Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources

Xiaoyong Zhou[1], Soteris Demetriou[2], Dongjing He[2], Muhammad Naveed[2],
Xiaorui Pan[1], XiaoFeng Wang[1], Carl A. Gunter[2], Klara Nahrstedt[2]
[1]School of Informatics and Computing, Indiana University, Bloomington, IN USA
[2]Department of Computer Science, University of Illinois, Urbana-Champaign, IL USA
{zhou, xiaopan, xw7}@indiana.edu,
{sdemetr2, dhe6, naveed2, cgunter, klara}@illinois.edu

ABSTRACT

The design of Android is based on a set of unprotected shared resources, including those inherited from Linux (e.g., Linux public directories). However, the dramatic development in Android applications (*app* for short) makes available a large amount of public background information (e.g., social networks, public online services), which can potentially turn such originally harmless resource sharing into serious privacy breaches. In this paper, we report our work on this important yet understudied problem. We discovered three unexpected channels of information leaks on Android: per-app data-usage statistics, ARP information, and speaker status (on or off). By monitoring these channels, an app without *any* permission may acquire sensitive information such as smartphone user's identity, the disease condition she is interested in, her geo-locations and her driving route, from top-of-the-line Android apps. Furthermore, we show that using existing and new techniques, this zero-permission app can both determine when its target (a particular application) is running and send out collected data stealthily to a remote adversary. These findings call into question the soundness of the design assumptions on shared resources, and demand effective solutions. To this end, we present a mitigation mechanism for achieving a delicate balance between utility and privacy of such resources.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Security kernels*; K.4.1 [**COMPUTERS AND SOCIETY**]: Public Policy Issues—*Privacy*

Keywords

Mobile Security, Privacy, Information Leaks

1. INTRODUCTION

Compared with the phenomenal progress in smartphone technologies, the ways smartphones are being used today are even more stunning. Increasingly, they become primary devices not only for the traditional phone calling, but for email checking, messaging, mapping/navigation, entertainment, social networking and even for activities as important as healthcare and investment management. This abrupt development challenges the limits of smartphone designers' imagination, and naturally calls into question whether the original security designs of smartphone systems, like Android, are ready for those highly diverse, fast-evolving applications. Indeed, recent years have seen waves of Android-based malware [3, 7] that exploit different vulnerabilities within this new computing platform, highlighting the security and privacy threats it is facing.

Android's security protection has been under scrutiny for years. A few weaknesses of the system, such as permission re-delegation [21] and capability leaks [27], are revealed by researchers. Particularly, a recent blog [16] from Leviathan Security Group describes the malicious activities that could be performed by an app without any permissions, including reading from SD card, accessing a list of installed apps, getting some device and system information (GSM and SIM vendor ID, the Android ID and kernel version) and pinging through a Linux Shell. Most of the problems reported here are either implementation weaknesses or design ambiguities that lead developers to inadvertent exposures. Examples include SD card read and unauthorized ping, which have all been fixed. The rest turn out to be less of a concern, as they are almost impossible to exploit in practice (GSM/SIM/Android ID, which leads to nothing more than mobile country code and mobile network code).

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

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

For example, consider the list of apps installed on an Android phone, which can be obtained by calling the pubic API `PackageManager.getInstalledApplications()` or looking at

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516661 .

User IDs (UID) within a Linux Shell. Such an app list is not considered sensitive by Google and UIDs are never thought to be confidential on Linux. However, for some healthcare apps (e.g., diabetes app) and life-style apps (e.g., gay social network like Hornet), simply knowing their names can lead to a grave infringement on their users' privacy. As another example, network data usage of an app is deliberately published by Android through its Linux public directory, aiming to help a smartphone user keep track of the app's mobile data consumption. However, we found in our research that this piece of apparently harmless and also useful data can actually be used to fingerprint a user's certain online activity, such as tweeting. This knowledge, combined with the background information that comes with the Twitter app (i.e., public tweets), can be used to infer the user's true identity, using an inference technique we developed (Section 3.2).

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

Our study inspects public resources disclosed at both the Android and its Linux layers and analyzes the impact such exposures can have on the private information maintained by a set of popular apps, in the presence of the rich background information they bring in. This research leads to a series of stunning discoveries on what can actually be inferred from those public resources by leveraging such auxiliary information. Specifically, by monitoring the network-data usage statistics of high-profile Android apps, such as Twitter, WebMD (app by a leading healthcare information provider [1]) and Yahoo! Finance (one of the most widely-used stock apps), one can find out a smartphone user's true identity, disease conditions and stocks one is interested in, without any permissions at all. Looking at the public trace of Address Resolution Protocol (ARP) stored under a public Linux directory, a zero-permission adversary can locate the user with good accuracy. Even the status of a smartphone's speaker (on or off), which can be conveniently identified through a public Android API, `AudioManager.isMusicActive`, turns out to be a serious information leak. We show that when a GPS navigator is being used, from the audio-on/off status, an inference technique we built can figure out the place the smartphone user goes and her driving route. A video demo of these attacks is posted online [12].

All such information leaks are found to be strongly related to the fallacies of design assumptions instead of mere implementation bugs. Every piece of information here is actually meant to be disclosed by Android and most of such data has been extensively used by legitimate Android apps: for example, hundreds of data usage monitors are already out there [5], relying on the usage statistics to keep track of a user's mobile data consumption. Therefore, fixing these information leaks must be done carefully to avoid undermining the basic functions of these existing apps and to strike a balance between system utility and data protection. To this end, we have designed a preliminary mitigation approach that controls the way data usage information is released, which helps to better understand how to achieve such a delicate trade-off.

Contributions. We summarize the contributions of the paper as follows:

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

• *First step toward mitigating this new threat*. We have developed a new mitigation approach, aiming to preserve the utility of such public data to the legitimate parties while controlling the way that an adversary can use it to derive user secrets.

Roadmap. The rest of the paper is organized as follows: Section 2 analyzes the privacy risks of Android public resources and the adversary model; Section 3, 4 and 5 elaborate the new side channels we discovered and the techniques we use to infer confidential information from them; Section 6 describes our mitigation approach; Section 7 compares our work with prior related research and Section 8 concludes the paper.

2. THREATS TO PUBLIC RESOURCES

2.1 Leaks from Public Resources

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

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

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

Most of such resources are provided through two virtual filesystems, the proc filesystem (procfs) and the sys filesystem (sysfs). The procfs contains public statistics about 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 [5] to keep track of individual apps' mobile data consumption.

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

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

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

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

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

• *App network-data usage* (Section 3). 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 develop a suite of inference techniques that can reveal a phone user's identity from the network-data consumption of Twitter app, the disease conditions she is interested in from that of WebMD app and the stock she is looking at from Yahoo! Finance app.

• *Public ARP information* (Section 4). 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 5). 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. A video demo is posted online [12].

2.2 Zero-Permission Adversary

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

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

• *Networking*. Leviathan's blog describes a zero-permission technique to smuggle out data across the Internet [16]. 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.

• *Situation awareness*. 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. GETTING YOUR IDENTITY, HEALTH AND INVESTMENT INFORMATION

In this section, we report our study on information leaks from public data usage statics.

3.1 Usage Monitoring and Analysis

Mobile-data usage statistics. Mobile data usages of Android are made public under `/proc/uid_stat/` (per app) and

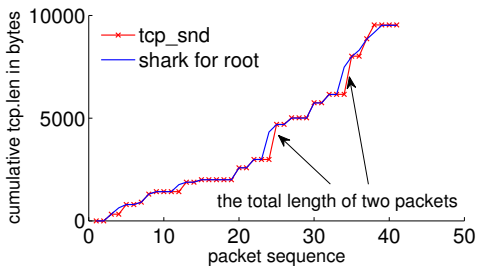

Figure 1: Monitor tool precision

Table 1: Performance overhead of the monitor tool: there the baseline is measured by AnTuTu [8]

	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%

`/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 [5]. 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.

Stealthy and realtime monitoring. 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 1, in which we compare the packet payloads observed by Shark for Root (a network traffic sniffer for 3G and WiFi), when the user is using Yahoo! Finance, with the cumulative outbound data usage detected by our usage monitor.

From the figure we can see that most of the time, our monitor can separate different packets from each other. However, there are situations in which only the cumulative length of multiple packets is identified (see the markers in the figure). This requires an analysis that can tolerate such non-determinism, which we 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 1 measured using AnTuTu [8] with a sampling rate of 10Hz for network usage and 50Hz for audio logging (Section 5). 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 2.2).

Analysis methodology. The monitor cannot always produce deterministic outcomes: when sampling the same packet sequence twice, it may observe two different sequences of increments from the usage statistics. To obtain a reliable traffic fingerprint of a target app's activity we 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 (e.g., click on a button) offline - i.e in a controlled context - and generate a payload-sequence signature for the behavior. Once our monitor collects a sequence of usage increments from the app's runtime on the victim's Android phone, we compare this usage sequence with the signature as follows. Consider a signature $(\cdots, s_i, s_{i+1}, \cdots, s_{i+n}, \cdots)$, where $s_{i,\cdots,i+n}$ are the payload lengths of the TCP packets with the same direction (inbound/outbound), and a sequence $(\cdots, m_j, \cdots)$, where m_j is an increment on a usage statistic (`tcp_rcv` or `tcp_snd`) of the direction of s_i, as observed by our monitor. Suppose that all the elements before m_j match the elements in the signature (those prior to s_i). We say that m_j also matches the signature elements if either $m_j = s_i$ or $m_j = s_i + \cdots + s_{i+k}$ with $1 < k \leq n$. The whole sequence is considered to *match* the signature if all of its elements match the signature elements.

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, which we elaborate in Appendix A.

3.2 Identity Inference

A person's identity, such as name, email address, etc., is always considered to be highly sensitive [35, 19, 15, 29] 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 [26]. 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.

The idea. In our attack, a zero-permission app monitors the mobile-data usage count `tcp_snd` of the Twitter 3.6.0 app when it is running. When the user send tweets to the Twitter server, the app detects this event and send its timestamp to the malicious server

stealthily. This gives us a vector of timestamps for the user's tweets, which we then use to search the tweet history through public Twitter APIs for the account whose activities are consistent with the vector: that is, the account's owner posts her tweets at the moments recorded by these timestamps. Given a few of timestamps, we can uniquely 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 4).

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

Specifically, during the offline analysis, we observed the following TCP payload sequence produced by the Twitter app: $(420|150, 314, 580\text{–}720)$. 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 3.1) 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 $(420|150, 894\text{–}1034)$.

Identity discovery. From the tweeting events detected, we obtain a sequence of timestamps $T = [t_1, t_2, \cdots, t_n]$ that describe when the phone user tweets. This sequence is then used to find out the user's Twitter ID from the public index of tweets. Such an index can be accessed through the Twitter Search API [4]: 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 4). 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 [11] 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 4), 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 2: City information and Twitter identity exploitation

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

Attack evaluation. We evaluated the effectiveness of this attack at three cities, Urbana, Bloomington and Chicago. Table 2 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 2. As we can see here, for small towns with populations below 100 thousand, all the tweets within one hour and a half can be retrieved through a single query, which is sufficient for our attack: it is conceivable that the victim's phone screen will dim out within that period after she tweets, allowing the malicious app to send out the timestamp through the browser. However, for Chicago, the query outcome only covers 2 minutes of tweets. Therefore, we need to continuously dump tweets from the Twitter server to a local database to make the attack work.

In 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.

3.3 Health and Investment

In this section, we show that the data-usage statistics our zero-permission app collects also leak out apps' sensitive inputs, e.g.,

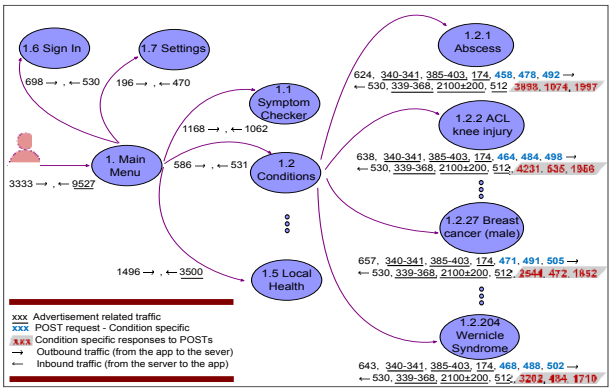

Figure 2: WebMD Finite State Machine

disease conditions a user selects on WebMD mobile [6]. This has been achieved by fingerprinting her actions with data-usage sequences they produce. The same attack technique also works on Yahoo! Finance, which is elaborated in Appendix B.

WebMD mobile. 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 [6]. To use the app, one first clicks to select 1 out of 6 sections, such as "Symptom Checker", "Conditions" and others. 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. All such 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.

Our attack. Again, 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 2. 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, the app produces 4 `GET` requests (with 3 `GET`s for ads); and then 3 `POST` requests to acquire the content for that condition. Among them, all ad-related requests and responses have predictable payload lengths (e.g., 174 bytes for `GET ads/dcfc.gif`) and can therefore be easily identified. The rest of the traffic (1 GET request, 3 `POST` requests and their responses) is characterized by distinctive payload lengths that can be used to fingerprint individual disease conditions. Just based on the POST requests, we can already classify all 204 conditions into 32 categories. The response payload sizes further help us uniquely identify all 204 conditions.

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. This issue was addressed in our research using the technique described in Section 3.1, which compares an observed increment to the cumulative length of multiple packets.

Attack evaluation. We ran our malware on a Google Nexus S 4G phone while using WebMD mobile. The usage data collected and delivered by our app was further compared to the traffic signatures we built offline. We found that the increment sequences matched well with the signatures in all 204 cases, in which we unequivocally identified the disease conditions being visited.

4. FINDING WHERE YOU ARE

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.getCurrentPosition`), 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 whereabout, 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 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.

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

BSSID-based geo-location. 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

Table 3: 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

concerns that our analysis could impact people's perceptions about the privacy implications of BSSID collection.

Exploiting commercial location services. Many commercial apps that offer location-based service 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 [2], 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.

4.2 Attack Evaluation

Data collection. 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.

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

5. KNOWING WHERE YOU GO

As discussed before, information leaks happen not only on the Linux layer of Android but also on its API level. In this section, we report our study of an audio public API that gives away one's driving route.

5.1 Driving Route Inference

Speaker usage information. 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

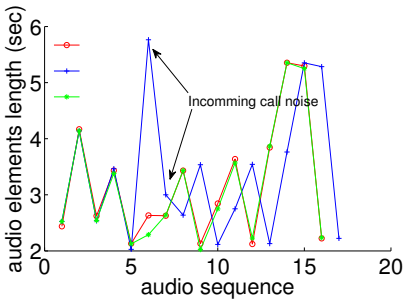

Figure 3: Audio elements similarity when driving on the same route

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.

Audio status logger. 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 3.1). In this case, we are looking for the process `com.google.android.apps.maps`, Google's navigation app. Once the process is discovered, our 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 2.2), 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.

Route fingerprinting simulator. Using the audio-status logger, we recorded the speech length sequences when we drove from home to office three times at Bloomington. Figure 3 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 [9] 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.

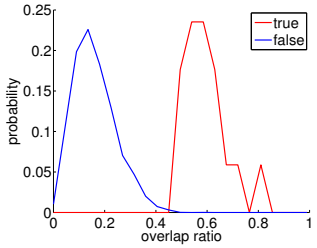

Figure 4: Audio length sequence distinguishability

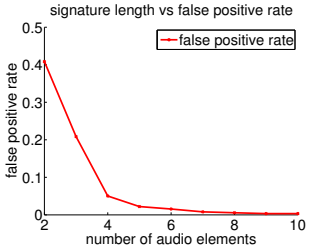

Figure 5: False positive rate vs number of audio elements

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 4. 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 4 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 4 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 5 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.

5.2 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 routes for the target speech-length sequence.

Location fingerprinting. 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 5).

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.

Scalable sequence matching. More challenging here is to locate a user's driving route on the map. For this purpose, we need some background knowledge to roughly determine the area that covers the route. As discussed before (Section 3.2 and 4), we can geolocate 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 destination 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 [10]. 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 [13], 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 4. 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 4 shows a TTS sequence that matches a subsequence on the real sequence. As a result of the search, 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.

Table 4: 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

Table 5: 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 6.

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	

Figure 6: Three FP Routes and Their Corresponding TP Routes. Each FP/TP pair has most of their routes overlapped.

5.3 Attack Evaluation

Location determination. 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.

Driving-route identification. We further 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 [10] 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 5.2 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 5). Among them, 8 are true positives, the real routes we drove.

Also, the 3 false positives are actually the routes that come very close to the routes of 3 real-drive sequences (see Figure 6), 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 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.

6. MITIGATION AND DISCUSSION

Given the various public resources on Android, the information leaks we found are very likely to be just a tip of the iceberg. Finding an effective solution to this problem is especially challenging with rich background information of users or apps gratuitously available on the web. To mitigate such threats, we first take a closer look at the attacks discovered in our research. The ARP data has not been extensively utilized by apps and can therefore be kept away from unauthorized parties by changing the related file's access privilege to `system`. A simple solution to control the audio channel can be to restrict the access to its related APIs, such as `isMusicActive`, only to system processes whenever sensitive apps (e.g. navigation related) are running in the foreground. The most challenging facet of such a mitigation venture is to address the availability mechanism of the data usage statistics, 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.

6.1 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.

Round up and round down. One strategy is to reduce the accuracy of the available information by rounding up or down the actual number of bytes sent or received by an app to a multiple of a given integer before disclosing that value to the querying process. This approach is reminiscent of a predominant defense strategy against traffic analysis, namely packet padding [20, 36]. 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 6.2). 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.

Aggregation. A limitation of the simple rounding strategy results from the fact that it still gives away the payload size of each packet, even though the information is perturbed. As a result, it cannot 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.

6.2 Enforcement Framework

To enforce the aforementioned policies, we designed and implemented a preliminary framework, which is elaborated below.

Permission Design. A naive idea would be adding yet another permission to Android's already complex permission system and have any data monitoring app requesting this permission in AndroidManifest.xml. However, prior research shows that the users do not pay too much attention to the permission list when installing apps, and the developers tend to declare more permissions than needed [25]. 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 6.1) or periodically updating the information.

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

Defense Evaluation. To understand the effectiveness our technique, we first evaluated the round up and round down scheme using the WebMD app. Figure 7 illustrates the results: with α increasing from 16 to 1024, the corresponding number of conditions that can be uniquely identified drops from 201 to 1. In other words, except a peculiar condition *DEMENTIA IN HEAD INJURY* whose total reply

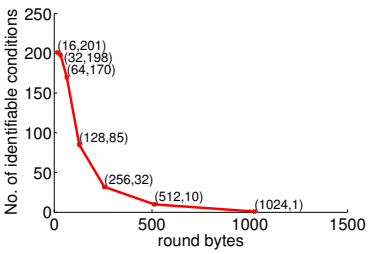

Figure 7: Effectiveness of round up and round down

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.

Limitations of our Defence. We found that it is challenging to come up with a bullet proof defense against all those information leaks 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 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.

7. RELATED WORK

Information leaks have been studied for decades and new discoveries continue to be made in recent years [33, 39, 37]. Among them, most related to our work is the work on the information leaks from procfs, which includes using the ESP/EIP data to infer keystrokes [38] and leveraging memory usages to fingerprint visited websites [31]. However, it is less clear whether those attacks pose a credible threat to Android, due to the high non-determinism of its memory allocation [31] and the challenges in keystroke analysis [38]. In comparison, our work shows that the usage statistics under procfs can be practically exploited to infer an Android user's sensitive information. The attack technique used here is related to prior work on traffic analysis [20]. However, those approaches assume the presence of an adversary who sees encrypted packets. Also, their analysis techniques cannot be directly applied to smartphone. Our attack is based upon a different adversary model, in which an app uses public resources to infer the content of the data received by a target app on the same device. For this purpose, we need to build different inference techniques based on the unique features of mobile computing, particularly the rich background in-

formation (i.e., social network, BSSID databases and Google Maps) that comes with the target app and the mobile OS.

Information leaks have been discovered on smartphone by both academia and the hacker community [21, 27, 16]. Most of known problems are caused by implementation errors, either in Android or within mobile apps. By comparison, the privacy risks come from shared resources in the presence of emerging background information have not been extensively studied on mobile devices. Up to our knowledge, all prior research on this subject focuses on the privacy implications of motion sensors or microphones [34, 17, 32, 18, 28]. What has never been done before is a systematic investigation on what can be inferred from the public resources exposed by both Linux and Android layers.

New techniques for better protecting user privacy on Android also continue to pop up [22, 23, 27, 30, 14, 24, 21]. Different from such research, our work focuses on the new privacy risks emerging from the fast-evolving smartphone apps, which could render innocuous public resources related to sensitive user information.

8. CONCLUSION

In this paper, we report our study on information leaks from Android public resources. The study reveals that highly sensitive data of a smartphone user, such as her identity, interested disease conditions, geo-location, driving route and more can actually be reliably inferred from such resources by analyzing popular apps. Our findings call into question the design assumption made by Android developers on public resources and demand new effort to address such privacy risks. To this end, we further present a preliminary design for mitigating the threats to selected public resources, while still maintaining their utility.

9. ACKNOWLEDGMENTS

The authors at Indiana University are supported in part by National Science Foundation CNS-1017782 and CNS-1117106. We also acknowledge the grant HHS-90TR0003/01 from the Office of the National Coordinator for Health Information Technology at the Department of Health and Human Services (HHS). The views in this paper represent opinions of the authors only.

10. REFERENCES

[1] ReadWrite A Tech Blog. http://readwrite.com/2011/05/10/doctor_in_your_pocket_webmd_comes_to_android. Accessed: 13/02/2013.

[2] Wifi coverage map. http://www.navizon.com/navizon_coverage_wifi.htm. Accessed: 13/02/2013.

[3] Fbi issues android smartphone malware warning. http://www.forbes.com/sites/billsinger/2012/10/15/fbi-issues-android-smartphone-malware-warning/, 2012.

[4] Get search, twitter api. https://dev.twitter.com/docs/api/1/get/search, 2012.

[5] Google play. https://play.google.com/store/search?q=traffic+monitor&c=apps, 2012.

[6] Google play: Webmd for android. http://www.webmd.com/webmdapp, 2012.

[7] Smart phone malware: The six worst offenders. http://www.nbcnews.com/technology/technolog/smart-phone-malware-six-worst-offenders-125248, 2012.

[8] Antutu benchmark. https://play.google.com/store/apps/details?id=com.antutu.ABenchMark, 2013.

[9] The google directions api. https://developers.google.com/maps/documentation/directions/, 2013.

[10] Locate family. http://www.locatefamily.com/, 2013.

[11] Lookup ip address location. http://whatismyipaddress.com/ip-lookup, 2013.

[12] Online demo. https://sites.google.com/site/sidedroid/, 2013.

[13] Standard address abbreviations. http://www.kutztown.edu/admin/adminserv/mailfile/guide/abbrev.html, 2013.

[14] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM.

[15] H. Berghel. Identity theft, social security numbers, and the web. *Commun. ACM*, 43(2):17–21, Feb. 2000.

[16] 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.

[17] L. Cai and H. Chen. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX conference on Hot topics in security*, HotSec'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[18] L. Cai and H. Chen. On the practicality of motion based keystroke inference attack. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 273–290, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] 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*, DIM '05, pages 20–27, New York, NY, USA, 2005. ACM.

[20] 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*, pages 191 –206, may 2010.

[21] 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.

[22] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[23] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[24] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM CCS*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[25] 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*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[26] 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*, 9, 2005.

[27] 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.

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

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

[30] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM CCS*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[31] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 143–157, Washington, DC, USA, 2012. IEEE Computer Society.

[32] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems Applications*, HotMobile '12, pages 9:1–9:6, New York, NY, USA, 2012. ACM.

[33] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM CCS*, pages 199–212, New York, NY, USA, 2009. ACM.

[34] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*. The Internet Society, 2011.

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

[36] 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.

[37] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. Uncovering spoken phrases in encrypted voice over ip conversations. *ACM Trans. Inf. Syst. Secur.*, 13(4):35:1–35:30, Dec. 2010.

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

[39] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

APPENDIX

A. USER-AGENT CONSTRUCTION

Many mobile apps use HTTP for data exchange and the traffic generated during this process always involves a User-Agent field. 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 we need here 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 our app, without any permission, from `android.os.Build` and `System.getProperty("http agent")`.

B. PERSONAL INVESTMENT INFERENCE

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

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

Stock news signature. From the dynamics of mobile data usage produced by the suggestion lists, we can identify a set of candidate stocks. To narrow it down, we further 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.