© 2014 Soteris Demetriou

ANDROID AT RISK: CURRENT THREATS STEMMING FROM UNPROTECTED LOCAL AND EXTERNAL RESOURCES

BY

SOTERIS DEMETRIOU

THESIS

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

Urbana, Illinois

Adviser:

Professor Carl A. Gunter

ABSTRACT

Android is an open source platform derived from Linux OS. It utilizes a plethora of resources both local and external. Most of its local resources (e.g procfs nodes) were inherited from Linux with some of them being eventually removed, while new ones were added to meet the requirements of a mobile multi-purpose platform. Moreover, such a platform compels the introduction of external resources which can be used in tandem with a variety of sensors (e.g Bluetooth and NFC) that the device is equipped with. This thesis demonstrates the subtlety involved in this adaptation which, if not performed correctly, can lead to severe information leaks stemming from unprotected local and external resources. It also presents new defense solutions and mitigation strategies that successfully tackle the found vulnerabilities.

In particular, this thesis unearths three new side channels on Android OS. Prior to this work, these side channels were considered to be innocuous but here we illustrate that they can be used maliciously by an adversary to infer a user's identity, geo-location, disease condition she is interested in, investment information and her driving route. These information leaks, stem from local resources shared among all installed apps on Android: **per-app data-usage statistics**; **ARP (Address Resolution Protocol) information**; and **speaker status** (on or off). While harmless on a different setting, these public local resources can evidently disclose private information on a mobile platform and thus we maintain that they should not be freely available to all third-party apps installed on the system. To this end, we present mitigation strategies which strike a balance between the utility of apps that legitimately need to access such information and the privacy leakage risk involved.

Unfortunately the design assumptions made while adapting Linux to create Android is not the only flaw of the latter. Specifically this work is also concerned with the security and privacy implications of using external to the OS resources. Such resources generate dynamic, hard to mediate channels

of communication between the OS and an external source through usually a wireless protocol. We explore such implications in connecting smartphones with external Bluetooth devices. This thesis posits that Android falls short in providing secure Bluetooth connections with external devices; ergo its application in privacy critical domains is at the very least premature. We present a new threat, defined as **external-device mis-bonding** or DMB for short. To demonstrate the severity of the threat, we perform realistic attacks on popular medical Bluetooth devices. These attacks delineate how an unauthorized app can capture private data from Bluetooth external devices and how it can help an adversary spoof those devices and feed erroneous data to legitimate applications. Furthermore, we designed an OS-level defense mechanism dubbed **Dabinder**, that addresses the system's shortcomings, by guaranteeing that a Bluetooth connection is established only between a legitimate app and its respective accessory.

Nevertheless, Bluetooth is not the only inadequately protected external resource with grave privacy ramifications. We have also studied NFC, Audio and SMS as potential channels of communication with alarmingly low confidentiality guarantees. We show with real world attacks, that Android's permission model is too coarse-grained to safeguard such channels while preserving the utility of the apps. To better understand the prevalence of the problem we perform a measurement study on the Android ecosystem and discuss our findings.

Finally this work presents **SEACAT**, a novel defense strategy, enhancing Android with flexible security capabilities. SEACAT is a scalable, effective and efficient solution, built on top of SELinux on Android, that enables the protection of channels used to communicate with external to Android resources. It achieves both MAC and DAC protection through straightforward and SELinux-compatible policies as the policy language and structure used, is in accordance with the current policy specifications. The system's design encompasses mirror caching on both the kernel and the middleware layer which facilitates rapid policy enforcement through appropriate and carefully positioned hooks in the system.

To my parents, for their unconditional love and support.

ACKNOWLEDGMENTS

This thesis would have not been possible to realized without the invaluable support and daily guidance by my advisor and academic father Dr Carl A. Gunter.

Dr Xiaofeng Wang's indispensable guidance was also continuous throughout my work.

Special thanks to Dr Klara Nahrstedt for her feedback and guidance.

I would also like to thank Muhammad Naveed a PHD candidate at the University of Illinois at Urbana-Champaign and Dongjing He a former Master's student at the University of Illinois at Urbana-Champaign and now at Google for our collaboration.

Furthermore I greatly value my collaboration with Xiaoyong Zhou, a former PHD student at the University of Indiana, Bloomington and now at Samsung Mobile. Xiaorui Pan, Yeonjoon Lee and Kan Yuan from Indiana University, Bloomington also contributed to different parts of this collaborative work.

TABLE OF CONTENTS

LIST OF TABLES . . . viii

LIST OF FIGURES . . . x

LIST OF ABBREVIATIONS . . . xii

CHAPTER 1 INTRODUCTION . . . 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 3
- 1.3 Approach . . . 7
- 1.4 Thesis Contributions . . . 9
- 1.5 Thesis Organisation . . . 11

CHAPTER 2 BACKGROUND . . . 12
- 2.1 Android OS . . . 12
- 2.2 Android Security Model . . . 17
- 2.3 Android's Resources . . . 24

CHAPTER 3 SIDE-CHANNEL ATTACKS USING LOCAL RESOURCES . . . 27
- 3.1 Adversary Model . . . 29
- 3.2 Side-Channel 1: per-App Network Traffic . . . 30
- 3.3 Side-Channel 2: ARP Info . . . 46
- 3.4 Side-Channel 3: Speaker Status . . . 48

CHAPTER 4 ATTACKS ON EXTERNAL RESOURCES . . . 57
- 4.1 Bluetooth Mis-Bonding Attacks . . . 58
- 4.2 Other External-Resources Attacks . . . 76

CHAPTER 5 DEFENCE: GUARDING THE VULNERABLE LOCAL AND EXTERNAL ANDROID RESOURCES . . . 82
- 5.1 Mitigating the Side-Channel Threats on Local Resources . . . 82
- 5.2 DABINDER: Thwarting the DMB Threat . . . 86
- 5.3 SEACAT: DAC and MAC on External Resources . . . 92

CHAPTER 6 CONCLUSION AND DISCUSSION . . . 112

REFERENCES 118

LIST OF TABLES

2.1 Android Versions [1] 13
2.2 Android Permissions 20

3.1 Performance overhead of the monitor tool: there the baseline is measured by AnTuTu [2] 32
3.2 WebMD. Comparison of Bytes Transmitted between two Conditions of different Categories 37
3.3 WebMD. Traffic Analysis for the *ACUTE SINUSITIS* condition navigation 39
3.4 City information and Twitter identity exploitation 43
3.5 Geo-location with a Single BSSID 49
3.6 Comparison between a Navigation Sequence and a Text Direction/TTS Sequence 54
3.7 Route Identification Result. The third column is the highest overlap ratio of a wrong route within the top 10 TTS sequences. FP indicates false positive. All FP routes (actually similar routes) are marked out in Figure 3.10. 55

4.1 Success rate of data-stealing attack. This table depicts the successful connections made by the malicious app on 100 trials. 66
4.2 Average power consumption over 10 minutes per surveillance technique using PowerTutor[3]. 67
4.3 Average power consumption over an hour. Comparison between our surveillance technique and 2 popular applications using PowerTutor[3]. 67
4.4 Data-injection attack launched 1ft and 20ft away from the victim's phone, with the original device touching the phone. In both cases, the experiments were repeated 100 times. 72
4.5 Sampled apps 73
4.6 Manual analysis on 20 apps. The other 48 apps were automatically filtered out by the locations of their suspicious APIs. 76
4.7 Critical Examples 77

5.1 Dabinder performance evaluation. (mean / sd) 92
5.2 *SEACAT* API 102
5.3 Threats to Android external resources 108
5.4 A list of operations affected by *SEACAT* enhancements 109
5.5 Detailed Performance Measurements in milliseconds (ms) . . . 111

LIST OF FIGURES

2.1 Android Software Stack [4] 14
2.2 Activity Lifecycle [5] 15
2.3 Application Isolation on Android 16
2.4 Android Permission Check 21

3.1 Monitor tool precision 31
3.2 Monitor tool UI 32
3.3 WebMD: First Screen 34
3.4 WebMD: A Condition's Screen 34
3.5 WebMD Finite State Machine 35
3.6 First Order Traffic Classification of WebMD's conditions . . . 36
3.7 Audio elements similarity when driving on the same route . . 50
3.8 Audio length sequence distinguishability 52
3.9 False positive rate vs number of audio elements 52
3.10 Three FP Routes and Their Corresponding TP Routes. Each FP/TP pair has most of their routes overlapped. 56

4.1 Data-stealing Attack 61
4.2 Normal Scenario 69
4.3 Adversary injecting fake data 69
4.4 Classifications of the sampled apps. Some of them collect information in multiple categories. 74

5.1 Effectiveness of round up/down mitigation technique 85
5.2 Bluetooth Subsystem and our defense mechanism: *DaBinder* is built into AdapterService and checks the interaction between apps and Bluetooth devices. It only allows authorized app to access Bluetooth device and keeps bonding policy in a secure storage. *Reference Monitor* and *Bonding Policy* blocks (both shown in light-blue) constitute Dabinder. 88
5.3 *SEACAT* architecture 95
5.4 *SEACAT* Policy Compliance Check 103

5.5 *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 106

LIST OF ABBREVIATIONS

AOSP	**A**ndroid **O**pen **S**ource **P**roject
apps	Smartphone **App**lication**s**
ARP	**A**ddress **R**esolution **P**rotocol
AVC	**A**ccess **V**ector **C**ache
DAC	**D**iscretionary **A**ccess **C**ontrol
DMB	external-**D**evice **M**is-**B**onding
IMEI	**I**nternational **M**obile station **E**quipment **I**dentity
MAC	**M**andatory **A**ccess **C**ontrol
SEACAT	**S**security **E**nhanced **A**ndroid **C**h**A**nnel con**T**rol

CHAPTER 1

INTRODUCTION

This thesis performs a detailed study on Android security and presents and discusses appropriate mitigation strategies to address found vulnerabilities. It specifically scrutinizes over the adoption of Linux to Android and the inefficiencies of the system when it comes to the interplay of smartphones with external devices and sources of information. But why should someone study mobile device's security? To address this question, we dedicate most of this Section to elaborate on the role of smartphones in the contemporary society, epitomising the significance of the provision of both security and privacy guarantees when it comes to their design. We will also define the problem with local and external resources and discuss our approach to effectively tackle it. Subsequently we list the contributions made by this work and finally provide the structure of this treatise on Android security.

1.1 Motivation

Seven years now, after the first iOS and Android enabled smartphones, the technology behemoths are now responsible for 90% [6] of total smartphone sales in 2013. 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 apps of any perceived purpose, led to the release of an astounding number of mobile applications in official application markets. These applications 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.

Furthermore, these multipurpose "phones" are equipped with a variety of sensors, with receivers and transmitters that enable them to communicate with a plethora of external sources through wireless protocols such as Bluetooth, NFC (Near Field Communication) and SMS (Short Message Service), or through the Internet. 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 [7] 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 Internet sockets are vital for the utility of web-based apps or just apps that make profit out of targeted advertising.

Of particular interest, is the Android OS which dominates the smartphone marketshare [6]. 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, 2GB 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 vast 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 userspace and Android is the ideal candidate for doing just that. As malware targeting Android increases, we have witnessed a

large scale of malicious attempts [8] exploiting the system's vulnerability to gain root access, or charging users money, by sending SMSs or calling premium numbers [9]. Furthermore the scientific community delineated another spectrum of the popular system's vulnerabilities, using more sophisticated attacks such as permission re-delegation [10] and capability leaks [11].

Android marketshare, penetration, use in sensitive settings and the fact that is being targeted by the vast majority of smartphone malware, highlight the significance of both studying the risks associated with its use and designing novel, efficient and effective defense mechanisms that minimize those risks. Next we elaborate on the current problems of the platform in respect to the aforementioned issues.

1.2 Problem Statement

Android was adapted from Linux with necessary modifications to reflect and address the needs of a mobile platform. This adaptation can lead to privacy violations if not performed correctly. The risks involved in this process stem from two main factors. Firstly, local OS resources used in a stationary machine sometimes require different access control management when used on a mobile platform. Secondly a mobile platform generated the need for enhancements that can take advantage of its ubiquitous nature. For this purpose, smartphones are equipped with a variety of transmitters and receivers that enable the use of multiple wireless protocols. This new capabilities produce new security and privacy requirements as information flows not only within the OS, but seamlessly to and from external sources.

Firstly we postulate that Android suffers from information leaks stemming from unprotected local resources. Android is a complicated system dealing with a lot of user private information. In addition such a mobile platform can accommodate the use of a myriad of third-party applications that may or not, require access to this information to function properly and offer their services to their users. To mediate access to such sensitive data, Android integrates a permission model 2.2.2. This model dictates that any app that wants to access a piece of information must request it at install time and the user will decide whether to grant that permission to the app or reject its installation. Even if this model has its inefficiencies, it does provide some

control over its local resources. Nevertheless, not every piece of information is protected by this model. The protection of resources that are left behind, 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 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. In addition, Android offers a rich API to provide the means to third-party apps to readily access resources. Some of these API calls are protected from the aforementioned permission model, however not all of them are. This work studies the possibility that some unprotected API calls can lead to privacy breaches.

To realize the significance of this problem considering the following example. Lets assume that Alexander (Alex for short), owns a smartphone running Android. Alex suffers from diabetes type II, which impels him to install a popular web-based medical app that helps him understand the implications of his condition and how he can better manage it. Such apps usually deal with scores of information and is preferable to store that in a remote location. Thus a user will download only the information he or she needs in real-time without encumbering its device with tons of unnecessary data. Web-based apps also have the advantage to serve both browser and mobile apps requests while keeping their content easily up-to-date and synchronized in all clients. Alex is also gay and he has decided to install a popular dating app to help him find an appropriate partner. In addition Alex enjoys classic mobile games and he installed a free game found online. Consider the fact that the latter app is malicious and disguises its purposes with a smart User Interface (UI) that allows Alex to throw screaming birds against pieces of wood and rocks. If this malicious app gains access to the installed applications, Alex's sexual preferences will immediately leak, a severe privacy infringement. Furthermore, if the malicious app, can access the network traffic generated by the web-based medical app, potentially it can infer Alex's medical condition. This is unequivocally another privacy infringement and in addition this kind of information can be very valuable to insurance companies. Such companies increase the annual fee to users that are more likely to need medical care and

if they knew that Alex belongs to such category they would have changed its fees appropriately.

Secondly we argue that Android also suffers from information leaks due to unprotected communication with remote resources. Mobile platforms are getting increasingly powerful and irrefutably are now more than devices used to make a phone call. In particular, a plethora of business, medical, entertainment and other accessories are being used daily from smartphone users. For example Bodymedia Link Armband [12] can be used to monitor ones daily activities, Nonin Pulse Oximeter [13] is used to monitor a patient's pulse and oxygen saturation and Entra Health System Blood Glucose Meter [14] to monitor a patients blood glucose levels. All these devices use Bluetooth to connect to a smartphone app than enables users to better manage their data and even share it with family, friends or their personal physicians. Furthermore, NFC tags can be used to automate tasks on a phone. One can use a tag to automatically input a WiFi password, change its smartphone profile or account according to the physical location that the tag is placed. Moreover, NFC can be used to receive credit card payments or exchange data between NFC devices in general. The audio jack on a smartphone can also be used for credit card payments [7] or receive and display data from fitness devices [15]. Furthermore, SMSs can be used for 2-factor authentication, where a user can connect and use a service using both a password and a PIN received on her device through an SMS. Popular examples that use this feature are among others Facebook, WhatsApp and the Chase Bank app. Last but not least, Internet sockets (e.g TCP) are being utilized by apps to exchange data with their respective servers. This is gaining increasing attention as we embark on the era of the Internet Of Things (IoT) where all devices are connected through the Internet and managed from a smartphone. IoT devices controlled by smartphones can be used for home automation and security [16], remotely control vehicles [17] or for health and fitness purposes [18, 19, 20].

Bluetooth, NFC, Audio, SMS and Internet constitute channels of 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 developers correctly protected access to those channels with permissions. However, not only permissions

are being neglected or granted without scrutiny from users [21] 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 NFC permission can access any NFC device in vicinity. Also apps with the READ_SMS permission can read any SMS message, whether that comes from a friend or from Chase currying security critical 2-factor authentication data. Moreover, an app with the AUDIO permission cannot only be used to support a speaker but can read data transitted to a connected fitness accessory [15]. Lastly, almost all free apps use the INTERNET permission, which is needed for them to be able to support targeted advertising. Targeted advertising is a common source of income for free apps, as they can display advertisements to their users and profit on every click on that advertisement. Such an app can use the Internet permission to surreptitiously create sockets that enable it to send out stolen information, receive malicious payloads or even take screenshots [22].

It is clear that a more fine-grained control over these channels is needed to control access to information communicated through them. Such control will allow 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 headset but restrict it from talking to a protected blood glucose meter and so on.

To better understand the problem lets consider Maria (Maria for short too). Maria is an athletic person and uses a fitness accessory that connects via Bluetooth to its respective app installed on her smartphone. This allows her to keep track of her diet, fitness and body status. Maria has another app installed and she granted it the Bluetooth permission as well, because in its description it posits that it needs it to connect to Maria's headset and it indeed does. However, the latter app exploits that fact and connects to the fitness accessory too. It furtively steals that information and it sells it to advertising companies along with Maria's IMEI number (a unique per device number used by a GSM network to identify valid devices). Furthermore, Maria is a Chase Bank customer and she uses its mobile application to conveniently pay her credit card and deposit cheques. The Chase Bank

app uses 2-factor authentication and requires Maria to input along with her password, a PIN that she can receive through an SMS. However, Maria installed another app which can scan barcodes on groceries and supply her with calorie information. That app requires the READ_SMS permission as it allows Maria to swiftly text such information with her friend Anna (also Anna for short) who is too a fitness enthusiast. However that app manipulates its access to the received SMSs to read authentication PINs sent by Chase and uses them to obtain access to Maria's Chase account.

Anna, Maria's friend, uses a fitness bracelet that connects to her smartphone app through the Audio jack and allows her to better manage their training progress. Anna also like music and she has installed an app to manage her playlists and listen to her favorite songs. The latter app though, abuses the fact that Anna has granted it the AUDIO permission and reads the data sent by the fitness bracelet when connected to the phone. In turn it sells that data to advertising companies.

Evidently Android local and external resources can leak critically private information if not adequately protected. This work performs a systematic study to help unearth vulnerabilities stemming from such resources and understand the magnitude of the problem. We will also discuss mitigation strategies which we designed to protect the OS from such vulnerabilities. Before delving into the details of this work, we will expatiate on the approach to address this fundamental design problem.

1.3 Approach

This work studies the vulnerabilities of Android OS in respect to inadequately protected local and external resources and further proposes mitigation strategies.

Firstly, we look into the risks involved in adapting Linux to Android. This adaptation if not done carefully can expose seemingly innocuous information to unauthorised apps that can exploit it for their malicious purposes. Vulnerabilities can arise due to mainly two factors: erroneously configured Linux DAC or the absence of Android permissions when necessary. On the one hand the Linux file system is protected with a Discretionary Access Control Model to mediate access to directories and files on the system. Such files

can carry information that might not be considered sensitive on a stationary machine, and thus processes are granted unfettered access to them. However, when applied on a mobile platform this public access can lead to severe information leaks. For example, knowing the router a stationary machine is connected to might not pose a risk as usually such a machine is always connected to a private network, e.g at home or at work. Therefore processes can be allowed to read such information as the risk is minimal. Nonetheless, when such information is unprotected on a mobile platform, third-party apps can utilize it to know the user's current location at all times, given the fact that people carry their smartphones wherever they go. On the other hand, Android controls access to sensitive information with the use of permissions. During installation time, a third-party app asks the user to grant it the permission to access a particular resource. When an app wants to accesss a resource, it can use Android's API to call the appropriate method that will return the data requested. During that request, the OS will check whether the appropriate permission was granted by the user and decide whether to allow access to the resource or through a security exception. However, if an API method that allows access to private information is not mediated by a permission check from the OS, it can lead to information leaks.

To study whether there are files insufficiently protected through Linux DAC access control and whether there are information leaks stemming from API calls not protected with permissions, this thesis inspects resources disclosed at both the Android and Linux layer and scrutinizes over their security implications. When an unprotected resource leading to a privacy breach is unearthed, a detailed real world attack example is being presented to delineate the risk associated with such an inefficiency.

Secondly, we will examine the communication of Android powered phones with external or remote resources. We consider external resources to be any kind of accessory or remote source of information that can transmit or receive information to and from the smartphone to enhance its capabilities. To this end, a study is being performed on the communication of smartphones with Bluetooth and NFC devices, the receipt of SMS messages and communication with accessories connected to the phone through the audio jack. In this work we refer to SMS, Bluetooth, NFC, Internet and Audio as *channels* of communication with external resources. As previously mentioned Android guards access to resources through permissions and it doesn't fail to do that

for the aforementioned channels. For example if an app wants to establish a Bluetooth connection with a Bluetooth device it has to be granted the *Bluetooth* permission by the user, to be able to make the appropriate method call. Here we argue that the permission model is too coarse-grained to protect access to those channels. Consider an app that wants to talk to a Bluetooth headset and for that is granted the Bluetooth permission. That privilege allows it to access any Bluetooth device in vicinity that is paired with the phone.

To study the risks involved with the coarse-granularity of the permission system when it comes to protecting communication with external resources, this work performs a meticulous security study on the most prominent channels of such communication. It looks into the communication of Android-enabled phones with Bluetooth devices and designs attacks that demonstrate the system's incapacity to protect such interaction. Then we will perform a measurement study to help better understand the prevalence of the problem. Next, we will attempt a generalisation to other channels of communication with external resources, such as Internet, SMS, Audio and NFC. We will delineate the severity of the problem with a survey on Google Play (the official Android App Market) and we will design and demonstrate attacks when needed, to exemplify it.

Lastly, this work will discuss mitigation strategies for the problems and vulnerabilities found. We will discuss known solutions and design new ones when appropriate. For example there is no known solution for allowing fine-grained access control to external resources, and this work will attempt to design an effective, efficient and novel system to address that threat.

The next Section will summarize the contributions made by this thesis and the last Section will elaborate on the structure of this report.

1.4 Thesis Contributions

Here we summarize this work's jointly with Zhou et al. [23], Naveed et al. [24] and Demetriou et al. [25], contributions:

- *We found new information leaks from Android public local resources.* Related work has been focusing on implementation flaws on Android. Here we studied how Android's design has some demonstrably fallacious assumptions

that can lead to hazardous information leaks. Such assumptions result in Android local resources being left unprotected. However some seemingly innocuous local resources can leak user private information. We demonstrate the design flaws with a suite of new inference techniques that depict how an adversary can recover a user's data from Android public local resources.

• *We found new threats on Android's communication with external resources.* This work systematically studies Android's channels of communication with external resources. These channels, namely Bluetooth, SMS, NFC, and Audio are proven to be insufficiently protected by Android's Permission Model. We define a new threat that we call device mis-bonding (DMB) for the Bluetooth channel and further demonstrate that Android's Permissions are too coarse-grained to support the utility of the apps while guaranteeing the confidentiality of the data communicated through these channels. Furthermore we measure the prevalence of the problem in the Android ecosystem.

• *We developed strategies to mitigate the threat stemming from Android public local resources.* We have developed a new mitigation approach, designed to preserve the utility of legitimate apps, while at the same time allows control on how public data can be made available to an adversary.

• *We developed a new technique called Dabinder that mitigates the DMB threat through appropriate changes on the OS.* We have developed the first technique to mitigate the device mis-bonding (DMB) threat. This approach automatically generates security policies for protecting the bond between an external Bluetooth device and it's authorized app. Furthermore our defense effectively enforces these security policies without impeding normal operations on the phone.

• *We developed a new OS-level solution called SEACAT to safeguard the communication with Android external resources, using both MAC and DAC.* We have designed the first mechanism that provides comprehensive protection of different kinds of Android external resources over their channels in a uniform way. Our approach is built on top of SELinux on Android and achieves 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.

1.5 Thesis Organisation

In Section 2, we will present background knowledge needed to support the technicalities of the remaining sections. In particular, we briefly present the Android OS 2.1, explaining its major components. Next, we will look into the Android Security system (2.2), focusing on how Android Sandboxes applications (2.2.1), how it protects sensitive API calls (2.2.2) and the partial integration of SEAndroid [26] (2.2.3) on the latest releases. We will continue with a brief analysis on local (2.3.1) and external Android resources (2.3.2).

In Section 3 we present the study on Android local resources: after a brief overview, this thesis presents the adversary model we considered (3.1). Subsequently it shows how that adversary can use information from unprotected local resources to infer an Android user's Twitter account, Health information and Stock Exchange Preferences (3.2). Then based on other vulnerabilities from local resources an attack is presented that allows the adversary to infer a user's location (3.3) and another attack leads to a user's driving route inference (3.4).

In Section 4 we elaborate on our study on Android external resources. We will illustrate attacks on the Bluetooth channel (4.1) and then we will generalise the problem to other channels (4.2). We will depict the significance of the problems found with relevant surveys.

In Section 5 we will present known solutions to the local resources attacks 5.1 and a new defense mechanism that mitigates the Bluetooth attacks (5.2). Thereafter we will illustrate a scalable, effective and efficient solution we have designed to tackle the external resources threat (5.3).

Lastly in Section 6 we will conclude this thesis and discuss its findings.

CHAPTER 2

BACKGROUND

In this Section we provide background to facilitate understanding of the work performed in analyzing vulnerabilities on Android stemming from insufficiently protected local and external resources and in designing a system to mitigate such inefficiencies. We will look into the architecture of the Android OS and its major components, we will discuss its current security model and finally explain what this work perceives as local and external resources.

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 [1]. 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.

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 Dalvik virtual machine and the application

Table 2.1: Android Versions [1]

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

framework written in Java. The Android software stack is displayed in figure 2.1.

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 [5].

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 pro-

Figure 2.1: Android Software Stack [4]

vided 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

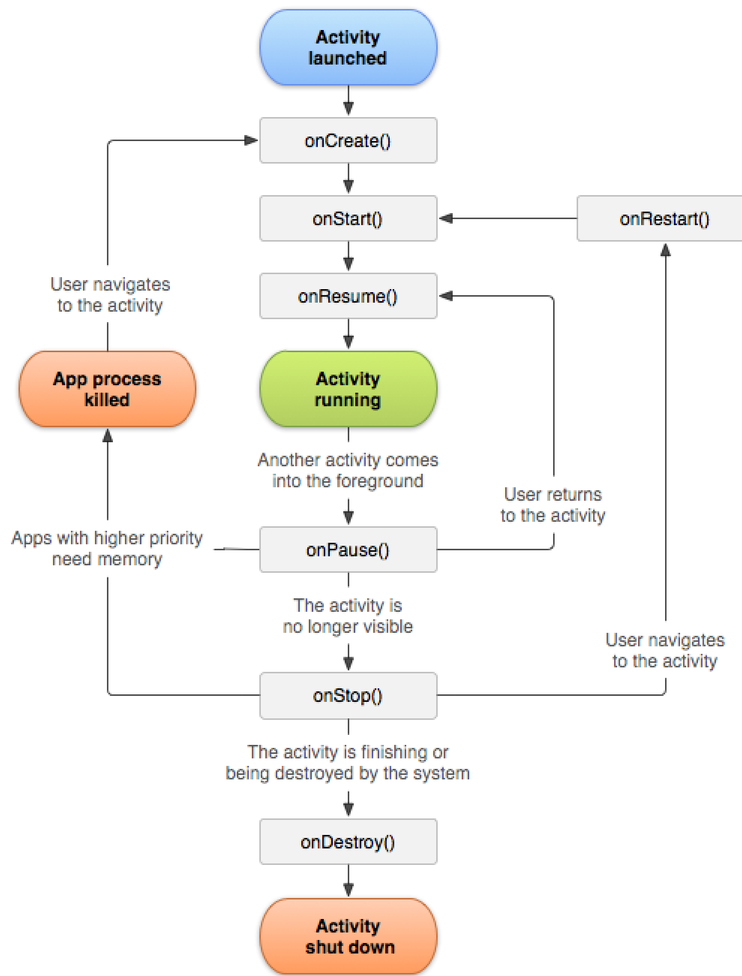

Figure 2.2: Activity Lifecycle [5]

(DVM) as shown in figure 2.3. Dalvik is an efficient process virtual machine 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.

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

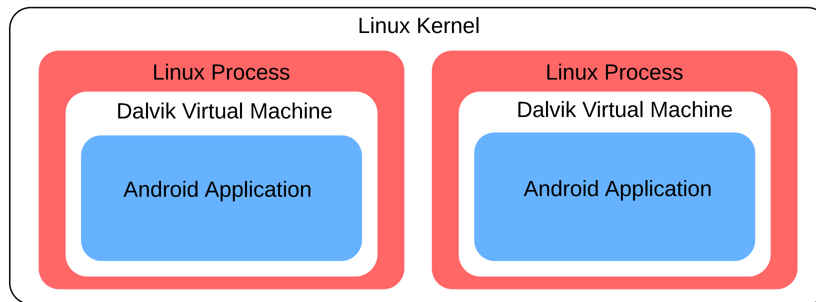

Figure 2.3: Application Isolation on Android

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. Thus 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. During the application's installation process handled by the *Package Manager*, the user is presented with a list of permissions the app is requesting. Each permission is presented alongside a description of what an app can do with that permission granted. The user can either accept all permissions requested and install the app or abort the installation process. These are known as the `system default permissions`. Furthermore, an application can declare its own permissions to control access to its own resources. This way other apps can request the permission declared and the data owner app can check data requesting apps whether they have the permission or not. For example this can be used by collaborative apps, either from the same developer or not.

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 instal-

lation 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 during an app's installation process. Dangerous permissions protect among others: Camera functions; Location data (GPS); Bluetooth functions, NFC functions, Audio functions, Telephony functions, SMS/MMS functions and Network/data connections. Table 2.2 is an excerpt from Android developers official website [27] listing Android system default permissions. 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. [21].

The Android OS checks permissions in 2 ways as shown in Figure 2.4: 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 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

Table 2.2: Android Permissions

Name	Description
BLUETOOTH	Allows applications to connect to paired Bluetooth devices
BLUETOOTH_ADMIN	Allows applications to discover and pair bluetooth devices
CAMERA	Required to be able to access the camera device
NFC	Allows applications to perform I/O operations over NFC
GET_TASKS	Allows an application to get information about the currently or recently running tasks
INTERNET	Allows applications to open network sockets
READ_SMS	Allows an application to read SMS messages
RECEIVE_BOOT_COMPLETE	Allows an application to receive the ACTION_BOOT_COMPLETED that is broadcast after the system finishes booting.
RECORD_AUDIO	Allows an application to record audio

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.

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. [26] published a detailed solution to port SELinux on Android, called Security Enhanced Android (SEAndroid).

Security-Enhanced Android is built on top of Android [26]. It is designed

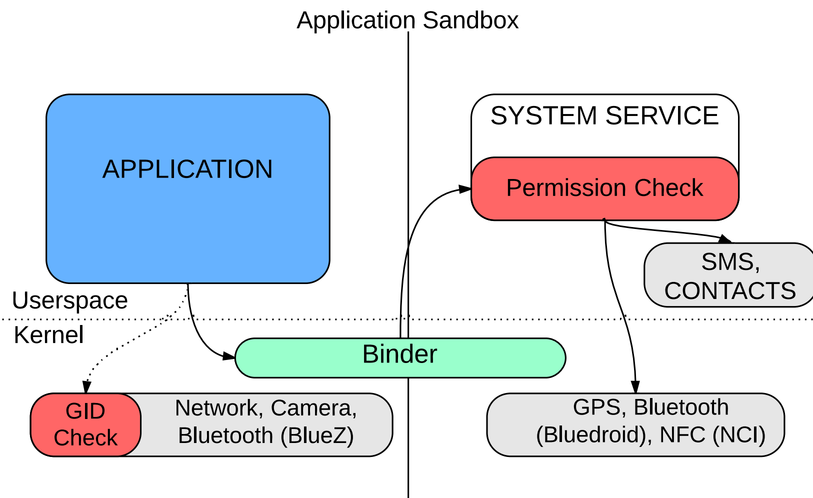

Figure 2.4: Android Permission Check

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 [26] 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

[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.

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 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 Android's Resources

Android is an operating system and as such it manages numerous resources. Some of resources can be utilized by the system for maintenance and scheduling whereas others are being provided to applications which leverage them to offer creative functionalities to the platform's user. This Section will define

the meaning of `local` and `external` resources as used throughout this work.

2.3.1 Android's Local Resources

Here we will focus on local resources that Android makes publicly available to all unprivileged processes and therefore all third-party apps without the need of a permission. We follow the classification on these local unprotected resources proposed in joint work with Zhou et al. [23]). Specifically these `public local resources` can be part of two categories: Linux public directories and Android public APIs.

• *Linux layer: public directories.* Linux historically makes available a large amount of resources considered harmless to normal users, to help them coordinate their activities. A prominent example is the process information displayed by the `ps` command (invoked through `Runtime.getRuntime.exec`), which includes each running process's user ID, Process ID (PID), memory and CPU consumption and other statistics.

Most of such resources are provided through two virtual filesystems, the proc filesystem (procfs) and the sys filesystem (sysfs). The procfs contains public statistics about 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 [28] to keep track of individual apps' mobile data consumption.

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

Access to those resources is allowed by design as the information one can acquire through them is not consider sensitive or security critical by the OS

developers.

2.3.2 Android's External 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). Examples include health and fitness systems (e.g., blood pressure monitors [18], Electrocardiography sensors [19], glucose meters [29]), remote vehicle controllers (e.g., Viper SmartStart [17]), home automation and security systems [16] and others. Those external devices and Internet resources are connected to smartphones through a variety of `channels`, which are essentially a set of hardware and software through which an app accesses the external resources. These channels are composed of Bluetooth, NFC, SMS, Internet and Audio.

CHAPTER 3

SIDE-CHANNEL ATTACKS USING LOCAL RESOURCES

As stated in Section 2.3.1 Android provides unprivileged applications with access to basic local 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 joint work with Zhou et al. [23] we identified two fundamental design/use gaps that are swiftly widening, affecting the Android ecosystem:

Firstly, we 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 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, we witnessed the manifestation of a gap between the assumptions on Android public resources and evolving app design, functionalities and background information throughout our 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, pub-

lic 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 our joint research [23], we carefully analyzed the ways public local resources are utilized by the OS and popular apps on Android, together with the public online information related to their operations. Our study discovered three confirmed new sources of information leaks:

- *App network-data usage* (Section 3.2). We 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, we 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 3.3). We 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.

- *Audio status API* (Section 3.4). We show that the public audio status information (speaker on/off) collected from a GPS navigator can be used to fingerprint a driving route. We further present an inference technique that uses Google Maps and the status information to practically identify her driving route on the map.

We built a zero-permission app that stealthily collects information for these attacks. This Section elaborates on side-channel attacks designed and executed based on these newly found information leaks. Firstly we will see the capabilities that adversary (3.1) possess to be able to deploy such attacks.

3.1 Adversary Model

The adversary considered in our research runs a zero-permission app on the victim's smartphone. Such an app needs to operate in a stealthy way to visually conceal its presence from the user and also minimize its impact on a smartphone's performance. On the other hand, the adversary has the resources to analyze the data gathered by the app using publicly available background information, for example, through crawling the public information released by social networks, 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 [30]. The idea is to let the sender app use the `URI ACTION_VIEW` Intent to open a browser and sneak the payload it wants to deliver to the parameters of an HTTP `GET` from the receiver website. We re-implemented this technique in our 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. We improved this method to smuggle data right before the screen is off or the screen is being unlocked. Specifically, our app continuously monitors `/lcd_power` (`/sys/class/lcd/panel/lcd_power` on Galaxy Nexus), an LCD status indicator released under the sysfs. Note that this indicator can be located under other directory on other devices, for example, `sys/class/backlight /s6e8aa0` on Nexus Prime. When the indicator becomes zero, the phone screen dims out, which allows our app to send out data through the browser without being noticed by the user. After the data transmission is done, our app can redirect the browser to Google and also set the phone to its home screen to cover this operation.

A malicious app should also be aware of the system's situation or state. Our 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 (`getInstalledApplications()`) and then periodically calls `ps` to get a list of active apps and their PIDs. Once a target is found to be active, our app will start a thread that closely monitors the `/proc/uid_stats/[uid]` and the `/proc/[pid]/` of the target.

3.2 Side-Channel 1: per-App Network Traffic

3.2.1 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 is newly introduced by Android to keep track of individual apps. These directories can be read by *any* app directly or through `TrafficStats`, a public API class. Of particular interest here are two files `/proc/uid_stat/[uid]/tcp_rcv` and `/proc/uid_stat/[uid]/tcp_snd`, which record the total numbers of bytes received and sent by a specific app respectively. We 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 [28]. However, our 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, we 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 3.1, in which we compare the packet payloads observed by Shark for Root (a network traffic sniffer for 3G and

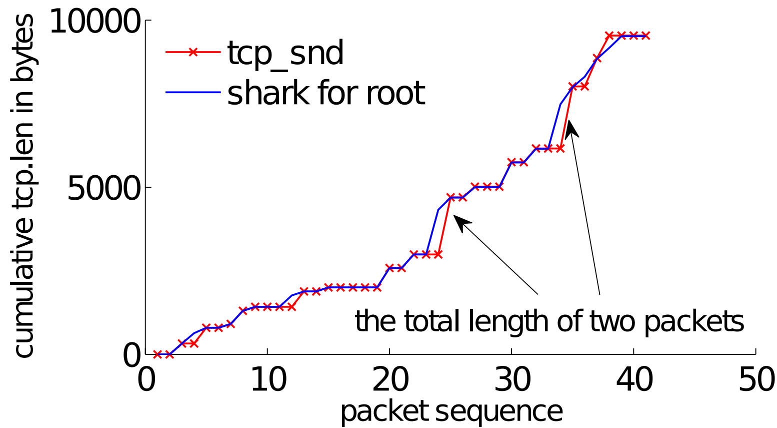

Figure 3.1: Monitor tool precision

WiFi) [31], when the user is using Yahoo! Finance, with the cumulative outbound data usage detected by our usage monitor.

From the figure 3.1 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 we discuss later.

In terms of performance, our monitor has a very small memory footprint, only 28 MB, even below that of the default Android keyboard app. When it is running at its peak speed, it takes 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 3.1 measured using AnTuTu [2] with a sampling rate of 10Hz for network usage 3.2 and 50Hz for audio logging (Section 3.4). To make this data collection stealthier, we adopted a strategy that samples intensively only when the target app is being executed, which is identified through `ps` (Section 3.1). The UI of the monitor tool is shown in Figure 3.2.

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.

Our approach first uses Shark for Root to analyze a target app's behavior

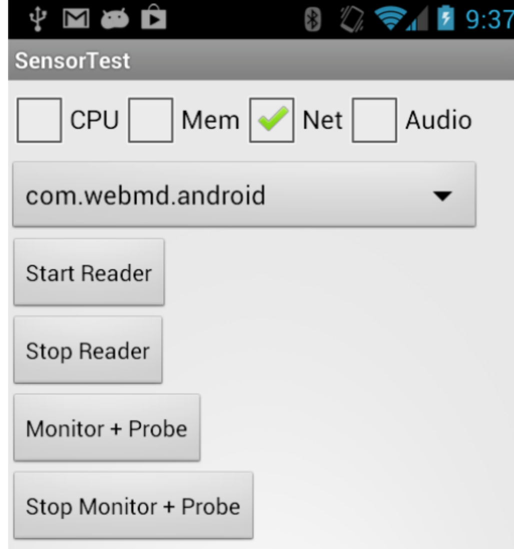

Figure 3.2: Monitor tool UI

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

Table 3.1: Performance overhead of the monitor tool: there the baseline is measured by AnTuTu [2]

	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%

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")`.

3.2.2 Health Data

In this section, we show that the data-usage statistics our zero-permission app collects leak out apps' sensitive inputs, e.g., disease conditions a user selects on WebMD mobile [32]. This has been achieved by fingerprinting her actions with data-usage sequences they produce. The same attack technique also works on Twitter 3.2.3 and Yahoo! Finance 3.2.4.

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 [32]. 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 3.3. In our research, we 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 3.4, all such

Figure 3.3: WebMD: First Screen

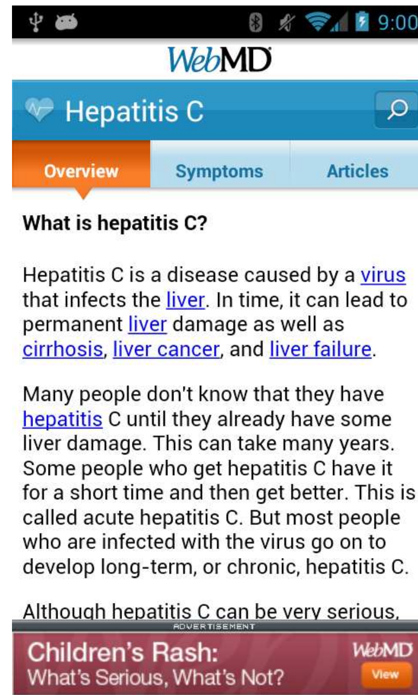

Figure 3.4: WebMD: A Condition's Screen

information is provided through a simple, fixed user interface running on the phone, while the data there is downloaded from the web. We 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

We 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 3.5. 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`

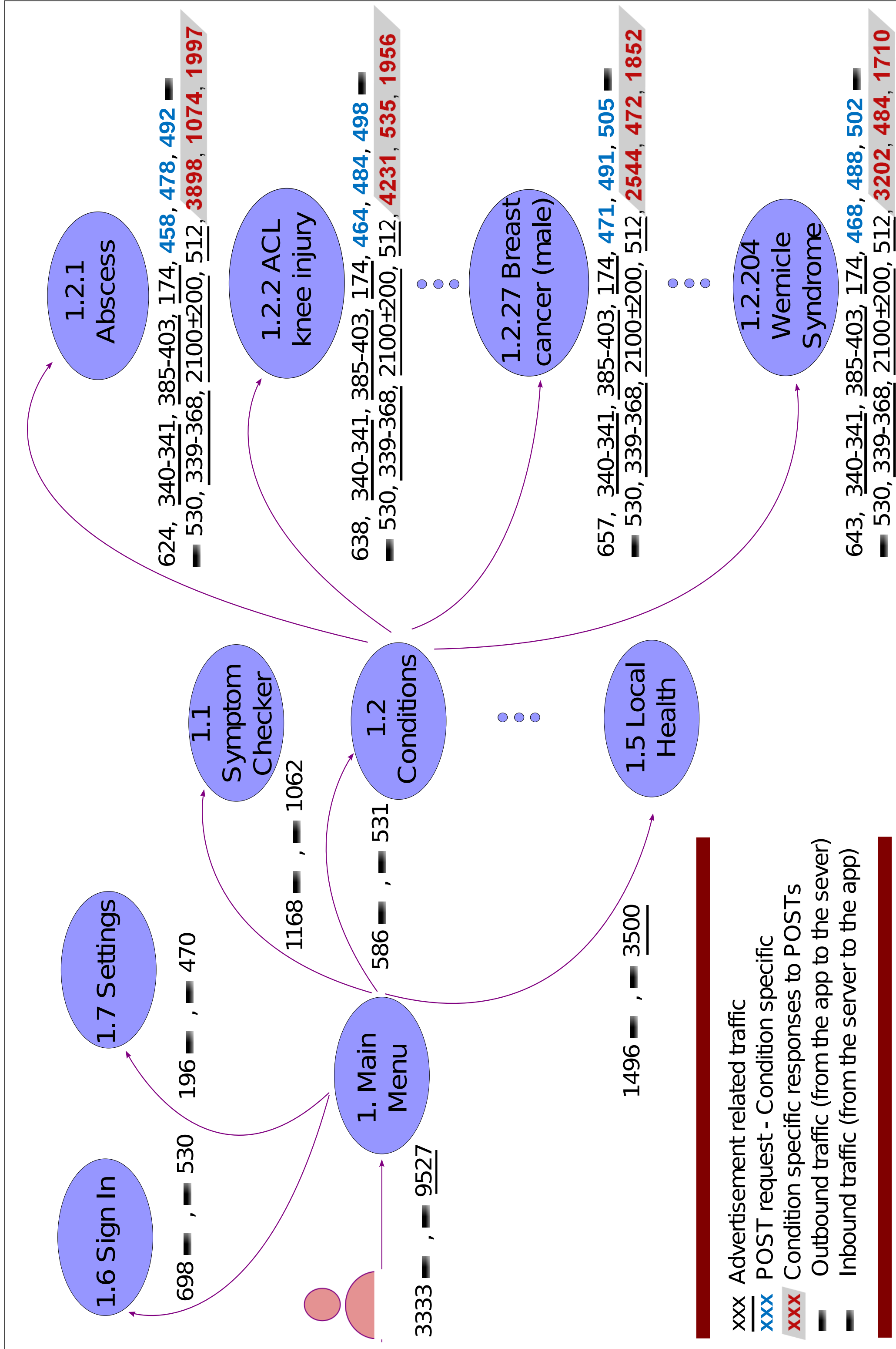

Figure 3.5: WebMD Finite State Machine

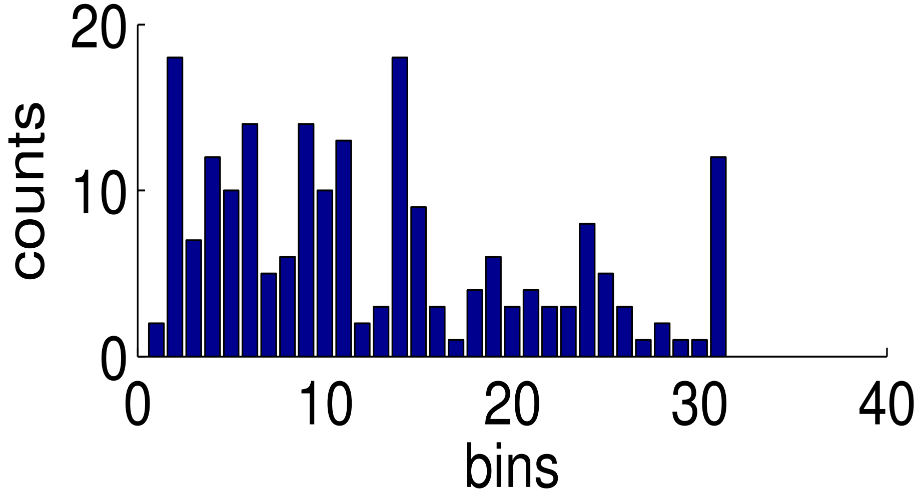

Figure 3.6: First Order Traffic Classification of WebMD's conditions

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 [33], 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 3.6). Table 3.2 shows an example of distinct transmission traffic patterns between "Anemia. iron deficiency" and "Vulvodynia".

Furthermore, we 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 our 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 3.2.1. Subsequently we use the information from $\{r_1, r_2, r_3\}$ to further differentiate between conditions of the same category.

In a real attack, however, our 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

Table 3.2: 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

technique described in Section 3.2.1, 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 our 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 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 we proceed according to the following methodology: As stated before, using Shark for Root we have created a detailed map of the states the application can be at any possible time. We refer to states as screens being displayed to the user as denoted by the simplified state diagram on Figure 3.5 . For each state of

the application we recorded 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 3.4. 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-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 3.6). The request on row 1 of Table 3.3 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 we have used the requests made for the three aforementioned condition specific information, which we mark at the fifth, sixth and seventh row of the Table...(example of a data collection). 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 3.6). 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

Table 3.3: 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

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 3.3) generated with each condition click and also understand the application protocol in place, we ran a set of experiments. For those experiments we have used a Google Nexus S 4G device running Android 4.1.1 with root access to the Operating System, available. On the device we installed Shark for Root which can capture the traffic and generate pcap files that we can analyze using an appropriate tool such as Wireshark. We have also installed WebMD and our monitor tool on the device. Before every experiment, we launch our tool set to monitor WebMD's traffic's and Shark which captures all network traffic on the device. Then we launch WebMD and navigate to a particular condition. Subsequently we stop our tool and Shark and analyze the results matching our tool's recordings with the measurements from Shark. Based on our analysis we generate tables (see table data sample again) 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 ().

Attack Evaluation

To evaluate the effectiveness of our attack on WebMD, we repeated our experiments. This time, we didn't mark our tool's output with the Condition

being visited on the device by the user. Conversely we 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 we have performed 221 experiments for 204 available Conditions. Our shuffling tool rejected 2 outputs which left us with 219 results to analyze. We 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 we further analyze the inbound traffic to identify the precise condition in the Category that has similar traffic with the observed one. Our tool's sampling rate has been proven instrumental to this effort as in most cases, a single reading of it could disclose to us 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 we were 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 we 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).

3.2.3 Identity

In joint work with Zhou et al. [23] we show that the data-usage statistics collected by our zero-permission app, 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 [34, 35, 36, 37] and should not be released to an untrusted party. For a smartphone user, unauthorized disclosure of her identity can

immediately reveal a lot of private information about her (e.g., disease, sex orientation, etc.) simply from the apps on her phone. Here we show 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 [38]. We also 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 we describe below poses a realistic threat to Android users' identity.

Attack Methodology

In our attack, a zero-permission app monitors the mobile-data usage count `tcp_snd` of the Twitter 3.6.0 app when it is running. When the user send tweets to the Twitter server, the app detects this event and send its timestamp to the malicious server stealthily. This gives us a vector of timestamps for the user's tweets, which we can then use to search the tweet history through public Twitter APIs for the account whose activities are consistent with the vector: that is, the account's owner posts her tweets at the moments recorded by these timestamps. Given a few of timestamps, we can uniquely 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. Our solution is using less protected data, the coarse location (e.g, city) of the person who tweets, to narrow down the search range (see Section 3.3 for an attack that allows an adversary to gain such information).

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

Specifically, during the offline analysis, we observed the following TCP payload sequence produced by the Twitter app: $(420|150, 314, 580\text{–}720)$. 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, what we want to do here is to use the signature to find out the timestamp when the user tweets. The problem here is that our usage monitor running on the victim's phone does not see those packets and can only observe the increments in the data-usage statistics. Our offline analysis shows that the payload for Client Hello can be reliably detected by the monitor. However, the time interval between 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 3.2.1) 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 [39]: 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, we found that the user's fine location could be inferred once she connects her phone to a Wi-Fi hotspot (see Section 3.3). Getting her coarse location in this case is much easier: our zero-permission app can invoke the mobile browser to visit a malicious website, which can then search her IP in public IP-to-location databases [40] to find her city. This allows us to set the query parameters using Google Maps. Note that smartphone users tend to use Wi-Fi whenever possible to conserve their mobile data (see Section 3.3), which gives our app chances to get their coarse locations. Please note that we do not require the user to geo-tag each tweet. The twitter search results include the tweets in a area as long as the user specified her geo-location in her profile.

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

Table 3.4: 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

Attack Evaluation

We evaluated the effectiveness of this attack at three cities, Urbana, Bloomington and Chicago. Table 3.4 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 3.4. 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.

3.2.4 Investment Data

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

Attack Methodology

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

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 our study, we ran our zero-permission app to monitor the Yahoo! Finance app on a Nexus S 4G smartphone. From the data-usage statistics collected while the suggestion list 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.

3.3 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. [23].

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 3.2.3 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.

3.3.1 Location Inference

We found that the BSSID of a Wi-Fi hotspot and signal levels perceived by the phone are disclosed by Android through procfs. Such information is location-sensitive because hotspots' BSSIDs have been extensively collected by companies (e.g., Google, Skyhook, Navizon, etc.) for location-based ser-

vices 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 we 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. Actually we talked to 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.

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 [41], 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.

3.3.2 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 3.5.

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 3.5 describes the numbers of the hotspots good for geo-locations at different spots and their accuracy.

3.4 Side-Channel 3: Speaker Status

As discussed before, information leaks happen not only on the Linux layer of Android but also on its API level. This section, reports our study with Zhou et al, [23] of an audio public API that gives away one's driving route.

Table 3.5: 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

3.4.1 Driving Route Inference

Android offers a set of public APIs that any apps, including those without any permissions, can call. An example is `AudioManager.isMusicActive`, through which an app can find out whether any sound is being played by the phone. This API is used to coordinate apps' access to the speaker. This seemingly harmless capability, however, turns out to reveal sensitive information in the presence of the powerful Google Maps API.

Consider a GPS navigation app one uses when she is driving. Such an app typically gives turn-by-turn voice guidance. During this process, a zero-permission app that continuously invokes the `isMusicActive` API can observe when the voice is being played and when it is off. In this way, it can get a sequence of *speech lengths* for voice direction *elements*, such as "turn left onto the Broadway avenue", which are typically individual sentences. The content of such directions is boilerplate but still diverse, depending on the names of street/avenues, and one driving route typically contains many such direction elements. These features make the length sequence a high dimensional vector that can be used to fingerprint a driving route, as discovered in our research.

To collect such speech-length sequences, we implemented an audio-state logger into our zero-permission app. Similar to the data-usage monitor, this component is invoked only when the target app is found to be running (Section 3.2.1). In this case, we are looking for the process `com.google.android.apps.maps`, Google's navigation app. Once the process is discovered, our

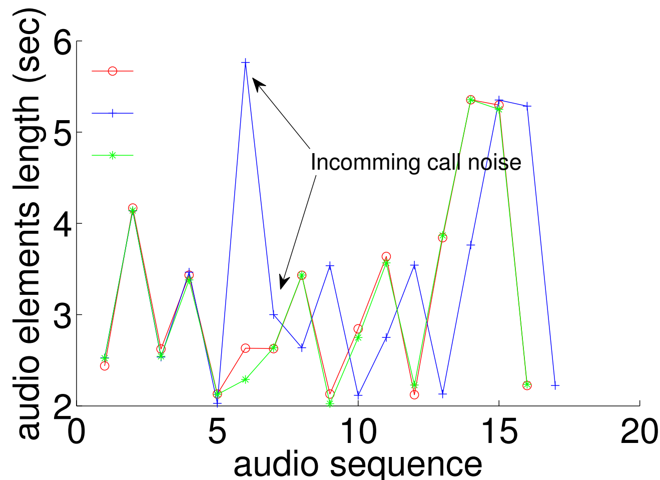

Figure 3.7: Audio elements similarity when driving on the same route

app runs the logger to continuously call `isMusicActive`, with a sampling rate of 50 per second. Whenever a change to the speaker status is detected, it records the new status and the timestamp of this event. At the end of the navigation, the app sneaks out this sequence (see Section 3.1), which is then used to reconstruct individual elements in the speech-length sequence through looking for the timestamp of an "on" speaker state with its preceding state being "off", and the timing for its subsequent state change that goes the other way around.

Using the audio-status logger, we recorded the speech length sequences when we drove from home to office three times at Bloomington. Figure 3.7 shows the comparisons among those sequences. Here we consider that two speech-length elements match if their difference is below 0.05 second, which is the error margin of the speaker status sampling. As we can see from the figure, those sequences all match well, except a spike caused by an incoming call in one trip.

To understand whether such sequences are sufficiently differentiable for fingerprinting a route, we further built an app that simulates the navigation process during driving. The app queries the Google Maps API [42] to get a route between a pair of addresses, which is in the form of a polyline, that is, a list of nodes (positions with GPS coordinates) and line segments between the consecutive nodes. Specifically, we turn on the "*allow mock gps*" option of an Android phone. This replaces the physical GPS with a simulator, which replays the gps coordinates of the route to Google Navigator and records the

voice guidance along the route to measure the lengths of its speech elements.

In our research, we randomly chose 1000 *test routes* in Bloomington with the similar driving time and number of elements as those of the routes for 10 real drives to get their speechlength sequences using our simulator. These sequences were compared with the length sequences recorded from the real routes, as illustrated in Figure 3.8. Here we use a variant of Jaccard index, called *overlap ratio*, to measure the similarity of two length sequences in a normalized way: given sequences s and s' and their longest common subsequence $\bar{s}$, their overlap ratio is defined as $R(s, s') = \frac{|\bar{s}|}{|s|+|s'|-|\bar{s}|}$. Figure 3.8 shows the distribution of the ratios between the sequences in real and test sets (which are associated with *different* routes) together with the distribution of the ratios between the speech-length sequences of real drives and the simulated drives on the same routes. As we can see here, for two different routes, their speech-length sequences are very different (mean: 0.1827, standard deviation: 0.0817), while two identical routes always have highly similar length sequences (mean: 0.6146, standard deviation: 0.0876). Based on these two distributions, we set a threshold of 0.5 for determining when two sequences "match": that is, they are considered to be related to the same route.

Figure 3.8 shows that speech-length sequences can effectively fingerprint their driving routes. A caveat here is that such a sequence should not be too short. Figure 3.9 illustrates what happens when comparing short sequences extracted from real driving routes with those of the same lengths randomly sampled from the 1000 test sequences. We can see here that false positives (i.e., matches between the sequences from different routes) begin to show up when sequence lengths go below 9.

3.4.2 Attack Methodology

Given a speech-length sequence, we want to identify its route on the map. To this end, we developed a suite of techniques for the following attacks: (1) fingerprinting a few "Points of interest" (PoI) the user might go, such as hospitals, airport and others, to find out whether the user indeed goes there and when she goes; (2) collecting a large number of possible routes the user might use (based on some background information) and searching these

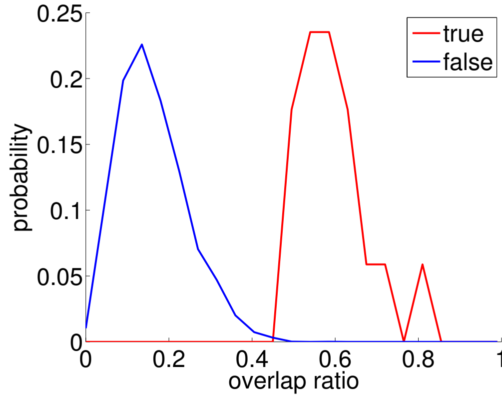

Figure 3.8: Audio length sequence distinguishability

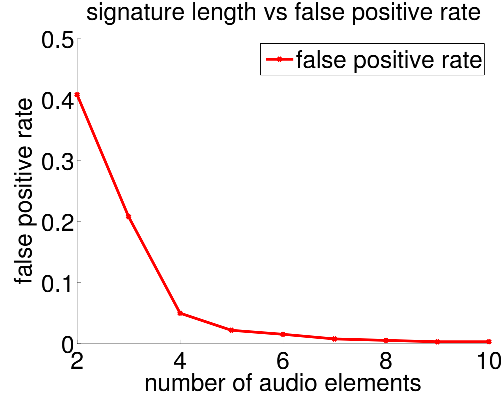

Figure 3.9: False positive rate vs number of audio elements

routes for the target speech-length sequence.

To fingerprint a PoI, we first find a set of start addresses surrounding it from Google Maps and then run our driving-simulation app from these addresses to the target. This gives us a set of speech-length sequences for the driving routes to the PoI, which we treat as a signature for the location. To ensure that such a signature is unlikely to have false positives, the start addresses are selected in a way that their routes to the PoI have at least 10 speech elements (Figure 3.9).

For each speech length sequence received from our zero permission app, our approach extracts a substring at the end of the sequence according to the lengths of a target PoI's signature sequences. On such a substring are the last steps of the route our app observes, which are used to compare with the signature. If the substring matches any signature sequences (i.e., with an overlap ratio above the threshold), we get strong evidence that the smartphone user has been to the fingerprinted location.

Locating a user's driving route on the map can be extremely challenging given the numerous possible routes. For this purpose, we need some background knowledge to roughly determine the area that covers the route. As discussed before (Section 3.2.3 and 3.3), we can geo-locate the user's home and other places she frequently visits when her phone is set to auto connect. At the very least, finding the city one is currently in can be easily done in the presence of Wi-Fi connection. Therefore, in our research, we assume that we know the user's start location or a place on her route and the rough area (e.g., city) she goes. Note that simply knowing one's start and destina-

tion cities can be enough for getting such information: driving between two cities typically goes through a common route segment, whose speech-length sequence can be used to locate the entrance point of the destination city (e.g., a highway exit) on the length sequence recorded during the victim's driving. Furthermore, since we can calculate from the timestamps on the sequence the driving time between the known location and the destination, the possible areas of the target can be roughly determined.

However, even under these constraints, collecting speech-length sequences in a large scale is still difficult: our driving simulator takes 2 to 5 minutes to complete a 10-mile route in Bloomington (necessary for getting all the speech elements on the route), which is much faster than a real drive, but still too slow to handle thousands (or more) of possible routes we need to inspect. Here we show how to make such a large-scale search possible.

Given a known point on the route and a target area, we developed a crawler using Google API to download the routes from the point to the residential addresses in the target area [43]. Each route here comes with a set of driving directions (e.g. *html_instructions*) in text and an estimated driving time. Such text directions are roughly a subset of the audio directions used by Google Navigator for the same route, with some standard road name abbreviations ("Rd", "Dr", etc.).

For each route with text directions, our approach replaces their abbreviations with the full names [44], calls the Google text-to-speech (TTS) engine to synthesize the audio for each sentence, and then measures the length of each audio element. This produces a sequence of speech lengths, which we call a *TTS sequence*. Comparing a TTS sequence with its corresponding speech-length sequence from a real drive (the *real sequence*), the former is typically a subset of the latter. An example is illustrated in Table 3.6. Based on this observation, we come up with a method to search a large number of TTS sequences, as follows.

We first extract all the subsequences on a real sequence under the constraint that two neighboring elements on a subsequence must be within a distance of 3 on the original sequence: that is, on the real sequence, there must be no more than 2 elements sitting in between these two elements. These subsequences are used to search TTS sequences for those with substrings that match them. The example in Table 3.6 shows a TTS sequence that matches a subsequence on the real sequence. As a result of the search,

Table 3.6: Comparison between a Navigation Sequence and a Text Direction/TTS Sequence

Google Navigator	**Real Length**	**Google Direction API**	**Synthesis Audio Length**
Turn left onto south xxx street, then turn right onto west xxx road	4.21	N/A	N/A
Turn right onto west xxx road	2.05	Turn right onto west xxx Road	2.15
Continue onto west xxx Road for a mile	2.53	N/A	N/A
In one thousand feet, turn right onto xxxxx ** north	4.07	N/A	N/A
Turn right onto xxxxx ** north	2.74	Turn right onto xxxxx ** north	2.72

we get a list of TTS sequences ranked in a descending order according to each sequence's overlap ratio with the real sequence calculated with the longest subsequence (under the above constraint) shared between them. We then pick up top TTS sequences, run our simulator on their source and destination addresses to generate their full speech-length sequences, and compare them with the real sequence to find its route.

3.4.3 Attack Evaluation

To determine whether our attack can be successfully used to detect a user's location, we fingerprinted two PoIs in Bloomington, i.e Bloomington Hospital and Indianapolis International Airport (IND) using our driving simulator. For the hospital, 19 routes with at least 10 audio elements on their speech-length sequences were selected from Google Maps, which cover all the paths to the place. The airport has only 4 paths to get in and get out, each having at least 10 audio elements. We first evaluated the false positives of these signatures with 200 routes with similar lengths, and did not observe any false match. Then we compared the signatures with 4 real driving sequences collected by our zero-permission app from the trips to these PoIs (2 for each PoI), they all matched the right routes in the signatures.

Next, to evaluate whether this attack can be used to identify a driving route, we tried to locate 10 speech-length sequences our zero-permission app collected from real drives from a highway exit to 10 random locations in Bloomington. To this end, we randomly selected 1000 residential addresses

from each of the 5 ZIP code areas in the town using the local family website [43] and called the Google Direction API to get the routes from the highway exit (which was supposed to be known) to these 5000 addresses, together with the Google driving routes for the 10 real sequences. Then, the TTS sequences of those 5010 routes were compared with the 10 real-drive speech length sequences collected by our malicious app. For each real sequence, 10 TTS sequences with the highest overlap ratios as described in Section 4.2.1 were picked out for a validation that involves simulating drives on these TTS sequences' routes, measuring their speech-length sequences and comparing them with the length sequences of the real drives. In the end, we identified 11 routes using the 0.5 threshold for the overlap ratio (see Table 3.7). Among them, 8 are true positives, the real routes we drove.

Table 3.7: Route Identification Result. The third column is the highest overlap ratio of a wrong route within the top 10 TTS sequences. FP indicates false positive. All FP routes (actually similar routes) are marked out in Figure 3.10.

Route No.	**result(ratio)**	**Top ratio of a wrong route**	**Notes(error)**
1	found (0.813)	0.579 (FP)	similar route (0.2mi)
2	found (1.0)	0.846 (FP)	similar route (0.5mi)
3	found (0.615)	0.462	
4	missed	0.412	
5	missed	0.32	
6	found (0.846)	0.667 (FP)	similar route (0.3mi)
7	found (0.714)	0.415	
8	found (0.5)	0.345	
9	found (0.588)	0.261	
10	found (0.6)	0.292	

Also, the 3 false positives are actually the routes that come very close to the routes of 3 real-drive sequences (see Figure 3.10), with two within 0.3 miles of the real targets and one within 0.5 miles. Note that in all these cases, the real routes were also found and ranked higher than those false

Figure 3.10: Three FP Routes and Their Corresponding TP Routes. Each FP/TP pair has most of their routes overlapped.

positives. This actually indicates that our approach works very well: even when the real-drive routes were not among the routes we randomly sampled on the map (from the highway exit to 5000 random addresses), the approach could still identify those very close to the real routes, thereby locating the smartphone user to the neighborhood of the places she went.

In this Chapter we saw how Android local unprotected resources can leak private information that can adversary can utilize to infer a user's health and financial information, her identity, her location and her private route. The side-channel attacks describe, leverage the system's erroneous security design at both the Linux (publicly available files) and the framework layer (unprotected API). Unfortunately the story does not end here. Android's security model is flawed also when it comes to safeguarding its external resources as these are defined in 2.3.2. In the next chapter (4) we elaborate on such vulnerabilities and demonstrate real-world attacks on popular external resources.

CHAPTER 4

ATTACKS ON EXTERNAL RESOURCES

As a mobile platform Android is equipped with capabilities to use an assortment of external resources. As we seen on Section 2.2, the Android Security Model only controls the access right on the channel used for communicating with such external resources, such as Bluetooth, NFC, Audio and audio devices, SMSs and network information though sockets. As long as an app acquires a 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 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 we content 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 joint work [24, 25] we studied the risks associated with this coarse granularity of the Android security model. On Section 4.1 we 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. We will refer to this as the `Device Mis-Bonding Threat` or simply `DMB`. After that we will discuss (see Section 4.2) other risks rising from the coarse granularity of the OS's security model and its inherent inability to safeguard the SMS, Audio, NFC and Internet channel.

4.1 Bluetooth Mis-Bonding Attacks

The fundamental cause of the DMB problem is the inadequacy of the Android security model in protecting 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 (Section 4.1.3).

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 [7], 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 4.1.4), 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.

4.1.1 Adversary Model and Targeted Bluetooth Devices

In our research, we conducted a study on this under-researched yet critical security problem. As the first step, our study focuses on Bluetooth healthcare devices, which are becoming increasingly popular in recent years. The security risks we discovered and the new technique we built are extended to other types of external devices, as we will see on Section 4.2.

We 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, our study focuses on Bluetooth devices. Specifically, we analyze four popular healthcare devices. All of them except the iThermometer are FDA approved Class II medical devices [45], 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* [12] is one of the most popular activity monitoring systems, which has been used in over 120 clinical studies [46]. 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* [13] 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 [45]. 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 [45].

- *Entra Health System MyGlucoHealth Blood-Glucose Meter* [14] 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 [47]. Along with FDA, this device is also approved by CE[2] and is fully HL7[3] compliant [14].

- *iThemometer* [48] 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 the 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, we show that such data becomes extremely vulnerable to the DMB threat.

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

[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.

4.1.2 Data-stealing Attacks

In our research, we 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 we describe how we addressed such technical challenges and designed end-to-end attacks on real devices.

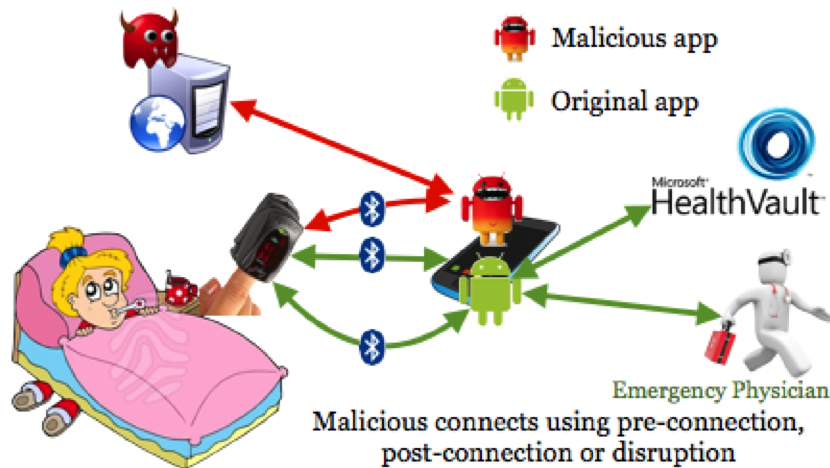

Figure 4.1: Data-stealing Attack

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, 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 [3]. 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 our research, we adopt 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 we 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 collec-

tion. The approach makes the attack less stealthy but more reliable in getting the data from the target device.

Here we elaborate how we utilize these techniques to launch data-stealing attacks on the healthcare devices.

Attack Implementation

In our study, we execute the data-stealing attacks on all four healthcare devices. To prepare for the attacks, we analyzed the code of these devices' official apps and their Bluetooth traffic captured using `hcidump` [49] to facilitate our 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 we 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, our 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, our 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, our 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, we designed the malicious app to check 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

We run the malicious app on a Nexus 4 development phone running JellyBean (4.2), together with all target devices' apps. We 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. Our study shows that the malicious app is often successful in capturing this window. The experimental results are presented in Table 4.1.

For the Bodymedia armband device, we 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. We achieve 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. We also tried the disruption strategy, which also worked, allowing our app to discontinue the official app's connection and get the health data. A problem with this attack strategy, as discussed before, is that the legitimate connection needs to be interrupted, which could be noticed by the user.

Table 4.1: 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. We evaluated `getRunningTasks` (the strategy we 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. We ran the app using each strategy independently for 10 minutes. The average power consumption of the strategy under scrutiny is illustrated

Table 4.2: Average power consumption over 10 minutes per surveillance technique using PowerTutor[3].

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 4.3: Average power consumption over an hour. Comparison between our surveillance technique and 2 popular applications using PowerTutor[3].

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

in Table 4.2. As depicted, the one we decided to adopt (`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). We further compared the power consumption of this strategy with that of two popular apps, as described in Table 4.3. As we can see here, our 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 we used for the power consumption evaluation provides accurate measurement for a very limited number of phones and rough measurements for all Android phones. This rough measurement suffices for this evaluation.

4.1.3 Data-injection Attacks

In addition to the threat of the data-stealing attacks, we found that the presence of the insider, the malicious app, on the phone also makes it possible to inject fake data into the device's official app and its online account for the user. This part will elaborate on our attack strategy, technical challenges we had to overcome and the evaluation of our implemented attack.

Attack Strategy

The data-injection attack works as follows:

1. The malicious app first uses its Bluetooth permissions to gain access to the channel and collect part of the bonding information for the target device. Then it delivers the information to the adversary either using the INTERNET permission which will allow it to use the network sockets or by invoking the browser with the adversary's remote domain and the stolen data encapsulated into it (e.g with JSON).

2. The adversary clones the device using the bonding information (MAC address, UUID and device name) and places the clone in the neighborhood of the original device or other places where the user may come close. With a standard Class 1 Bluetooth device, the clone can stay as far as 100m from the phone.

3. When the user gets into the clone's Bluetooth transmission range, the malicious app resets the link key (in the presence of secure communication) by unpairing the phone from the original device and pairing it with the clone, then it invokes the target app (if it is not already running). The clone then talks to the official app and transmits falsified data to the app.

4. To make the attack stealthy, the malicious app can choose to pair the phone back with the original device after the attack, thus the phone user will not notice that her device has been unpaired. This succeeds most of the time, as the PIN for most Bluetooth devices is either "0000" or "1234" [50]. It should be noted that the original Bluetooth device does not need to be discoverable for pairing, as long as its MAC address is known. Moreover, most of the new devices with Bluetooth 2.1+ use Secure Simple Pairing (SSP) [51], which does not need any user intervention for pairing.

To make this attack happen, we need to address a few technical challenges, particularly, how to clone the original device, how to stealthily reset the link key when secure connections are in use, and how to connect to the spoofed device in the presence of the original one. Here we elaborate these problems and our solutions.

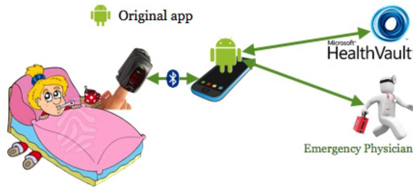

Figure 4.2: Normal Scenario

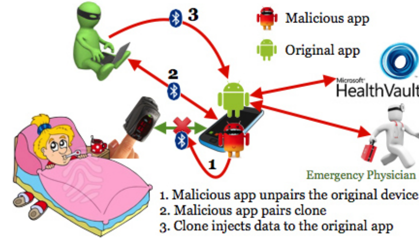

Figure 4.3: Adversary injecting fake data

Addressing Technical Challenges

To clone a Bluetooth device, all an attacker needs is the target device's name, MAC address and UUID. As discussed before, such information can be easily obtained by calling Android's `getBondedDevices()` method of `BluetoothAdapter` class [52], which gives a list of devices paired with the phone and their bonding data. In our experiment, we ran SpoofTooph [53] on a Linux laptop that masqueraded the original device using its name, MAC and class (i.e., UUID). The spoofed device can be placed wherever the user may come close if the official app of the original device does not have a soft button for activating its Bluetooth connections. An example here is the Bodymedia armband in automatic connection mode. Otherwise, the adversary needs to set it up in the vicinity of the original device. The presence of two devices with the same name, MAC and UUID is hard to detect, as Bluetooth scanners show only one of them. Also, the spoofed device can be much further away from the user's phone than the original device (even outside the door) and still ensure the success of the attack, which we elaborate later in this section.

What gives trouble to this data-injection attack is the link key stored in the operating system that the malicious app cannot get. The link key is used to encrypt Bluetooth packets when the device's official app invokes `createsecureRfcommSocket`. Without knowing this piece of information, the clone cannot talk properly with the app. This is not always a problem: an app can connect to its device without encryption protection (through `createInsecureRfcommSocket`) and even some devices that offer encryption may not provide adequate security. Consider the Nonin pulse oximeter as an example. Its app first tries secure connections, but, if this attempt fails, the app automatically switches to the insecure channel to ensure that the

communication can still go through. However, for the device that always sticks to the secure channel, an attacker needs a way to circumvent the defense provided by the link key.

Here is our strategy: if we cannot get the link key, we can simply replace it, setting it to one known to our spoofed device. Given the `Bluetooth` and `Bluetooth_ADMIN` permissions, the malicious app can easily unpair the phone from any Bluetooth device by calling the API `IBluetooth`, which removes the current link key shared with the device. To enable the official app to talk to the clone, however, we need to pair it with the spoofed device. This operation sets a new link key for the clone, which requires the user to enter a PIN to authenticate her phone to the device. (Note that the PIN here is for device-device authentication, not app-device authentication). The PIN itself is not a big issue, as the clone will accept whatever it receives.[4] The problem is that there is no public Android API for programmatically entering the PIN. The phone user's intervention seems inevitable.

A close look at Android source code, however, shows that the OS has a hidden interface that allows entering a PIN programmatically. All we need to do here is to find a way to use it. The APIs provided by Android communicate with different system services through IPC (Inter-Process Communication) calls 2.4, based on those services' interfaces specified by Android Interface Definition Language (AIDL). Actually, a method `setPin()` under Android `IBluetooth` API (Android's private API for Bluetooth services) can be used to programmatically input the PIN. The problem is that this interface is not open to ordinary apps. In our research, we managed to get this interface by using a technique [54] that retrieves the AIDL description of the method from the Android source code and then compiles it together with the code of the malicious app. As a result, the interface becomes visible to the app. Through the interface, a PIN can then be automatically entered for a Bluetooth pairing. To avoid showing any user interfaces that might arouse suspicion from the phone user, the app performs this pairing operation when the screen is off, which can be determined without requesting any permission [23]. We provide a demonstration of this attack on a web page [55].

Another technical challenge comes from some apps' soft buttons, which needs to be clicked by the phone user to initiate their Bluetooth communi-

[4]After the attack, if the malicious app wants to restore the pairing with the original device, oftentimes a default PIN (0000 or 1234) works just fine [50].

cation with the devices. All four device apps we studied can be set in this "button" mode. In this case, automatic triggering of those apps' Bluetooth connections is difficult. Therefore we have to place the clones somehow close to their original devices (within tens of meters), in hopes that when the user starts running the official apps, they will mistakenly talk to the clones. This turns out to be pretty realistic: we found that we can set the spoofed device in a way that it almost always wins this connection race, as elaborated below.

A Bluetooth device typically waits for a smartphone to initiate a connection. During this waiting process, it continues to switch between two modes, *page sleep* where it sleeps to save power [56] and *page scan* where it wakes up to look for connection requests. This process is called paging. Let T_{sleep} be the time interval between two scans and T_{scan} the duration of a scan. Obviously, a larger T_{scan} and a smaller T_{sleep} lead to more power consumption, but are more likely to timely respond to the phone's connection requests. According to Bluetooth specifications [56, 57], these parameters are supposed to be set such that $11.25ms \leq T_{sleep} \leq 2.56s$ and $10.625ms \leq T_{scan} \leq T_{sleep}$. For most Android external devices, including all those used in our research, their parameters are chosen for power saving and cannot be modified by the user. The adversary, however, can be more aggressive. In our research, we used the Linux configuration tool `hciconfig` to set these parameters for the Bluetooth dongle on our attack laptop to $T_{scan} = 11.25ms$ and $T_{sleep} = 1.28s$, and ran this spoofed device against all four healthcare devices. We found that the clone almost always won such connection races. This happened even when the original device was more powerful than the clone in terms of radio signal strengths. For example, Nonin pulse oximeter [13] is a Class I Bluetooth device, with a $100mW$ radio and a range up to $100m$, whereas clone was Class II with a $2.5mW$ radio and a range up to $10m$; even under such a disparity, the clone always managed to first connect to the phone even behind a wall and $7m$ away.

Effectiveness Evaluation

In our research, we implemented the attack and evaluated it on our NEXUS 4 phone (Android 4.2, 1.5GHz quad-core CPU and 2GB memory) and the aforementioned medical devices. The experiments were conducted in the presence of both the original devices and their clones (a VM with Intel Core

Table 4.4: Data-injection attack launched 1ft and 20ft away from the victim's phone, with the original device touching the phone. In both cases, the experiments were repeated 100 times.

Distance of cloned device	**1 ft**	**20 ft***
Number of observations	100	100
Distance of original device	0 ft	0 ft
No. of times original device responded	0	0
No. of times cloned device responded	100	100

* with a wall in between

i7 CPU (shared), 1GB memory, a Bluetooth 2.0 dongle), though this is unnecessary when these devices' apps are in the automatic connection mode that enables the malicious app to trigger them to automatically connect to the clones whenever the phone user comes close to the spoofed devices. To make our attacks realistic, we deliberately placed the original devices closer to the phone (0 feet) than the spoofed ones (20 feet away, with a wall in-between). Among all 100 executions of the official apps, the phone was always first connected to the clones, despite their larger distance from the phone. The results are presented in Table 4.4. Note that in the presence of secure connections, the phone cannot talk to the original device after the reset of the link key. When this happens, however, the phone will get a notification of connection failure if the response from the original device comes first. In our experiment, we never observed such a notification: each time, the phone always smoothly established a connection with the clone.

In all the tests, our app first unpaired the phone from the original device and then paired it with the clone, using a random PIN as an input. All these unpairing and pairing attempts succeeded. Once a connection was established, we observed that the spoofed device easily dumped fake data into the official app, this data was displayed on the phone or the web page for the user's account. A demo is posted online [55].

4.1.4 Measuring the DMB threat

As discussed before, the DMB problem comes 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.

Table 4.5: Sampled apps

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

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. Our study, for which we give details below, 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 (Sections 5.2,5.3).

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 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 4.4.

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,

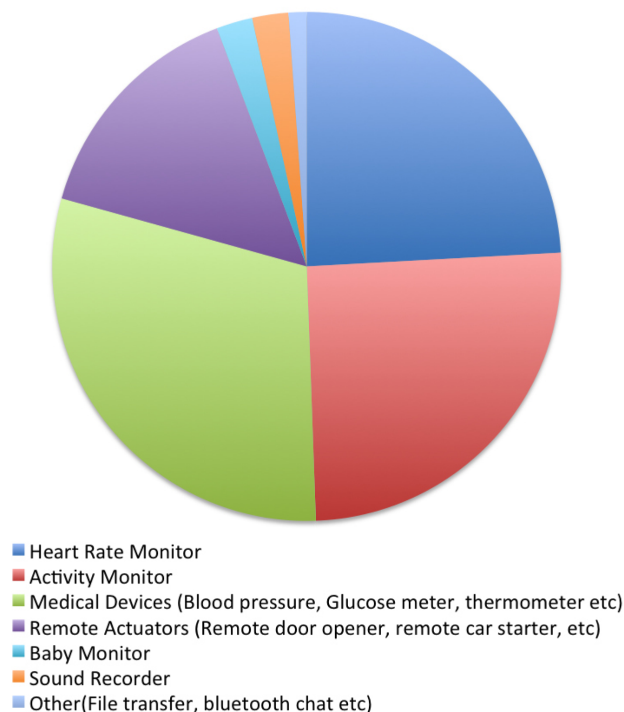

Figure 4.4: Classifications of the sampled apps. Some of them collect information in multiple categories.

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 4.6) 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 [58, 59, 60]) 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 4.6) 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.

4.1.5 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 4.6 summarizes our findings. The 48 apps we removed automatically either did not have any suspicious APIs or 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.

Our 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 renders such device-device authentication insufficient for protecting private user information.

Table 4.6: 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

Our 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 (even the integrity 4.1.3) of the information communicated through that channel. These findings led us to study more such channels of communication with external resources which we report on the next Section (4.2).

4.2 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 4.1 and related studies on the Internet (local socket connections) [22] 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 demonstrated that the SMS, Audio and NFC channels are equally under-protected, exposing private

Table 4.7: 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.

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 using external resources.

4.2.1 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 4.7 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 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.

4.2.2 Study Results

Our analysis shows that most external resources we 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 [61]. 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 [15], 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 [62, 63]. 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[5]. 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[6]. 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

[5]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.

[6]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.

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.

This ends our discussion for this Chapter. We have seen side-channel attacks stemming from unprotected local resources on Android and confused deputy attacks due to the coarse granularity of the Android Permission Model. We further seen the prevalence of affected applications highlighting the magnitude of the of the threat. The natural question to follow is: How do we protect the system and legitimate applications against such attacks? On the next Section 5 we will discuss some strategies to tackle the side-channel attacks, a defense mechanism called Dabinder which can protect against Bluetooth mis-bonding attacks and a a system called SEACAT which places control MAC and DAC access control on the channels of communication with external sources.

CHAPTER 5

DEFENCE: GUARDING THE VULNERABLE LOCAL AND EXTERNAL ANDROID RESOURCES

Until know, we have been discussing how the Android Security Model is incapable of protecting its local and external resources, which lead to a suite of attacks compromising a user's privacy 3, and the confidentiality, and integrity in some cases, of the data being communicated with external sources 4. The natural next step after the exposure of a vulnerability is to try finding ways to protect against possible attacks. In this Section we will address this challenge, and discuss solutions we designed and implemented [23, 24, 25].

5.1 Mitigating the Side-Channel Threats on Local Resources

Given the various unprotected local resources on Android, the information leaks we found 3 are very likely to be just a tip of the iceberg. Finding an effective solution to this problem is especially challenging with rich background information of users or apps gratuitously available on the web. To mitigate such threats, we first take a closer look at the attacks discovered in our research. The ARP data (see 3.3) 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 (see 3.4) 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 3.2), 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, we report our approach on mitigating the threat deriving from

the statistics availability, while maintaining their utility.

5.1.1 Mitigation Strategies

To suppress information leaks from the statistics available through `tcp_rcv` and `tcp_snd`, we can release less accurate information. Here we 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 [33, 64]. 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 5.1.2). 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`.

5.1.2 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 [21]. 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 5.1.1) 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 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

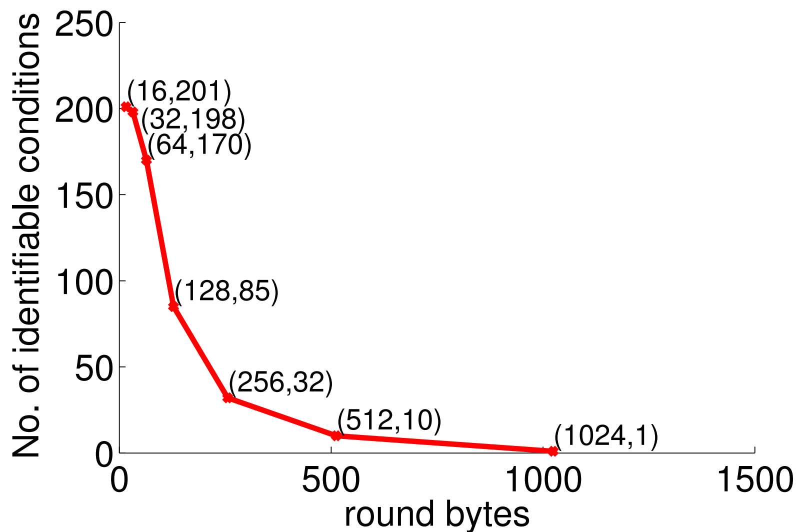

Figure 5.1: Effectiveness of round up/down mitigation technique

deliberately kept the API interface unchanged so existing data monitor apps can still run.

5.1.3 Defence Evaluation

To understand the effectiveness our technique, we first evaluated the round up and round down scheme using the WebMD app. Figure 5.1 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.

5.2 DABINDER: Thwarting the DMB Threat

Our security analysis of existing Android Bluetooth devices 4 shows that most of them are completely unprotected from mis-bonding (or DMB) attacks. Although theoretically each device manufacturer can provide its own app-device authentication to address the problem, this requires upgrading not only its software (the app) and also the hardware (the external device), thus making the device more expensive. Also, this case-by-case fix renders the quality of security protection for different external devices hard to control. A better solution is to enhance Android to provide an OS-level access control that bonds each external device to authorized app(s). This way, even if a malicious app has the privilege to access the channel (BLUETOOTH and BLUETOOTH_ADMIN permission), it won't be able to compromise the confidentiality of the data being communicated between an external device bonded with a legitimate app. In this section, we elaborate our design and

implementation of such a technique, called *Dabinder*, and evaluation of its efficacy. *Dabinder* assumes that the underlying Android OS is not compromised. The protection mechanism (namely *Dabinder* is developed on the framework layer of Android OS.

5.2.1 Overview

We built Dabinder to effectively control app-device bonding in both pairing and communication stages and to minimize user involvements in setting access-control policies. Here we describe a high-level design that achieves these two goals.

Figure 5.2 illustrates the architecture for Bluetooth socket communication on Android 4.2, which includes Dabinder components (Reference Monitor and Binding Policy database). To pair a device programmatically, the system calls `setPairingConfirmation` and `setPin` or `setPassKey` of `Bluetooth Device`. To unpair a device, the app uses the API `removeBond`. Alternatively, it can invoke the settings program to control the Bluetooth adapter. In both cases, an IPC request needs to be sent to `AdapterService` to control the Bluetooth device. Once a bond (pairing) is established, the app can make a socket connection to access the device. To this end, again it first needs to talk to the `BluetoothAdapater`, to get a list of paired devices. From this list, the app identifies the target device (MAC) and further requests a socket through the object `BluetoothDevice`. This request is also delivered using an IPC, through the `IBluetooth` interface, to `AdapterService`, which creates the socket for the connection.

In our design, the whole security mechanism is built into the `Adapter Service`, including a component that controls socket establishment (within an authorized app-device pair[1]) and one that manages the unpairing operation (which can only be performed by an authorized app[2]). Such access controls are based on a set of security policies that unambiguously bonds a device to its authorized app, which are generated automatically by the system from what is observed from the phone's Bluetooth operations.

[1] App-device pair is our terminology for creating a bond between an app and Bluetooth device as opposed to conventional Bluetooth pairing which creates a bond between the phone and the device.

[2] Authorized app: The app that has already established bond to the device.

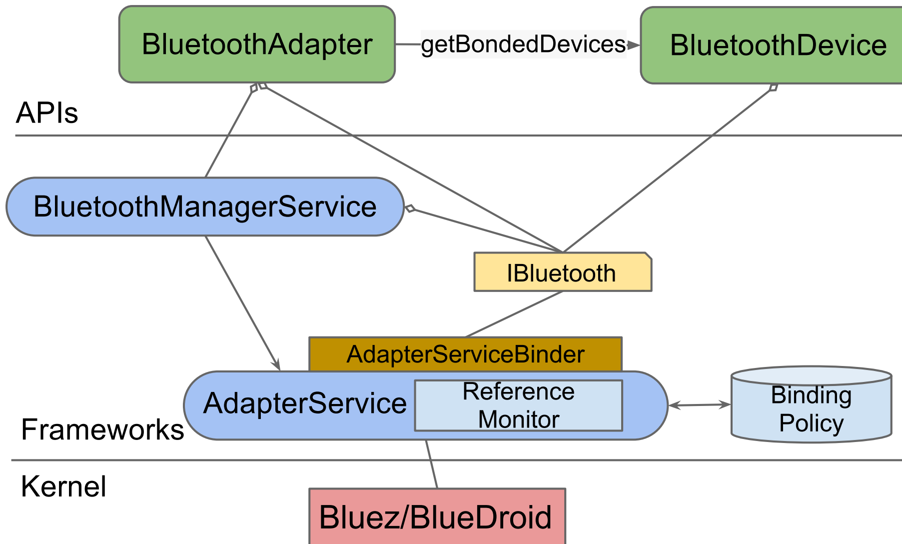

Figure 5.2: Bluetooth Subsystem and our defense mechanism: *DaBinder* is built into AdapterService and checks the interaction between apps and Bluetooth devices. It only allows authorized app to access Bluetooth device and keeps bonding policy in a secure storage. *Reference Monitor* and *Bonding Policy* blocks (both shown in light-blue) constitute Dabinder.

Now let's see with an example, how exactly Dabinder operates: Once a the Bluetooth device is activated, it is paired with its authorized app by the phone user. This pairing operation is observed by Dabinder, which then generates a *bonding policy* that associates each device (name, MAC and UUID) to its official app (that is, its Linux user ID or UID). Whenever Android receives a Bluetooth socket-connection request from an app, the policy enforcement mechanism checks whether the app is associated in the bonding policy to the device it is trying to talk to: if the app is not on the device's bonding policy, the request is denied; otherwise, it is allowed to proceed. In this way, our mechanism defeats the data-stealing attacks (see 4.1.2). Also, Dabinder runs an unpairing controller to manage the operations to dissolve a pairing relation between the phone and a device: in the absence of the bonding policy between the device and the app that requests such an operation, the app is considered to be unauthorized and its unpairing attempt is stopped. As the data-injection attack 4.1.3 is contingent on resetting the link key for the phone-device communication, it cannot work without unpairing the phone from the original device. Therefore, such an attack cannot go through.

5.2.2 Design and Implementation

Here we present the detailed design of Dabinder, which we implemented in our research on a Galaxy NEXUS 4 phone with Android 4.2. As described before, our design includes mechanisms for generating and maintaining security policies, and for enforcing these policies during phone-device pairing and app-device connection establishment. All these mechanisms were implemented within `Adapter Service`.

Critical to Dabinder's mission is the security policies on the legitimate bond between an app and an external device. Such a policy can certainly be manually specified, but it is highly desired that it can also be automatically generated, without the user's intervention if she prefers to do so. In our design, this policy-identification operation is performed within `AdapterService`, when our policy enforcement mechanism inspects a pairing request and its follow-up connection request: if an app is the first one to make a socket connection to the device after the device is paired with the phone, our mechanism automatically adds this app-device relation to a policy database as a new bonding policy.

To securely and persistently maintain these policies in the system, `Adapter Service` keeps a Bluetooth MAC Address and `UID` mapping in the `Settings.Secure` key-value storage, which is persistent and read-only to the apps, and can only be modified by the phone user or a system program. The user can manage these policies from `BluetoothManagerService` through a user interface built upon two functions exposed by `AdapterService`, `addDevApp` and `removeAppDev`. In particular, she can explicitly declare an exemption policy for a device, thus allowing it to be accessed by any app.

Furthermore, to communicate with an external device, an app needs to establish a connection with it through a Bluetooth socket. Such a socket is created by the system, through a call to the `BluetoothDevice` APIs: `createRfcommSocket`, `createRfcommSocketToServiceRecord`, `createInsecureRfcommSocket` or `createInsecureRfcommSocketToServiceRecord`. A straightforward solution here is to instrument these APIs in order to control the creation of Bluetooth sockets. The problem is that such mediation actually happens in the user land, inside individual apps' address space. As a result, there is no guarantee that it cannot be circumvented. Also, such a policy compliance checking needs an additional IPC to `AdapterService`,

to get the policies from the system. Instead, in our research, we modified `AdapterService.connectSocket`, a system function all these APIs have to invoke, for policy compliance checking and enforcement. Whenever the function is called, it first searches the policy database for the device according to its MAC address. If the device is found, our enforcement mechanism continues to look for its related bonding policies. In case, the app does not appear on any of them (i.e. the device has been connected before and is not exempted from the bonding protection), we consider that a policy violation is detected and the request is denied. Otherwise, `connectSocket` returns a socket and allocates the corresponding resources, such as file descriptors for the connection.

This policy enforcement is implemented on the Android framework layer. Apparently, the app including native code such as `createBondNative` and `removeBondNative` may still touch the Bluetooth device on the Linux layer, as illustrated in Figure 5.2. In our research, we inspected the Bluetooth interface on Linux and found that it is actually included in the Linux group `bluetooth`. For the app with `BLUETOOTH_ADMIN` and `BLUETOOTH` permissions, it can get into the groups `net_bt_admin` and `net_bt`, but not `bluetooth`. As a result, it will not be able to directly access the Linux Bluetooth resources, even through its native code, due to the Linux access control. Under all circumstances, the app needs IPC calls to transfer the execution to a system process in order to use kernel resources. Therefore, we conclude that our protection mechanism cannot be circumvented even by the native code.

Unpairing control. Despite policy identification, policy management and connection control, Dabinder has to control the "unpair" operation too. In the presence of secure Bluetooth communication, a malicious app needs to first unpair the phone from the original device before pairing it with the clone to reset the link key. To prevent this unauthorized unpairing, Dabinder interposes on the function `removeBond` within `AdapterService`. Whenever an unpairing request is received from the `IBluetooth` interface, our mechanism checks it against the bonding policy retrieved from the policy dataset: if the app that sends the request is not the authorized one on the policy, this request is denied. Alternatively, we can pop up an interface on the phone to alert the user to the unpairing request and allow the operation to proceed with her permission.

A problem here is that some devices do not use secure Bluetooth communication, which enables the spoofed device to talk to the official app even without knowing the link key. Fortunately, our measurement study shows that most of devices collecting sensitive user data do support encrypted communication (Section 4.1.4), though some of them can also automatically switch to the insecure one when the secure connection fails. To address this issue, Dabinder provides an optional policy through which the phone user can require that any communication with a certain device must be encrypted. In case, the policy is violated (that is, a device is switching to the insecure communication), we can choose to stop the communication and alert the user for further instructions. This happens in `AdapterService`, within the method `connectSocket` which checks whether `SEC_FLAG_AUTH` and `SEC_FLAG_ENCRYPT` on `flag` are set.

5.2.3 Evaluation

We evaluated our implementation to understand its effectiveness in protecting the communication with external devices and its performance impacts on the phone's normal operations. All the experiments were conducted on the Galaxy Nexus phone with a Dual-core 1.2 GHz Cortex-A9 and 1GB memory, Android 4.2 operating system and BlueDroid stack.

To understand the effectiveness of our approach, we ran it against all the data-injection and data-stealing attacks discussed in Section 4.1. All these attack attempts were thwarted. Specifically, for all the data-stealing attacks, Dabinder stopped the malicious app from making socket connections to the target device, as these connections violated the policy it automatically detected during the pairing stage of the phone. When it comes to the data-injection attacks, our implementation blocked all the attempts to unpair the phone from the devices and therefore defeated the attacks when the secure communication was in use. Also, our approach denied the establishment of a socket for insecure connection required by the Pulse Oximeter app.

We further evaluated the performance of Dabinder, comparing the execution times for establishing sockets and unpairing a device with and without its policy inspection and enforcement. Specifically, we measured the performance of a set of functions using the code instrumented before and after their

Table 5.1: Dabinder performance evaluation. (mean / sd)

Functions	Original	Dabinder	Delays
BluetoothSocket	0.0317 / 0.0059 ms	0.0353 / 0.0153 ms	0.0036 ms
connectSocket	63.1670 / 14.7098 ms	86.5152 / 14.2201 ms	23.3482 ms
removeBond	0.5319 / 0.1863 ms	0.5493 / 0.1822 ms	0.017ms

executions. The results are illustrated in Table 5.1.

Here, `BluetoothSocket` creates a Bluetooth socket, `connectSocket` builds a socket connection and `removeBond` unpairs the phone from a device. As we can see from the table, for all these functions, the delay is mainly caused by `connectSocket`, about 23ms on average. It should be noted that only 8 devices can have Bluetooth connection simultaneously to a smartphone. Moreover, a phone cannot have more than 30 RFCOMM sockets active at the same time, as RFCOMM have only 30 channels [65]. For external devices, they typically can accommodate only one or two connections. Actually, all data from the device is downloaded through a single Bluetooth connection. Therefore, this 23ms delay will not bring in any noticeable inconvenience to the phone user.

Dabinder is a solution tailored to the Bluetooth channel. As Android allows apps to utilize information from other external resources, such as NFC, Audio, SMS and Internet, solutions are needed to warrant the utility of those apps while preserving the confidentiality of the data being communicated through thsoe channels. Next we discuss such a system, which uniformly addresses this challenge for all the known channels of communication with external resources.

5.3 SEACAT: DAC and MAC on External Resources

Our studies on Bluetooth, SMS, Audio and NFC 4, and prior findings on Internet [22] emphasize the urgent need to enhance Android access control

to protect external resources. In this section, we present the first uniform design for this purpose. This 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. We implemented SEACAT on Android 4.4 with an SEAndroid enhanced kernel 3.4.0.

5.3.1 Design Overview

Our 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. However, achieving this goal is by no means a smooth sail. Here are a few technical challenges that need to be overcome in our design and implementation.

- *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[3]. How to integrate SEACAT into the current Android in the most efficient way is challenging.

To address these challenges, we have come up with a design that integrates policy compliance checks from both the framework and the kernel layer, and

[3]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.

enforces MAC and DAC policies within the same security hooks (Figure 5.3). 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 approval. Those policies and the security contexts of the labeled resources are uploaded

to the DAC base and the mapping tables respectively.

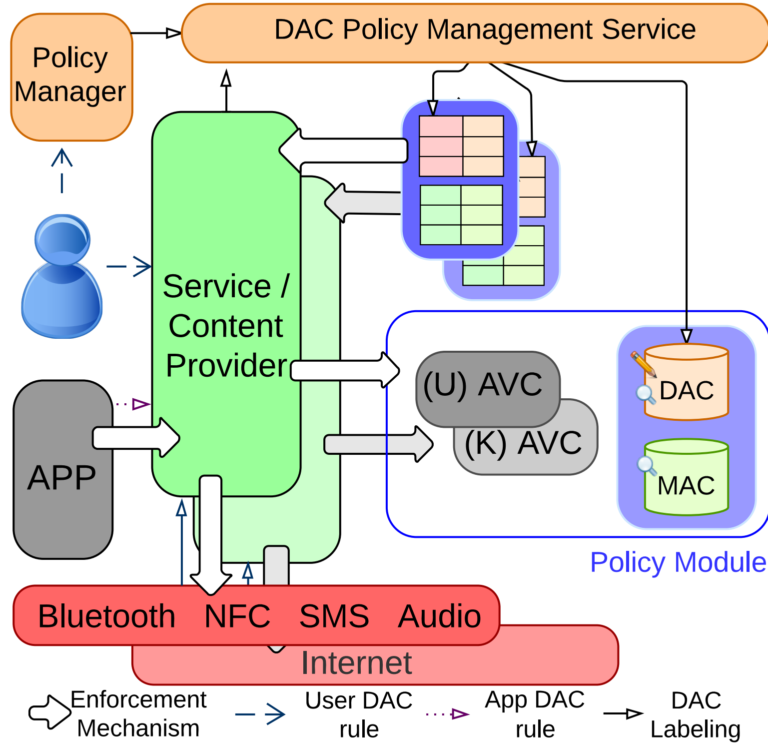

Figure 5.3: *SEACAT* architecture

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, our design makes sure that even when it is misconfigured, the adversary still cannot bypass the MAC protection in place. Finally, we assume the presence of malicious apps on the user's device, with proper permissions to access all aforementioned channels.

5.3.2 Policy Specification and Management

To control external resources, we first need to specify the right policies and identify the subjects (i.e., apps) and objects (e.g., Bluetooth glucose meter,

the Chase bank, etc.) to apply them. This is done within the policy module and our 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 our research, we 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 we 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, we provide 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, we 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. Our implementation 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.

We further modified `libselinux`, which is used by Zygote, to assign the appropriate security context to the process forked for an app. Our 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 5.1.2).

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 our implementation, we modified a few system apps and services to accommodate this mechanism. For Bluetooth, we changed `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. In our implementation, we instrumented 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[4]. Use of Internet resources should be specified by the app. For example, those using ADB-level services [22] can state the local IP address and their services' port numbers to let our system label them.

Pertaining Audio, we 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

[4]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.

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.

5.3.3 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 5.4). 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. In our research, we put the enforcement of DAC policies 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, we implemented a 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, we built another mapping table for the DAC policy[5]. This table is synchronized automatically

[5]Note that we did not build the table for MAC here, since SELinux already has

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 we created for manipulation and retrieval of information from the memory buffers and exposed through libselinux to the rest of the system, are listed within Table 5.2.

Table 5.2: *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`. In our research, we wrapped this SEAndroid function, adding program logic for retrieving an object's security context from the mapping table. 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 pa-

a table for enforcing MAC policies on IPs. Also, all other channels are enforced on the framework layer.

rameters. 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 we had to interject the security hooks in the appropriate functions of the framework or the kernel. This instrumentation allows us to perform our 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.

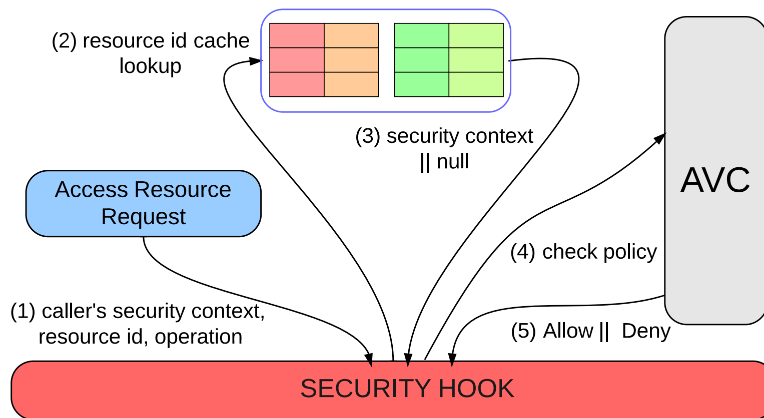

Figure 5.4: *SEACAT* Policy Compliance Check

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. In our implementation, we inserted 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. What we did is to turn to Binder, which proxies the inter-process call (IPC) from the real caller app. Specifically, our 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 we instrumented the actual connection attempt, which uses 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.

What about NFC? For the broadcomm chip on Google Nexus 4 devices, the NFC stack has been implemented on the framework/library layer through `libnfc-nci`. As a result, all our 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. In our research, we instrumented 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 ex-

isting 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 5.5 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

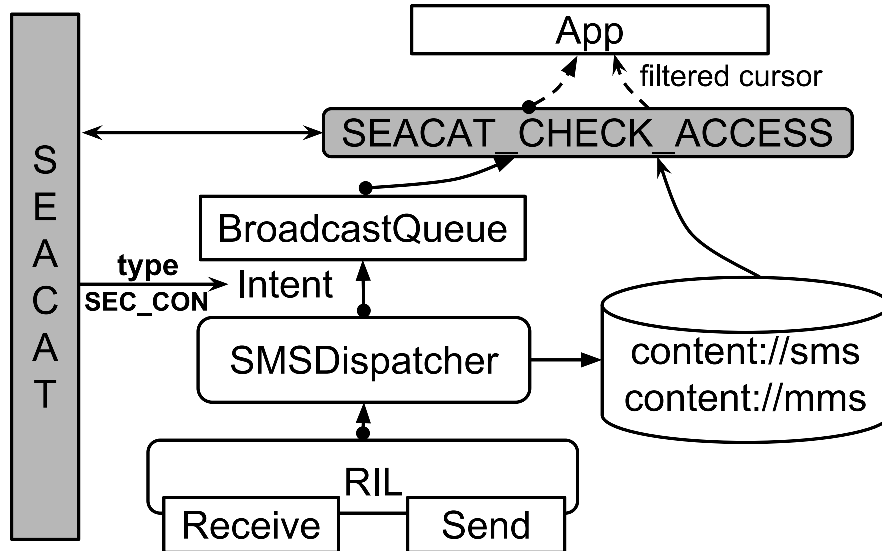

Figure 5.5: *SEACAT*'s enforcement on SMS: *SEACAT* labels each sms message intent and checks if an app can access the message before delivering the intent to the app. Also `SEACAT` filters the sms content provider query results according to the security context of the app

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`[6], 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.

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

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

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.

5.3.4 Evaluation

In our research, we evaluated the effectiveness of SEACAT against all existing threats to Android external resources and measured the performance overhead it introduces. Our study was performed on a pair of Nexus 4 phones with Android 4.4 (android-4.4_r12), kernel KRT16S, with the 3.4 kernel (android-msmmako3.4kitkatmr0): one installed with an unmodified OS to serve as a baseline, and the other with the SEACAT-enhanced kernel. Following we report what we found. The video demos for this study can be found online [61].

Firstly we want to make sure that SEACAT actually solves the problem and can successfully safeguard all known external resources. Table 5.3 presents 5 known threats to external resources used in our research, which include collection of data from iThermometer through Bluetooth misbonding (see Section 4.1), unauthorized use of an ADB proxy based screenshot service through local socket connections [22], 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) 4.2. 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 [22] and Sections 4.1

and 4.2.

Table 5.3: 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 our policy management service, we found that only the official app of iThermometer, which was assigned to an authorized domain, was able to get data from the device [61]. 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` [61]. Similarly, the screenshot attack reported in prior research [22] 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 5.3.2). Also, we 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 we manually injected such a policy into our 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, we 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 5.4 shows examples of the operations used in our research. In the experiments, we conducted 10 trials for each operation to compute its average duration.

Table 5.4: 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, we 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, we just looked at the execution time of `BluetoothSocket.connect`. Regarding SMS, we measure the time from when a SMS message is received (`process-MessagePart`) 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 `setForegroundDispatch` was called. For the Audio channel, we recorded the time for the call `AudioRecord.start Recording` to go through.

The results of this evaluation are presented in Table 5.5. 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 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.

Table 5.5: 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

In this Chapter we have seen some mitigation strategies for attacks stemming from local resources, a defense solution to tackle the unfettered use of the Bluetooth channel and lastly we elaborated on a more uniform approach taking advantage of the SELinux infrastructure incorporated to AOSP since version 4.3, to protect all known channels of communication with external Android resources. In the last Chapter 6 we will summarize our findings and scrutinize over the ramifications and limitations of our defense strategies in safeguarding Android's local and external resources.

CHAPTER 6

CONCLUSION AND DISCUSSION

Since its advent in 2007, Android dominated the mobile world. Its open source nature lead to its adoption by major hardware and telecommunication companies offering an unprecedented experience to its users. Built on top of Linux, it inherits a well studied and functional paradigm in terms of its utility and security. Nevertheless, its model for supporting third-party apps to offer creative and novel functionalities, generated both new requirements and new challenges for its developers. Android seems to take security seriously and it employs a number of security models, featuring Linux's DAC access control for local resources, its permission model and most recently SELinux on Android for MAC access control. Nevertheless, as seen in Chapter 3 and Chapter 4, a number of vulnerabilities exist that can compromise the system. Such vulnerabilities can stem from its ineptitude to sufficiently protect its local and external resources.

Firstly we saw (Section 3) how the gap between Linux design and the actual smartphone use and the gap between assumptions on publicly available local resources and evolving app design and functionality, entail grave user-privacy leaks. Some local resources, while being publicly available to processes on a stationary machine is of minimal privacy risk, the same resources result in critical privacy violations when available to all users/apps on a mobile platform. For example network usage statistics can be used to infer a user's medical condition, her identity and financial preferences. In addition, information regarding the MAC addresses of access points the smarpthone is connected to, can leak a user's location. Furthermore, Android's rich API, employs the permission model to control access to sensitive information. However, there exist information that at a first look doesn't seem privacy critical and thus are not protected by a permission. Nonetheless, an adversary can use such information to infer a user's driving route. The attacks possible due to unprotected local resources that we have seen, are mere examples of what

an adversary can do by leveraging access to seemingly innocuous resources. These findings call into question the design assumptions made by Android developers on public local resources and emphasize the demand for new techniques to address such privacy risks. To this end we discussed solutions (see 5.1) focusing on protecting attacks based on divulged network traffic while preserving the utility of the apps that legitimately need such information. In particular we first restricted direct access to resources. This way an app can only access the information given by the respective Linux files through the framework's API which an app can use only if the appropriate permission is being granted by the user. By doing that, an app cannot hide the fact that is accessing the resource in question. As a second step, we proposed that apps can actually choose how its own network traffic will be made available to other apps. For this we allowed a developer to choose between the following 4 policies:

- no access: no other app can access the network traffic generate by this app.
- rounding: the length of bytes sent or received by the app is rounded up or down to a multiple of a coefficient given by the developer.
- aggregation: the system reports accummulated data network traffic information when a time interval expires.
- no protection (default policy): any app can access this app's network traffic as before.

Using the no access policy, highly sensitive web-based apps can dictate that no one can read their generated network traffic. On one hand, a banking app can ask the system to conceal all its traffic due to the critically private data it uses. On the other hand, a gaming app can use the default policy, for knowing what the user clicks during a game is of no real value to an adversary. Rounding and aggregation can be used by privacy aware apps and in fact is the recommended option. These two techniques, preserve the utility of apps that provide the user with detailed network statistics allowing her to better manage her data plan and apps, while at the same time, minimize the risk of information leaks. In other words, they allow apps to get information that can be utilized for general statistical traffic usage reporting, but not as

detailed as to allow inference attacks on what is actually being clicked on the monitored app.

The limitation of this technique is that it relies on app developers to provide these policies. Most of the developers are concerned with ease of development and how rapidly they can get the product in the market and then about user privacy. Therefore it becomes debatable whether they will actually use such a mechanism. Nevertheless, highly privacy-critical applications such as banking, business and healthcare apps are expected to utilize this solution once available as potential privacy violations can acutely jeopardize their business.

Secondly we studied the risk associated with Android's channels of communication with external resources. We contended that the Android permission model is too coarse-grained to safeguard these channels while preserving the utility of the apps. To demonstrate that we elaborated on Device Mis-Bonding Attacks (DMB) on Bluetooth and discussed similar confused deputy attacks on the Internet, Audio, NFC and SMS channel. Consider for example an app with the Bluetooth permission talking to a Bluetooth-powered blood glucose meter. A diabetic user, will install the app and use the Bluetooth device. However, any app with the appropriate Bluetooth permissions can connect to the blood glucose meter and steal the user's data. Furthermore, a malicious entity can spoof the blood glucose meter and feed falsified data to the legitimate app, an action that might lead to severe medical implications for the user: erroneous data can compel her to use higher dosages of insulin which can lead even to death. Also consider a user, installing the Chase app. The app can send through SMS account balances to the user, or credit card payments which surpass a defined limit. It can even send a PIN that can be used to change the users' password on her Chase account. Unfortunately any app with the RECEIVE_SMS and/or READ_SMS permission can read these SMSs.

To address this we first designed Dabinder, which can protect against the Bluetooth data-stealing attacks. Dabinder allows the user to map a legitimate application with a Bluetooth accessory. Once this association is established, only the legitimate app can use the Bluetooth channel to communicate with the accessory. Moreover, Dabinder tackle on the Bluetooth data-injection attacks is two-fold. In the case of encrypted communication, it only allows a mapped application to remove the bond (the link/encryption

key) with the correct Bluetooth device. However, in the case of unencrypted communication, the spoofed device trying to feed erroneous data to a legitimate app does not need to reset the established encryption key and thus can talk directly to the app. To address this, Dabinder allows the user to specify which app-device communication must always be encrypted.

Dabinder has of course its limitations. For example, it is a solution tailored specifically to Bluetooth and cannot protect other channels such as Internet, NFC, SMS and Audio. In addition, it always relies on the user to map an application with the appropriate device. Enterprises and accessory companies cannot safeguard their devices directly and it is up to the user's discretion to deploy the security mechanism. We thus say that Dabinder is a DAC solution for the Bluetooth channel. Furthermore, its design is somewhat rigid as it does not provide any management capabilities. For example once a user has made a decision about an app-device pairing this cannot change even if she decides to use another app with that device or the company itself develops a new app for its accessory. Also, its enforcement happens at the framework layer and inside the Bluetooth service. Consequently, a malicious app can directly use the Bluetooth stack (Broadcomm's Bluedroid since 4.2) to bypass the enforcement mechanism and communicate with the external device. Lastly, Dabinder is effective, provided the user pairs the right app with the right device. For example if she pairs the right app with a spoofed device then it is subjected to data-injecting attacks. In fact she won't be able to use the right device with the phone after that, unless they find another compatible app.

To address this need for a more general approach that can safeguard all Android's channels of communication with external resources, we designed SEACAT. SEACAT protects Bluetooth, SMS, NFC, Audio and Internet communications through a unified approach built on top of the implementation of SELinux on Android (SEAndroid). SEACAT naturally extends SEAndroid MAC control to cover these external channels, by providing the capability to smartphone vendors and enterprise IT's to restrict access to external resources to only clearly defined apps through MAC policies. Furthermore, SEACAT has a DAC component that allows a user to protect these channels when she deems it as appropriate. For example if she uses a personal banking app that receives SMSs carrying private information, she can define through SEACAT's DAC policy management that only the Chase app

can read those messages, despite the fact that other apps might have the RECEIVE_SMS and/or READ_SMS permission. This is important as a lot of not-so-trustworthy apps have valid reasons to request such permissions: e.g., FunForMobile Ringtones & Chat (5,000,000+ downloads) needs the READ_SMS and SEND_SMS permissions to allow users share jokes. However, this permission allows the app to read any SMS even if that comes from Chase Bank service. With SEACAT, a user can still allow FunForMobile Ringtones & Chat to access SMSs but not the ones carrying private information.

SEACAT has negligible performance overhead and strives to use the same policy structure and syntax as SEAndroid. This allows its MAC policy configuration to be not more complex than the current SEAndroid MAC configuration. Moreover its DAC policies can be automatically configured by answering simple questions such as Does this app come with an accessory? and Please select the app you downloaded for this accessory., or by mapping the ID FACEBOOK to the Facebook app.

But how does SEACAT compare with Dabinder? SEACAT is a more general solution than Dabinder as it covers all known channels of communication with external resources and can be readily expanded to cover new ones as they appear in a highly dynamic and evolving system such as Android. Furthermore is more capable as it offers smarpthone vendors, accessory manufactures and service providers the capability to configure MAC policies and users the capability to manage their DAC policies. Even for Bluetooth, Dabinder's solution is on the framework layer while SEACAT enforcement happens directly in the Bluetooth stack. Thus SEACAT offers higher security guarantees than Dabinder.

In par with the other security solutions we explored, SEACAT has its own limitations. In particular, it has a hard time protecting against spoofing (or data-injecting) attacks. For secure channels, things are easier as the spoofed device must know the encryption key shared between the OS and the external source. In the Bluetooth case this protection can be subverted by a malicious app removing this key (by calling removeBond()). This malicious act can be thwarted by SEAndroid policies dictating that only system apps can execute such task. However in the absence of such a key, the system will have troubles defending itself. For example SEACAT recognizes external devices by their IDs (MAC address for Bluetooth, Serial number for NFC, SMS ID for SMS,

IP/Domain name for Internet). A malicious entity can spoof such IDs and attack the respective channel. Here an approach like Dabinder's would work for the Bluetooth channel. Specifically SEACAT DAC policies can be defined to restrict certain app-device communications to a secure protocol. For MAC defined app-resource pairs, SEACAT can automatically require the use of a secure Bluetooth channel. This will create a link key between the OS and the external Bluetooth device which can only be used by the right app. Nevertheless, the system is still vulnerable from spoofing attacks on other channels unless an encryption key is shared between the app and the external resource.

With this discussion we reach the end of this thesis. By know you should be convinced that Android might dominate the mobile market but it is by no means a perfect system. In fact Android is continuously evolving and is hard to predict how it is going to be leveraged by developers or utilized by users. Nonetheless, as we use our smartphones to guide every facet of our lives, our data privacy on mobiles becomes progressively significant. This thesis epitomizes the fact that Android's security model is still problematic when it comes to protecting its local and external resources. Unprotected resources result in severe privacy breaches. This generates the need to intensify both the scientific community's and industry's efforts to buttress Android's security. As we saw here, doing that is not always trivial. Solutions exist or can be designed that can protect against specific adversary models but they have their limitations. Essentially, the goal is to strike a balance between designing a solution with the strongest possible adversary model and designing a practical solution that can be deployed in the real world. This is exactly the approach we followed here.

REFERENCES

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

[2] "Antutu benchmark," https://play.google.com/store/apps/details?id=com.antutu.ABenchMark, 2013, accessed: 10/07/2014.

[3] 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.

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

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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

[12] "Bodymedia link armband," http://www.bodymedia.com/, accessed: 10/07/2014.

[13] "Nonin onyx ii pulseoximeter," http://www.nonin.com/PulseOximetry/Finger/Onyx9560, accessed: 10/07/2014.

[14] "Foracare testngo," http://www.myglucohealth.net/, accessed: 10/07/2014.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] J. Wick, "New interactive diabetes support tools manage mealtime insulin dosing," http://www.hcplive.com/articles/New-Interactive-Diabetes-Support-Tools-Manage-Mealtime-Insulin-Dosing, 2014, accessed: 10/07/2014.

[21] 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.

[22] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets," 2014.

[23] 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.

[24] 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.

[25] 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," unpublished.

[26] 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.

[27] "Android Developers Official Website manifest.permission," https://developer.android.com/reference/android/Manifest.permission.html, accessed: 10/07/2014.

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

[29] 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.

[30] 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.

[31] "Shark for root," https://play.google.com/store/apps/details?id=lv.n3o.shark&hl=en, 2012, accessed: 10/07/2014.

[32] "Google play: Webmd for android," http://www.webmd.com/webmdapp, 2012, accessed: 10/07/2014.

[33] 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.

[34] D. J. Solove, "Identity Theft, Privacy, and the Architecture of Vulnerability," *Hastings Law Journal*, vol. 54, pp. 1227 – 1276, 2002-2003.

[35] 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.

[36] 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

[37] S. B. Hoar, "Identity Theft: The Crime of the New Millennium," *Oregon Law Review*, vol. 80, pp. 1423–1448, 2001.

[38] 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.

[39] "Get search, twitter api," https://dev.twitter.com/docs/api/1/get/search, 2012, accessed: 10/07/2014.

[40] "Lookup ip address location," http://whatismyipaddress.com/ip-lookup, 2013, accessed: 10/07/2014.

[41] "Wifi coverage map," http://www.navizon.com/navizon_coverage_wifi.htm, accessed: 13/02/2013.

[42] "The google directions api," https://developers.google.com/maps/documentation/directions/, 2013, accessed: 10/07/2014.

[43] "Locate family," http://www.locatefamily.com/, 2013, accessed: 10/07/2014.

[44] "Standard address abbreviations," http://www.kutztown.edu/admin/adminserv/mailfile/guide/abbrev.html, 2013, accessed: 10/07/2014.

[45] "Nonin onyx ii pulseoximeter specs," http://www.nonin.com/products.asp?ID=39&sec=2&sub=9, accessed: 10/07/2014.

[46] "Bodymedia puts a spin on the ordinary testing procedure," http://blog.bodymedia.com/page/6/, 2012, accessed: 10/07/2014.

[47] C. Daniels, "Why is too much insulin bad?" http://www.livestrong.com/article/423665-why-is-too-much-insulin-bad/, accessed: 10/07/2014.

[48] "ithermometer," http://www.ithermometer.info/, accessed: 10/07/2014.

[49] "hcidump," http://www.linuxcommand.org/man_pages/hcidump8.html, accessed: 10/07/2014.

[50] D. Spill and A. Bittau, "Bluesniff: Eve meets alice and bluetooth," in *Proceedings of USENIX Workshop on Offensive Technologies (WOOT)*, 2007.

[51] S. Bluetooth, "Bluetooth core specification version 2.1+ edr," *Specification of the Bluetooth System*, 2007.

[52] "Android Developers Official Website android bluetoothadapter class," http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html, 2013, accessed: 10/07/2014.

[53] "Spooftooph," http://www.hackfromacave.com/projects/spooftooph.html, 2013, accessed: 10/07/2014.

[54] "Getting ibluetooth instance," http://snipplr.com/view/49526/, 2011, accessed: 10/07/2014.

[55] "Demos for our bluetooth misbonding attacks," [https://sites.google.com/site/edmbdroid/], accessed: 10/07/2014.

[56] M. Handy and D. Timmermann, "Time-slot-based analysis of bluetooth energy consumption for page and inquiry states," accessed: 10/07/2014.

[57] S. Bluetooth, "Specification of the bluetooth systemversion 2.0, 4. november 2004," https://www.bluetooth.org, 2004, accessed: 10/07/2014.

[58] "Java cryptography extension," http://www.oracle.com/technetwork/java/javase/documentation/index.html, accessed: 10/07/2014.

[59] "Bouncycastle library," http://www.bouncycastle.org/, accessed: 10/07/2014.

[60] "Spongycastle library," http://rtyley.github.io/spongycastle/, accessed: 10/07/2014.

[61] "SEACAT demos website," https://sites.google.com/site/seacatchannelcontrol/, accessed: 10/07/2014.

[62] "Square Security official website," https://squareup.com/security, accessed: 10/07/2014.

[63] B. Dwyer, "Paypal here vs. square," http://www.cardfellow.com/blog/paypal-here-vs-square/, accessed: 10/07/2014.

[64] 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.

[65] S. Bluetooth, "Rfcomm with ts 07.10," *Bluetooth SIG*, 2003.