytes sent on the network link.

[7]In practice this will only affect real-time streaming services offered by such app-device connections. Actuators and sensors will not exhibit a noticeable effect.

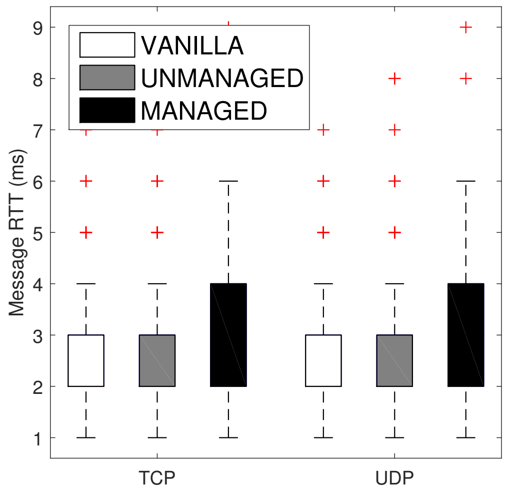


Figure 8.12: Application-level communication latency for Android.

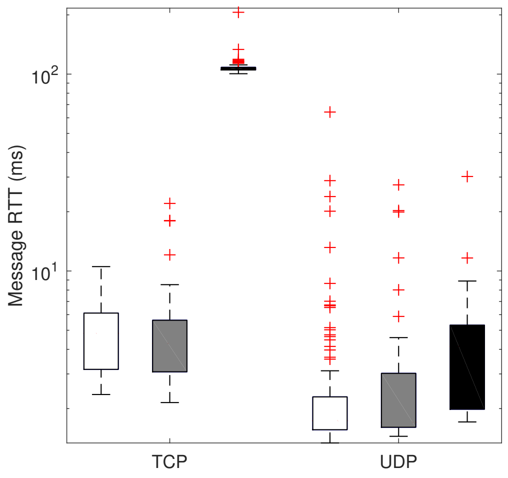


Figure 8.13: Application-level communication latency for iOS. The $x - axis$ is in log_{10} scale.

packet for managed apps (see Figure 8.4). This is a security, performance trade-off we had to address. We opted-in for security.

8.5 DISCUSSION

HAN users smartphones OS integrity. The application-level enforcement assumes that HAN user phones are not compromised. Preventing phone compromises is out of the scope of this study since other solutions already exist and even deployed on commodity smartphones [240, 241, 242, 243, 244, 245]. For example, SELinux for Android [246] uses mandatory access control to ensure that even compromised system processes are restricted, and is

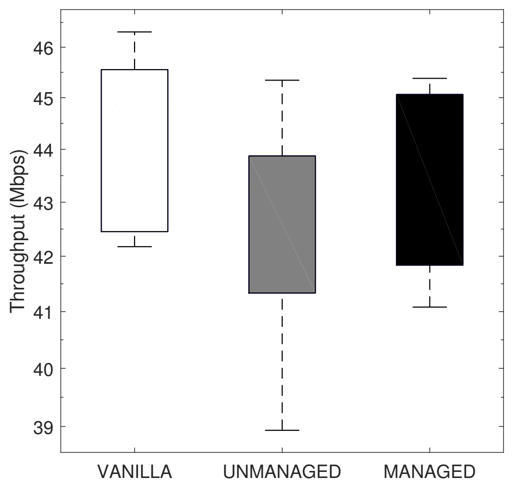


Figure 8.14: Application-level throughput for Android.

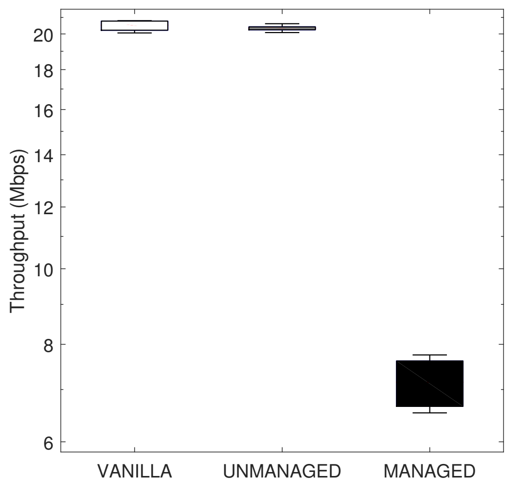


Figure 8.15: Application-level throughput for iOS.

deployed on all Android phones with version 4.4. and higher (more than 60% in 2015 [247]). Most Android phones are equipped with ARM processors [248] with TrustZone [243] which can be utilized for solutions stemming from the trusted computing domain. TZ-RKP [244] is a real-time kernel protection technique deployed on Samsung Galaxy phones that ensures the kernel integrity using the ARM TrustZone secure world. iOS devices have the Secure Enclave, a secure co-processor that is used to guarantee secure boot [245]. However, even if a user device is compromised (and in the case of all guest phones), Hanguard can guarantee *phone-level* protection.Furthermore, Hanguard helps its users detect spurious access attempts by (1) keeping a log, (2) sending out-of-band notifications to the admin user when a violation of the policy is attempted.

Switching Between Information Gathering Approaches. iOS follows a far more stringent approach than Android in isolating processes. In fact our Android solution for traffic monitoring does not work on iOS. Instead we utilize Apple's `NEPacketTunnel Provider` API with a per-app VPN configuration. The latter requires an MDM (Mobile Device Management) server: the *router* vendor will need to enrol their users' iOS devices and push an over-the-air (OTA) configuration profile on the phone, just like the cell phone carriers (e.g. AT&T, T-mobile e.t.c.) do. This process is already mature and streamlined for users who just need to accept the configuration. Apple does offer the non-enterprise `NEVPNManager` API but that would entail Hanguard iOS Monitors proxying traffic not only from a selected set of apps but from all apps, imposing the overheads we demonstrate in Section 6.4 for both unmanaged and managed apps. In the proposed design we opted for security and runtime performance in the expense of an initial bootstrapping usability burden, that allows us to selectively proxy traffic only from a handful of apps when used in the HAN environment. This work illustrates how such capabilities can facilitate novel solutions on the iOS platform. Also note that any of the two aforementioned techniques can be used in practice with Hanguard iOS Monitors. Hanguard's design, allows router vendors to readily switch between monitoring techniques with a mere application update. Similarly, if access to the Android `procfs` as a whole is forbidden in the future (not a straightforward decision since this would break a lot of legitimate apps), Hanguard can switch to a VPNService-based Android Monitor by merely pushing an app update.

8.6 SUMMARY

In this chapter I presented my study on the security of smartphone communications with shared smart-home WiFi IoT devices. My analysis revealed that introducing smartphones in such environments introduces attack surfaces stemming from the smartphone's multi-tenancy. Indeed I showed how a malicious application on an otherwize authorized phone to the smart-home network, can gain unauthorized access to connected IoT devices on the same network. To tackle such adversaries I presented an effective, efficient and backward compatible system solution distributed between trusted phones and the home area network router. The solution leverages the vantage position of the router to mitigate atacks even from compromised devices but also collaborates with a userspace applications on trusted phones to tackle other malicious apps on those phones from compromising the shared IoT devices. Figure 8.16 visualizes the userspace traffic monitoring addition to the smartphone operating system.

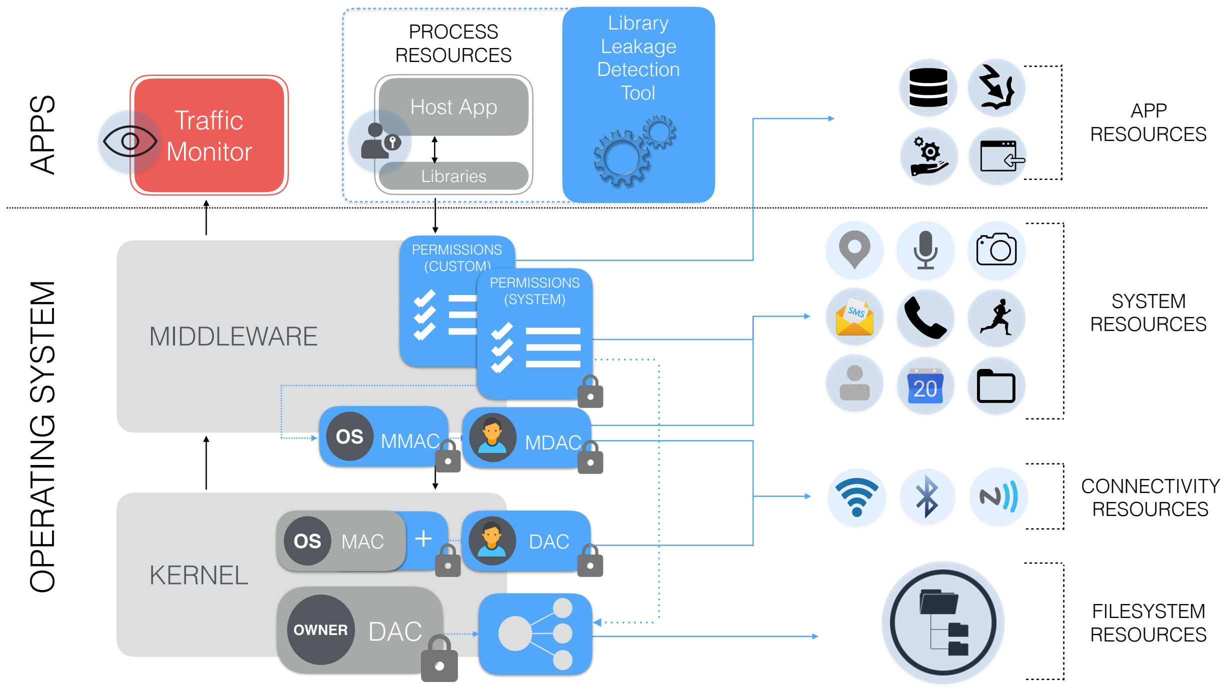


Figure 8.16: Userspace traffic monitors can help network routers enforce fine-grained policies on network access request to shared IoT devices.

CHAPTER 9: DESIGN PRINCIPLES

Over the years, stakeholders came up with fundamental design principles for secure systems. However, none of this would have completely prevented the problems I found in my work. In this chapter, I will present a set of new security design principles stemming from my analysis on modern, smartphone operating systems.

Saltzer et al. [249] discusses 8 principles to guide the design of a system which facilitate reduction in security flaws. These principles are listed below:

- **Economy of mechanism**: this refers to keeping a design as simple and small as possible.
- **Fail-safe defaults**: fail-safe defaults support a design where by default access is denied, unless there is a condition under which access should be permitted.
- **Complete mediation**: authorization should be checked on every attempt to access every object. This is a fundamental principle fo every access control design.
- **Open design**: the security of a system should not depend on its design secrecy. For example the Android Open Source Project follows this principle. In contrast iOS is a closed system. However, we should not assume it is more secure just based on that.
- **Separation of privilege**: when possible, require two keys to unlock a protection mechanism.
- **Least privilege**: every principal should only have the least amount of privileges possible in order to complete their tasks. Mandatory Access Control mechanisms can achieve that with an ideal policy.
- **Least common mechanism**: shared mechanisms should be minimized as they can be the source of potential information path between isolated principals.
- **Psychological acceptability**: user interfaces should be simple and protection models should match the users' mental models. This is to help user make correct security decisions.

Similarly, the GenCyber program, targets building a cyber security curriculum based on 10 cyber security first principles. First principles should be the fundamental building blocks of any security design. GenCyber aims to be more specific since it targets K-12 students and instructors. According to Payne et al. [250], these principles are:

- **Minimization**: aims to reduce the attack surface by disabling unneeded functionality. This is a corollary from Saltzer et al. *Economy of mechanism.*
- **Simplicity**: facilitates clear understanding of a system's functionality. This is a corollary from Saltzer et al. *economy of mechanism.*
- **Abstraction**: remove any distracting details when not needed.
- **Information hiding**: prevent certain features from being available publicly (users, other apps etc.).
- **Least privilege**: same as Saltzer et al.. Any tunning program should have the minimum set of privileges possible to perform its tasks.
- **Modularity**: break up complex functionality into small components (modules). This improves manageability, interoperability, security and protection. This is a corollary from Saltzer et al. *economy of mechanism.*
- **Layering**: implement multiple layers of defense. If one layer is defeated, another one might stop the attack. This is also discussed by Lampson [251] which defines it as *defense in depth.*
- **Resource encapsulation**: all resources should be separated and use as intended. This is a corollary from Saltzer et al. *complete mediation.*
- **Process isolation**: isolate processes sharing a platform so they do not interfere with each other. Most operating systems implement this. Every program runs in its own process with its own address space.
- **Domain separation**: separate areas where resources are located. For example, SELinux uses assigns principals (subjects) to domains to dictate which resource each domain can access.

All of these are valuable guidelines for building secure systems. However, during my studies, I have identified some new valuable principles that could have helped in the design of modern smartphone operating systems. These are: *contextual threat model*; *granularity of mechanism* and; *layered responsibility.*

9.1 CONTEXTUAL THREAT MODEL

The security of a system is designed with a threat model in mind. However the threat model itself should be designed taking into account the peculiarities of the context or environment in which the system will be deployed.

Consider for example the analysis we performed on shared filesystem resources on smartphone operating systems (see Chapter 5). Android is built on top a stripped down version of a Linux kernel. The Linux kernel does perform *process isolation* to prevent users from interfereing with each other. It also uses a Discretionary Access Control mechanism to control access to filesystem resources. By adopting the Linux kernel on Android, Android engineers also made the decision to adopt its security models. This decision let to a number of problems where seemingly innocuous information made available on a desktop machine becomes dangerous when applied on a smartphone operating system. There are two main differences between traditional Linux-based machines and smartphone operating systems which change the threat model: (a) the notion of a user; (b) mobility.

In a traditional Linux system, a user ID (UID) is assigned to every user of the system. A user is typically a human that uses the machine. The DAC mechanism which protects resources on the device, relies on checks based on that UID and group ID (GID, users can be grouped). On Android, UIDs are now assigned to applications. That is, every application is considered as a different user of the device. Android engineers wrongly assumed that they could merely reuse the Linux UID and inherit all Linux protection mechanisms. The threat model though, is not trivially transferable between these environments. For example, revealing UIDs to users in a traditional Linux environment is of low risk. On smarpthone operating systems, revealing the UIDs to applications (not users) means that every application can be made aware of other applications that are installed on the smartphone (from the UID one can derive the unique package name of an app on Android). This can be problematic since the mere presence of apps reveals sensitive information: a user might install a dating app (e.g. Tinder) or lifestyle app (e.g. a gay social network like Hornet), or healthcare apps (e.g. a diabetes app or a pregnancy app). Other apps might target the user or harvest and sell this information about the user to third-parties. I further showed in my analysis on shared process resources (see Chapter 4) that the list of installed apps on a smarpthone can be highly indicative of its user's personally identifiable information (PII) such as age range, gender and, zip code, but also if they suffer from a medical condition or allergies, salary range, marital status and more.

Also, other information seemingly harmless on traditional *static* Linux environment, become dangerous when available on a *mobile* platform. In Chapter 5 I show how address resolution protocol (ARP) information available to a physical user on a traditional Linux machine, can be leveraged by third-party applications on smarpthones to track the user's location. In a published related work[11] we further showed that just by knowing whether the speaker is on or off, can also enable a third-party app to track the smartphone user's driving route. These are adversarial capabilities not present on a static machine.

Thus it is very important when designing a new system to re-consider the adversary model before adopting any security mechanisms or decisions from another system.

Contextual Threat Model: The threat model for a system should be desinged taking into account the context within which the system will be deployed.

9.2 GRANULARITY OF MECHANISM

Saltzer et al. discusses *complete mediation* as a fundamental property for access control design and GenCyber highlights the importance of *domain separation* and *process isolation*. However there is no guideline on the granularity of isolation and mediation. The *granularity of mechanism* refers to the importance of selecting principals (or subjects) at the right granularity for isolation and mediation decisions.

For example, even though Android uses *complete mediation* this does not stop the attacks from advertising libraries we discussed on Chapter 4. This is because all mediation mechanisms control access at the process level. Nonetheless, advertising libraries (or in principle any third-party library) on smartphone applications run within the same process boundaries as their host apps. Thus any security mechanism enforced at the process level will not be able to control access to resources by third-party libraries. One could split the libraries from their host apps such that they run on their own processes something suggested by Shekhar et al. [17]. In this case, mediation at process boundaries would be enough. Nonetheless, the security mechanism should be deployed respecting the granularity of principals. This principle would further help design distributed systems. In Chapter 8 we saw how a router in a smart home could only enforce device-level policies. When smartphones are introduced in that environment this mechanism becomes insufficient. An authorized smartphone might carry unauthorized apps which might try to gain access to smart home devices in the network. The mechanism I introduced is based on the observation that access could happen at application-level granularity. A corollary derived from the *granularity of mechanism* principle is the *separation of origin* which could further guide the implementation of some security mechanisms.

• *Separation on origin*: Principals or subjects in an access control scheme could be selected based on their origins. An instantiation of this took place in web security with the *same-origin policy*. The same-origin policy tries to tackle untrusted code running on a trusted web page from accessing data on another web page. More recently Chromium (one of the most popular open source browsers) introduced *out-of-process iframes* (OOPIF) which allow a child frame of a web page to be rendered on a different process. The important security decision here was the identification of trusted principals not at the web page level but at the iframe level. Basically the focus should be to identify the lowest level where an origin can execute code. Similarly, smartphone operating systems would benefit from security mechanisms designed with principal isolation and mediation at lower than the process levels. For example, code could be indentified based on the package name of their Java class hierarchy. This would allow the design of mechanims to protect against third-party libraries who take advantage of their shared host app's privileges. Moreover, in Chapter 6 we discussed how we can utilize the same principle for preventing permission escalation attacks on Android. For example, custom permissions on Android which are declared by untrusted principals (identified by their unique app identifiers on Android) should be decoupled from system permissions which should only be defined by system principals. Even within the realm of custom permissions, a custom permission should be strictly associated with the principal that defined it. We have seen how this stops permission hijacking attacks.

• *A case study for Android*: Lets see how we can re-design security mechanisms given this principle. Modern processors leverage the notion of protection rings. Protection rings are basically successively less privileged domains with the data they can access. x86 supports 4 rings ranging from 0 (most privileged) to 3 (least privileged). Linux, only leverages ring 0 for the kernel and ring 3 for userspace apps. The problem with the implementation of this mechanism on Android, is that third-party libraries run in the same ring as the host apps. Android also leverages a permission model at its middleware which is essentially a capability-based access control scheme. Resources are protected by permissions and apps are granted permissions to access them. A reference monitor checks on such resource requests, whether the requestor app has the appropriate permission. However, these permissions are granted at the process granularity which means they can be inherited by an app's libraries. Lastly, SELinux on Android further separates apps/processes in domains to enforce a *type enforcement* MAC scheme. Even this domain separation does not help when the granularity of the mechanism is coarse-grained.

Bearing the *separation of origin* principle in mind, we could identify third-party libraries as principals. Once we do that, we can revisit the protection mechanism implementations. At the lowest level, we could utilize an extra protection ring for third-party libraries. Apps could run on ring 2, whereas libraries could run in the less privileged ring 3. Of course this would require certain support from the underlying architecture to utilize more rings. Inside that ring, principals should not have the ability to directly use system calls, access sensor drivers etc. At the middleware layer, Android would grant permissions differently to apps and their libraries. This would allow the system to restrict the permissions that can be granted to third-party libraries. For example, we could disallow libraries from using middleware APIs to access sensors, dynamically load code, run background processes etc. Furthermore, we could easily present the user with clear indications on whether a permission is requested or used by an app or an advertising library. Lastly at the SELinux type enforcement mechanism we could also further isolate resources from libraries.

Evidently, the granularity of mechanism is a powerful principle which has a direct impact on systems' security. The origin of principal can facilitate granularity decisions to set privilege boundaries.

Granularity of Mechanism: The granularity of a security mechanism (isolation, privilege assignment) should match the trust model.

9.3 LAYERED RESPONSIBILITY

This principle refers to placement of security mechanisms at the right layers of the system. Here layers could be layers in the software stack or privileged layers/domains. Lampson [251] discusses *defense in depth* and GenCyber uses *layering* to argue about the placement of **multiple** layers of defense. Here I provide arguments to reason about the **responsibility** of each layer of defense. Saltzer et al. [252] posed the end-to-end argument which provides reasoning against low-level function implementation in layered communication protocols. In a nutshell the paper advocates avoiding mandatory functions in lower layers, if layers above it have the necessary semantics to decide whether a function should be used. This avoids unnecessary performance penalties. The paper discusses the subtleties in making those decisions. *Layered responsibility* adds to that discourse focusing on security mechanisms.

While it might be tempting to move a security function in a layered system closer to the application that leverages that function, sometimes relying on higher layers introduces security problems. Let us consider the Android operating system as a layered architecture with the following layers from bottom (higher privilege) to top (lower privilege): the kernel; the middleware including system applications and; third-party applications. Note that the layers I consider are based on their privileges. You can also think of them as protection rings, or isolation domains. In most networking operations, relying only on the OS for security is insufficient: relying solely on the kernel (with its networking stacks) to authenticate the external resource (a web domain, a Bluetooth device, an NFC device, a WiFi device) raises mis-bonding issues (see Chapter 7). In particular, encryption integrated in the protocol can tackle network adversaries but remains vulnerable to internal adversaries in the form of third-party apps competing for access to the network channel. These apps have access to the information after it is being decrypted by the protocol. Moreover, authentication happens at the device level since the kernel does not have the necessary semantics to enforce app-level or user-level authentication. On the other hand, solely relying on third-party applications/developers for security is bad practice. We found that most Android apps which interact with external sources of information (SMSs, Bluetooth, NFC and Internet devices) suffer from lack or weak authentication (see Chapter 7), while others have found that security is commonly implemented incorrectly by third-party developers [231]. In the context of a smart home, we can think of an equivalent security architecture with the router being the most trusted device and third-party IoT devices in the network as untrusted. In that setting we found that most of the security vulnerabilities manifest again because of lack or weak authentication, or implementation flaws on third-party IoT devices [29]. In contrast, not only we should have security in multiple layers, but it is important to think about the responsibilities of each layer in the overall security of a system.

For example, the networking protocol could ensure device level authentication and encryption to guarantee secure inter-device communication. Most communication protocols already do or support that. However, the OS responsibility is to also guarantee secure access to resources, whether these are internal (filesystem resources) or external (devices). Thus the OS should have mechanisms to make sure that the expected application is utilizing a networking channel at any given point. Currenty, on Android, this is mediated through permissions and Linux group IDs (GIDs). Nonetheless, as I showed on Chapter 7, it does not guarantee that apps will utilize their permissions only to connect to external resources we expect them too; the permission provides access to the channel irrespective of what is on the other end. The OS responsibility does not stop at deciding who should access the networking channel as a whole. This might be true for files, which are mostly static and predictable resources. In contrast, networking sockets can be used to access any unpredictable external resource. *Since we expect the OS to manage resources, the OS should have mechanisms to guarantee correct multiplexing of such information to applications.* For example, a Bluetooth Fitbit should interact only with the Fitbit app. We discussed how the OS can leverage user-driven decision making to support such decisions, or enforce MAC rules in enterprise settings where admins have knowledge of the enterprise applications and resources. Lastly, authentication should be performed by the application, as it is expected to do so in current implementations. The applications have the necessarry semantics to decide whether the correct user is logged into the app/service, and also can authenticate the remote resource. For example, an app developer can decide whether two-factor authentication is mandated or not (a gaming app might not need it, a banking app will benefit from it); an app developer can decide whether the remote service is the expected one.

It is also tempting to move security operations in more privileged domains of responsibility. ARMTrustZone and Intel SGX are technologies which support trusted execution environments (TEE) in systems. Programs and information in TEEs are strongly (harware) isolated even from the kernel of the OS. Nonetheless, logic in TEEs should remain as simple and small as possible (*economy of mechanism*). The security operations in the TEE should match its responsibility. For example, we could ensure root-of-trust and enable remote attestation in the TEE but we should not expect it to perform application access control to resources.

Layered Responsibility: Respect each layer's responsibility when placing security mechanisms in a layered system.

CHAPTER 10: SUMMARY AND FUTURE DIRECTIONS

10.1 SUMMARY

In my work, I have performed a holistic security analysis on the security of modern smartphone operating systems. My analysis used Android as a usecase which is the most popular smartphone operating system ($>$ 80% of smartphone OS marketshare). Smartphones, are equipped with advanced sensors and networking capabilities. Their always-on and always-with-us nature renders them the de facto devices users utilize to perform a vast array of daily tasks, from socializing, to performing financial transactions and mangement, medical condition tracking, navigation etc. To maximize utility, modern smartphone operating systems rely on a multi-process architecture which supports concurrent execution of userspace programs from a variety of sources. This allows third-party developers to create mobile applications which can utilize a number of resources made available to them by the smartphone operating system to offer creative services. This creates a multi-tenancy issue on smartphones where third-party applications compete for access to resources. In my work I analyzed how shared resources in this environments, can be used by third-party tenants in unexpected ways, to compromise the confidentiality of user and other tenant data on smartphones. This understanding allows us to develop more accurate adversary models for smartphone operating systems. I used this understanding to design effective and efficient mechanisms for information leakage detection and prevention. My solutions are tested on real smartphone applications, operating systems and devices.

Figure 1.1 illustrates the shared resources studied in my work. The Figure is redrawn here for readability (Figure 10.2). In particular this thesis analyzed the security of the following shared resources.

- Intra-process privileges
- Filesystem resources
- System and Application Resources
- Communication Channels
- Smart External Devices

I have proven that shared resources can be the source of vulnerabilities on smartphone operating systems. I have shown that existing isolation and security mechanisms are either

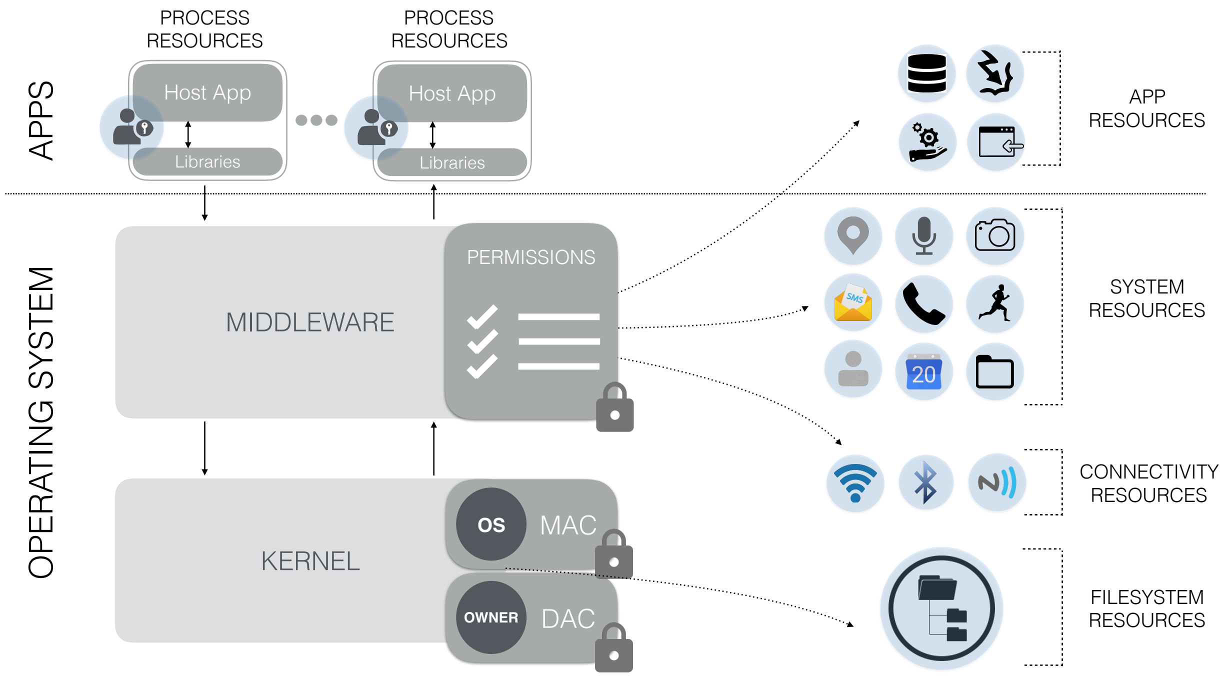


Figure 10.1: Smartphone shared resources.

too coarse-grained or merely flawed, and as such are insufficient to guarantee the confidentiality of user or application data. Unfortunately, current smartphone operating systems focus mostly on isolating third-party applications as a whole from the system. This is important for the integrity of the device. However, this treatment overlooks the severe private information leakage across third-party apps, made possible through shared resources. My work shows how by including this in our threat models can help us design more effective systems for detection and prevention of such leakage.

My designs are shown to be efficient and effective. More importantly I took an approach which retrofits security into existing products. Figure 10.2 illustrates the main security enhancements my work introduces to smartphone operating systems. For example, in the case of mobile advertising, it is unclear whether a prevention mechanism would be practical since the OS vendors have strong business incentives to enable advertising. In this case I showed how we can develop tools that can help users and app developers to discover how their information can be accessed by advertising libraries. In sharing filesystem resources I showed how an abstraction layer can help us release contents of files at different granularities as dictated by users through permissions. I also showed how by redesigning the permission model to separate management of custom and system permissions can prevent privilege escalation attacks. I have also showed how we can extend the use a hybrid mandatory

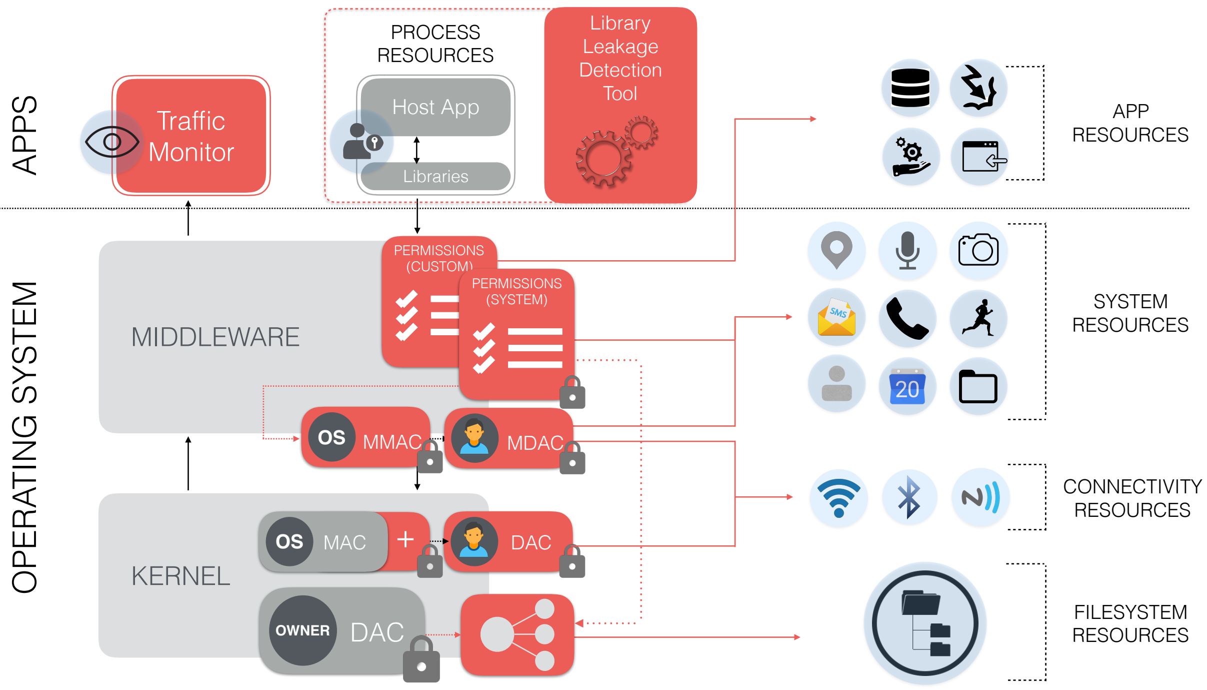


Figure 10.2: Proposed security additions on smartphone operating systems.

and discretionary access control scheme distributed across the middleware and the kernel, to offer fine-grained protection to external to the smarpthone OS resources. Moreover, I demonstrated how we can build userspace network traffic monitors which can can help IoT endpoints such as routers enforce fine-grained application level decisions on flows targetting IoT devices. Lastly, based on my security analysis on shared smartphone operating systems, I introduced three new needed principles to guide the design of secure systems (see Chapter 9): *contextual integrity*; *granularity of mechanism* and; *layered responsibility*.

10.2 CONTRIBUTIONS

Next I list the main contributions of this work.

• *Provides a systematic analysis of information reach of advertising libraries embedded in smartphone apps.* Previous work has focused on past and current behaviors of advertising libraries, overlooking the fact that these behaviors can change opportunistically. This thesis focuses on the fact that such libraries share process space and privileges with their hosts and as such can eventually take advantage of those privileges. It systematically models all shared privileges a library has with its hosts which leverages for the design of an automatic open-source tool for estimating the risk of sensitive user information exposure by a host app to its advertising library.

• *Discovers new side-channels hidden in shared filesystem resources and demonstrates new adversarial inference techniques.* An analysis of Android shared filesystem resources led to the discovery of new side-channels which can be exploited by malicious applications with a suite of new inference techniques to bypass the process isolation boundaries and infer a user's identity, medical condition and financial preferences. This work led to Google introducing further restrictions on Android filesystem resource access by third-party apps.

• *Redesigns the Android Runtime Permission Model to tackle attacks on shared system and application components.* This thesis introduces a redesign of the Android permission model which (a) separates management of system-defined and third-party-defined permissions and (b) tracks ownership of custom permissions. This solves a perennial problem on Android where vulnerabilities kept arising because of this non separation of trust between permissions. The proposed model is backward compatible and formally verified to be correct with respect to fundamental security properties.

• *Unearths threats on Android's communication with external resources.* This work systematically studies Android's shared channels of communication with external resources such as Bluetooth and NFC devices, devices that connect through the Audio port, incoming SMSs and, WiFi smart-home devices. It defines a new threat, called the device mis-bonding (DMB) problem, to highlight the system's incapacity to create application-level bonds. It further demonstrates that Android's system permissions are too coarse-grained to support the utility of the apps while guaranteeing the confidentiality and intergrity of the data communicated through these channels. Furthermore it measures the prevalence of the problem in the Android ecosystem.

• *Introduces smartphone OS-level enhancements to safeguard the communication with Android external resources, using both MAC and DAC.* This is the first mechanism that provides comprehensive protection of different kinds of Android external resources over their channels in a uniform way. The enhancements are built on top of SELinux on Android and achieve both MAC and DAC in an integrated, highly efficient way, without undermining their security guarantees. These new techniques help both system administrators and ordinary Android users to specify their policies and safeguard their accessories and other external resources.

• *Introduces a novel distributed application-level access control system to safeguard vulnerable shared smart-home devices from malicious smartphone apps.* It shows how smartphones can collaborate with enforcing points in smart environments to enable fine-grained access control for WiFi smart-home devices. The design focuses on OS-level enahancement at the enforcing point (the router) which makes device-level decisions and utilizes trusted applications on smartphones for application-level decisions. The trusted applications utilize novel traffic monitoring techniques while the overall solution is independent from IoT device manufacturers.

• *Impact on real-world smartphone operating systems.* Threats revealed in this work were acknowledged by Google, who overhauls the development of Android (the most popular smartphone operating system) which is used by millions of users. Google introduced security enhancements to Android to address these issues.

• *New design principles for the secure systems.* Proposed three new principles for the design of secure systems.

10.3 MOVING FORWARD

Shared resources can be the source of security issues in every multi-tenant environment (e.g. cloud, smartphones). As devices become more interconnected, this creates a complicated system of systems, where devices compete for and share not only networking, but also physical, event and other resources.

For example, IoT devices are now increasingly controlled through new user interaction modalities such as voice and gestures. In those environments, the control signals are communicated through the physical channel which is essentially shared between all devices and humans in proximity. This is problematic since these signals can be spoofed by a proximity adversary. An interesting direction would be the study of how these physical channels can be exploited and how we can build better security mechanisms to mitigate such threats.

In addition, a lot of effort is going into enabling smart cities and connected vehicles. The vision is to enable next generation vehicles to communicate with each other and the infrastructure to improve navigation, positioning, awareness and optimize traffic scheduling among others. In this ecosystem, connected vehicles will share massive amounts of media streams or outcomes based on those with each other and the infrastructure. Also the infrastructure would offer services to connected vehicles based on crowdsourced and historic data. Sharing these rich sensing streams and historic information, come with grave privacy concerns. An interesting direction would be to study how an adversary can utilize such shared data to fingerprint and profile individuals. From the defense perspective we need new technologies which can enable such next generation services in a privacy preserving manner without destroying utility.

REFERENCES

[1] I. W. M. P. Tracker, "Apple Cedes Market Share in Smartphone Operating System Market as Android Surges and Windows Phone Gains, According to IDC," http://www.idc.com/getdoc.jsp?containerId=prUS24257413, accessed: 10/07/2014.

[2] statista.com, "Number of available applications in the Google Play Store from December 2009 to December 2017," https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, accessed: 03/17/2018.

[3] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *WiSec*, 2012.

[4] A. Srivastava, P. Jain, S. Demetriou, L. Cox, and K.-H. Kim, "Camforensics: Understanding visual privacy leaks in the wild," in *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2017.

[5] "Square up," https://squareup.com/, accessed: 10/07/2014.

[6] M. Honorof, "79http://www.tomsguide.com/us/mobile-malware-targets-android,news-17452.html, accessed: 10/07/2014.

[7] 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.

[8] 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.

[9] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, Feb. 2012. [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/pubs/NDSS12_WOODPECKER.pdf

[10] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. Gunter, "Free for all! assessing user data exposure to advertising libraries on android," in *Network and Distributed System Security (NDSS) Symposium*, 2016.

[11] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516661 pp. 1017–1028.

[12] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter, "Resolving the predicament of android custom permissions," in *Network and Distributed System Security (NDSS) Symposium*, 2018.

[13] 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," in *Network and Distributed System Security (NDSS) Symposium*, 2014.

[14] S. Demetriou, X. Zhou, M. Naveed, Y. Lee, K. Yuan, X. Wang, and C. A. Gunter, "What's in your dongle and bank account? mandatory and discretionary protection of android external resources," in *Network and Distributed System Security (NDSS) Symposium*, 2015.

[15] S. Demetriou, N. Zhang, Y. Lee, X. Wang, C. Gunter, X. Zhou, and M. Grace, "Hanguard: Sdn-driven protection of wifi smart-home devices from malicious mobile apps," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[16] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly, "Powerspy: Location tracking using mobile device power analysis." in *USENIX Security Symposium*, 2015, pp. 785–800.

[17] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating smartphone advertising from applications," in *USENIX Security*, 2012.

[18] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *TRUST*, 2012.

[19] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in Android applications," in *SAC*, 2012.

[20] 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," in *PLDI*, 2014.

[21] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: driving apps to test the security of third-party components," in *USENIX Security*, 2014.

[22] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in Android ad libraries," in *MoST*, 2012.

[23] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in android for privacy leakage detection," in *CCS*, 2013.

[24] L. Vigneri, J. Chandrashekar, I. Pefkianakis, and O. Heen, "Taming the Android appstore: Lightweight characterization of Android applications," *arXiv.org*, 2015.

[25] 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 *OSDI*, 2010.

[26] V. Moonsamy, M. Alazab, and L. Batten, "Towards an understanding of the impact of advertising on data leaks," *IJSN*, vol. 7, no. 3, 2012.

[27] 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.

[28] "Jawbone official website," https://jawbone.com/up, accessed: 2014-05-13.

[29] N. Zhang, S. Demetriou, X. Mi, W. Diao, K. Yuan, P. Zong, F. Qian, X. Wang, K. Chen, Y. Tian et al., "Understanding iot security through the data crystal ball: Where we are now and where we are going to be," *arXiv preprint arXiv:1703.09809*, 2017.

[30] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, 2015.

[31] "Wikipedia android version history," http://en.wikipedia.org/wiki/Android_version_history, accessed: 10/07/2014.

[32] "Wikipedia android (operating system)," http://en.wikipedia.org/wiki/Android_%28operating_system%29, accessed: 10/07/2014.

[33] "Android Developers Official Website activity," http://developer.android.com/reference/android/app/Activity.html, accessed: 10/07/2014.

[34] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wijesekera pp. 499–514.

[35] "Android permissions." https://tinyurl.com/y863owbb.

[36] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android," in *20th Annual Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[37] G. A. Miller, "Wordnet: a lexical database for english," *CACM*, vol. 38, no. 11, 1995.

[38] C. Leacock and M. Chodorow, "Combining local context and wordnet similarity for word sense identification," *WordNet: An electronic lexical database*, 1998.

[39] "Alloy: A language and tool for relational models." http://alloy.mit.edu.

[40] S. Nath, "MAdScope: Characterizing mobile in-app targeted ads," in *Mobisys*, 2015.

[41] "Google play," https://play.google.com/store/search?q=traffic+monitor&c=apps, 2012, accessed: 10/07/2014.

[42] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.

[43] S. Stein, "Withings wireless blood pressure monitor supports android/ios, now available," http://www.cnet.com/news/withings-wireless-blood-pressure-monitor-supports-androidios-now-available, 2014, accessed: 10/07/2014.

[44] N. Wanchoo, "Fda approves mega electronic's android-based emotion ecg mobile monitor," http://www.medgadget.com/2013/12/mega-electronics-gets-fda-approval-for-android-based-ecg-monitor.html, 2013, accessed: 10/07/2014.

[45] J. R. W. J. Joseph Tran, Rosanna Tran, "Smartphone-based glucose monitors and applications in the management of diabetes: An overview of 10 salient "apps" and a novel smartphone-connected blood glucose monitor," http://clinical.diabetesjournals.org/content/30/4/173.full, 2012, accessed: 10/07/2014.

[46] "Viper official website," http://www.viper.com/SmartStart/, accessed: 10/07/2014.

[47] K. Voss, "Top 10 phone apps for home security," http://www.securityoptions.com/top-10-apps-for-home-security-systems/, 2014, accessed: 10/07/2014.

[48] nest.com, "Nest thermostat," https://goo.gl/oSIFfQ, 2017.

[49] honeywell.com, "Honeywell," http://goo.gl/9yuiTX, 2017.

[50] belkin.com, "Belkin netcam," http://goo.gl/60dfkg, 2017.

[51] ibabylabs.com, "ibabylabs.com," https://goo.gl/y6Gdzd, 2017.

[52] samsung.com, "Samsung family hub refrigerator," http://goo.gl/ddwlxb, 2017.

[53] I. Ullah, R. Boreli, M. A. Kaafar, and S. S. Kanhere, "Characterising user targeting for in-app mobile ads," in *INFOCOM WKSHPS*, 2014.

[54] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in Android," in *ASIACCS*, 2012.

[55] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, "Don't kill my ads!: balancing privacy in an ad-supported mobile application market," in *HotMobile*, 2012.

[56] H. Haddadi, P. Hui, and I. Brown, "Mobiad: private and scalable mobile advertising," in *MobiArch*, 2010.

[57] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, “Supor: Precise and scalable sensitive user input detection for android apps,” in *USENIX Security*, 2015.

[58] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, “Uipicker: User-input privacy identification in mobile applications,” in *USENIX Security*, 2015.

[59] 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 CCS*. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687 pp. 199–212.

[60] 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.

[61] 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

[62] K. Zhang and X. Wang, “Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems,” *analysis*, vol. 20, p. 23, 2010.

[63] 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.

[64] S. Chen, R. Wang, X. Wang, and K. Zhang, “Side-channel leaks in web applications: A reality today, a challenge tomorrow,” in *Security and Privacy (SP), 2010 IEEE Symposium on*, may 2010, pp. 191 –206.

[65] P. Brodley and leviathan Security Group, “Zero Permission Android Applications,” http://leviathansecurity.com/blog/archives/17-Zero-Permission-Android-Applications.html, accessed: 13/02/2013.

[66] 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*. The Internet Society, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#SchlegelZZIKW11

[67] 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.

[68] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "Accessory: password inference using accelerometers on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems Applications*, ser. HotMobile '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2162081.2162095 pp. 9:1–9:6.

[69] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30921-2_16 pp. 273–290.

[70] J. Han, E. Owusu, T.-L. Nguyen, A. Perrig, and J. Zhang, "Accomplice: Location inference using accelerometers on smartphones," in *Proceedings of the 4th International Conference on Communication Systems and Networks*, Bangalore, India, 2012.

[71] 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.

[72] 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 CCS*, ser. CCS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046780 pp. 639–652.

[73] 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.

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

[75] A. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: and defenses." in *USENIX Security*, 2011.

[76] E. Chin, A. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *MobiSys*, 2011.

[77] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *ACSAC*, 2012.

[78] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "Detection of design flaws in the android permission protocol through bounded verification," in *International Symposium on Formal Methods*, 2015.

[79] G. Betarte, J. Campo, C. Luna, and A. Romano, "Verifying android's permission model," in *Theoretical Aspects of Computing*, 2015.

[80] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A formal model to analyze the permission authorization and enforcement in the android framework," in *SocialCom*, 2010.

[81] Y. Zhauniarovich and O. Gadyatskaya, "Small changes, big changes: an updated view on the android permission system," in *RAID*, 2016.

[82] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. Wang, Z. Qian, and H. Chen, "revdroid: code analysis of the side effects after dynamic permission revocation of android apps," in *Asia CCS*, 2016.

[83] D. Bogdanas, N. Nelson, and D. Dig, "Analysis and transformations in support of android privacy," Tech. Rep., 2016.

[84] "Custom permission vulnerabilities." https://tinyurl.com/y7yoae52.

[85] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A small but non-negligible flaw in the android permission scheme," in *POLICY*, 2010.

[86] J. Sellwood and J. Crampton, "Sleeping android: The danger of dormant permissions," in *SPSM*, 2013.

[87] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2009.39 pp. 340–349.

[88] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing android-powered mobile devices using selinux," *IEEE Security & Privacy*, vol. 8, no. 3, pp. 36–44, 2010.

[89] M. Ongtang, K. Butler, and P. McDaniel, "Porscha: policy oriented secure content handling in android," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920295 pp. 221–230.

[90] M. Conti, V. T. N. Nguyen, and B. Crispo, "Crepe: context-related policy enforcement for android," in *Proceedings of the 13th international conference on Information security*, ser. ISC'10. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949317.1949355 pp. 331–345.

[91] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920313 pp. 347–356.

[92] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[93] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1755688.1755732 pp. 328–332.

[94] M. Rahman, B. Carbunar, and M. Banik, "Fit and vulnerable: Attacks and defenses for a health monitoring device," *CoRR*, vol. abs/1304.5672, 2013.

[95] C. Li, A. Raghunathan, and N. K. Jha, "Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system," in *e-Health Networking Applications and Services (Healthcom)*, 2011. [Online]. Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6026732&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D6026732 pp. 150 – 156.

[96] R. Marti, J. Delgado, and X. Perramons, "Security specification and implementation for mobile e-health services," *eee*, vol. 00, pp. 241–248, 2004.

[97] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on android." ser. IEEE Symposium on Security and Privacy, 2015.

[98] M. Stanislav and T. Beardsley, "Hacking iot: A case study on baby monitor exposures and vulnerabilities," https://goo.gl/Uh7y4e, 2015.

[99] S. Notra, M. Siddiqi, H. H. Gharakheili, V. Sivaraman, and R. Boreli, "An Experimental Study of Security and Privacy Risks with Emerging Household Appliances," ser. M2MSec '14, 2014.

[100] V. Sivaraman, H. H. Gharakheili, A. Vishwanath, R. Boreli, and O. Mehani, "Network-level security and privacy control for smart-home iot devices," ser. WiMob '15, 2015.

[101] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *IEEE Symposium on Security and Privacy*, 2016.

[102] V. Sivaraman, D. Chan, D. Earl, and R. Boreli, "Smart-phones attacking smart-homes," ser. WiSec '16, 2016.

[103] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou, "Antmonitor: A system for monitoring from mobile devices," ser. SIGCOMM '15, 2015.

[104] C. Gomez and J. Paradells, "Wireless home automation networks: A survey of architectures and technologies," *Communications Magazine, IEEE*, 2010.

[105] B. Ur, J. Jung, and S. Schechter, “The current state of access control for smart devices in homes,” ser. HUPS ’13, 2013.

[106] T. Denning, T. Kohno, and H. M. Levy, “Computer security and the modern home,” *Commun. ACM*, 2013.

[107] G.-J. Ahn, H. Hu, and J. Jin, “Towards role-based authorization for osgi service environments,” ser. FTDCS ’08, 2008.

[108] S. Das, S. Chita, N. Peterson, B. Shirazi, and M. Bhadkamkar, “Home automation and security for mobile devices,” ser. PERCOM Workshops ’11, 2011.

[109] J. E. Kim, G. Boulos, J. Yackovich, T. Barth, C. Beckel, and D. Mosse, “Seamless integration of heterogeneous devices and access control in smart homes,” ser. IE ’12, 2012.

[110] A. Lioy, A. Pastor, F. Risso, R. Sassu, and A. Shaw, “Offloading security applications into the network,” ser. eChallenges e-2014, 2014.

[111] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, “Flowfence: Practical data protection for emerging iot application frameworks,” in *USENIX Security Symposium*, 2016.

[112] T. H.-J. Kim, L. Bauer, J. Newsome, A. Perrig, and J. Walker, “Challenges in access right assignment for secure home networks,” ser. HotSec’10, 2010.

[113] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter, “Access control for home data sharing: Attitudes, needs and practices,” ser. CHI ’10, 2010.

[114] S. Smalley and R. Craig, “Security enhanced (se) android: Bringing flexible mac to android.” ser. NDSS ’13, 2013.

[115] S. Bugiel, S. Heuser, and A.-R. Sadeghi, “Flexible and fine-grained mandatory access control on android for diverse security and privacy policies,” ser. USENIX Security 13, 2013.

[116] S. Demetriou, X. Zhou, M. Naveed, Y. Lee, K. Yuan, X. Wang, and C. A. Gunter, “What’s in your dongle and bank account? mandatory and discretionary protection of android external resources.” ser. NDSS ’15, 2015.

[117] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, “Traffic classification through simple statistical fingerprinting,” *SIGCOMM Comput. Commun. Rev.*, 2007.

[118] P. Judge and M. Ammar, “Gothic: a group access control architecture for secure multicast and anycast,” ser. INFOCOM ’02, 2002.

[119] G. Appenzeller, M. Roussopoulos, and M. Baker, "User-friendly access control for public network ports," ser. INFOCOM '99, 1999.

[120] W. R. Cheswick and S. M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*, 1994.

[121] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *TRUST*, 2011.

[122] C. E. Wills and C. Tatar, "Understanding what they do with what they know," in *WPES*, 2012.

[123] P. Barford, I. Canadi, D. Krushevskaja, Q. Ma, and S. Muthukrishnan, "Adscape: Harvesting and analyzing online display ads," in *WWW*, 2014.

[124] S. Kaplan and B. J. Garrick, "On the quantitative definition of risk," *Risk Analysis*, 1981.

[125] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu, "Riskmon: Continuous and automated risk assessment of mobile applications," in *CODASPY*, 2014.

[126] Y. Jing, G. Ahn, Z. Zhao, and H. Hu, "Towards automated risk assessment and mitigation of mobile application," *Dependable and Secure Computing, IEEE Transactions on*, 2014.

[127] E. Steel, C. Locke, E. Cadman, and B. Freese, "How much is your personal data worth?" *Financial Times*, June 2013. [Online]. Available: \url{http://www.ft.com/intl/cms/s/2/927ca86e-d29b-11e2-88ed-00144feab7de.html#axzz3Zqb8gG3Y}

[128] L. Sweeney, "K-anonymity: A model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 2002.

[129] C. Duhigg, "How companies learn your secrets," *The New York Times*, vol. 16, 2012. [Online]. Available: \url{http://128.59.177.251/twiki/pub/CompPrivConst/HowCompaniesLearnOurConsumingSecrets/How_Companies_Learn_Your_Secrets_-_NYTimes.com.pdf}

[130] Twitter, "What is app graph in Twitter?" https://goo.gl/scmc69, Observed in May 2015.

[131] S. Dredge, "Twitter scanning users' other apps to help deliver 'tailored content'," *The Guardian*, November. [Online]. Available: \url{http://www.theguardian.com/technology/2014/nov/27/twitter-scanning-other-apps-tailored-content}

[132] J. Turow, *The daily you: How the new advertising industry is defining your identity and your worth.* Yale University Press, 2012.

[133] M. D. Buhrmester, T. Kwang, and S. D. Gosling, "Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality data?" *Perspectives on Psychological Science*, 2011.

[134] developer.android.com, "Ui/application exerciser monkey," http://goo.gl/cH9wPR, Observed in May 2015.

[135] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *ACM ASE*, 2012.

[136] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Mobisys*, 2014.

[137] M. A. Finlayson, "Java libraries for accessing the princeton wordnet: Comparison and evaluation," in *GWC*, 2014.

[138] R. Rada, H. Mili, E. Bicknell, and M. Blettner, "Development and application of a metric on semantic nets," *IEEE SMC*, 1989.

[139] D. Lin, "An information-theoretic definition of similarity." in *ICML*, 1998.

[140] S. Banerjee and T. Pedersen, "An adapted lesk algorithm for word sense disambiguation using wordnet," in *CICLing*, 2002.

[141] Z. Wu and M. Palmer, "Verbs semantics and lexical selection," in *ACL*, 1994.

[142] J. J. Jiang and D. W. Conrath, "Semantic similarity based on corpus statistics and lexical taxonomy," *arXiv.org*, 1997.

[143] P. Resnik, "Using information content to evaluate semantic similarity in a taxonomy," *arXiv.org*, 1995.

[144] G. Hirst and D. St-Onge, "Lexical chains as representations of context for the detection and correction of malapropisms," *WordNet: An electronic lexical database*, 1998.

[145] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *SIGMOD*, 2000.

[146] R. Agrawal, R. Srikant et al., "Fast algorithms for mining association rules," in *VLDB*, 1994.

[147] J. Carroll and T. Briscoe, "High precision extraction of grammatical relations," in *COLING*, 2002.

[148] D. Smith, "businessinsider.com," http://goo.gl/LNn0pi, Observed in May 2015.

[149] P. Sawers, "thenextweb.com," http://goo.gl/8g34xB, Observed in May 2015.

[150] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, “Do we need hundreds of classifiers to solve real world classification problems?” *J. Mach. Learn. Res.*, Jan. 2014.

[151] “Fda.gov,” http://goo.gl/guSsMM, accessed: 2015-01-05.

[152] “Shark for root,” https://play.google.com/store/apps/details?id=lv.n3o.shark&hl=en, 2012, accessed: 10/07/2014.

[153] “Antutu benchmark,” https://play.google.com/store/apps/details?id=com.antutu.ABenchMark, 2013, accessed: 10/07/2014.

[154] “Google play: Webmd for android,” http://www.webmd.com/webmdapp, 2012, accessed: 10/07/2014.

[155] D. J. Solove, “Identity Theft, Privacy, and the Architecture of Vulnerability,” *Hastings Law Journal*, vol. 54, pp. 1227 – 1276, 2002-2003.

[156] 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.

[157] H. Berghel, “Identity theft, social security numbers, and the web,” *Commun. ACM*, vol. 43, no. 2, pp. 17–21, Feb. 2000. [Online]. Available: http://doi.acm.org/10.1145/328236.328114

[158] S. B. Hoar, “Identity Theft: The Crime of the New Millennium,” *Oregon Law Review*, vol. 80, pp. 1423–1448, 2001.

[159] T. Govani and H. Pashley, “Student awareness of the privacy implications when using facebook,” *unpublished paper presented at the ”Privacy Poster Fair” at the Carnegie Mellon University School of Library and Information Science*, vol. 9, 2005.

[160] “Get search, twitter api,” https://dev.twitter.com/docs/api/1/get/search, 2012, accessed: 10/07/2014.

[161] “Lookup ip address location,” http://whatismyipaddress.com/ip-lookup, 2013, accessed: 10/07/2014.

[162] “Wifi coverage map,” http://www.navizon.com/navizon_coverage_wifi.htm, accessed: 13/02/2013.

[163] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu, “Statistical identification of encrypted web browsing traffic,” in *IEEE Symposium on Security and Privacy.* Society Press, 2002.

[164] “Apktool decompiler,” http://ibotpeaches.github.io/Apktool/.

[165] "Jd-gui." http://jd.benow.ca/.

[166] "Dex2jar." https://github.com/pxb1988/dex2jar.

[167] "Privilege escalation through custom permission update," https://issuetracker.google.com/issues/37130844.

[168] "Google cloud messaging," https://tinyurl.com/ybocrrqw.

[169] "Upload applications to appaloosa," https://tinyurl.com/y94pb3cv.

[170] "Privilege escalation by exploiting fcfs property of custom permissions," https://issuetracker.google.com/issues/37131935.

[171] "Creating apps with plugin architecture," https://tinyurl.com/ydfdk9z7.

[172] "Android plugin application," https://tinyurl.com/ycfd9pot.

[173] "Yoga guru," https://tinyurl.com/yb3dqopp.

[174] "Android users have an average of 95 apps installed on their phones, according to yahoo aviate data," https://tinyurl.com/ybc7dqbn.

[175] "Program correctness, the specification," https://tinyurl.com/y8r8cze8.

[176] K. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming AI and Reasoning*, 2010.

[177] "Nonin onyx ii pulseoximeter specs," http://www.nonin.com/products.asp?ID=39&sec=2&sub=9, accessed: 10/07/2014.

[178] "Bodymedia link armband," http://www.bodymedia.com/, accessed: 10/07/2014.

[179] "Bodymedia puts a spin on the ordinary testing procedure," http://blog.bodymedia.com/page/6/, 2012, accessed: 10/07/2014.

[180] "Nonin onyx ii pulseoximeter," http://www.nonin.com/PulseOximetry/Finger/Onyx9560, accessed: 10/07/2014.

[181] "Foracare testngo," http://www.myglucohealth.net/, accessed: 10/07/2014.

[182] C. Daniels, "Why is too much insulin bad?" http://www.livestrong.com/article/423665-why-is-too-much-insulin-bad/, accessed: 10/07/2014.

[183] "ithermometer," http://www.ithermometer.info/, accessed: 10/07/2014.

[184] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, 2010, pp. 105–114.

[185] "hcidump," http://www.linuxcommand.org/man_pages/hcidump8.html, accessed: 10/07/2014.

[186] "Java cryptography extension," http://www.oracle.com/technetwork/java/javase/documentation/index.html, accessed: 10/07/2014.

[187] "Bouncycastle library," http://www.bouncycastle.org/, accessed: 10/07/2014.

[188] "Spongycastle library," http://rtyley.github.io/spongycastle/, accessed: 10/07/2014.

[189] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets," 2014.

[190] "SEACAT demos website," https://sites.google.com/site/seacatchannelcontrol/, accessed: 10/07/2014.

[191] "Square Security official website," https://squareup.com/security, accessed: 10/07/2014.

[192] B. Dwyer, "Paypal here vs. square," http://www.cardfellow.com/blog/paypal-here-vs-square/, accessed: 10/07/2014.

[193] Gartner, "Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015," http://goo.gl/L9ubfl, 2015.

[194] nest.com, "Nest protect," https://goo.gl/jM8ALk, 2017.

[195] belkin.com, "Wemo switch + motion," https://goo.gl/sjUsi3, 2017.

[196] I. Kelly, "Hacking the wemo wifi switch part 1," https://goo.gl/PKeO1A, 2012.

[197] M. Noble, "Wemo hacking," http://goo.gl/C97vKv, 2013.

[198] M. Smith, "500,000 belkin wemo users could be hacked; cert issues advisory," http://goo.gl/HBN9HB, 2014.

[199] M. Smith, "Eavesdropping made easy: Remote spying with wemo baby and an iphone," http://goo.gl/OUxdUy, 2013.

[200] securityfocus.com, "Belkin wifi netcam video stream backdoor with unchangeable admin/admin credentials," http://goo.gl/XnmwAk, 2013.

[201] D. Storm, "Eerie music coming from wireless baby cam; is it a haunting? no, it's a hacker," http://goo.gl/49Larp, 2015.

[202] D. Storm, "2 more wireless baby monitors hacked: Hackers remotely spied on babies and parents," http://goo.gl/UIbWvA, 2015.

[203] D. Goodin, "Welcome to the"internet of things" where even lights aren't hacker safe2 more wireless baby monitors hacked: Hackers remotely spied on babies and parents," http://goo.gl/l0qh05, 2013.

[204] "Internet of things research study," Hewlett-Packard Enterprise, Tech. Rep., 2015.

[205] S. Machlis, "Iot's dark side: Hundreds of unsecured devices open to attack," http://goo.gl/pM9TNk, 2015.

[206] S. Shekyan and A. Hartutyunyan, "Watching the watchers:hacking wireless ip security cameras," in *HITB*, 2013.

[207] I. Kim, "Is cctv a spy? backdoor that was secretly hidden in chinese products were found," http://goo.gl/3xQ7Dy, 2015.

[208] t. Greene, "Spike malware toolkit can infect windows, linux and arm-based linux devices," http://goo.gl/KQgSyT, 2014.

[209] K. Hill, "This guy's light bulb performed a dos attack on his entire smart house," http://goo.gl/24skXK, 2015.

[210] yahoo.com, "Proofpoint uncovers internet of things (iot) cyberattack," http://goo.gl/GBqies, 2014.

[211] P. Paganini, "Internet of things - symantec has discovered a new linux worm," http://goo.gl/DwPGnM, 2013.

[212] J. Poushter, "Smartphone ownership and internet usage continues to climb in emerging economies," *Pew Research Center: Global Attitudes & Trends*, 2016.

[213] statista.com, "Number of apps available in leading app stores as of june 2016," http://goo.gl/LO6umz, 2016.

[214] securityintelligence.com, "2015 mobile threat report - the rise of mobile malware," https://goo.gl/lhZ1yb, 2015.

[215] mcafee.com, "Mobile malware -the rise continues," http://goo.gl/3f1LP8, 2013.

[216] S. M. Poremba, "Studies show rise of the mobile malware threat," http://goo.gl/VfUKB4, 2016.

[217] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*, 2014.

[218] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection," ser. MobiSys '12, 2012.

[219] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," ser. CODASPY '12, 2012.

[220] D. Palmer, "This android malware has infected 85 million devices and makes its creators 300,000 a month," http://goo.gl/4YbaWg, 2016.

[221] indianexpress.com, “Android malware ’godless’ has affected over 8.5 lakh devices globally,” http://goo.gl/RE5ffK, 2016.

[222] H. Solomon, “Mobile malware, unpatched android devices are increasing problems say studies,” http://goo.gl/EUGmDC, 2016.

[223] “Hanguard demo website,” https://goo.gl/dfYeop, 2017.

[224] darkreading.com, “Firms slow to secure flaws in embedded devices,” http://goo.gl/b7Cltt, 2011.

[225] J. Saarinen, “Vendors slow to patch openssl vulnerabilities,” http://goo.gl/9EFXAT, 2014.

[226] microsoft.com, “Azure iot hub - microsoft azure,” https://goo.gl/RYZTGC, 2017.

[227] google.com, “Onhub - google,” https://goo.gl/igIM5c, 2017.

[228] IDC, “Idc: Smartphone os market share,” http://goo.gl/y1uN4Q, 2016.

[229] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, “A large-scale analysis of the security of embedded firmwares,” in *USENIX Security*, 2014.

[230] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, “Oauth demystified for mobile application developers,” in *CCS*, 2014.

[231] F. Wei, S. Roy, X. Ou et al., “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps,” in *CCS*, 2014.

[232] R. A. Fisher, “On the interpretation of χ 2 from contingency tables, and the calculation of p,” *Journal of the Royal Statistical Society*, 1922.

[233] J. M. Carlson, D. Heckerman, and G. Shani, “Estimating false discovery rates for contingency tables,” *Microsoft Res*, 2009.

[234] belkin.com, “Wemo insight switch,” http://goo.gl/0WGDFe, 2017.

[235] myn3rd.com, “My n3rd: Connect and control anything from anywhere,” http://goo.gl/8gpa0D, 2017.

[236] android.com, “Vpnservice-android developers,” http://goo.gl/0cKFyO, 2017.

[237] qualcomm.com, “Trepn power profiler,” https://goo.gl/KGrswV, 2017.

[238] apple.com., “Instruments: ios performance analysis tool.” https://goo.gl/6XnAXF, 2017.

[239] apple.com, “Xcode: Apple’s ide,” https://goo.gl/TgMco6, 2017.

[240] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun, “Smartphones as practical and secure location verification tokens for payments,” in *NDSS ’14*, 2014.

[241] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, “Trustdump: Reliable memory acquisition on smartphones,” ser. ESORICS ’14, 2014.

[242] J. Winter, “Trusted computing building blocks for embedded linux-based ARM trustzone platforms,” in *STC ’08*, 2008.

[243] “ARM Security Technology,” ARM Limited, Tech. Rep., 2008.

[244] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, “Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world,” ser. CCS ’14, 2014.

[245] “iOS Security,” Apple Inc., Tech. Rep., 2015.

[246] S. Smalley and R. Craig, “Security enhanced (SE) android: Bringing flexible MAC to android,” ser. NDSS ’13, 2013.

[247] statista.com, “Android version market share distribution among smartphone owners as of september 2016,” http://goo.gl/vMm2t2, 2017.

[248] “ARM Strategic Report,” ARM Limited, Tech. Rep., 2015.

[249] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[250] B. R. Payne, T. Abegaz, and K. Antonia, “Planning and implementing a successful nsa-nsf gencyber summer cyber academy,” *Journal of Cybersecurity Education, Research and Practice*, vol. 2016, no. 2, p. 3, 2016.

[251] B. W. Lampson, “Computer security in the real world,” *Computer*, vol. 37, no. 6, pp. 37–46, 2004.

[252] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.