ProfileDroid: Multi-layer Profiling of Android Applications

Xuetao Wei Lorenzo Gomez Iulian Neamtiu Michalis Faloutsos
Department of Computer Science and Engineering
University of California, Riverside
{xwei, gomezl, neamtiu, michalis}@cs.ucr.edu

ABSTRACT

The Android platform lacks tools for assessing and monitoring apps in a systematic way. This lack of tools is particularly problematic when combined with the open nature of Google Play, the main app distribution channel. As our key contribution, we design and implement ProfileDroid, a comprehensive, multi-layer system for monitoring and profiling apps. Our approach is arguably the first to profile apps at four layers: (a) static, or app specification, (b) user interaction, (c) operating system, and (d) network. We evaluate 27 free and paid Android apps and make several observations: (a) we identify discrepancies between the app specification and app execution, (b) free versions of apps could end up costing more than their paid counterparts, due to an order of magnitude increase in traffic, (c) most network traffic is not encrypted, (d) apps communicate with many more sources than users might expect—as many as 13, and (e) we find that 22 out of 27 apps communicate with Google during execution. ProfileDroid is the first step towards a systematic approach for (a) generating cost-effective but comprehensive app profiles, and (b) identifying inconsistencies and surprising behaviors.

Categories and Subject Descriptors

C.2.1 [**Computer-communication Networks**]: Network Architecture and Design—*Wireless communication*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.4.8 [**Operating Systems**]: Performance—*Measurements*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation/methodology*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Android apps, Google Android, Profiling, Monitoring, System

1. INTRODUCTION

Given an Android app, how can we get an informative thumbnail of its behavior? This is the problem we set to address in this paper, in light of the 550,000 apps currently on Google Play (ex Android Market) [1, 18]. Given this substantial number of apps, we consider scalability as a key requirement. In particular, we devise a profiling scheme that works even with limited resources in terms of time, manual effort, and cost. We define limited resources to mean: a few users with a few minutes of experimentation per application. At the same time, we want the resulting app profiles to be comprehensive, useful, and intuitive. Therefore, given an app and one or more short executions, we want a profile that captures succinctly what the app did, and contrast it with: (a) what it was expected or allowed to do, and (b) other executions of the same app. For example, an effective profile should provide: (a) how apps use resources, expressed in terms of network data and system calls, (b) the types of device resources (e.g., camera, telephony) an app accesses, and whether it is allowed to, and (c) what entities an app communicates with (e.g., cloud or third-party servers).

Who would be interested in such a capability? We argue that an inexpensive solution would appeal to everyone who "comes in contact" with the app, including: (a) the app developer, (b) the owner of an Android app market, (c) a system administrator, and (d) the end user. Effective profiling can help us: (a) enhance user control, (b) improve user experience, (c) assess performance and security implications, and (d) facilitate troubleshooting. We envision our quick and cost-effective thumbnails (profiles) to be the first step of app profiling, which can then have more involved and resource-intense steps, potentially based on what the thumbnail has revealed.

Despite the flurry of research activity in this area, there is no approach yet that focuses on profiling the behavior of an Android app *itself* in all its complexity. Several efforts have focused on analyzing the mobile phone traffic and show the protocol related properties, but they do not study the apps *themselves* [14, 16]. Others have studied security issues that reveal the abuse of personal device information [22, 31]. However, all these works: (a) do not focus on *individual* apps, but report general trends, or (b) focus on a single layer, studying, e.g., the network behavior or the app specification in isolation. For example, some apps have negligible user inputs, such as `Pandora`, or negligible network traffic, such as `Advanced Task Killer`, and thus, by focusing only on one layer, the most significant aspect of an application could be missed. In Section 5, we discuss and compare with previous work in detail.

We design and implement ProfileDroid, a systematic and comprehensive system for profiling Android apps. A key novelty is that our profiling spans four layers: (a) static, i.e., app specification, (b) user interaction, (c) operating system, and (d) network. To the best of our knowledge, this is the first work[1] that considers all these layers in profiling individual Android apps. Our contributions are twofold. First, designing the system requires the careful selection of informative and intuitive metrics, which capture the essence of each layer. Second, implementing the system is a nontrivial task, and we have to overcome numerous practical challenges.[2]

We demonstrate the capabilities of our system through experiments. We deployed our system on Motorola Droid Bionic phones, with Android version 2.3.4 and Linux kernel version 2.6.35. We profile 19 free apps; for 8 of these, we also profile their paid counterparts, for a total of 27 apps. For each app, we gather profiling data from 30 runs for several users at different times of day. Though we use limited testing resources, our results show that our approach can effectively profile apps, and detect surprising behaviors and inconsistencies. Finally, we show that *cross-layer* app analysis can provide insights and detect issues that are not visible when examining single layers in isolation.

We group our most interesting observations in three groups, although it is clear that some observations span several categories.

A. Privacy and Security Issues.

1. Lack of transparency. We identify discrepancies between the app specification and app execution. For example, `Instant Heart Rate` and `Dictionary` use resources without declaring them up-front (Section 3.1).

2. Most network traffic is unencrypted. We find that most of the network traffic is not encrypted. For example, most of the web-based traffic is over HTTP and not HTTPS: only 8 out of the 27 apps use HTTPS and for `Facebook`, 22.74% of the traffic is not encrypted (Section 4.5).

B. Operational Issues.

3. Free apps have a cost. Free versions of apps could end up costing more than their paid versions especially on limited data plans, due to increased advertising/analytics traffic. For example, the free version of `Angry Birds` has 13 times more traffic than the paid version (Section 4.3).

4. Apps talk to "strangers". Apps interact with many more traffic sources than one would expect. For example, the free version of `Shazam` talks to 13 different traffic sources in a 5-minute interval, while its paid counterpart talks with 4 (Section 4.6).

5. Google "touches" almost everything. Out of 27 apps, 22 apps exchange data traffic with Google, including apps that one would not have expected, e.g., Google accounts for 85.97% of the traffic for the free version of the health app `Instant Heart Rate`, and 90% for the paid version (Section 4.7).

C. Performance Issues.

6. Security comes at a price. The Android OS uses virtual machine (VM)-based isolation for security and reliability, but as a consequence, the VM overhead is high: more than 63% of the system calls are introduced by the VM for context-switching between threads, supporting IPC, and idling (Section 4.4).

2. OVERVIEW OF APPROACH

We present an overview of the design and implementation of ProfileDroid. We measure and profile apps at four different layers: (a) static, or app specification (b) user interaction, (c) operating system, and (d) network. For each layer, our system consists of two parts: a monitoring and a profiling component. For each layer, the monitoring component runs on the Android device where the app is running. The captured information is subsequently fed into the profiling part, which runs on the connected computer. In Figure 1, on the right, we show a high level overview of our system and its design. On the left, we have an actual picture of the actual system the Android device that runs the app and the profiling computer (such as a desktop or a laptop).

In the future, we foresee a light-weight version of the whole profiling system to run exclusively on the Android device. The challenge is that the computation, the data storage, and the battery consumption must be minimized. How to implement the profiling in an incremental and online fashion is beyond the scope of the current work. Note that our system is focused on profiling of an *individual* app, and not intended to monitor user behavior on mobile devices.

From an architectural point of view, we design ProfileDroid to be flexible and modular with level-defined interfaces between the monitoring and profiling components. Thus, it is easy to modify or improve functionality within each layer. Furthermore, we could easily extend the current functionality to add more metrics, and even potentially more layers, such as a physical layer (temperature, battery level, etc.).

2.1 Implementation and Challenges

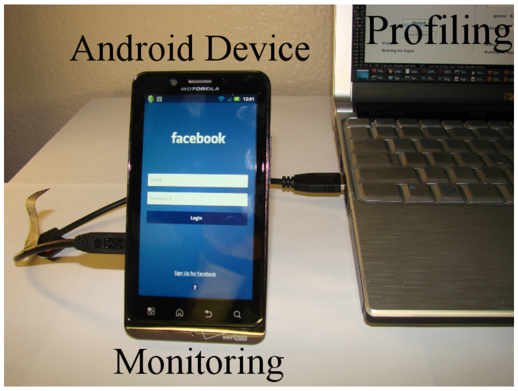
We describe the implementation of monitoring at each layer, and briefly touch on challenges we had to surmount when constructing ProfileDroid.

To profile an application, we start the monitoring infrastructure (described at length below) and then the target app is launched. The monitoring system logs all the relevant activities, e.g., user touchscreen input events, system calls, and all network traffic in both directions.

2.1.1 Static Layer

At the static layer, we analyze the APK (Android application package) file, which is how Android apps are distributed. We use `apktool` to unpack the APK file to extract relevant data. From there, we mainly focus on the `Manifest.xml` file and the bytecode files contained in the `/smali` folder. The manifest is specified by the developer and identifies hardware usage and permissions requested by each app. The `smali` files contain the app bytecode which we parse and analyze statically, as explained later in Section 3.1.

[1]An earlier work [12] uses the term "cross-layer," but the layers it refers to are quite different from the layers we use.
[2]Examples include fine-tuning data collection tools to work on Android, distinguishing between presses and swipes, and disambiguating app traffic from third-party traffic.



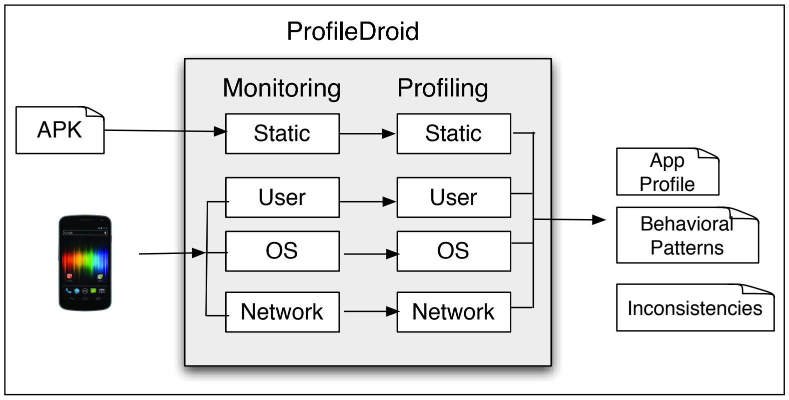


Figure 1: Overview and actual usage (left) and architecture (right) of ProfileDroid.

2.1.2 *User Layer*

At the user layer, we focus on user-generated events, i.e., events that result from interaction between the user and the Android device while running the app. To gather the data of the user layer, we use a combination of the `logcat` and `getevent` tools of `adb`. From the `logcat` we capture the system debug output and log messages from the app. In particular, we focus on events-related messages. To collect the user input events, we use the `getevent` tool, which reads the `/dev /input/event*` to capture user events from input devices, e.g., touchscreen, accelerometer, proximity sensor. Due to the raw nature of the events logged, it was challenging to disambiguate between swipes and presses on the touchscreen. We provide details in Section 3.2.

2.1.3 *Operating System Layer*

At the operating system-layer, we measure the operating system activity by monitoring system calls. We collect system calls invoked by the app using an Android-specific version of `strace`. Next, we classify system calls into four categories: filesystem, network, VM/IPC, and miscellaneous. As described in Section 3.3, this classification is challenging, due to the virtual file system and the additional VM layer that decouples apps from the OS.

2.1.4 *Network Layer*

At the network layer, we analyze network traffic by logging the data packets. We use an Android-specific version of `tcpdump` that collects all network traffic on the device. We parse, domain-resolve, and classify traffic. As described in Section 3.4, classifying network traffic is a significant challenge in itself; we used information from domain resolvers, and improve its precision with manually-gathered data on specific websites that act as traffic sources.

Having collected the measured data as described above, we analyze it using the methods and the metrics of Section 3.

2.2 Experimental Setup

2.2.1 *Android Devices*

The Android devices monitored and profiled in this paper were a pair of identical Motorola Droid Bionic phones, which have dual-core ARM Cortex-A9 processors running at 1GHz. The phones were released on released September 8, 2011 and run Android version 2.3.4 with Linux kernel version 2.6.35.

2.2.2 *App Selection*

As of March 2012, Google Play lists more than 550,000 apps [2], so to ensure representative results, we strictly follow the following criteria in selecting our test apps. First, we selected a variety of apps that cover most app categories as defined in Google Play, such as Entertainment, Productivity tools, etc. Second, all selected apps had to be popular, so that we could examine real-world, production-quality software with a broad user base. In particular, the selected apps must have at least 1,000,000 installs, as reported by Google Play, and be within the Top-130 free apps, as ranked by the Google Play website. In the end, we selected 27 apps as the basis for our study: 19 free apps and 8 paid apps; the 8 paid apps have free counterparts, which are included in the list of 19 free apps. The list of the selected apps, as well as their categories, is shown in Table 1.

2.2.3 *Conducting the experiment*

In order to isolate app behavior and improve precision when profiling an app, we do not allow other manufacturer-installed apps to run concurrently on the Android device, as they could interfere with our measurements. Also, to minimize the impact of poor wireless link quality on apps, we used WiFi in strong signal conditions. Further, to ensure statistics were collected of only the app in question, we installed one app on the phone at a time and uninstalled it before the next app was tested. Note however, that system daemons and required device apps were still able to run as they normally would, e.g., the service and battery managers.

Finally, in order to add stability to the experiment, the multi-layer traces for each individual app were collected from tests conducted by multiple users to obtain a comprehensive exploration of different usage scenarios of the target application. To cover a larger variety of running conditions without burdening the user, we use *capture-and-replay*, as explained below. Each user ran each app one time for 5 minutes; we capture the user interaction using event logging. Then, using a replay tool we created, each recorded run was replayed back 5 times in the morning and 5 times at night, for a total of 10 runs each per user per app. The runs of each app were conducted at different times of the day to avoid time-of-day bias, which could lead to uncharacteristic interaction with the app; by using the capture-and-replay tool, we are able to achieve this while avoiding repetitive manual runs from the same user. For those apps that had both free and paid

App name	Category
Dictionary.com, Dictionary.com-$$	Reference
Tiny Flashlight	Tools
Zedge	Personalization
Weather Bug, Weather Bug-$$	Weather
Advanced Task Killer, Advanced Task Killer-$$	Productivity
Flixster	Entertainment
Picsay, Picsay-$$	Photography
ESPN	Sports
Gasbuddy	Travel
Pandora	Music & Audio
Shazam, Shazam-$$	Music & Audio
Youtube	Media & Video
Amazon	Shopping
Facebook	Social
Dolphin, Dolphin-$$	Communication (Browsers)
Angry Birds, Angry Birds-$$	Games
Craigslist	Business
CNN	News & Magazines
Instant Heart Rate, Instant Heart Rate-$$	Health & Fitness

Table 1: The test apps; app-$$ represents the paid version of an app.

App	Internet	GPS	Camera	Microphne	Bluetooth	Telephony
Dictionary.com	✓			I		I
Dictionary.com-$$	✓			I		I
Tiny Flashlight	✓		✓			
Zedge	✓					
Weather Bug	✓	✓				
Weather Bug-$$	✓	✓				
Advanced Task Killer	✓					
Advanced Task Killer-$$	✓					
Flixster	✓	✓				
Picsay	✓					
Picsay-$$	✓					
ESPN	✓					
Gasbuddy	✓	✓				
Pandora	✓				✓	
Shazam	✓	✓		✓		
Shazam-$$	✓	✓		✓		
YouTube	✓					
Amazon	✓		✓			
Facebook	✓	✓	I			✓
Dolphin	✓	✓				
Dolphin-$$	✓	✓				
Angry Birds	✓					
Angry Birds-$$	✓					
Craigslist	✓					
CNN	✓		✓			
Instant Heart Rate	✓		✓		I	I
Instant Heart Rate-$$	✓		✓		I	I

Table 2: Profiling results of *static* layer; '✓' represents use via permissions, while 'I' via intents.

versions, users carried out the same task, so we can pinpoint differences between paid and free versions. To summarize, our profiling is based on 30 runs (3 users × 10 replay runs) for each app.

3. ANALYZING EACH LAYER

In this section, we first provide detailed descriptions of our profiling methodology, and we highlight challenges and interesting observations.

3.1 Static Layer

The first layer in our framework aims at understanding the app's functionality and permissions. In particular, we analyze the APK file on two dimensions to identify app functionality and usage of device resources: first, we extract the permissions that the app asks for, and then we parse the app bytecode to identify intents, i.e., indirect resource access via deputy apps. Note that, in this layer only, we analyze the app without running it—hence the name *static layer*.

Functionality usage. Android devices offer several major functionalities, labeled as follows: Internet, GPS, Camera, Microphone, Bluetooth and Telephony. We present the results in Table 2. A '✓' means the app requires permission to use the device, while 'I' means the device is used indirectly via intents and deputy apps. We observe that Internet is the most-used functionality, as the Internet is the gateway to interact with remote servers via 3G or WiFi—all of our examined apps use the Internet for various tasks. For instance, `Pandora` and `YouTube` use the Internet to fetch multimedia files, while `Craigslist` and `Facebook` use it to get content updates when necessary.

GPS, the second most popular resource (9 apps) is used for navigation and location-aware services. For example, `Gasbuddy` returns gas stations near the user's location, while `Facebook` uses the GPS service to allow users to *check-in*, i.e., publish their presence at entertainment spots or places of interests. Camera, the third-most popular functionality (5 apps) is used for example, to record and post real-time news information (`CNN`), or for for barcode scanning `Amazon`. Microphone, Bluetooth and Telephony are three additional communication channels besides the Internet, which could be used for voice communication, file sharing, and text messages. This increased usage of various communication channels is a double-edged sword. On the one hand, various communication channels improve user experience. On the other hand, it increases the risk of privacy leaks and security attacks on the device.

Intent usage. Android intents allow apps to access resources indirectly by using deputy apps that have access to the requested resource. For example, `Facebook` does not have the camera permission, but can send an intent to a deputy camera app to take and retrieve a picture.[3] We decompiled each app using `apktool` and identified instances of the `android.content.Intent` class in the Dalvik bytecode. Next, we analyzed the parameters of each intent call to find

[3]This was the case for the version of the Facebook app we analyzed in March 2012, the time we performed the study. However, we found that, as of June 2012, the Facebook app requests the Camera permission explicitly.

the intent's type, i.e., the device's resource to be accessed via deputy apps.

We believe that presenting users with the list of resources used via intents (e.g., that the `Facebook` app does not have direct access to the camera, but nevertheless it can use the camera app to take pictures) helps them make better-informed decisions about installing and using an app. Though legitimate within the Android security model, this lack of user forewarning can be considered deceiving; with the more comprehensive picture provided by ProfileDroid, users have a better understanding of resource usage, direct or indirect [3].

3.2 User Layer

At the user layer, we analyze the input events that result from user interaction. In particular, we focus on *touches*—generated when the user touches the screen—as touchscreens are the main Android input devices. Touch events include *presses*, e.g., pressing the app buttons of the apps, and *swipes*—finger motion without losing contact with the screen. The intensity of events (events per unit of time), as well as the ratio between swipes and presses are powerful metrics for GUI behavioral fingerprinting (Section 4.2); we present the results in Figure 2 and now proceed to discussing these metrics.

Technical challenge. Disambiguating between swipes and presses was a challenge, because of the nature of reported events by the *getevent* tool. Swipes and presses are reported by the touchscreen input device, but the reported events are not labeled as swipes or presses. A single press usually accounts for 30 touchscreen events, while a swipe usually accounts for around 100 touchscreen events. In order to distinguish between swipes and presses, we developed a method to cluster and label events. For example, two events separated by less than 80 milliseconds are likely to be part of a sequence of events, and if that sequence of events grows above 30, then it is likely that the action is a swipe instead of a press. Evaluating and fine-tuning our method was an intricate process.

Touch events intensity. We measured touch intensity as the number of touch events per second—this reveals how interactive an app is. For example, the music app `Pandora` requires only minimal input (music control) once a station is selected. In contrast, in the game `Angry Birds`, the user has to interact with the interface of the game using swipes and screen taps, which results in a high intensity for touch events.

Swipe/Press ratio. We use the ratio of swipes to presses to better capture the nature of the interaction, and distinguish between apps that have similar touch intensity. Note that swipes are used for navigation and zooming, while touches are used for selection. Figure 2 shows that apps that involve browsing, news-page flipping, gaming, e.g., `CNN`, `Angry Birds`, have a high ratio of swipes to presses; even for apps with the same touch intensity, the swipe/press ratio can help profile and distinguish apps, as seen in the following table:

App	Touch intensity	Swipe/Press ratio
`Picsay`	*medium*	*low*
`CNN`	*medium*	*high*

Phone event intensity. The bottom chart in Figure 2 shows the intensity of events generated by the phone itself during the test. These events contain a wealth of contextual data that, if leaked, could pose serious privacy risks. The most frequent events we observed were generated by the accelerometer, the light proximity sensor, and for some location-aware apps, the compass. For brevity, we omit details, but we note that phone-event intensity, and changes in intensity, can reveal the user's proximity to the phone, the user's motion patterns, and user orientation and changes thereof.

App	**Syscall intensity (calls/sec.)**	**FS (%)**	**NET (%)**	**VM& IPC (%)**	**MISC (%)**
Dictionary.com	1025.64	3.54	1.88	67.52	27.06
Dictionary.com-$$	492.90	7.81	4.91	69.48	17.80
Tiny Flashlight	435.61	1.23	0.32	77.30	21.15
Zedge	668.46	4.17	2.25	75.54	18.04
Weather Bug	1728.13	2.19	0.98	67.94	28.89
Weather Bug-$$	492.17	1.07	1.78	75.58	21.57
AdvTaskKiller	75.06	3.30	0.01	65.95	30.74
AdvTaskKiller-$$	30.46	7.19	0.00	63.77	29.04
Flixster	325.34	2.66	3.20	71.37	22.77
Picsay	319.45	2.06	0.01	75.12	22.81
Picsay-$$	346.93	2.43	0.16	74.37	23.04
ESPN	1030.16	2.49	2.07	87.09	8.35
Gasbuddy	1216.74	1.12	0.32	74.48	24.08
Pandora	286.67	2.92	2.25	70.31	24.52
Shazam	769.54	6.44	2.64	72.16	18.76
Shazam-$$	525.47	6.28	1.40	74.31	18.01
YouTube	246.78	0.80	0.58	77.90	20.72
Amazon	692.83	0.42	6.33	76.80	16.45
Facebook	1030.74	3.99	2.98	72.02	21.01
Dolphin	850.94	5.20	1.70	71.91	21.19
Dolphin-$$	605.63	9.05	3.44	68.45	19.07
Angry Birds	1047.19	0.74	0.36	82.21	16.69
Angry Birds-$$	741.28	0.14	0.04	85.60	14.22
Craigslist	827.86	5.00	2.47	73.81	18.72
CNN	418.26	7.68	5.55	71.47	15.30
InstHeartRate	944.27	7.70	1.73	75.48	15.09
InstHeartRate-$$	919.18	12.25	0.14	72.52	15.09

Table 3: Profiling results: *operating system* layer.

3.3 Operating System Layer

We first present a brief overview of the Android OS, and then discuss metrics and results at the operating system layer.

Android OS is a Linux-based operating system, customized for mobile devices. Android apps are written in Java and compiled to Dalvik executable (Dex) bytecode. The bytecode is bundled with the app manifest (specification, permissions) to create an APK file. When an app is installed, the user must grant the app the permissions specified in the manifest. The Dex bytecode runs on top of the Dalvik Virtual Machine (VM)—an Android-specific Java virtual machine. Each app runs as a separate Linux process with a unique user ID in a separate copy of the VM. The separation among apps offers a certain level of protection and running on top of a VM avoids granting apps direct access to hardware resources. While increasing reliability and reducing the potential for security breaches, this vertical (app–hardware) and horizontal (app–app) separation means that apps do not

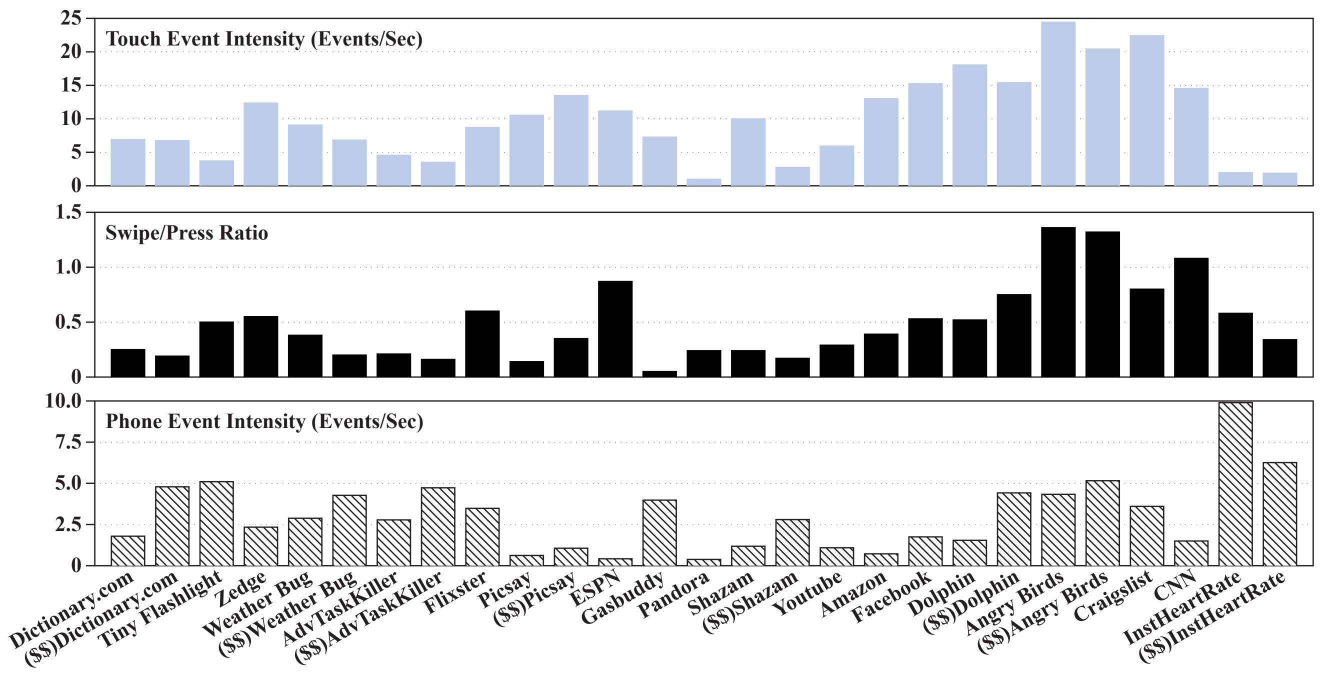


Figure 2: Profiling results of *user* layer; note that scales are different.

run natively and inter-app communications must take place primarily via IPC. We profile apps at the operating system layer with several goals in mind: to understand how apps use system resources, how the operating-system intensity compares to the intensity observed at other layers, and to characterize the potential performance implications of running apps in separate VM copies. To this end, we analyzed the system call traces for each app to understand the nature and frequency of system calls. We present the results in Table 3.

System call intensity. The second column of Table 3 shows the system call intensity in system calls per second. While the intensity differs across apps, note that in all cases the intensity is relatively high (between 30 and 1,183 system calls per second) for a mobile platform.

System call characterization. To characterize the nature of system calls, we group them into four bins: file system (FS), network (NET), virtual machine (VM&IPC), and miscellaneous (MISC). Categorizing system calls is not trivial.

Technical challenge. The Linux version running on our phone (2.6.35.7 for Arm) supports about 370 system calls; we observed 49 different system calls in our traces. While some system calls are straightforward to categorize, the operation of virtual filesystem calls such as `read` and `write`, which act on a file descriptor, depends on the file descriptor and can represent file reading and writing, network send/receive, or reading/altering system configuration via `/proc`. Therefore, for all the virtual filesystem calls, we categorize them based on the file descriptor associated with them, as explained below. FS system calls are used to access data stored on the flash drive and SD card of the mobile device and consist mostly of `read` and `write` calls on a file descriptor associated with a space-occupying file in the file system, i.e., opened via `open`. NET system calls consist mostly of `read` and `write` calls on a file descriptor associated with a network socket, i.e., opened via `socket`; note that for NET system calls, reads and writes mean receiving from and sending to the network. VM&IPC system calls are calls inserted by the virtual machine for operations such as scheduling, timing, idling, and IPC. For each such operation, the VM inserts a specific sequence of system calls. We extracted these sequences, and compared the number of system calls that appear as part of the sequence to the total number, to quantify the VM and IPC-introduced overhead. The most common VM/IPC system calls we observed (in decreasing order of frequency) were: `clock_gettime`, `epoll_wait`, `getpid`, `getuid32`, `futex`, `ioctl`, and `ARM_cacheflush`. The remaining system calls are predominantly `read` and `write` calls to the `/proc` special filesystem are categorized as MISC.

The results are presented in Table 3: for each category, we show both intensity, as well as the percentage relative to all categories. Note that FS and NET percentages are quite similar, but I/O system calls (FS and NET) constitute a relatively small percentage of total system calls, with the VM&IPC dominating. We will come back to this aspect in Section 4.4.

3.4 Network Layer

The network-layer analysis summarizes the data communication of the app via WiFi or 3G. Android apps increasingly rely on Internet access for a diverse array of services, e.g., for traffic, map or weather data and even offloading computation to the cloud. An increasing number of network traffic sources are becoming visible in app traffic, e.g., Content Distribution Networks, Cloud, Analytics and Advertisement. To this end, we characterize the app's network

App	Traffic intensity (bytes/sec.)	Traffic In/Out (ratio)	Origin (%)	CDN+Cloud (%)	Google (%)	Third party (%)	Traffic sources	HTTP/HTTPS split (%)
Dictionary.com	1450.07	1.94	–	35.36	64.64	–	8	100/–
Dictionary.com-$$	488.73	1.97	0.02	1.78	98.20	–	3	100/–
Tiny Flashlight	134.26	2.49	–	–	99.79	0.21	4	100/–
Zedge	15424.08	10.68	–	96.84	3.16	–	4	100/–
Weather Bug	3808.08	5.05	–	75.82	16.12	8.06	13	100/–
Weather Bug-$$	2420.46	8.28	–	82.77	6.13	11.10	5	100/–
AdvTaskKiller	25.74	0.94	–	–	100.00	–	1	91.96/8.04
AdvTaskKiller-$$	–	–	–	–	–	–	0	–/–
Flixster	23507.39	20.60	2.34	96.90	0.54	0.22	10	100/–
Picsay	4.80	0.34	–	48.93	51.07	–	2	100/–
Picsay-$$	320.48	11.80	–	99.85	0.15	–	2	100/–
ESPN	4120.74	4.65	–	47.96	10.09	41.95	5	100/–
Gasbuddy	5504.78	10.44	6.17	11.23	81.37	1.23	6	100/–
Pandora	24393.31	28.07	97.56	0.91	1.51	0.02	11	99.85/0.15
Shazam	4091.29	3.71	32.77	38.12	15.77	13.34	13	100/–
Shazam-$$	1506.19	3.09	44.60	55.36	0.04	–	4	100/–
YouTube	109655.23	34.44	96.47	–	3.53	–	2	100/–
Amazon	7757.60	8.17	95.02	4.98	–	–	4	99.34/0.66
Facebook	4606.34	1.45	67.55	32.45	–	–	3	22.74/77.26
Dolphin	7486.28	5.92	44.55	0.05	8.60	46.80	22	99.86/0.14
Dolphin-$$	3692.73	6.05	80.30	1.10	5.80	12.80	9	99.89/0.11
Angry Birds	501.57	0.78	–	73.31	10.61	16.08	8	100/–
Angry Birds-$$	36.07	1.10	–	88.72	5.79	5.49	4	100/–
Craigslist	7657.10	9.64	99.97	–	–	0.03	10	100/–
CNN	2992.76	5.66	65.25	34.75	–	–	2	100/–
InstHeartRate	573.51	2.29	–	4.18	85.97	9.85	3	86.27/13.73
InstHeartRate-$$	6.09	0.31	–	8.82	90.00	1.18	2	20.11/79.89

Table 4: Profiling results of *network* layer; '–' represents no traffic.

behavior using the following metrics and present the results in Table 4.

Traffic intensity. This metric captures the intensity of the network traffic of the app. Depending on the app, the network traffic intensity can vary greatly, as shown in Table 4. For the user, this great variance in traffic intensity could be an important property to be aware of, especially if the user has a limited data plan. Not surprisingly, we observe that the highest traffic intensity is associated with a video app, `YouTube`. Similarly, the entertainment app `Flixster`, music app `Pandora`, and personalization app `Zedge` also have large traffic intensities as they download audio and video files. We also observe apps with zero, or negligible, traffic intensity, such as the productivity app `Advanced Task Killer` and free photography app `Picsay`.

Origin of traffic. The origin of traffic means the percentage of the network traffic that comes from the servers owned by the app provider. This metric is particularly interesting for privacy-sensitive users, since it is an indication of the control that the app provider has over the app's data. Interestingly, there is large variance for this metric, as shown in Table 4. For example, the apps `Amazon`, `Pandora`, `YouTube`, and `Craigslist` deliver most of their network traffic (e.g., more than 95%) through their own servers and network. However, there is no origin traffic in the apps `Angry Birds` and `ESPN`. Interestingly, we observe that ony 67% of the `Facebook` traffic comes from Facebook servers, with the remaining coming from content providers or the cloud.

Technical challenge. It is a challenge to classify the network traffic into different categories (e.g., cloud vs. ad network), let alone identify the originating entity. To resolve this, we combine an array of methods, including reverse IP address lookup, DNS and `whois`, and additional information and knowledge from public databases and the web. In many cases, we use information from CrunchBase (crunchbase.com) to identify the type of traffic sources after we resolve the top-level domains of the network traffic [6]. Then, we classify the remaining traffic sources based on information gleaned from their website and search results.

In some cases, detecting the origin is even more complicated. For example, consider the `Dolphin` web browser—here the origin is not the Dolphin web site, but rather the website that the user visits with the browser, e.g., if the user visits CNN, then cnn.com is the origin. Also, YouTube is owned by Google and YouTube media content is delivered from domain 1e100.net, which is owned by Google; we report the media content (96.47%) as Origin, and the remaining traffic (3.53%) as Google which can include Google ads and analytics.

CDN+Cloud traffic. This metric shows the percentage of the traffic that comes from servers of CDN (e.g., Akamai) or cloud providers (e.g., Amazon AWS). Content Distribution Network (CDN) has become a common method to distribute the app's data to its users across the world faster, with scalability and cost-effectively. Cloud platforms have extended this idea by providing services (e.g., computation) and not just data storage. Given that it is not obvious if someone using a cloud service is using it as storage, e.g., as a CDN, or for computation, we group CDN and cloud services into one category. Interestingly, there is a very strong presence of this kind of traffic for some apps, as seen in Table 4. For example, the personalization app `Zedge`, and the video-heavy app `Flixster` need intensive network services, and they use CDN and Cloud data sources. The high percentages that we observe for CDN+Cloud traffic point to how important CDN and Cloud sources are, and how much apps rely on them for data distribution.

Google traffic. Given that Android is a product of Google, it is natural to wonder how involved Google is in Android traffic. The metric is the percentage of traffic exchanged with Google servers (e.g., 1e100.net), shown as the second-to-last column in Table 4. It has been reported that the percentage of Google traffic has increased significantly over the past several years [7]. This is due in part to the increasing penetration of Google services (e.g., maps, ads, analytics, and Google App Engine). Note that 22 of out of the 27 apps exchange traffic with Google, and we discuss this in more detail in Section 4.7.

Third-party traffic. This metric is of particular interest to privacy-sensitive users. We define third party traffic as network traffic from various advertising services (e.g., Atdmt) and analytical services (e.g., Omniture) besides Google, since advertising and analytical services from Google are included in the Google traffic metric. From Table 4, we see that different apps have different percentages of third-party traffic. Most apps only get a small or negligible amount of traffic from third parties (e.g., `YouTube`, `Amazon` and `Facebook`). At the same time, nearly half of the total traffic of `ESPN` and `Dolphin` comes from third parties.

The ratio of incoming traffic and outgoing traffic. This metric captures the role of an app as a consumer or producer of data. In Table 4, we see that most of the apps are more likely to receive data than to send data. As expected, we see that the network traffic from `Flixster`, `Pandora`, and `YouTube`, which includes audio and video content, is mostly incoming traffic as the large values of the ratios show. In contrast, apps such as `Picsay` and `Angry Birds` tend to send out more data than they receive.

Note that this metric could have important implications for performance optimization of wireless data network providers. An increase in the outgoing traffic could challenge network provisioning, in the same way that the emergence of p2p file sharing stretched cable network operators, who were not expecting large household upload needs. Another use of this metric is to detect suspicious variations in the ratio, e.g., unusually large uploads, which could indicate a massive theft of data. Note that the goal of this paper is to provide the framework and tools for such an investigation, which we plan to conduct as our future work.

Number of distinct traffic sources. An additional way of quantifying the interactions of an app is with the number of distinct traffic sources, i.e., distinct top-level domains. This metric can be seen as a complementary way to quantify network interactions, a sudden increase in this metric could indicate malicious behavior. In Table 4 we present the results. First, we observe that all the examined apps interact with at least two distinct traffic sources, except `Advanced Task Killer`. Second, some of the apps interact with a surprisingly high number of distinct traffic sources, e.g., `Weather bug`, `Flixster`, and `Pandora`. Note that we count all the distinct traffic sources that appear in the traces of multiple executions.

App	Static (# of func.)	User (events/ sec.)	OS (syscall/ sec.)	Network (bytes/ sec.)
Dictionary.com	L	M	H	M
Dictionary.com-$$	L	M	M	M
Tiny Flashlight	M	L	M	L
Zedge	L	M	M	H
Weather Bug	M	M	H	M
Weather Bug-$$	M	M	M	M
AdvTaskKiller	L	M	L	L
AdvTaskKiller-$$	L	M	L	L
Flixster	M	M	L	H
Picsay	L	M	L	L
Picsay-$$	L	M	M	M
ESPN	L	M	H	M
Gasbuddy	M	M	H	M
Pandora	M	L	L	H
Shazam	H	L	M	M
Shazam-$$	H	L	H	M
YouTube	L	M	M	H
Amazon	M	M	M	H
Facebook	H	H	H	M
Dolphin	M	H	M	H
Dolphin-$$	M	H	M	M
Angry Birds	L	H	M	M
Angry Birds-$$	L	H	H	L
Craigslist	L	H	H	H
CNN	M	M	M	M
InstHeartRate	M	L	H	M
InstHeartRate-$$	M	L	H	L

Table 5: Thumbnails of multi-layer intensity in the *H*-*M*-*L* model (*H*:high, *M*:medium, *L*:low).

The percentage of HTTP and HTTPS traffic. To get a sense of the percentage of secure Android app traffic, we compute the split between HTTP and HTTPS traffic, e.g., non-encrypted and encrypted traffic. We present the results in the last column of Table 4 ('–' represents no traffic). The absence of HTTPS traffic is staggering in the apps we tested, and even `Facebook` has roughly 22 % of unencrypted traffic, as we further elaborate in section 4.4.

4. ProfileDroid: PROFILING APPS

In this section, we ask the question: *How can* ProfileDroid *help us better understand app behavior?* In response, we show what kind of information ProfileDroid can extract from each layer in isolation or in combination with other layers.

4.1 Capturing Multi-layer Intensity

The intensity of activities at each layer is a fundamental metric that we want to capture, as it can provide a thumbnail of the app behavior. The multi-layer intensity is a tuple consisting of intensity metrics from each layer: static (number of functionalities), user (touch event intensity), operating system (system call intensity), and network (traffic intensity).

Presenting raw intensity numbers is easy, but it has limited intuitive value. For example, reporting 100 system calls per second provides minimal information to a user or an application developer. A more informative approach is to present the relative intensity of this app compared to other apps.

We opt to represent the activity intensity of each layer using labels: H (high), M (medium), and L (low). The three levels (H, M, L) are defined relative to the intensities observed at each layer using the five-number summary from statistical analysis [10]: minimum (Min), lower quartile (Q_1), median (Med), upper quartile (Q_3), and maximum (Max). Specifically, we compute the five-number summary across all 27 apps at each layer, and then define the ranges for H, M, and L as follows:

$$Min < L \leq Q_1 \qquad Q_1 < M \leq Q_3 \qquad Q_3 < H \leq Max$$

The results are in the following table:

Layer	Min	Q_1	Med	Q_3	Max
Static	1	1	2	2	3
User	0.57	3.27	7.57	13.62	24.42
OS	30.46	336.14	605.63	885.06	1728.13
Network	0	227.37	2992.76	6495.53	109655.23

Note that there are many different ways to define these thresholds, depending on the goal of the study, whether it is conserving resources, (e.g., determining static thresholds to limit intensity), or studying different app categories (e.g., general-purpose apps have different thresholds compared to games). In addition, having more than three levels of intensity provides more accurate profiling, at the expense of simplicity. To sum up, we chose to use relative intensities and characterize a wide range of popular apps to mimic testing of typical Google Play apps.

Table 5 shows the results of applying this H-M-L model to our test apps. We now proceed to showing how users and developers can benefit from an H-M-L-based app thumbnail for characterizing app behavior. Users can make more informed decisions when choosing apps by matching the H-M-L thumbnail with individual preference and constraints. For example, if a user has a small-allotment data plan on the phone, perhaps he would like to only use apps that are rated L for the intensity of network traffic; if the battery is low, perhaps she should refrain from running apps rated H at the OS or network layers.

Developers can also benefit from the H-M-L model by being able to profile their apps with ProfileDroid and optimize based on the H-M-L outcome. For example, if ProfileDroid indicates an unusually high intensity of filesystem calls in the operating system layer, the developer can examine their code to ensure those calls are legitimate. Similarly, if the developer is contemplating using an advertising library in their app, she can construct two H-M-L app models, with and without the ad library and understand the trade-offs.

In addition, an H-M-L thumbnail can help capture the nature of an app. Intuitively, we would expect interactive apps (social apps, news apps, games, Web browsers) to have intensity H at the user layer; similarly, we would expect media player apps to have intensity H at the network layer, but L at the user layer. Table 5 supports these expectations, and suggests that the the H-M-L thumbnail could be an initial way to classify apps into coarse behavioral categories.

4.2 Cross-layer Analysis

We introduce a notion of *cross-layer analysis* to compare the inferred (or observed) behavior across different layers. Performing this analysis serves two purposes: to identify potential discrepancies (e.g., resource usage via intents, as explained in Section 3.1), and to help characterize app behavior in cases where examining just one layer is insufficient. We now provide some examples.

Network traffic disambiguation. By cross-checking the user and network layers we were able to distinguish advertising traffic from expected traffic. For example, when profiling the `Dolphin` browser, by looking at both layers, we were able to separate advertisers traffic from web content traffic (the website that the user browses to), as follows. From the user layer trace, we see that the user surfed to, for example, cnn.com, which, when combined with the network traffic, can be used to distinguish legitimate traffic coming from CNN and advertising traffic originating at CNN; note that the two traffic categories are distinct and labeled Origin and Third-party, respectively, in Section 3.4. If we were to only examine the network layer, when observing traffic with the source cnn.com, we would not be able to tell Origin traffic apart from ads placed by cnn.com.

Application disambiguation. In addition to traffic disambiguation, we envision cross-layer checking to be useful for behavioral fingerprinting for apps (outside the scope of this paper). Suppose that we need to distinguish a file manager app from a database-intensive app. If we only examine the operating system layer, we would find that both apps show high FS (filesystem) activity. However, the database app does this without any user intervention, whereas the file manager initiates file activity (e.g., move file, copy file) in response to user input. By cross-checking the operating system layer and user layer we can distinguish between the two apps because the file manager will show much higher user-layer activity. We leave behavioral app fingerprinting to future work.

4.3 Free Versions of Apps Could End Up Costing More Than Their Paid Versions

The Android platform provides an open market for app developers. Free apps (69% of all apps on Google Play [2]) significantly contributed to the adoption of Android platform. However, the free apps are not as free as we would expect. As we will explain shortly, considerable amounts of network traffic are dedicated to for-profit services, e.g., advertising and analytics.

In fact, we performed a cross-layer study between free apps and their paid counterparts. As mentioned in Section 2.1, users carried out the same task when running the free and paid versions of an app. We now proceed to describe findings at each layer. We found no difference at the static layer (Table 2). At the user-layer, Figure 2 shows that most of behaviors are similar between free and paid version of the apps, which indicates that free and paid versions have similar GUI layouts, and performing the same task takes similar effort in both the free and the paid versions of an app. The exception was the photography app `Picsay`. At first we found this finding counterintuitive; however, the paid version of `Picsay` provides more picture-manipulating functions than the free version, which require more navigation (user input) when manipulating a photo.

Differences are visible at the OS layer as well: as shown in Table 3, system call intensity is significantly higher (around 50%–100%) in free apps compared the their paid counterparts, which implies lower performance and higher energy consumption. The only exception is `Picsay`, whose paid version has higher system call intensity; this is due to increased GUI navigation burden as we explained above.

We now move on to the network layer. Intuitively, the paid apps should not bother users with the profit-making extra traffic, e.g., ads and analytics, which consumes away the data plan. However, the results only partially match our expectations. As shown in Table 4, we find that the majority of the paid apps indeed exhibit dramatically reduced network traffic intensity, which help conserve the data plan. Also, as explained in Section 4.6, paid apps talk to fewer data sources than their free counterparts. However, we could still observe traffic from Google and third party in the paid apps. We further investigate whether the paid apps secure their network traffic by using HTTPS instead of HTTP. As shown in Table 4, that is usually not the case, with the exception of `Instant Heart Rate`.

To sum up, the "free" in "free apps" comes with a hard-to-quantify, but noticeable, user cost. Users are unaware of this because multi-layer behavior is generally opaque to all but most advanced users; however, this shortcoming is addressed well by ProfileDroid.

4.4 Heavy VM&IPC Usage Reveals a Security-Performance Trade-off

As mentioned in Section 3.3, Android apps are isolated from the hardware via the VM, and isolated from each other by running on separate VM copies in separate processes with different UIDs. This isolation has certain reliability and security advantages, i.e., a corrupted or malicious app can only inflict limited damage. The flip side, though, is the high overhead associated with running bytecode on top of a VM (instead of natively), as well as the high overhead due to IPC communication that has to cross address spaces. The VM&IPC column in Table 3 quantifies this overhead: we were able to attribute around two-thirds of system calls (63.77% to 87.09%, depending on the app) to VM and IPC. The precise impact of VM&IPC system calls on performance and energy usage is beyond the scope of this paper, as it would require significantly more instrumentation. Nevertheless, the two-thirds figure provides a good intuition of the additional system call burden due to isolation.

4.5 Most Network Traffic is not Encrypted

As Android devices and apps manipulate and communicate sensitive data (e.g., GPS location, list of contacts, account information), we have investigated whether the Android apps use HTTPS to secure their data transfer. Last column of Table 4 shows the split between HTTP and HTTPS traffic for each app. We see that most apps use HTTP to transfer the data. Although some apps secure their traffic by using HTTPS, the efforts are quite limited. This is a potential concern: for example, for `Facebook` 77.26% of network traffic is HTTPS, hence the remaining 22.74% can be intercepted or modified in transit in a malicious way. A similar concern is notable with `Instant Heart Rate`, a health app, whose free version secures only 13.73% of the traffic with HTTPS; personal health information might leak in the remaining 86.27% HTTP traffic. We further investigate which traffic sources are using HTTPS and report the results in Table 6. Note how HTTPS data sources (Origin, CDN, Google) also deliver services over HTTP. These results reveal that deployment of HTTPS is lagging in Android apps—an undesirable situation as Android apps are often used for privacy-sensitive tasks.

App	HTTPS traffic sources	HTTP
Pandora	Pandora, Google	yes
Amazon	Amazon	yes
Facebook	Facebook, Akamai	yes
Instant Heart Rate	Google	yes
Instant Heart Rate-$$	Google	yes

Table 6: Traffic sources for HTTPS.

4.6 Apps Talk to Many More Traffic Sources Than One Would Think

When running apps that have Internet permission, the underlying network activity is a complete mystery: without access to network monitoring and analysis capabilities, users and developers do not know where the network traffic comes from and goes to. To help address this issue, we investigate the traffic sources; Table 4 shows the number of distinct traffic sources in each app, while Table 7 shows the number of distinct traffic sources per traffic category. We make two observations here. First, Table 4 reveals that most of the apps interact with at least two traffic sources, and some apps have traffic with more than 10 sources, e.g., `Pandora` and `Shazam`, because as we explained in Section 3.4, traffic sources span a wide range of network traffic categories: Origin, CDN, Cloud, Google and third party. Second, paid apps have fewer traffic sources than their free counterparts (3 vs. 8 for `Dictionary.com`, 4 vs. 13 for `Shazam`, 9 vs. 22 for `Dolphin`), and the number of third-party sources is 0 or 1 for most paid apps. This information is particularly relevant to app developers, because not all traffic sources are under the developer's control. Knowing this information makes both users and developers aware of the possible implications (e.g., data leaking to third parties) of running an app.

4.7 How Predominant is Google Traffic in the Overall Network Traffic?

Android apps are relying on many Google services such as Google maps, YouTube video, AdMob advertising, Google Analytics, and Google App Engine. Since Google leads the Android development effort, we set out to investigate whether Google "rules" the Android app traffic. In Table 4, we have presented the percentage of Google traffic relative to all traffic. While this percentage varies across apps, most apps have at least some Google traffic. Furthermore, Google traffic dominates the network traffic in the apps `Tiny Flashlight` (99.79%), `Gasbuddy` (81.37%) and `Instant Heart Rate` (85.97%), which shows that these apps crucially rely on Google services. However, some apps, such as `Amazon` and `Facebook`, do not have Google traffic; we believe this information is relevant to certain categories of users.

In addition, we further break down the Google traffic and analyze the ratio of incoming traffic from Google to outgoing traffic to Google. The ratios are presented in Table 7.

App	CDN+ Cloud	Google	Third party	Google In/Out
Dictionary.com	3	1	4	2.42
Dictionary.com-$$	2	1	0	1.92
Tiny Flashlight	0	1	3	2.13
Zedge	2	1	1	2.06
Weather Bug	5	1	7	4.93
Weather Bug-$$	3	1	1	13.20
AdvTaskKiller	0	1	0	0.94
AdvTaskKiller-$$	0	0	0	–
Flixster	4	1	4	0.90
Picsay	1	1	0	0.93
Picsay-$$	1	1	0	0.94
ESPN	1	1	3	3.84
Gasbuddy	2	1	2	17.25
Pandora	3	1	6	3.63
Shazam	3	1	8	2.61
Shazam-$$	1	1	1	0.84
YouTube	0	1	0	11.10
Amazon	3	0	0	–
Facebook	2	0	0	–
Dolphin	0	1	17	5.10
Dolphin-$$	0	1	4	2.99
Angry Birds	1	1	6	2.26
Angry Birds-$$	2	1	0	1.04
Craigslist	6	0	3	–
CNN	1	0	0	–
InstHeartRate	1	1	1	2.41
InstHeartRate-$$	1	1	0	1.21

Table 7: Number of distinct traffic sources per traffic category, and the ratio of incoming to outgoing Google traffic; '–' means no Google traffic.

We find that most apps are Google data receivers (in/out ratio > 1). However, `Advanced Task Killer`, `Picsay` and `Flixster`, are sending more data to Google than they are receiving (in/out ratio < 1); this is expected.

5. RELATED WORK

To our knowledge, none of the prior efforts have focused on multi-layer monitoring and profiling of *individual* Android app.

Smartphone Measurements and Profiling. Falaki et al. [14] analyzed network logs from 43 smartphones and found commonly used app ports, properties of TCP transfer and the impact factors of smartphone performance. Furthermore, they also analyzed the diversity of smartphone usage, e.g., how the user uses the smartphone and apps [16]. Maier et al. [13] analyzed protocol usage, the size of HTTP content and the types of hand-held traffic. These efforts aid network operators, but they do not analyze the Android apps themselves. Recent work by Xu et al. [26] did a large scale network traffic measurement study on usage behaviors of smartphone apps, e.g., locality, diurnal behaviors and mobility patterns. Qian et al. [12] developed a tool named ARO to locate the performance and energy bottlenecks of smartphones by considering the cross-layer information ranging from radio resource control to application layer. Huang et al. [19] performed the measurement study using smartphones on 3G networks, and presented the app and device usage of smartphones. Falaki et al. [15] developed a monitoring tool SystemSens to capture the usage context, e.g., CPU and memory, of smartphone. Livelab [9] is a measurement tool implemented on iPhones to measure iPhone usage and different aspects of wireless network performance. Powertutor [21] focused on power modeling and measured the energy usage of smartphone. All these efforts focus on studying other layers, or device resource usage, which is different from our focus.

Android Security Related Work. Enck et al. [30] presented a framework that reads the declared permissions of an application at install time and compared it against a set of security rules to detect potentially malicious applications. Ongtang et al. [24] described a fine-grained Android permission model for protecting applications by expressing permission statements in more detail. Felt et al. [5] examined the mapping between Android APIs and permissions and proposed Stowaway, a static analysis tool to detect overprivilege in Android apps. Our method profiles the phone functionalities in static layer not only by explicit permission request, but also by implicit Intent usage. Comdroid found a number of exploitable vulnerabilities in inter-app communication in Android apps [11]. Permission re-delegation attack were shown to perform privileged tasks with the help of an app with permissions [4]. Taintdroid performed dynamic information-flow tracking to identify the privacy leaks to advertisers or single content provider on Android apps [31]. Furthermore, Enck et al. [29] found pervasive misuse of personal identifiers and penetration of advertising and analytics services by conducting static analysis on a large scale of Android apps. Our work profiles the app from multiple layers, and furthermore, we profile the network layer with a more fine-grained granularity, e.g., Origin, CDN, Cloud, Google and so on. AppFence extended the Taintdroid framework by allowing users to enable privacy control mechanisms [25]. A behavior-based malware detection system Crowdroid was proposed to apply clustering algorithms on system calls statistics to differentiate between benign and malicious apps [17]. pBMDS was proposed to correlate user input features with system call features to detect anomalous behaviors on devices [20]. Grace et al. [23] used Woodpecker to examined how the Android permission-based security model is enforced in pre-installed apps of stock smartphones. Capability leaks were found that could be exploited by malicious activities. DroidRanger was proposed to detect malicious apps in official and alternative markets [33]. Zhou et al. characterized a large set of Android malwares and a large subset of malwares' main attack was accumulating fees on the devices by subscribing to premium services by abusing SMS-related Android permissions [32]. Colluding applications could combine their permissions and perform activities beyond their individual privileges. They can communicate directly [8], or exploit covert or overt channels in Android core system components [27]. An effective framework was developed to defense against privilege-escalation attacks on devices, e.g., confused deputy attacks and colluding attacks [28].

6. CONCLUSIONS

In this paper, we have presented ProfileDroid, a monitoring and profiling system for characterizing Android app behaviors at multiple layers: static, user, OS and network. We proposed an ensemble of metrics at each layer to capture the essential characteristics of app specification, user activities, OS and network statistics. Through our analysis of top free and paid apps, we show that characteristics and behavior of Android apps are well-captured by the metrics selected in our profiling methodology, thereby justifying the selection of these metrics. Finally, we illustrate how, by using ProfileDroid for multi-layer analysis, we were able to uncover surprising behavioral characteristics.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Srdjan Capkun for their feedback. This work was supported in part by National Science Foundation awards CNS-1064646, CNS-1143627, by a Google Research Award, by ARL CTA W911NF-09-2-0053, and by DARPA SMISC Program.

7. REFERENCES

[1] Google Play. https://play.google.com/store, May 2012.

[2] Androlib. Number of New Applications in Android Market by month, March 2012. http://www.androlib.com/appstats.aspx.

[3] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *SOUPS*, 2012.

[4] A.P. Felt, H. Wang, A. Moshchuk, S. Hanna and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium*, 2011.

[5] A.P.Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *ACM CCS*, 2011.

[6] B. Krishnamurthy and C. E. Willis. Privacy diffusion on the web: A longitudinal perspective. In *WWW*, 2009.

[7] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet inter-domain traffic. In *ACM SIGCOMM*, 2010.

[8] C. Marforio, F. Aurelien, and S. Capkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. In *Technical Report 724, ETH Zurich*, 2011.

[9] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. LiveLab: Measuring Wireless Networks and Smartphone Users in the Field. In *HotMetrics*, 2010.

[10] D.C. Hoaglin, F. Mosteller and J. W. Tukey. Understanding robust and exploratory data analysis, 1983. Wiley.

[11] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *ACM MobiSys*, 2011.

[12] F. Qian, Z. Wang, A. Gerber, Z. Morley Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile apps: a Cross-layer Approach. In *ACM MobiSys*, 2011.

[13] G. Maier, F. Schneider, and A. Feldmann. A First Look at Mobile Hand-held Device Traffic. In *PAM*, 2010.

[14] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A First Look at Traffic on Smartphones. In *ACM IMC*, 2010.

[15] H.Falaki, R.Mahajan, and D. Estrin. SystemSens: A Tool for Monitoring Usage in Smartphone Research Deployments. In *ACM MobiArch*, 2011.

[16] H.Falaki, R.Mahajan, S. Kandula, D.Lymberopoulos, R.Govindan, and D.Estrin . Diversity in Smartphone Usage. In *ACM MobiSys*, 2010.

[17] I. Burguera,U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *SPSM*, 2011.

[18] IDC. Android- and iOS-Powered Smartphones Expand Their Share of the Market. http://www.idc.com/getdoc.jsp?containerId=prUS23503312, May 2012.

[19] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing app Performance Differences on Smartphones. In *ACM MobiSys*, 2010.

[20] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu. pBMDS: A Behavior-based Malware Detection System for Cellphone Devices. In *ACM WiSec*, 2010.

[21] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *CODES+ISSS*, 2010.

[22] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Detecting Privacy Leaks in iOS apps. In *NDSS*, 2011.

[23] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones . In *NDSS*, 2012.

[24] M. Ongtang, S. McLaughlin, W. Enck and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *ACSAC*, 2009.

[25] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These arenâĂŹt the Droids youâĂŹre looking for: RetroïňĄtting Android to protect data from imperious applications. In *ACM CCS*, 2011.

[26] Q. Xu, J. Erman, A. Gerber, Z. Morley Mao, J. Pang, and S. Venkataraman. Identify Diverse Usage Behaviors of Smartphone Apps. In *IMC*, 2011.

[27] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS*, 2011.

[28] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B.Shastry. Towards Taming Privilege-Escalation Attacks on Android . In *NDSS*, 2012.

[29] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, 2011.

[30] W. Enck, M. Ongtang and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *ACM CCS*, 2009.

[31] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[32] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE S&P*, 2012.

[33] Y. Zhou, Z. Wang, Wu Zhou and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets . In *NDSS*, 2012.