Characterizing Mobile Open APIs in Smartphone Apps

Li Zhang, Chris Stover, Amanda Lins, Chris Buckley, Prasant Mohapatra
Computer Science Department, University of California, Davis
Email: {jxzhang, cjstover, amicoskilins, cmbuckley, pmohapatra}@ucdavis.edu

***Abstract*—Mobile applications used in smartphones are increasingly using Open APIs, and the trend is likely to continue in the foreseeable future. However, the performance of the Open APIs integrated in smartphone apps (Mobile Open APIs) remains hidden from app developers and app users because of the lack of a method to isolate the Open API calls from the whole app execution process. In this paper, we present the very first effort on characterizing Mobile Open APIs and analyzing their performance in terms of four metrics: response latency, network traffic, energy consumption, and CPU usage. We first develop *APIExtractor (APIX)*, a software tool to extract the Mobile Open API calls as fine-grained as in the function level from Android app files (.apk). Then the popularity of the Open API functions were ranked by running *APIX* on 200 top popular apps downloaded from the Android app store. We then perform in-depth case studies on the the top 17 most popular Mobile Open APIs, by wrapping each of them in a specifically designed app (called *APISymphone*) and test the apps on both Wi-Fi and cellular network. Furthermore, we conduct a global scale measurements of the Mobile Open APIs by using Amazon Elastic Computing service. Our comprehensive measurement-based results provides very intriguing as well as interesting insights to the performance characteristics of Mobile APIs.**

I. Introduction

Open Application Programming Interface (Open API) is a new technology that enables web devices to interact with each other through certain web service protocols, e.g. Single Object Access Protocol (SOAP) and Representational State Transfer (REST) [1].

Recent years have witnessed a fast trend in the implementation of commercial software products by making use of Open APIs. The number of companies which have published their own Open APIs increased by a factor more than 200% every 6 months since 2010 [2]. Correspondingly, the usage of these Open APIs has also increased remarkably. By the end of May 2011, there were 16 Open APIs each of which was called more than 1 billion times per month [2]. For example, Twitter Open APIs are called 13 billion times/day, and AccuWeater Open APIs get 1.1 billion calls/day [2]. The increase in the popularity of integrating Open APIs into commercial software products has also led to the similar trends in designing smartphone applications (hereafter called as "apps"). Upon examination of the top 200 apps in terms of the number of downloads in Android App Market, we found that 179 (89.5%) of these apps integrate at least one kind of third-party Open APIs.

With the rapidly increasing usage and implementation of Open APIs, it is important and necessary to understand the resource usage and overheads of these popular web functions. Furthermore, the increasing adoption in resource constrained mobile devices warrants prudent implementation and design trade-offs. To the best of our knowledge, no studies have been reported yet on a detailed characterization of Open APIs on smartphones (we refer to these as Mobile Open APIs). Many Open APIs have similar functionality and it is important for application developers to understand the difference between these Open APIs. In order to decide which Open API to integrate, developers should know what the time and resource costs are for the popular APIs. Thus a detailed characterization would provide app developers guidance to optimize their apps with respect to performance, energy consumption, and bandwidth consumption.

In order to characterize Open APIs, we analyzed 200 popular android applications. We decompiled the source code and then were able to search for calls to APIs. This approach offers insight into the Open APIs that are implemented most frequently in popular Android applications. We have designed and implemented a tool called *APIX* to automate the extraction of APIs from the source codes. Analysis of the *APIX* output helps us identify the most popular Open APIs. To profile the mobile APIs, we then developed a series of apps called *APISymphone* that helps in deriving the latency, CPU, and power usage of the popular mobile APIs. Furthermore, we also performed a global-scale study, aided through Amazon EC2 service, to characterize Mobile Open APIs when accessed from various parts of the world.

The following contributions can be highlighted from this paper:

- Our *APIX* tool achieved extremely high coverage on the potential search space for identifying the Mobile Open APIs. Using *APIX*, we were able to identify the Mobile Open APIs integrated in the popular Android apps. We compared our ranking of the popularity of the Mobile Open APIs with the ranking of the popularity of the overall Open APIs.
- We tested the top popular Mobile Open APIs on an Android device, and evaluated each Open API's performance by several metrics closely related to users' quality of experience. We compared the performance of each Open API in different networks (Wi-Fi and Cellular 3G). We compared the performance of the Open APIs with similar functions.
- We carried out global-scale measurements on the characteristics of the Mobile Open APIs in multiple cities around the world. We studied the impact of different

ISBN 978-3-901882-58-6

geographical locations on the characteristics of each Mobile Open API. We also compared the performance of the Mobile Open APIs with similar functions in location-by-location level. Our results show that different Mobile Open APIs have different "sweet locations".

- The characterization of mobile Open APIs will be useful for app designers and app users. App designers can leverage the analysis to build models to predict the behavior of their apps and learn the impact of frequently called APIs. They can do various trade-off analysis and can provide advice for users regarding the access networks and the impacts therein. Overall the fine-grained measurement based characterization would lead to the design of resource-aware smartphone apps.

The rest of the paper is organized as follows. An overview of system-level issues of Open API is presented in Section II. The *APIX* implementation and ranking of Mobile Open APIs are done in Section III. The most common Open APIs are profiled in Section IV. The global-scale evaluation is reported in Section V. Section VI provides an overview of related work, followed by the inferences derived in Section VII.

II. OPEN API

In this section, we present the work flow of Open API through an example, and describe the mechanisms associated with the associated protocols.

An Example. In Figure 1, an app integrating AT&T Speech Open API is used as an example to demonstrate the basic work flow of the Open APIs [3]. AT&T Speech API provides speech recognition service, which translates spoken word into text, by using the well-known Watson Speech Recognition Engine. The app is developed by Sphero, a company that manufactures voice controlled robotic balls. When the user says a command to the app, the AT&T Speech API is invoked and the spoken voice will be streamed to the AT&T Watson Server. Then the Watson Server carries out speech recognition and sends the text back to the phone. Up to now, the whole process of invoking the Speech API is done. Afterwards, the Sphero app can start to process the text received and send the corresponding commands to the robotic ball. In Figure 1, the Apigee Gateway is a third-party API management tool to provide traffic control, statistics and security provision services [4]. The function of the AT&T API server is to distribute different kinds of AT&T Open API requests to proper handlers [3].

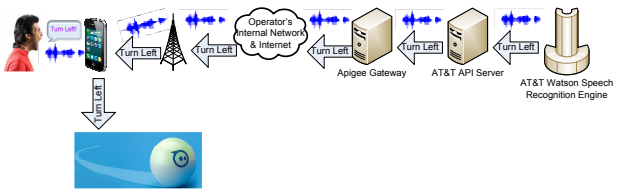


Fig. 1: An Example of Invoking AT&T Speech Open API [3]

Web Service Protocol Stack. Web service protocol is one kind of *application level* protocol which is used to define, locate, implement and let two web services to interact with each other. REST and SOAP are two widely used web service protocol stacks. According to [2], 73% of the Open APIs follow REST protocol, and 17% of the Open APIs use SOAP protocol, by the end of year 2011.

REST Protocol. REST was introduced and defined in 2000 by Fielding [5]. The REST architecture is client/server based. Each network resource has a representation. The service requests and responses between the client and servers are built around transfer of representation of the resources. Since REST standardizes http and https as its service transport protocols, a key feature of the REST-style Open APIs is that they can only be invoked by visiting http or https format Uniform Resource Locators (URLs) [6].

SOAP Protocol. SOAP protocol was the dominating web service protocol in the Open API market. It is also based on the client/server model. The key feature of SOAP is that it relies on XML format as its service messaging protocol. Since SOAP does not specify the service transport protocol, the client can use any transport protocols, e.g. HTTP and FTP, to invoke a SOAP-style Open API, by visiting the desired URL [6].

III. APIX AND RANKING MOBILE API

To the best of our knowledge, there has been no reported statistical studies on the popularity of the Open APIs among mobile app developers. The existing Open API marketing analysis [7] did not distinguish between Mobile Open APIs and the Open APIs for other non-mobile platforms. Here, we account for the Open APIs used in the most popular Android apps and present a ranking of the popularity of Mobile Open APIs. It is interesting to note that our Mobile Open API popularity ranking is completely different from the overall Open API popularity ranking presented in [7].

A. Apps Selection

We performed a comprehensive study of the popularity of the apps available in Android app stores. By the end of October 22, 2012, there are about $33,000$ popular apps each of which has more than $50,000$ downloads in Google's Android app store [8]. The statistics showed that these popular apps were distributed asymmetrically in all the 32 categories of Android apps. We then selected a total of 200 popular apps from the market according to the ratio of the number of popular apps in each category.

B. Design and Implementation of APIX

As mentioned in Section II, the existing Open APIs generally use SOAP or REST web service protocol. Both of these protocols require the apps to visit certain URLs to invoke API service requests [6]. As an example, Twitter OAuth, a popular social network Open API, allows third-party apps to authenticate an app user by verifying the user's Twitter account and password at Twitter's website. To invoke the Twitter OAuth Open API, the apps must visit the URL "https://api.twitter.com/oauth". By tracing the URLs the apps are interacting with, we are able to extract the Open API calls integrated in the apps. Therefore the problem of achieving full coverage of the URLs to invoke Open APIs becomes vital.

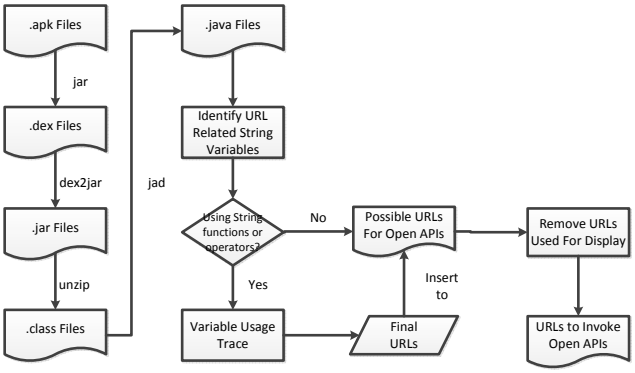


Fig. 2: Flow Chart of APIExtractor (APIX).

Earlier studies on Android security and privacy concerns [9], [10] have pointed out that the external method by monitoring the network traffic can not achieve full coverage on the API calls due to the lack of automated Java program test generators to execute all the subroutines of the mobile apps. The internal identification approach usually parses through the Android manifest file and the compiled Java files (.class) generated by reverse engineering techniques from the Dalvik Executable files (.dex). However, the existing internal identification method [10] requires a pre-known database of the targeted APIs, e.g. Android standard APIs. Moreover, the variance of the coding styles of the app developers further increase the hardness of identifying API invoking URLs, i.e. some programmers may prefer using the string.append() or the sring.replace() functions to build URL strings than assigning the values to string variables directly. Since the number of organizations which have already published Open APIs has exceeded $7,000$ and is still doubling every 6 months, a method which can identify both known and unknown Open APIs is desperately needed.

We have developed a software tool "APIExtractor" (APIX) to automatically accomplish the Open API extraction task by parsing the Java source code (.java) of apps and tracing the URLs. *APIX* is implemented in a mix of Python language and Linux Shell scripts. Before running *APIX*, 4 free Linux libraries have to be pre-installed, "jar", "unzip", "JAD (JAva Decompiler)" and "dex2jar". *APIX* takes a set of compiled Android application package files (.apk) as input, and outputs the profile of each Open API integrated in these apps. The flow chart of *APIX* is presented in Figure 2.

1) Extraction from Java source code: As shown in Figure 2, *APIX* first transfers the compiled app (.apk) files to Dalvik Executable format (.dex) which is the format to be run on Android's Dalvik Virtual Machine. Then "dex2jar" is applied to transfer the .dex files to Java Archive files which is typically used to aggregate Java virtual machine readable files (.class) and associate resources, e.g. images and text. The .class files are inserted into the "JAD" module to be decompiled to human readable Java source code files (.java). By applying *APIX* on the 200 selected apps, we successfully decompiled $13,540(99.5\%)$ out of a total of $13,615$ .class files to their corresponding original Java source code. The 65 unsuccessfully decompiled .class files are written in Java Development Kit (JDK) versions lower than 4, which are not commonly used nowadays.

2) Identification of Open API calls in Java Files: By parsing through the Java source code files and accounting the variables containing Internet URLs, e.g. http://*, https://*, ftp://* and fb-connect://*, APIX creates a list $L_{\text{potential}}$ of all possible Open API invoking URLs. We observed from the source code that sometimes the app developers constructed the URL strings by using the Java string related functions or operations, i.e. string.replace(), "+" and "-". In the $13,540$ .java files decompiled, we observed a total of 167 code blocks which include URL related Java string function calls or operators. Then we did variable usage tracing on all these 167 places and inserted the finally built URL related strings to $L_{\text{potential}}$. After removing the URLs related to Internet resources aimed for display (images, text, flashes) from $L_{\text{potential}}$, APIX finally outputs a list of the invoking URLs of the integrated Open API calls.

C. Ranking Mobile Open APIs

By running APIX on the selected apps, we identified a total of 1715 distinct Open APIs, where the term "distinct" means the URLs to invoke the Open API calls are different. From the results, we observe that majority of the invoking URLs begin with "http://" (74.1%) or "https://" (18.3%). There are a few invoking URLs that start with other headers, e.g. "fb://" (4.2%), "scoreloop://" (0.3%) and "ftp://" (0.3%).

Figure 3 shows the cumulative density function of the number of Open APIs in the selected apps. There are only $21(10.5\%)$ apps that do not use any Open APIs. Analyzing the number of distinct Open APIs in each app, we found that each app utilizes 17.8 different Open APIs on an average. There are 2 apps, Line Camera and PicsArt - Photo Studio, which integrate more than 100 distinct Open API calls. For example, Line Camera integrated 43 distinct "pinterest.com" Open APIs, 26 different "navercorp.jp" Open APIs and 12 disparate "twitter.com" Open APIs.

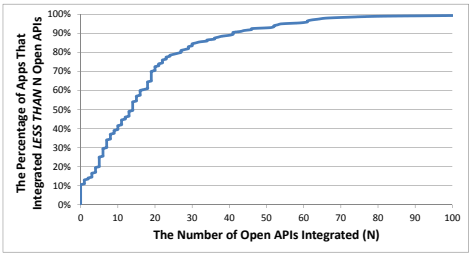


Fig. 3: The Cumulative Density Function (CDF) of the Number of Open APIs in Apps

Finally, we categorize the identified Mobile Open APIs by software development kits (SDKs) and functions, and rank the popularity of the identified Mobile Open APIs. The top 15 popular SDKs and the top 17 popular mobile Open API functions are presented in Table I and Table II respectively.

In Table I, we compare the popularity of the SDKs for Mobile Open APIs with the popularity of the SDKs for overall Open APIs. For the Mobile Open APIs identified by

TABLE I: The Ranking of Popular Mobile Open API SDKs and Overall Open API SDKs

Mobile Open API Ranking	SDK Name	The Number of Selected Mobile Apps	Overall Open API Ranking	SDK Name	The Number of Application Mashups
1	Google's AdMob	100	1	Google Maps	2424
2	Facebook	81	2	Twitter	752
3	Google API	74	3	Youtube	652
4	flurry.com	54	4	Flickr	615
5	inmob.com	39	5	Amazon eCommerce	416
6	w3.org	28	6	Facebook	387
7	Dropbox	28	7	Twillo	353
8	Twitter	23	8	Last.fm	226
9	tapjoyads.com	23	9	eBay Search	220
10	chartboost.com	20	10	Google Search	184
11	Amazon AWS	17	11	Microsoft Bing Maps	175
12	Yahoo Search	15	12	Twillo SMS	172
13	appspot.com	13	13	del.icio.us	160
14	adwhirl.com	12	14	Yahoo Search	144
15	greystripe.com	10	15	Yahoo Maps	135

TABLE II: The Identified Popular Mobile Open API Functions

Ranking	Mobile Open API Functions	The Number of Selected Mobile Apps	Web Service Protocol	The Core of The URLs to Invoke
1	AdMob: ask server to send basic ads	82	REST	http://media.admob.com/sdk-core-v40.js
2	Facebook: user authentication	64	REST	https://api.facebook.com/restserver.php
3	Facebook: post content to Facebook	64	REST	https://graph.facebook.com/
4	Facebook: download friend list	63	REST	https://m.facebook.com/com/dialog
5	Admob: mobile rich media ad interface (MRAID)	57	REST	http://media.admob.com/mraid/
6	Mydas: ask server to send basic ads	36	REST	http://androidsdk.ads.mp.mydas.mobi/getAd
7	Mydas: get rich media ads	36	REST	http://androidsdk.ads.mp.mydas.mobi/getAd.php5?
8	InMobi: get ads for testing	35	REST	http://i.w.sandbox.inmobi.com/showad.asm/
9	InMobi: ask server to send basic ads	35	REST	http://i.w.inmobi.com/showad.asm/
10	InMobi: send tracker information to server	31	REST	http://ma.inmobi.com/downloads/
11	Dropbox: download a file	25	REST	http://dl.dropbox.com/u/
12	Chartboost: ask server to send basic ads	20	REST	https://www.chartboost.com/
13	Twitter: user authentication	18	REST	https://api.twitter.com/oauth/
14	Tapjoy: get a list of offers	17	REST	https://ws.tapjoyads.com/get_offers/
15	Tapjoy: user authentication	17	REST	https://ws.tapjoyads.com/connect?/
16	Twitter: post content to Twitter	14	REST	https://api.twitter.com/1/
17	Amazon: upload a file to server	13	SOAP	http://s3.amazonaws.com/

APIX, the most commonly used SDKs are basically related to three categories: ads, social networks, and cloud storage. For the overall Open API SDKs, the most widely used ones basically fall in the areas of map, social networks, and search. There are only 4 SDKs appearing in both the top 15 Mobile Open API SDK ranking and the top 15 overall Open API SDK ranking. This observation further enhance our motivation to characterize the popular Open APIs specified for mobile apps.

In Table II, we present the top 17 Mobile Open API functions identified by APIX. These functions basically engage in 4 aspects: get ads, open standard for authentication (OAuth), social networks, and cloud storage operations. We also observe that REST is the dominating web service protocol among the popular Mobile Open APIs. Among the top 17 identified Mobile Open APIs, only Amazon AWS API uses SOAP. The popularity of REST protocol also leads to the fact that almost all the URLs to invoke API calls are based on http and https protocols, which are the only two transport protocols standardized in REST. The URLs listed in the fifth column of Table II are the core part of the URLs to invoke the Open API calls after removing the Open API user identification information and the session IDs.

IV. Profiling Mobile Open APIs

In this section, we characterize the performance of the Mobile Open APIs by using four metrics that largely impact the users' quality of experience. The four metrics are: latency, energy consumption, network traffic generated, and CPU usage. Latency is defined as the time from invoking the Open API call to getting the desired response from the server. Energy consumption quantifies the amount of Joules used in executing the API and its associated operations. The network traffic reflects the bandwidth usage requirements of the mobile APIs. The utilization of CPU is also measured for each of the APIs.

It is extremely challenging to isolate the Open API calls from local operations, assuming we do not have access to the apps' original source code, which could then be run in debug

mode. We propose an approach to profile isolated Open API calls. The basic idea is to implement apps which include only one Open API, then insert probes into the source code to profile the Open APIs. We have implemented a series of apps called APISymphone, each of the app corresponds to an API listed in Table II. We profile all the 17 Open APIs and did a comprehensive analysis on their performance.

A. APISymphone

We implemented the APISymphone series apps by using the Android SDK and 9 Open API SDKs. The pseudo-code of the framework to implement each APISymphone app is presented in Algorithm 1. All APISymphone apps include only the minimum functions and resources to invoke the Open API call. Each APISymphone app is consisted of 4 potential operations:

- Get Open API access token: Most of the Open API servers require the app identification to be submitted and verified before assigning an access token. The app identification usually consists of a unique app name and an encrypted key. The 9 SDKs we utilized offer various functions for the verification process. For example, Facebook uses "Utility.mFacebook = new Facebook(APP_ ID);".
- Prepare the local resources required for Open API calls (optional): Some Open APIs need to access certain local resources. For example, social network Open APIs need to access the corresponding user name and password. To ensure fair comparism among the APIs with similar functions, all the corresponding resources are kept constant, e.g. social network friend list and the file (193KByte) to be accessed by cloud storage APIs.
- Loop Open API calls: Since some Open API calls target on very simple tasks, some of the metrics may be too tiny to be detected, e.g. latency. In each app, we loop calling the APIs for multiple runs and calculate the average value of the metrics. To avoid the interference of caching, we clear the cache after each run.
- Probes to detect latency and CPU usage: To measure the latency and the CPU usage, probes are inserted into the app. The details of implementing the latency probe and the CPU usage probe is described in the next section (Section IV-B).

B. Probes Implementation and Overhead Analysis

We present how the latency probe and the CPU usage probe are implemented, and then analyze the overhead caused by the probes. The latency probe is implemented by using the Calendar class defined in java.util.Calendar. We measured the running time of calling one Calendar instance by running it 5 million times on a Nexus S Android phone. The overhead lies in the range of $(0.0616, 0.0667)$ milliseconds.

The CPU usage probe is implemented by making use of the ActivityManager class defined in android.app.ActivityManager and the Android log files stored in the directory "/proc". An instance of ActivityManager will return the process ID (PID) of each running process in Android phones. In the Android operating system, the process name is the same as the app package name by default. We first search for the PID of the app being tested in all the current running processes by matching the process name with the app's package name. Since Android system automatically logs and stores the total CPU usage in "/proc/stat" file and the CPU usage of each process "/proc/PID/stat" file, each time we carry out CPU measurement by deleting the old log files and reading the new log files created when the Open APIs are being invoked. Since the log file only contains the CPU usage during the Open API call, the CPU usage for the app is actually the CPU usage of the Open API. The implementation of the CPU usage probe is done by reading the log files automatically created by the Android system. Thus the CPU usage probe itself will not introduce any overhead in our measurement.

Algorithm 1 APISymphone Apps

```
Step 1 Get Open API access token;
Step 2 Prepare the required local resources;
Step 3 SumLatency=0;
Step 4 SumCPU=0;
Step 5 repeat K times
         ProbeCPU.start();
         ProbeLatency.start();
         Open API call;
         ProbeLatency.end();
         ProbeCPU.end();
         Clear cache;
         SumLatency+=ThisRunLatency(ProbeLatency);
         SumCPU+=ThisRunCPU(ProbeCPU);
Step 6 endrepeat
Step 7 AverageLatency=SumLatency/K;
Step 8 AverageCPU=SumCPU/K;
```

C. Experimental Setup

Our experimental setup consisted of a Nexus S Android phone, 17 APISymphone series apps, a Monsoon power monitor, a Dell E5400 laptop and a Cisco WRT310N wireless router. The operating system running the phone is Android 4.1.1. We set the sampling rate of the Monsoon power monitor to be 20kHz. The Dell E5400 laptop was used as the console of the monsoon power monitor. The Cisco wireless router was utilized to provide dedicated Wi-Fi access to the Nexus S phone.

To prevent our experiments from being interfered by other apps, we uninstalled as many apps as possible from the phone and disabled all the unnecessary background services by using the app management tool offered by Android. To ensure there are no other apps running in the background, we also used a well-known third-party task manager app called "Advanced Task Killer" [11].

Latency and CPU usage were measured by using the probes implemented in the apps. The power consumption metric was measured by using the monsoon power monitor. To monitor the network traffic generated during the Open API calls, we used a packet sniffing app "tPacketCapture", available in Android app store. Running "tPacketCapture" in background consumes considerable energy, and incurs latency and CPU usage overhead to the measurement. We therefore carried out the packet sniffing experiments separately.

D. Data Analysis

We ran each APISymphone app on the Nexus phone 10 times and calculated the average values of the metrics measured. The network traffic monitoring experiments were done separately and were also repeated 10 times. The experimental results are presented in Figure 4. Due to space limitations, we abbreviate the description of each Open API. The sequence of the Open APIs follows as listed in Table II.

1) Latency: Figure 4(a) shows the response latency of each Open APIs. The Open APIs are tested both on Wi-Fi and AT&T 3G (HSPA+) cellular network. Majority of the Open APIs running in 3G network incurred longer latency compared to the Wi-Fi network. For example, the latency of the DB Download API on 3G is 4.9 times higher than its latency on the Wi-Fi network. On an average, the latency of the Open APIs on 3G network is 2.3 times higher than on Wi-Fi network. One of the obvious causes is that the Wi-Fi connection usually has higher bandwidth than the 3G connection. However, the magnitude of the difference indicated other probable causes. We observed that the Open API requests are handled by different servers which incur different amount of processing latency. We use the Twitter OAuth API as an example to explain this observation. By analyzing the packets captured during the Open API call, we found that the API requests were handled by two different Twitter servers, one at IP address 199.59.150.9 (for Wi-Fi accesses), the other at IP address 199.59.149.232 (for 3G accesses). In both 3G and Wi-Fi scenarios, the phone faced $2-4$ times packet loss in a portion of the runs. In Wi-Fi scenario, the uplink and the downlink lost packets were always retransmitted in less than 120ms. However, in several runs of 3G scenario, it was observed that the downlink retransmissions incur around 36 seconds latency.

From Figure 4(a), it is also observed that the performance of Open APIs with similar functions varies a lot in terms of latency. For example, using Wi-Fi, the loading of basic advertisement from Mydas cost 62 times more latency than loading basic advertisements from Admob. Another example is that the latency of the Twitter OAuth API is 4 times the latency of Facebook OAuth API. This observation sheds light on the motivations for the app developers to pick proper Open APIs to integrate, if there are no other constraints.

2) Network Traffic: In Figure 4(b), we compare the network traffic generated by each Open API. As expected, the APIs interacting with image or multimedia resources tend to generate more network traffic, e.g., posting pictures to Facebook and downloading ads video from Madas. The Wi-Fi scenario generated more traffic than the 3G scenario for some APIs, and less for the others. The average difference of the traffic values in Wi-Fi and 3G scenarios is only 0.7%. Unlike the latency comparison, we did not observe much difference between the APIs with similar functions in terms of traffic generated.

3) Energy Consumption: Battery life is also an important concern for the app users' quality of experience. In Figure 4(c), we present the energy consumed to carry out each Open API call. For each API, the energy consumption on 3G is usually higher than the power consumption on Wi-Fi network. This is not only because the 3G radio consumes more power than the Wi-Fi radio [12], but also due to the higher latency incurred in the 3G access network.

To compare the energy consumption of the API with similar function, we use the Nexus S battery as an example and assume there is no other apps running on the phone. Considering the $1,500$ mAh and 3.7V Nexus S battery, the total energy is $19,980J$. By using the corresponding Open APIs, users can post approximately $6,263$ photos to Facebook, but only about $5,328$ photos to Twitter, in Wi-Fi network. In 3G network, these numbers reduced dramatically to $1,237$ for Facebook and to $4,768$ for Twitter.

4) CPU Usage: CPU usage is also an important parameter that the app developers should consider before integrating the Open API into their apps. We present the CPU usage of each tested Open API in Figure 4(d). There are 2 CPU resource draining APIs: AdMob Basic and Twitter OAuth. These two APIs consume over 60% of the CPU time when connecting through Wi-Fi, and consume over 80% of the CPU time while connecting through 3G network. Considering the 15 second latency of running Twitter OAuth on 3G, it is highly probable that the users are driven into the risk of incurring long time system lag even freeze.

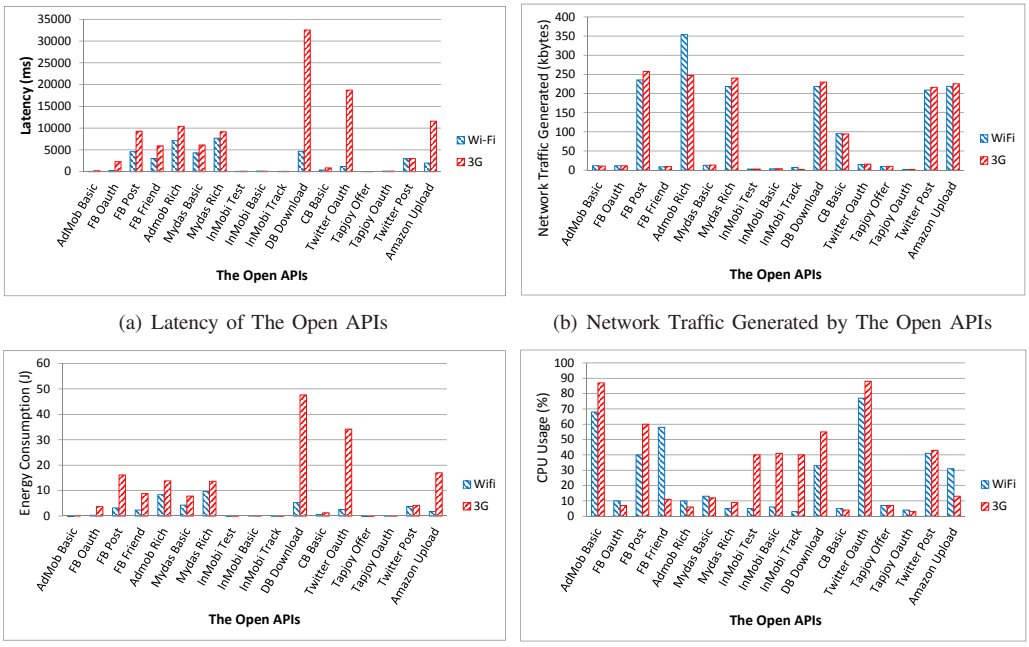
5) Inferences: The analysis of the data infers the following aspects:

- Latency of an Open API depends not only on the network bandwidth but also the behaviors of the Open API servers. The apps designers may choose to exploit the fact regarding the variance of latency because of the access (WiFi vs 3G).
- Open APIs with very similar functionality exhibit highly varied latency. The apps designers can analyze the trade-off between the level of desired functionality and the latency overheads before developing their apps.
- The network traffic consumption does not show any definitive trend for mobile Open APIs.
- The variance in energy consumption is quite significant for the Open APIs, especially while using different access networks. For certain specific APIs, if their frequency of usage is high, the preferred access network may be advertised to conserve energy.
- The CPU usage factor may impact the quality of certain apps if they overuse a few specific Open API calls.

Using the aforementioned analysis of latency, network traffic, energy consumption, and CPU usage, designers would be able to model and predict the behavior of the apps that use these APIs. Detailed and fine-granular models can be developed using the characteristics of the open APIs. Based on the analysis, the users can be apprised of the impact of access networks on their apps if they (the apps) use frequent calls to a few specific open APIs.

V. Global Scale Evaluation

Due to increase of the popularity of Open APIs among app developers, apps integrating Open APIs are being used by smartphone users all over the world. It is thus desirable to do a global-scale test on Mobile Open APIs to study and analyze their impact with respect to the geographical locations. In this section, we proposed a global-scale measurement-based study on the performance of the Open APIs by making use of the Amazon's Elastic Compute Cloud (EC2) services. The results and corresponding analysis are also discussed.



(a) Latency of The Open APIs

(b) Network Traffic Generated by The Open APIs

(c) Energy Consumption of The Open APIs

(d) CPU Usage of The Open APIs

Fig. 4: Measurement Results of The Open APIs

A. Amazon EC2 Instances Configuration

We set up Amazon instances at 7 locations in 5 continents, as listed in Table III. For all the instances, we chose the M1.Large type virtual machine with duo core 3.0GHz CPUs and 7.5GB RAM.

TABLE III: Virtual Machine Instances

EC2 Region	Location	Static IP Address
N. Virgina	Ashburn, VA	174.129.3.36
Oregon	Boardman, OR	54.245.235.96
N. California	San Jose, CA	54.241.15.165
Europe	Dublin, Ireland	79.125.123.198
Asia Pacific	Singapore, Singapore	54.251.114.141
Asia Pacific	Sydney, Australia	54.252.99.12
South America	Sao Paulo, Brazil	177.71.189.164

On each EC2 instance, we set up an AVD. The device type was set to be Nexus S with Android 4.1.1. In the next step, we selected 6 Open APIs from Table II to be tested on these instances:

- Two ads related Open APIs: AdMob and InMob
- Two social network related Open APIs: Facebook and Twitter
- Two cloud storage related Open APIs: Dropbox and Amazon.

The corresponding APISymphone apps were launched sequentially on the AVD to be tested. For the AVDs, we only profiled the latency and the network traffic generated. The experimental results are shown in Fig 5 through Fig 7. Each data in these figures is also the average value of 10 runs.

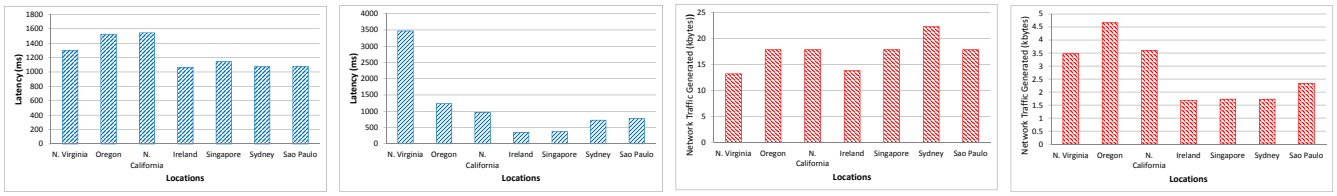
B. Case Studies of The Selected Open APIs

In this section, we provide an in-depth view into the performance of the Open APIs in 7 different locations.

1) Ads Related Open APIs: AdMob Basic Ads and InMob Basic Ads APIs are two representative ads related APIs. Their basic functions are quite similar. When Ads APIs are invoked, the phone will send an ad request, with users' context information, to the ads server. Then a thread is created to continuously receive ads from the server. In our implementation, each run of the APIs is from the invoking to the moment that the first ad is completely cached. Between each two runs, we clear the cached ad data.

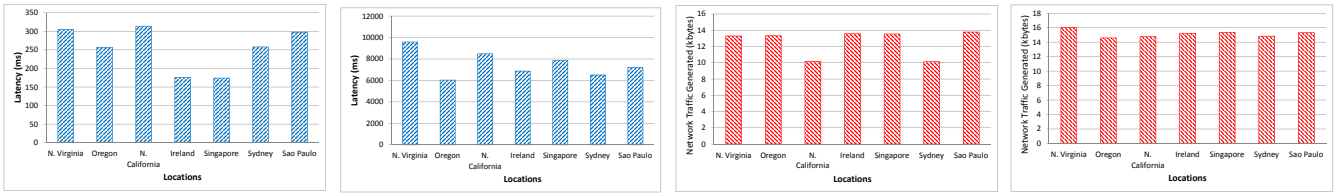
Figure 5(a) shows the latency of AdMob Basic Ads API is basically around 1200 ms and does not vary much around the average value. The AVDs in Oregon and N. California are observed to suffer roughly 37% more delay than the AVDs located in Ireland. On InMobi Basic Ads API, we observe another kind of characteristic of latency. As shown in Figure 5(b), the latency of the InMobi Basic Ads API in N. Virginia is much higher than in any other places. However, by comparing and analyzing the traffic packets captured at all 7 locations, we did not observe that the N. Virginia AVD is suffering higher retransmission rate than the other AVDs. We believe the difference in the server processing time is the main reason that explains the pattern presented in Figure 5(b).

Both Figure 5(c) and Figure 5(d) show that the traffic generated by the ads APIs varies from place to place. This behavior is expected as both the Ads APIs send the ads request as long as the users' context information to the servers. Since every AVD is with the same setting except the location, it is very likely that the amount of ads traffic difference is caused by the location context. It is also observed that AdMob API



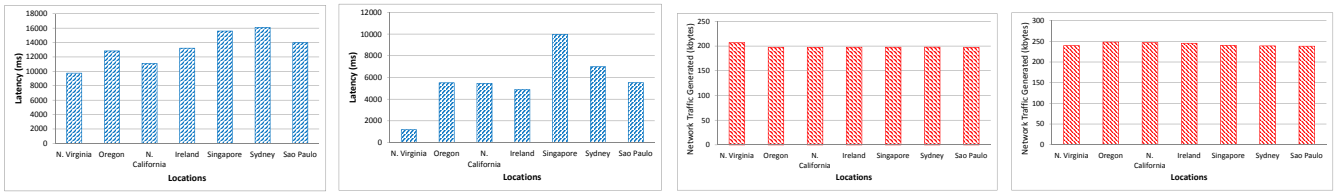
(a) Latency (AdMob Basic) (b) Latency (InMobi Basic) (c) Network Traffic Generated (AdMob Basic) (d) Network Traffic Generated (InMobi Basic)

Fig. 5: Basic Ads Open APIs in Different Locations



(a) Latency (FB Oauth) (b) Latency (Twitter Oauth) (c) Network Traffic Generated (FB Oauth) (d) Network Traffic Generated (Twitter Oauth)

Fig. 6: Oauth Open APIs in Different Locations



(a) Latency (DB Download) (b) Latency (Amazon Upload) (c) Network Traffic Generated (DB Download) (d) Network Traffic Generated (Amazon Upload)

Fig. 7: Cloud Storage Open APIs in Different Locations

generates much more traffic than InMobi API in all of the locations. We also observe from the captured packets that AdMob API's uplink and downlink traffic ratio is about $1:7$, while the InMobi API's uplink and downlink traffic ratio is $1:0.7$. In other words, AdMob uses a much richer media format for basic ads delivery than InMobi.

2) Social Network OAuth Open APIs: OAuth Open APIs allow users to get a token to access their private resources on the OAuth API publisher's website from a third-party website, by supplying their account and password of the OAuth API publisher's website. Facebook OAuth and Twitter OAuth APIs are two representative OAuth Open APIs.

Figure 6(a) shows the latency to get a Facebook token from different locations. It seems the Facebook OAuth API works pretty well in all the locations with less than $350ms$ latency. According to [13], human's average reaction time is 215ms. That means the users will feel like the authentication is done almost immediately.

The latency caused by Twitter OAuth API is much longer than Facebook. Figure 6(b) shows the AVDs need around 7 seconds to get a token from Twitter. We believe the main reason is the server's processing latency, since we did not observe considerable packet loss in packet traces.

In terms of the traffic generated, the two APIs perform almost the same. As shown in Figure 6(c) and Figure 6(d), each call of the Facebook OAuth API and the Twitter OAuth API generates 12.54kbyte and 15.15kbyte traffic on average respectively.

3) Cloud Storage Open APIs: We also analyze two cloud storage Open APIs in detail. The function of the Dropbox Download API is to download a file in the our Dropbox account from Dropbox server. The Amazon Upload API is to upload a file to our account on Amazon Simple Storage Service (S3). The file we choose to upload and download is the same image file (193Kbyte) used in the Nexus S based experiments.

The latency and traffic of the Dropbox Download Open API is presented in Figure 7. According to our observation, all the AVDs interact with the same server (IP 199.47.219.158) to carry out the downloads. From Figure 7(a), we can observe that the AVD in N. Virginia enjoys the lowest latency. Figure 7(c) shows the average overhead of downloading the 193 Kbyte file is 6 Kbyte, where overhead is defined as the difference of the file size and the traffic

generated.

We present the metrics of Amazon Upload Open API in Figure 7(b) and Figure 7(d). It is observed that the latency in N. Virginia is much lower than in other locations. For example, the AVD in Singapore suffers 7 times more latency than the AVD in N. Virginia. As shown in Figure 7(c) and Figure 7(d), Dropbox Download API generates 6.4Kbyte overhead on average, while Amazon Upload API introduces 49.5Kbyte overhead.

VI. Related Work

To the best of our knowledge, none of the prior efforts have focused on profiling the performance of the Mobile Open APIs. Prior works have investigated several related aspects such as user behaviors, energy consumption, signaling overhead, and security and privacy issues at the phone level or at the app level. In [14], Falaki *et al.* analyzed the diversity of smartphone user behaviors, e.g. the pattern how the smartphone users typically use the smartphone and apps.

Efforts in [15]–[17] profiled and analyzed the energy consumed by smartphone apps. Pathak *et al.* implemented *eprof*, a tool which can estimate the power consumption of apps by tracing the system calls. In [17], another study applied a model for Radio Resource Control (RRC) to estimate the power consumption on the 3G radios of Android phones.

Some previous works focused on profiling the overhead generated by mobile advertisement deliveries and analytic data collections. According to [18], 77% of the top free Android apps were implanted with at least one third-party ad component that downloads ads in real-time. In [11], the authors pointed out the analytics data collection will introduce overhead to the Android system. The amount of overhead is qualified by comparing the packets of the paid version and the free version of the apps. A comprehensive analysis of the mobile advertising ecosystem is presented a most recently work [19]. There are also a few multi-layer Android profiling efforts reported in the literature. For example, Wei *et al.* developed *ProfiledDroid*, a multi-layer system to monitor and profile the apps [20]. *ProfileDroid* profiles the apps in four layers: static, user interaction, operating system and network.

Unlike the previous efforts, APIX is not focused on the manifest file of the apps but on the source code directly to achieve extremely high coverage (99,54%) of the search space. Furthermore, we also proposed a methodology to test isolated Open APIs by implementing APISymphone series apps. Lastly, to the best of our knowledge, this is the first to propose a methodology for profiling the performance of the Open APIs in global scale by making use of Amazon EC2 services.

VII. Conclusions

The software *APIX* we built for this work provides an accurate approach with extremely high coverage (99.5%) to identify both known and unknown Open APIs integrated in the apps at function level. Relying on APIX, we identified the Mobile Open APIs integrated in the selected Android apps. By comparing our ranking of the popularity of the Mobile Open APIs with the ranking of the popularity of the overall Open APIs, we observed that the two rankings are almost distinct. We tested the top popular Mobile Open APIs on Android devices in both Wi-Fi and 3G networks, and characterized each Open API's performance by four metrics: latency, traffic generated, energy consumption, and CPU usage. Our results show the Open APIs basically suffers longer latency and higher energy consumption in 3G network than in Wi-Fi network. It is also shown the network traffic and the CPU usage does not follow definite trend in 3G versus Wi-Fi comparison. We also observed differing characteristics of the Open APIs with similar functions. Global-scale measurements were also performed to study the impact of different geographical locations on the characteristics of the Open APIs. Our experimental results expose the potential to optimize the performance of the smartphone apps by analyzing the performance and resource requirements of the popular mobile Open APIs.

References

[1] A. Nghiem, *IT Web Services: A Roadmap for the Enterprise*. Prentice Hall, Oct 8, 2002.

[2] J. Musser, *ProgrammableWeb, Open APIs: State of the Market*. AT&T Summit, 2012.

[3] M. Owens and J. Lieske, "Step-by-step with at&t speech," Sep 2012.

[4] Apigee, http://apigee.com/about/.

[5] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

[6] D. Chappell, *SOAP vs. REST: Complements or Competitors?* 2010.

[7] Programable Web Open API Popularity Ranking, http://www.programmableweb.com/apis/.

[8] AppBrain. http://www.appbrain.com/stats/.

[9] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," MobiSys '11, pp. 239–252, ACM, 2011.

[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," CCS '11, pp. 627–638, ACM, 2011.

[11] L. Zhang, D. Gupta, and P. Mohapatra, "How expensive are free smartphone apps?," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 16, no. 3, 2012.

[12] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," IMC '09, ACM, 2009.

[13] G. R. Grice, R. Nullmeyer, and V. A. Spiker, "Human reaction time: Toward a general theory.," *Journal of Experimental Psychology: General*, vol. 111, no. 1, pp. 135 –153, 1982.

[14] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," MobiSys '10, pp. 179–194, ACM, 2010.

[15] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," EuroSys '12, pp. 29–42, ACM, 2012.

[16] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," MobiSys '12, pp. 267–280, ACM, 2012.

[17] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: a cross-layer approach," MobiSys '11, pp. 321–334, ACM, 2011.

[18] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, "Don't kill my ads!: balancing privacy in an ad-supported mobile application market," HotMobile '12, pp. 2:1–2:6, ACM, 2012.

[19] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft, "Breaking for commercials: characterizing mobile advertising," IMC'12, ACM, 2012.

[20] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: multi-layer profiling of android applications," Mobicom '12, ACM, 2012.