# LIBSPECTOR: Context-Aware Large-Scale Network Traffic Analysis of Android Applications

Onur Zungur
*Boston University*
Boston, USA
zungur@bu.edu

Gianluca Stringhini
*Boston University*
Boston, USA
gian@bu.edu

Manuel Egele
*Boston University*
Boston, USA
megele@bu.edu

*Abstract*—**Android applications (apps) are a combination of code written by the developers as well as third-party libraries that carry out most commonly used functionalities such as advertisement and payments. Running apps in a monitoring environment allows researchers to measure how much network traffic is exchanged between an app and remote endpoints. However, current systems currently do not have the ability to reliably distinguish traffic that is generated by different libraries. This is important, because while mobile users are paying for data traffic without distinctions, some of this traffic is useful (e.g., data for core app functionalities), whereas the rest of the traffic can be considered a nuisance (e.g., excessive advertisements).**

**In this paper, we present LIBSPECTOR, a system that precisely attributes network traffic coming from an Android app to the library that generated it. To this end, we instrument the Android Framework to inspect the network connections initiated by apps, provide fine-grained information on the libraries in use, and calculate method coverage information while performing dynamic analysis. We then perform a measurement on 25,000 popular Android apps and investigate the relation between different categories of apps with the use of specific libraries. We analyze the method coverage of our dynamic analysis method, and further characterize the endpoint connections established by the Android apps. Our results indicate that advertisement libraries account for over a quarter of the total data transmission. We further observe that there is no strict 1-to-1 correlation between the similar categories of network endpoints and libraries which initiated the data transfer.**

## I. INTRODUCTION

Mobile applications, or apps, are a significant reason for the success of mobile smart devices over the last decade. Apps allow end-users to extend the capabilities of off-the-shelf mobile devices with functionality that the original designers did not anticipate. Today, the two most prominent mobile platforms, Google's Android and Apple's iOS, give users access to market places that each host in excess of 3.7 million third-party apps, many of which accumulated billions of installations [39]–[41]. A further testament to the success of mobile apps is the amount of revenue that app developers can generate. For example, SensorTower [37] reported that global mobile app revenue for Google PlayStore apps reached $7.1 billion for the first quarter of 2019 with a 20.2% increase year over year. In addition to revenues generated from App Store sales, developers can also tap into alternative revenue streams which frequently come in the form of advertising.

The popularity and diversity of the app-ecosystem has resulted in a multitude of measurement studies that analyze these systems from various angles. For example, Petsas et. al [31] investigated how rankings on app-stores affect the install base of apps, whereas Wang et al. [45] focused on third-party library prevalence, API levels, privileges and malware occurrences. Most closely related to our work are the ad-library network traffic detection by Xue et al. [47], Maier et al. [28] and Tongaonkar et al. [42]. In their studies, Xue et al. and Maier et al. used `User-Agent` field in `HTTP` headers, whereas Tongaonkar et al. used hostnames for identifying ad-library traffic.

Prior work focused on using the information contained in the network packets to identify which libraries generated the network traffic. Unfortunately, in general, modern mobile apps consist of an amalgamation of developer-authored and "external" library code, both of which can generate network traffic. Therefore, treating all network data equal when attributing it to an app, classifying the network traffic based on header information or network endpoints do not adequately consider this development, and might produce inaccurate results. For example, the prevalence of generic identifiers in HTTP headers, same hosts (i.e., companies) serving multiple apps and the use of Content Distribution Networks render a purely network-focused analysis of library traffic insufficient for reliable traffic attribution.

In this paper, we are interested in answering questions such as, how much network data that is sent or received by an app belongs to first-party (i.e., developer-authored) code, and how much of that data serves auxiliary purposes, such as advertisement, or statistic usage information collection. Additionally, we are interested in questions, such as which library categories (e.g., development aid) are responsible for generating what fraction of the network traffic.

To answer such detailed questions, it is not sufficient to attribute network traffic to an individual app or use network packets only. Instead, we argue that the attribution of network traffic to app components requires more precise runtime information, from which we can derive contextual information and gain insights on network activities.

To obtain this additional contextual information, we built LIBSPECTOR, a dynamic analysis system for Android apps. LIBSPECTOR installs each app in an Android emulator and exercises it while monitoring the network traffic and the app's execution in detail. Given that Android apps are inherently driven by user-interface interactions, we leverage the Android `monkey` [4] User Interface exerciser tool. Similar to prior work, we record packet captures of all network communications in and out of the emulated environment. However, to derive the information necessary to attribute network packets to classes and methods in an app, we modify the Android framework to capture this information at runtime. Specifically, every time the app connects a network socket, our modified system attributes the resulting socket-pair (i.e., the tuple of (srcIP, srcPort, dstIP, dstPort)) with the corresponding Java method. With this additional information at hand, we can attribute each packet to the method that connected the corresponding socket. Based on this detailed attribution, we can now measure, for each app, how much traffic is originating from which and what kind of library, what kind of network endpoints the packets are destined, and how much of the app code is leading to network-related activities.

We performed a large scale analysis on 25,000 apps that span 49 of the app categories in Google's Play store. We observed that runtime inspection provided us with more detail on the established network connections where the network-based classifications could lead to inaccurate results. To the best of our knowledge, our study is the first dynamic analysis on Android apps which can attribute network traffic to apps' third-party libraries using app runtime information with method-level granularity. Furthermore, we augment our measurement results with existing data-sources (e.g., the library categorization efforts of LibRadar [26]) to shine additional light on the network traffic behavior on entire library categories.

In summary, this paper makes the following contributions:

- We design and implement LIBSPECTOR, a fine-grained measurement system that attributes network packets to the method and library of an Android app that is responsible for sending or receiving that packet (§II-A).
- Based on this automated analysis capability, we analyze 25,000 Android apps via large-scale dynamic analysis, where we exercise each app via Android's `monkey`.
- By analyzing the resulting captured network traffic, we demonstrate the importance of library analysis using app-context, and make the following observations: i) advertisement libraries cause a quarter of the mobile app network traffic, ii) 35% of apps *only* had advertisement and tracker (AnT) traffic, whereas 89% of the apps had *some* AnT traffic, iii) there is not always a strict 1-to-1 correlation between libraries and connection endpoints of the same category, and iv) estimated advertisement traffic costs \$1.17 to users and causes 18.7% more energy consumption.

## II. LIBSPECTOR OVERVIEW

In this paper we aim to collect fine grained information on which network data is generated by which libraries in an Android app. As such, our main goals when designing our analysis system are to i) exercise an app to determine which libraries and methods cause a network connection and measure Java method coverage, ii) measure how much data flow we have in each direction per library, and iii) study the relationship between categories of apps, libraries, and domains in terms of network connectivity. Therefore, we design and implement a system with the following design choices:

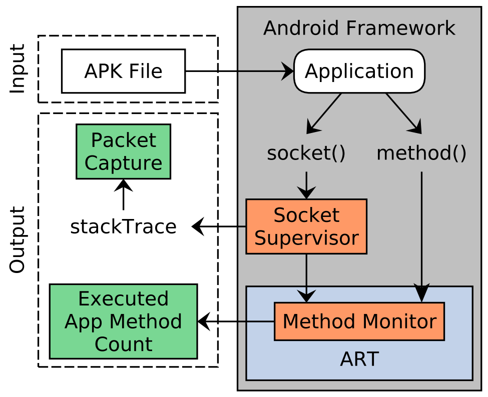


Fig. 1: Data Collection System Overview.

**Dynamic analysis:** Exercising apps provides network connectivity and interaction between apps and the external servers, which yields information on the mobile network traffic.

**Fine-grained network library analysis:** The information contained in network packets is not fine-grained or accurate enough to reliably attribute data flow to specific libraries [10], [46]. Thus, the analysis environment should provide sufficiently detailed information to associate libraries to individual connections.

**App integrity:** Apps under scrutiny should not be modified or have their integrity broken. This ensures that the internal structure of apps is not tampered with instrumentation and LIBSPECTOR is compatible with existing app stores.

**Scalability:** The data collection system should be highly parallelized and scalable in response to the millions of apps available on app stores.

**Coverage awareness:** When measuring network data in a dynamic analysis, it is important to measure what fraction of the app methods our system invokes, which determines the amount of code execution.

To this end, we implemented LIBSPECTOR, a scalable dynamic analysis framework with fine-grained network and Java method coverage measurement which does not require app modification.

### *A. System Design*

LIBSPECTOR comprises of two main high-level components for data collection during app exercising: *Socket Supervisor* and *Method Monitor*. The *Socket Supervisor* monitors the creation of sockets, extracts information on methods that leads to the creation of any socket and reports this information to a data collection server. The *Method Monitor* keeps track of all the Java methods that an app has executed, records observed methods, and then provides information on what fraction of the app's code is covered. A general overview of the data collection architecture is shown in Figure 1.

*1) Socket Supervisor:* Any network communication in Android starts with a `socket` system call, regardless of whether an Android app establishes the network connection from managed Dalvik code or natively. It is therefore fundamental for a fine-grained network-activity data collection system to monitor and examine the creation of sockets. Here, we define the fine-grained information (i.e., context) as the Java call stack that is related to the socket creation, which includes information on the libraries.

*2) Method Monitor:* One of the drawbacks of dynamic analysis is achieving complete coverage. Since every app may use different series of method calls leading to a network connection, lower coverage rates indicate unexplored methods. In such cases, the dynamic analysis would result in incomplete data for total traffic volume and missing attribution of data transfer to libraries, as unexplored methods may cause more network traffic or belong to different libraries than the already executed ones. Having an accurate method coverage measurement provides information on the lower bound of the collected data over the course of our dynamic analysis and quantify the accuracy of our system.

While the *Socket Supervisor* can provide information on network connections (i.e., Java methods in the call stack that leads to a socket), the *Socket Supervisor* cannot monitor non-socket-related information. Therefore, we design the *Method Monitor*, which monitors all the Java methods and creates a set of methods that that the app invokes. Subsequently, the *Method Monitor* disassembles the apps' `dex` file under the `apk` package to obtain a full set of methods that the app includes, and calculates the Java method coverage as a ratio of executed app methods over total app methods.

## B. System Implementation

*1) Method Monitor:* We implemented the *Method Monitor* as a combination of a modified ART runtime and Android's built-in debugging tools. The implementation of the ART runtime includes an API through Activity Manager to measure app performance with Android Profiler. This API provides provisions to monitor Java method calls using the Android Debug Bridge (`adb`), to register method signatures and timestamps when a Java method is entered or exited. Hence, we use this API to control the Android Profiler and register listeners for method invocations. The ART runtime, by default, stores the collected data in a user-specified buffer, which is insufficient (i.e., filled within seconds of app initialization) for the amount of data we collect during our experiments as listeners also record repeated calls to a method. Therefore, we modified the ART runtime such that Android Profiler *only* records unique methods when the app calls methods for the first time.

*2) Socket Supervisor:* We implemented the *Socket Supervisor* as a custom module for the Xposed Framework [35], which provides an API to modify the behavior of user-space apps, enabling LIBSPECTOR to monitor the creation of sockets. The *Socket Supervisor* collects information on a socket consisting of i) which app sent the data, ii) network connection's socket pair parameters (i.e., destination and source IPs and ports) and, iii) the stack trace at the time a socket is created.

Upon the establishment of a network connection, the *Socket Supervisor* gathers the active stack trace by invoking Java's built-in `getStackTrace` method. This method returns a list of active stack frames that are related to the creation of socket, and hence the network connection. Then, similar to our previous work BorderPatrol [50], we obtain a mapping of every active stack frame to their respective method signatures. Finally, the *Socket Supervisor* prepends the connection information to the list of method signatures, and sends this data to our data collection servers using UDP packets. For every unique socket that the app creates, the Xposed module includes a `sha256` checksum of the `apk` file and socket pair parameters along with the translated stack trace.

For the above operations, the *Socket Supervisor* relies on two submodules, the custom Xposed module for socket monitoring and and a custom shared library for obtaining connection parameters. In the following paragraphs, we describe these two submodules in detail.

*a) Xposed Framework and Custom Module:* Our custom Xposed module and framework serves as the socket call interceptor and context extractor. This module first places *post* hooks on the `socket` and `connect` method calls, and then parses the `dex` files that the app's `apk` package includes to obtain detailed information on methods including their class hierarchy and parameters. Using *post* hooks ensure that at the time of program-flow interception, there is a network connection with distinct parameters. We use the parsed `dex` file information to provide a translation from method names in a stack trace to their respective method signatures. In addition, this module obtains the socket pair parameters. This ensures that we can match the packet we observe in the network packet capture with its respective socket call stack when we obtain the contextual information from UDP packets (explained in *Shared Library* section below). Subsequently, the module creates one UDP packet per socket, which includes the method signature translations of the stack trace we gathered. Using method signatures ensures that we can differentiate the overloaded variants of methods with the same name within one class when we examine the stack traces of sockets.

*b) Shared Library:* LIBSPECTOR gathers socket-related stack traces and sends them in separate UDP sockets, which does not include the same connection parameters as the TCP socket. Consequently, we have to associate the stack trace information included in UDP packets with their respective TCP socket. To associate a stack trace with its socket pair, we use the socket's set of connection parameters (i.e., source and destination IP's and ports). Since all socket pairs have a unique set of connection parameters for any given point in time, any packet that includes these parameters in their header belongs to the connection of the aforementioned socket pair. To access the connection parameters of a given socket, we use the `getsockname` and `getpeername` system calls. For this purpose, we compile a shared library which exposes the required system calls to the Xposed module via the Java Native

Interface.

*3) Experimental Setup:* LIBSPECTOR's data collection framework consists of a job dispatcher and multiple workers which run different and fresh copies of the same modified Android 7.1.1 image (i.e., same user profile, advertising and device IDs without account logins) in Android emulators on CentOS 7 servers. Every worker pulls the assigned `apk` from a database server and exercises the app in the Android emulator using `adb monkey` [4] User Interface exerciser with 1,000 events and 500ms throttling between events. While the apps are exercised, we record all the network traffic of the emulator into a packet capture file and collect the set of visited methods in the emulator. At the end of each experiment, our modified framework writes the set of method signatures which the app invoked during experiment into a file and sends the packet capture of all the network traffic of the emulator to a central database for later evaluation. Our performance analysis shows that, LIBSPECTOR incurs a 0.5ms (9.75%) worst-case packet delay per request on the mobile device. Offline analysis and heuristics, excluding external data scraping and database activities, on average takes less than 5 seconds per app.

## III. DATA

In this section, we describe the data sources that we used as inputs to our system, and the methodology for our analysis.

### A. App Collection

To collect a representative set of Android apps, we extracted a list of the most downloaded and free Android app package names from AndroidRank [5], which is a website that keeps track of the Google PlayStore [16] app metadata. Then, we cross-referenced these package names with the list of available apps from the AndroZoo dataset [2], which is a repository of Android apps for research purposes. For certain apps, the AndroZoo dataset includes more than one version of the same app, collected at different times. Each app in the dataset also lists the date in which the app was created as specified in the `dex` file as well as the date of the latest VirusTotal [44] scan of the `apk`. For all the package names we collected from the AndroidRank, we retrieved the `apk` from the AndroZoo dataset with the latest `dex` time stamp. For packages with the default `dex` time stamps (i.e., 01-01-1980), we selected the `apk` that was most recently scanned via VirusTotal (VT). At the time of our experiments, there were no `apk`s that had neither the non-default `dex` time-stamp nor a VT scan date. We further filtered out apps that *only* included `ARM` shared libraries, as LIBSPECTOR supports `x86` compatible apps.

### B. Output Data Set

We exercised every app for 8 minutes using `adb monkey` UI exerciser [4] and issued 1,000 random events with 500ms delay while recording all network activity of the emulator. During our dynamic analysis, we collect (i) a set of method signatures of the methods that the app executed, (ii) stack traces of `socket` calls that we obtain via the `getStackTrace` Java method, (iii) the respective method signatures of each stack frame, and (iv) source and destination IPs and port numbers of each sockets. Additionally, we use the `dexlib2` [18] library to extract all the method signatures contained in a particular `apk`. Previous work by Reyes et al. [33] found that the `adb monkey` tool matched or exceeded humans' app screen coverage 61% of the time. However, we should note that, due to the randomness of the `monkey`, the results we present constitute a lower bound on the method coverage, and hence the extensiveness of these apps' network activities.

### C. Traffic Attribution

In this section, we present the methodology that allows us to determine how much data the app libraries consume.

Java classes are organized in packages according to a hierarchical naming pattern where dots separate the levels of hierarchy. The structure of the package hierarchy represents the relationship between classes and methods. Similar to Plumicke [32], we define the type signature of a method as a unique identifier which includes all the levels of this hierarchy, including the method signature with input and return value types. Furthermore, a disassembled `dex` file (i.e., `smali` code) clearly shows this structure in type signatures[1]. However, the hierarchical structure of the packages can be arbitrarily deep (i.e, length of the package names can be arbitrarily long). Since we can filter method and class names from a type signature, we use our custom Xposed module to obtain the type signatures of the respective call frame and obtain the package name for every active method call in the call stack.

As we are primarily interested in third-party libraries and their respective network connections, we then eliminate the method calls to Android's built-in packages. To do so, we refer to Android API 25 Class Index [15] and introduce a regular expression rule to filter out call frames of built-in packages[2]. We then use LibRadar [26], a tool that detects and categorizes third-party libraries in Android apps. For the majority of applications, LibRadar is able to detect the libraries contained in apps. However, as there are first party-authored packages that LibRadar has not encountered before with varying degrees of hierarchical depth, it cannot resolve all the libraries of an app. Consequently, we determine the library as the hierarchically greatest matching package structure (i.e., longest matching prefix) among all the libraries that LibRadar has detected across 25,000 apps.

Finally, we attribute the socket activity to the library of the chronologically first called method from a non built-in library in the stack trace. We then define *origin-libraries* as the libraries that such methods belong. While our analysis mainly focuses on the *origin-libraries* and their connections, we also provide analysis for libraries with a reduced-granularity. For these libraries, we select only the top two levels of hierarchy,

[1]Smali convention for a method's type signature is Lpackage/name/className$innerClassName;->methodName (inputTypes) returnTypes

[2]android.*, dalvik.*, java.*, javax.*, junit.*, org.apache.http.*, org.json.*, org.w3c.dom.*, org.xml.sax.*, org.xml.pull.v1.*

and name them *2-level libraries*. The reduced granularity provides us with the information on the activities of domains (and companies) that libraries belong.

Listing 1 shows a stack trace collected during our experiments and demonstrates the logic behind *origin-libraries* attribution. The first frame (line 1) represents the chronologically last method invocation before the creation of the socket. The frames on lines 13 and 14 include internal API calls, which we eliminate with regular expression rules. Consequently, we attribute the socket creation to the method call in line 12, which precedes all other method invocations. As per our library name extraction methodology, we determine the *origin-library* as `"com.unity3d.ads.android.cache"`. Consequently, the *two-level library* is `com.unity3d`.

```
1  java.net.Socket.connect
2  com.android.okhttp.internal.Platform.connectSocket
3  com.android.okhttp.Connection.connectSocket
4  com.android.okhttp.Connection.connect
5  com.android.okhttp.Connection.connectAndSetOwner
6  com.android.okhttp.OkHttpClient$1.connectAndSetOwner
7  com.android.okhttp.internal.http.HttpEngine.connect
8  com.android.okhttp.internal.http.HttpEngine.sendRequest
9  com.android.okhttp.internal.huc.HttpURLConnectionImpl.execute
10 com.android.okhttp.internal.huc.HttpURLConnectionImpl.connect
11 com.unity3d.ads.android.cache.b.a
12 com.unity3d.ads.android.cache.b.doInBackground
13 android.os.AsyncTask$2.call
14 java.util.concurrent.FutureTask.run
```

Listing 1: Stack Trace Example

## D. Library Categories

To extract the categories of *origin-libraries*, we again rely on the output of LibRadar. We first run LibRadar on all the apps that we collected (§ III-A). We then construct an aggregated list of libraries with their respective categories that LibRadar provides. Additionally, we use Li et al.'s work [23] to identify common advertisement/tracker (AnT) libraries, which increases precision and ensures a more comprehensive analysis regarding the AnT traffic.

For libraries where LibRadar cannot determine the corresponding category, we apply a majority voting heuristic. Listing 2 is an example of our library categorization methodology for *com.unity3d.example*, where LibRadar cannot provide a category. Here, we first find the longest matching organizational structure (i.e., common prefix) across all the libraries that LibRadar detects in our app dataset. (i.e., *com.unity3d*) Then, we collect all the libraries which start with the common prefix and their categories into a list (i.e., lines marked with [LibRadar] in Listing 2). Afterwards, we use majority voting within this list to predict the category of the unknown library, and hence determine the category of *com.unity.example* as Game Engine, which has the most votes. Similarly, the category of the *origin-library* of the stack trace in Listing 1 solely depends on *com.unity3d.ads*, as it is the longest prefix and the only matching library.

```
[LibRadar] com.unity3d -> Game Engine
[LibRadar] com.unity3d.ads -> Advertisement
[LibRadar] com.unity3d.plugin.downloader -> App Market
[LibRadar] com.unity3d.services -> Game Engine
[Predicted] com.unity3d.example -> {Game Engine:2,
     Advertisement:1, App Market:1} -> Game Engine
[Predicted] com.unity3d.ads.android.cache ->
     {Advertisement:1} -> Advertisement
```

Listing 2: LibRadar category results of unity3d and category prediction for two related libraries

## E. Traffic Volume

LIBSPECTOR sends a UDP packet which contains information on a socket right after the connection is established, and thus lacks the information on how much data is transmitted over a particular socket during an experiment. Consequently, we calculate the data transfer size *after* the connection is closed, which is the sum of all TCP packets within the same stream (i.e., the packets which possess the same connection parameters as the socket itself). First, we associate TCP packets that the socket sent by traversing packet capture file of the app run using socket parameters. Then, we sum packet sizes to find the data transfer size. Since the established network connections need to have a unique set of connection parameters at a given time, we ensure that stack traces of two different sockets with the same connection endpoint are counted separately. Finally, we associate the transfer size with *origin-libraries* based on the stack trace information we collected from the respective socket of the connection (§ III-C). We should note that the ratio of UDP traffic (excluding LIBSPECTOR's UDP packets) is 0.52% of the total traffic present in the dataset, the majority (97%) of which consists of DNS requests. Therefore, we chose to omit UDP traffic from our analysis.

## F. Determining DNS Domain Categories

As part of our analysis, we analyze which domains were part of the DNS resolution requests at the time of our experiments. To this purpose, we collected domain categories provided by VirusTotal [44] using their public API. For every domain, VirusTotal returns a list of domain categories aggregated from five different cybersecurity companies. As there are no universal baselines for domain category naming, it is possible to see multiple different classifications for the same domain. Hence, similar to the methodology of AVClass [36], we chose to simplify and tokenize various domain categories into 17 generic-categories. For every domain category that VirusTotal provides, we search for a list of hand-curated words (with regular expression rules) and classify it under a generic category. Table I shows the generic categories, number of domains that fall under each generic-category and the regular expression patterns used for the tokenization of categories.

To find the category of a domain, we first tokenize all the categories that VirusTotal returns. Then, we apply majority voting among the list of generic-categories for each domain and select the most occurring generic-category.

## IV. RESULTS AND ANALYSIS

In this section, we present the analysis of the extracted data and answer the following research questions:

TABLE I: Tokenization of Domain Categories

| Generic Category | Count | Regular Expression Pattern(s) |
|---|---|---|
| adult | 206 | adult,sex,obscene,personals, dating,porn,violence,lingerie, marijuana,alcohol,gambling |
| advertisements | 1,336 | ads,advert,marketing,exposure |
| analytics | 419 | analytics |
| business_and_finance | 3,394 | busines,financ,shop,bank, trading,estate,auctions, professional |
| cdn | 77 | proxy,dns,content,delivery |
| communication | 472 | im,chat,mail,text,radio,tv, forum,telephony,portal,file |
| education | 413 | education,reference |
| entertainment | 481 | entertainment,sport,videos, streaming,pay-to-surf |
| games | 288 | game |
| health | 40 | health,medication,nutrition |
| info_tech | 1,525 | information,technology, computersandsoftware, dynamic content |
| internet_services | 374 | hosting, url-shortening, search, download,collaboration, parked, online, infrastructure, storage,security, surveillance, government |
| lifestyle | 558 | blog,hobbies,lifestyle,travel, cultur,religi,politic, restaurant,vehicles, philanthropic,event,advice |
| malicious | 23 | malicious,infected,bot not recommended,illegal, hack,compromised, suspicious content |
| news | 415 | news,tabloids,journals |
| social_networks | 55 | social |
| unknown | 4064 | (all remaining) |
| Total | 14,140 | |

- RQ1 What are the properties of data transfer and flow ratios in terms of total and average transfer for different categories of apps, libraries, and domains?
- RQ2 Is it necessary to track data flows based on *origin-libraries* instead of using network analysis only?
- RQ3 How comprehensive is the empirical analysis in terms of Java method coverage?
- RQ4 What is the monetary and energy cost of third-party libraries to an average user?

First, we present the aggregated data transfer sizes, and investigate the data flow for different categories of apps, libraries and DNS domains. Secondly, we analyze average data transfer sizes per aforementioned categories, extract mean values, calculate the ratio of data transfer flows and present the prevalence of advertisement and tracker library traffic. Then, we investigate the Java method coverage of our experiments. Finally, we estimate the monetary and energy consumption cost of advertisement libraries based on our empirical results and previous studies.

## A. Data Transfer Across Categories

Our apps generated a total of 30.75 GB of data from monitored sockets, where 29.13 GB was received and 1.62 GB was sent. The total number of flows (i.e., number of distinct sockets) was 617,400, sending data originating from 8,652 *origin-libraries* across 13 categories to 14,140 different DNS domains with 17 generic (i.e., tokenized) categories. Figure 2 shows aggregate data transfer size of the *origin-libraries*' categories per app category, as well as the ratio of data transfer per *origin-libraries*.

As for the *origin-libraries*, the most data transferring category was "Advertisement", which initiated 28.28% of the total traffic, effectively amounting for more than a quarter of the total data sent by any *origin-libraries* with 8.69 GB of total data transfer. Surprisingly, we observe that the highest activity by "Advertisement" *origin-libraries* in gaming apps, even more dominant than "Game Engine" libraries, which mostly manifested themselves in simulation and action games. The second most data transferring library category is "Development Aid", which accounted for 26.34% of the data transfer with 8.1 GB. The libraries classified under "Development Aid" often include third-party development libraries such as `okhttp3` or companies' development infrastructure/API-related libraries such as `com.amazon.whispersync` (for Kindle). Finally, we see libraries with Unknown categories initiating connections that cause 25.3% of the total data transfer (7.75 GB), which includes app-specific, first-party developer code as well as the library traffic that could not otherwise be attributed.

Figure 3 demonstrates the top data transferring *origin-libraries*. Here, we see that `com.unity3d.player` is the top data transferring *origin-library* with 1.59 GB, which is classified as a Game Engine library. Based on our methodology of library name extraction (§ III-C), it is possible to see the same prefixes across different *origin-libraries*. We therefore also classify *origin-libraries* into more generic library names, and use the *2-level libraries*. Among the *origin-libraries*, Google's internal libraries (`com.google` and `com.android`) transferred 2.84 GB and 452 MB data, respectively, followed by the Advertisement/Game Engine[3] libraries `com.unity3d` and `com.gameloft` with 2.82 GB. We finally observe that *2-level libraries* transmitted 4.96 MB data on average, where the top 25 of the 4,793 *2-level libraries* accounted for 72.5% of the total data transmitted.

Figure 4 demonstrates the Cumulative Distribution Function of sent and received network data amount for apps, *origin-libraries* and DNS domains. We observe that all apps, *origin-libraries* and DNS domains always received more data than they sent, and the data transfer flow size is between 400B and 1GB. We then examine the *ratio* of data transfer flows. Figure 5 shows the ratio of sent data over received data per apps, *origin-libraries* libraries and DNS domains. We observe that on average, apps and *origin-libraries* receive 81 and 87 times more data than sent, while servers of domains send 104 times more data than received. The similar average ratios between apps and libs indicate a uniform distribution on *origin-libraries* across the apps we tested. The discrepancy between app and DNS transfer flow ratios is due to 25,000 apps sending data to 14,100 domain names only. In terms of the distribution, top 5,057 (out of 25,000) apps, 2,299

[3]Although primarily a Game Engine library, unity3d also includes advertisement classes, which manifested themselves during our experiments

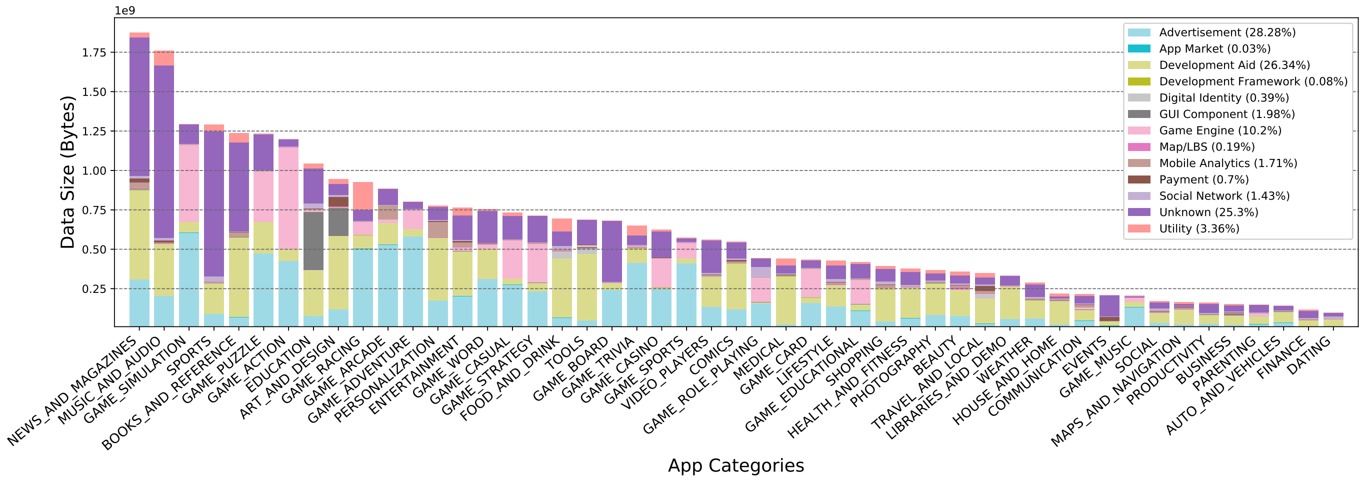
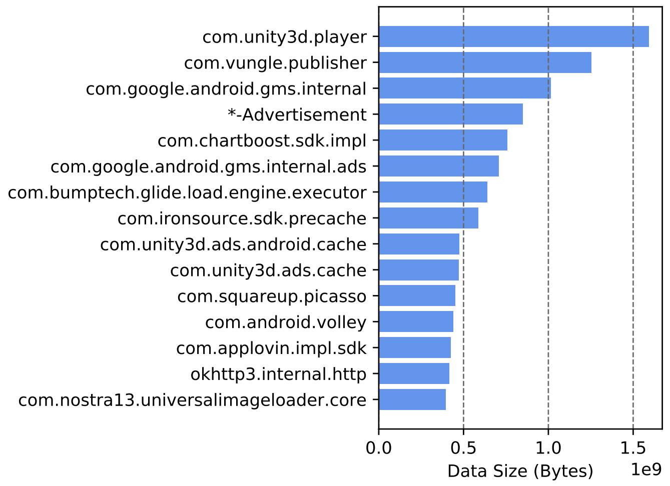


Fig. 2: Data transfer size of *origin-libraries*' categories per app category. Ratio of total data transfer per *origin-libraries* categories are presented in the legend.



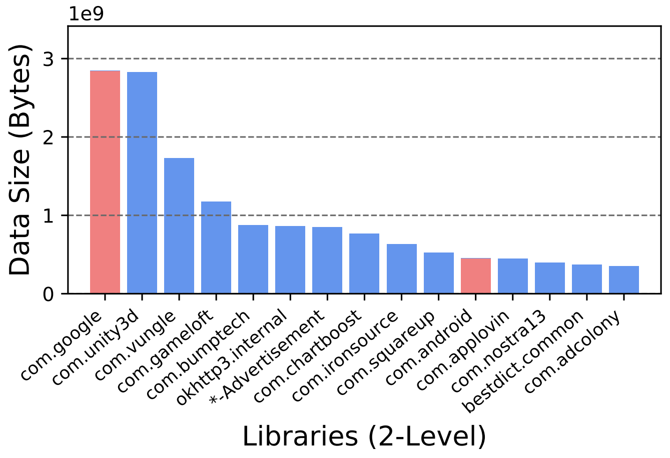


Fig. 3: (Top) Top data transferring *origin-libraries*. `*-Advertisement` represents total data transfer through sockets created by Android's built-in libraries which sends data to DNS domains categorized under Advertisement. (Bottom) Top data transferring *2-level libraries*. Traffic from Android built-in apps is shown red.

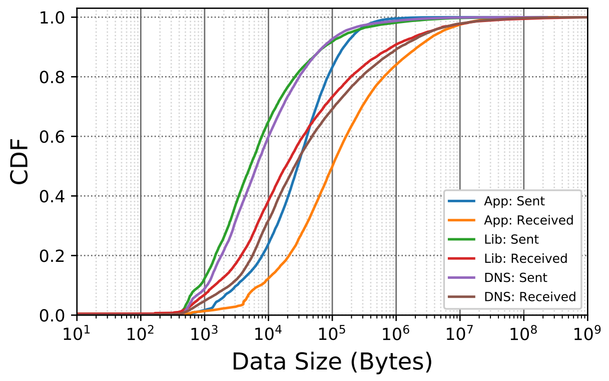


Fig. 4: CDF of data transfer flow sizes across apps, *origin-libraries*, and DNS domains

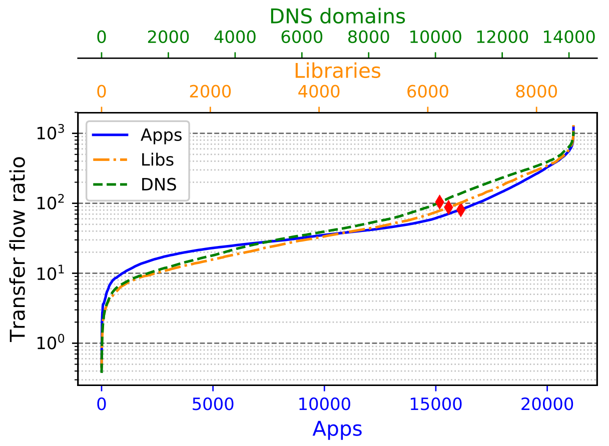


Fig. 5: Data transfer flow ratios across apps, *origin-libraries*, and DNS domains. Red diamonds indicate the average flow ratios for each X-axis.

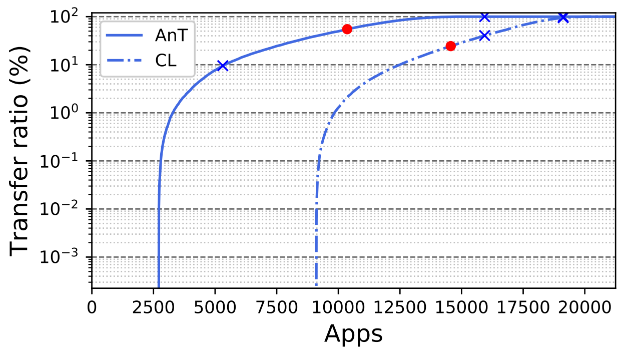


Fig. 6: Data transfer ratio of Advertisement and Tracker (AnT) libraries and Common Libraries. Means of transfer ratios are marked with red dots.

*origin-libraries* (out of 8,746) and 4,010 (out of 14,140) DNS domains are associated with half of the total data transfer, which suggests that a minority of libraries and domains cause the majority of network traffic in apps. We also see that the top 10% of *origin-libraries* received over 260 times data than they sent, which shows that the libraries which receive the most traffic can be more than 3 times as aggressive (i.e., higher flow ratio and more data consuming) as the average.

As a next step, we also investigate the prevalence of Advertisement and Tracker (AnT) libraries and the most common libraries (CL) that previous studies observed in their dataset [23] More specifically, we study if AnT and CL libraries appeared in the network stack *and* initiate connections to remote servers. Figure 6 shows the ratio of data which originated from AnT libraries and common libraries over total data transfer size per apps based on the library lists provided by Li et al. For AnT libraries, over ~2,500 apps do not send *any* data due to such libraries, whereas the network traffic of ~8,750 apps *entirely* consisted of AnT-related *origin-libraries*. Similarly, ~13,500 apps manifested network activity due to common libraries. On average, AnT libraries received 54.8 times more data than sent, which is more than twice the average of common libraries with a ratio of 24.4.

We finally investigate the average data transfer sizes for apps, *origin-libraries*, and DNS domains as neither are uniformly distributed among their respective categories. Figure 8 shows the average data transfer per app category. We observe that, the ”Music and Audio” and ”News and Magazines” categories transmit the most data on average, which indicates that the aggregate data transfer size from these categories is not only due to the higher number of apps in our dataset but because of their network-dependent functionalities. Figure 7 shows average data transfer per *origin-libraries* (left) and DNS domain categories (right). We see that Mobile Analytics, Game Engines and Advertisements are the top 3 data transmitting library categories, with averages of 35.6MB, 27.91MB and 12.66MB per library. On the other hand, the DNS domain categories where apps send the data portrays a profile much different than a 1-to-1 correlation between similar categories. That is, CDN domains receive an average 46.27MB per domain, which is almost 11 times more data than advertisements (4.32MB per domain). While previous approaches classified advertisement library traffic by name-based indicators of advertisement domains, CDN-bound traffic would cause inaccuracies during network traffic attribution. In comparison to other domain categories, there are very few social-network-related domains that apps had interaction with. That is, with 3.42MB, social network related domains are third highest data transferring libraries in average data transfer per domain rankings.

**In summary** we observed that: 1) Over a quarter of the mobile network traffic originates from advertisement libraries, 2) Google's internal libraries cause the most traffic, 3) apps receive more data than send in general, 4) AnT libraries receive twice as more data as common libraries, 5) 35% of app traffic was caused by AnT only, whereas 10% of apps were free of any AnT traffic, and 6) on average, CDN domains receive the most traffic, almost 11 times more than advertisement or gaming domains.

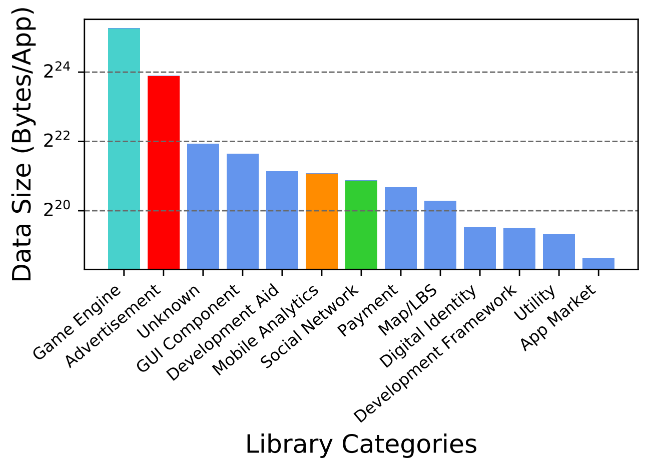
## B. Library vs DNS Domain Categories

Previous studies ( [28], [42], [43], [46], [47]) make use of the User-Agent field, domain names, hostnames, and URL parameters to identify and categorize the network traffic of apps. However, we see that the network traffic does not always originate and end up in similar categories of *origin-libraries* and domains (i.e., advertisement, mobile analytics, social media and games). For instance, advertisement libraries send ~29% of their traffic to CDN servers. Therefore, a system which simply examines the network traffic without the contextual information from *within* the app can misclassify the nature of a network connection. To confirm this intuition, we present the correlation of *origin-libraries* and DNS domains in the from of aggregate data transfer size from the former to the latter in Figure 9. It can be clearly seen from the heatmap that advertisement-related domains not only receive traffic from advertisement libraries, but also from development aid and mobile analytics libraries. Conversely, the traffic originating from advertisement libraries also ends up in CDN and business/finance domains. Similarly, the traffic from mobile analytics libraries often end up in business and finance related domains, instead of commonly known analytics-related domains. Consequently, the answer to RQ.2 is that when classifying a network traffic flow, it is necessary to analyze the *origin-libraries* in conjunction with the DNS domains.

## C. Method Coverage

One of the challenging aspects of dynamic analysis is to achieve complete method coverage. Since we analyze apps at a large scale, we are bound by the effectiveness of automated user input generators for our experiments.

As data measurements with dynamic analysis are closely coupled with the amount of executed code, we modified the method tracing of the Android framework to provide information on which methods execute during experiments. To this end, we first extract and compile all available method



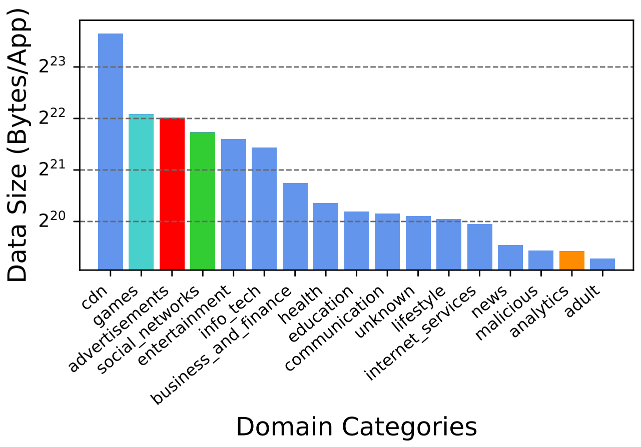


Fig. 7: Average data transfer per *origin-libraries* (left) and DNS domain (right) categories.

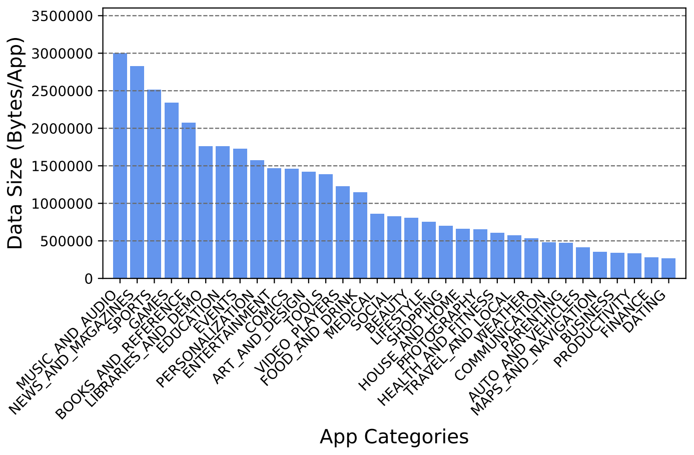


Fig. 8: Average data transfer per app category.

signatures from `dex` files that an `apk` includes. We then obtain a method trace file from Android Profiler at the end of app's evaluation, which lists all the methods that the app has called *including* native API calls. To distinguish overloaded variants of methods which share the same name within a class, we use the method type signatures. Finally, we compute the method coverage as the ratio of method signatures which are listed in the method trace file *and* available in the app's respective `dex` file divided by the total number of methods in the `dex` file.

Before our large-scale experiments, we conducted a study where we ran a subset of 100 random apps from our dataset with 10, 100, 500, 1,000, 5,000, 10,000 UI input events. Our empirical analysis showed that exercising an app beyond 1,000 UI input events did not provide any significant benefits over the number of methods called, as the startup activities often include AnT library loading activity that uses network as well.

Figure 10 shows the method coverage percentages per app. On average, `apks` in our dataset contain 49,138 methods, with 27.3% of apps including more than the average. The average method coverage of our experiment is 9.5%, where 40.5% of the exercised apps had above-average coverage. This result is consistent with other studies such as Zheng et al.'s [48], where the authors observed 10.3% coverage after using `monkey` for 18 hours. We should note that, the network traffic we obtained during our analysis constitutes a lower bound for the app network activities with an coverage method coverage of 9.5%, and consists of app activities before login screens.

## D. Estimating the Cost to Users

Two of the major impacts of apps' network traffic on users are: 1) the cost of data transmission over mobile plans and 2) energy consumption. To study the impact to users, we rely on current prices of mobile data plans and energy consumption measurements of previous studies.

Based on our results, average network traffic due to Advertisements and Mobile Analytics *origin-libraries* account for 15.58 and 2.2 MB of data transfer over 8 minutes of app runtime in our setup. As of 2019, Google Fi's mobile data plan charges \$10 per GB [17]. Therefore, the network traffic volumes we observed translate to an average of \$1.17 and \$0.17 worth of data usage every hour *only* due to non-app related Advertisements and Mobile Analytics traffic, respectively. Social Network and Digital Identity *origin-libraries* accounted for 1.92 MB of data transfer on average, costing users \$0.14 per hour[4]. Another costly *origin-libraries* category is Game Engines, with an average cost of \$3.02 per hour. As gaming apps have large initial file downloads, the total data transmission ratio of apps with `GAME_*` categories is higher than all the other categories combined.

To estimate the energy consumption of mobile advertising libraries, we rely on a study by Vallina et al. [43]. Although authors do not provide the energy cost of mobile advertisements per byte, the average advertisement content is presented as 31kB/day and the average current drain for four major ad libraries is 229mA with 20s refresh rate while idle current drain is reported as 144.6mA. Using the time series throughput

[4]This cost can be subsidized when there are zero-cost traffic agreements between the Internet Service Providers and social network websites.

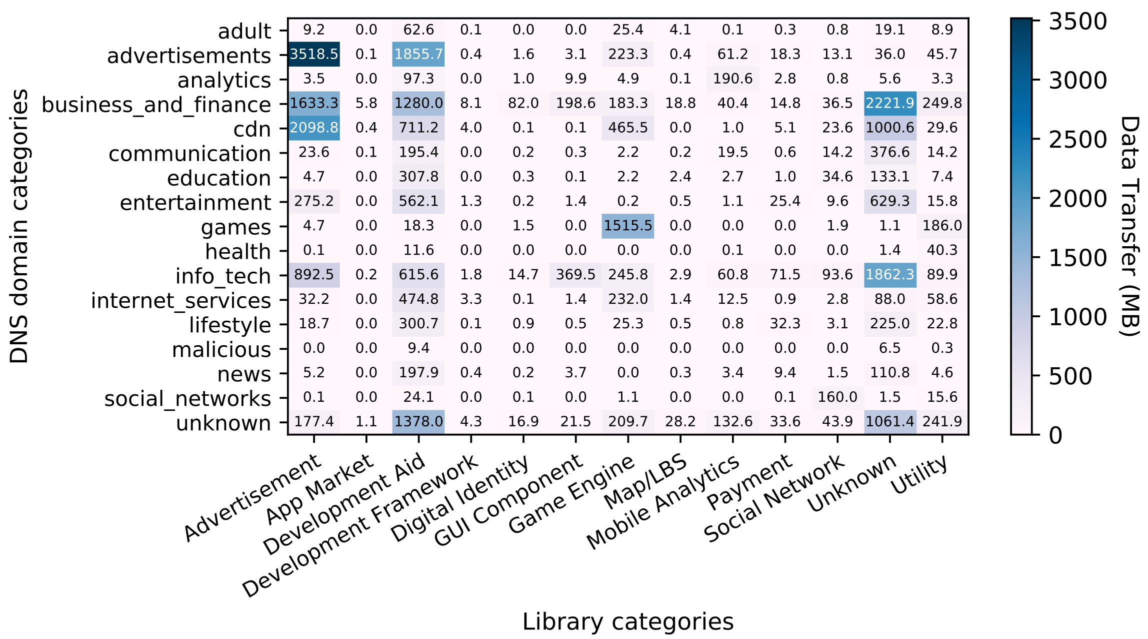


Fig. 9: Correlation of library categories with DNS categories

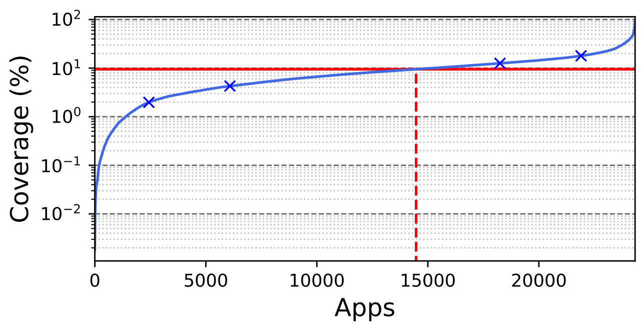


Fig. 10: Method coverage percentages per app

data by Vallina et al., we estimate active download time for advertisement libraries to be 9.3 seconds per minute. Rosen et al.'s study [34] indicates that apps continue to transmit data even when they are sent to background and 80% of the background traffic is sent within the first 60 seconds, conforming with the Pareto Principle. Therefore, we assume 5 minutes of runtime will be a good approximation of the app's overall power consumption resulting from advertisement libraries[5]. Hence, based on a typical smartphone battery ($11.55Wh/3000mAh = 3.85V$), we calculate the power consumption of advertisement libraries while they are active ($(229mA - 144.6mA) \times 3.85V = 0.325W$), data transmission rate ($(31kB \times 0.95)/(5min \times 9.3sec/min) = 635B/sec$) and the energy consumption per byte of transmitted data ($0.325W/635B/sec = 5 \times 10^{-3} J/B$). We have observed that advertisement libraries send 15.6 MB data on average, which costs 7794 Joules of energy, or 2.16Wh. For a typical 11.55Wh battery, that is 18.7% more energy consumption *only* because of advertising-related activities. The impact of the remaining categories of libraries can also be calculated with isolated energy consumption measurements of such libraries, which is out of scope of this study. However, prior empirical analysis of Li et al., [22] suggests that network components consume the most energy with over 40% of non-idle energy consumption.

[5]Pareto lower cumulative distribution (P) with scale and shape parameters $x_{\text{min}}$=1, $x_{\text{m}}$=1 and $\alpha$=1 yields P=0.8 for x=5 and P=0.95 for x=21. We consider x∈[1,5] as one minute of execution.

### *E. Applications of* LIBSPECTOR

**Security:** LIBSPECTOR associates network data flows with *origin-libraries* and predicts library categories to reveal network activities of third-party libraries. Policy control systems such as BorderPatrol [50] implement policy actions including blacklisting, where the level of enforcement can be selected as a library. In such cases, a-priori knowledge of the to-be-blacklisted library is required to determine policies. The information obtained from LIBSPECTOR can provide insights on which library to blacklist, and hence augment the capabilities of such policy enforcement systems.

**Measurement:** LIBSPECTOR can evaluate the connections at a finer granularity than conventional DNS based systems and categorize libraries with better accuracy. From Figure 9, we determine that a purely DNS based approach would misclassify all CDN-bound traffic from known origin-libraries (19.3% of the total traffic) and not all traffic has a 1-on-1 correlation between similar categories of DNS and *origin-libraries*. Hence, it is essential to use the app context information in a network traffic analysis, which LIBSPECTOR provides.

## V. RELATED WORK

Earlier works focused on distinguishing apps [1] and user activities [9] using mobile app traffic. Chung et al. [8] studied mobile network traffic volume and flow characteristics while attributing traffic to different types of origin devices. They performed an app-level traffic classification where network traffic volumes are divided into business-related categories of apps. They also characterized devices with commonly used OS fingerprinting techniques. ProfileDroid [46] is an app monitoring and profiling architecture. Authors here have performed a static analysis on app's `apk` and a dynamic analysis on input events, intent usage, system calls using network layer information. However, the authors have attributed the origin of network activities to individual apps, measured traffic flow ratios and determined third-party traffic based on network connection endpoints. Fukuda et al. [11] investigated network traffic volumes for WiFi and 3G networks on user devices and characterized usage patters for different times of the day. Further studies of Xu et al [47] and Maier et al [28] attributed traffic to ad-libraries using HTTP header and host information, while Tongaonkar et al. [42] used hostnames. Finally, authors of SmartGen [51] conducted a large-scale study to identify URL's that mobile apps connect to with symbolic execution and identified malicious links, however did not associate potential connections to libraries.

There are also studies on app markets and app characteristics. Petsas et al. [31] collected data from various app stores and studied app popularity, number of updates, comments, app categories, download counts and app pricing. Wang et al. [45] studied Chinese app markets for app categories, download counts, API levels, as well as most popular third-party libraries across different markets. However, these works did not focus on the network traffic analysis at large scale.

Some studies linked the network traffic to mobile energy consumption. Hao et al., [19] estimated mobile app energy consumption at code level with various granularity. However, they had to rely on manual analysis for estimating the implemetation-dependent network-related method invocations. Kundu et al. [21] analyzed malicious energy drainage on mobile systems, while Gao et al., [12] studied methods and attacks that can mislead energy consumption models, and built E-Android [13] as a defensive profiler. Falaki et al. [10] characterized smartphone traffic flow and investigated traffic size, network latencies and power consumption of wireless components of smartphones. Similar to other related works, they attributed traffic generation to different categories of applications as well. Rosen et al. [34] conducted a two-year user study to identify network energy efficiency of mobile apps. The authors studied network energy consumption per app, quantified the impact of background data transmission and provided case studies across different categories of apps with data flow analysis. Vallina et al. [43] studied data transfer flows and energy consumption of advertisement networks that are used in most popular apps on a purpose-build app. Their study identified advertisement traffic based on DNS lookups and HTTP traffic and studied data connection refresh intervals as well as presenting current drain statistics for cached/uncached data across different types of ads. Unfortunately, these works could not attribute network packets to individual libraries.

As for the ad-library focused studies, AdDroid [30] proposed a new API for privilege separation, AdDetect [29] used ML-based approach to identify advertisement libraries, AdSplit [38] and Aframe [49] identified and isolated ad processes and iframe displays, respectively. A similar cost analysis is performed in Adrob [14], where authors captured network data and studied the revenue loss and related impacts of app cloning due to advertisement libraries. These works primarily focused on isolating third-party processes and did not perform a large-scale analysis.

In the literature, there are other large-scale app evaluation frameworks such as Bierma et al.'s Andlantis [7]. Similar works such as Andrubis [25] focuses on malware detection with both static and dynamic analysis, and includes similar features with our framework such as method tracing. However, these works did not analyze the contextual information with respect to network activities, but rather focused on malware detection. Additionally, authors of CHIMP [3] developed a crowd sourced UI exercising framework, whereas authors of PUMA [20] and Dynodroid [27] used app and framework instrumentation respectively to increase `monkey` coverage.

Finally, previous efforts of LibScout [6], LibD [24], LibRadar [26] and Li et al.'s work [23] on detection of third-party libraries provided us with the insights to attribute traffic to libraries. However, the hierarchical organization of Java libraries and inclusion of developer-authored (i.e., app-specific and first-party) code in apps created a lack of comprehensiveness for classification. We tackled this problem by a multitude of heuristics combined with the fine-grained information we extracted from the apps during execution-time, and demonstrated the correlation of such categories with classes of domains.

## VI. Conclusions

In this paper, we designed and built a data collection framework which extracts fine-grained network-context information from app activities while recording method coverage. Using our framework, we analyzed 25,000 Android apps from Google PlayStore in a parallelized execution environment. We then presented our methodology and heuristics on how the data is categorized and important features are selected. Finally we demonstrated the trends on data transfer sizes, flows and usage statistics of Android libraries, DNS domains and app behavior by categories. We find that (i) 35% of apps *only* had advertisement and trackers (AnT) traffic, whereas 89% of the apps had *some* AnT traffic, (ii) AnT libraries are twice as aggressive as the common libraries in terms of data flow ratios, (iii) advertising libraries constitute 28.3% of the overall data traffic, costing an estimated $1.17 per hour to the user and causing 18.7% more energy consumption, and (iv) there is not always a strict 1-to-1 correlation between libraries and domains of the same category, and traffic often ends up on CDN domains, requiring a contextual analysis of the traffic.

## Acknowledgments

This work was partially funded by ONR under grants N00014-19-1-2364 and N00014-17-1-2541. We would like to thank the reviewers and our shepherd Timothy Wood for their insightful feedback in improving the final version of our paper.

## References

[1] H. F. Alan and J. Kaur. Can Android applications be identified using only TCP/IP headers of their launch time traffic? In *Proceedings of the 9th ACM conference on security & privacy in wireless and mobile networks*, pages 61–66, 2016.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of Android apps for the research community. In *Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE/ACM, 2016.

[3] M. Almeida, M. Bilal, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Varvello, and J. Blackburn. Chimp: Crowdsourcing human inputs for mobile phones. In *Proceedings of the World Wide Web Conference on World Wide Web*, pages 45–54, 2018.

[4] Android. ADB Monkey UI Exerciser. https://developer.android.com/studio/test/monkey.html, 2019.

[5] AndroidRank. Most downloaded free Android applications. https://www.androidrank.org/app/ranking/all?sort=4&price=free, 2019.

[6] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in Android and its security applications. In *Proceedings of the Conference on Computer and Communications Security*, pages 356–367. ACM SIGSAC, 2016.

[7] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. R. Choe. Andlantis: Large-scale Android dynamic analysis. *arXiv preprint arXiv:1410.7751*, 2014.

[8] J. Y. Chung, Y. Choi, B. Park, and J. W.-K. Hong. Measurement analysis of mobile traffic in enterprise networks. In *Asia-Pacific Network Operations and Management Symposium*, pages 1–4. IEEE, 2011.

[9] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde. Analyzing Android encrypted network traffic to identify user actions. *IEEE Transactions on Information Forensics and Security*, 11(1):114–125, 2015.

[10] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proceedings of the Internet Measurement Conference*, pages 281–287. ACM, 2010.

[11] K. Fukuda and K. Nagami. A measurement of mobile traffic offloading. In *International Conference on Passive and Active Network Measurement*, pages 73–82. Springer, 2013.

[12] X. Gao, D. Liu, D. Liu, and H. Wang. On energy security of smartphones. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, pages 148–150, 2016.

[13] X. Gao, D. Liu, D. Liu, H. Wang, and A. Stavrou. E-android: A new energy profiling tool for smartphones. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 492–502. IEEE, 2017.

[14] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of Android application plagiarism. In *Proceedings of the annual international conference on Mobile systems, applications, and services*, pages 431–444. ACM, 2013.

[15] Google. Android API 25 Class Index. https://developer.android.com/reference/classes.

[16] Google. Google Play Store. https://play.google.com/store.

[17] Google. Fi data plan. https://fi.google.com/about/plan/, 2019.

[18] B. Gruver. dexlib2 library. https://github.com/JesusFreke/smali/tree/master/dexlib2, 2017.

[19] S. Hao, D. Li, W. G. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *IEEE Proceedings of the International Conference on Software Engineering*, pages 92–101, 2013.

[20] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the annual international conference on Mobile systems, applications, and services*, pages 204–217. ACM, 2014.

[21] A. Kundu, Z. Lin, and J. Hammor. Energy attacks on mobile devices. https://arxiv.org/pdf/1704.04464.pdf, 2017.

[22] D. Li, S. Hao, J. Gui, and W. G. Halfond. An empirical study of the energy consumption of Android applications. In *IEEE International Conference on Software Maintenance and Evolution*, pages 121–130, 2014.

[23] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon. An investigation into the use of common libraries in Android apps. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 403–414. IEEE, 2016.

[24] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. Libd: scalable and precise third-party library detection in Android markets. In *International Conference on Software Engineering (ICSE)*, pages 335–346. IEEE/ACM, 2017.

[25] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17. IEEE, 2014.

[26] Z. Ma, H. Wang, Y. Guo, and X. Chen. Libradar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the international conference on software engineering companion*, pages 653–656. ACM, 2016.

[27] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[28] G. Maier, F. Schneider, and A. Feldmann. A first look at mobile handheld device traffic. In *International Conference on Passive and Active Network Measurement*, pages 161–170. Springer, 2010.

[29] A. Narayanan, L. Chen, and C. K. Chan. Addetect: Automated detection of Android ad libraries using semantic analysis. In *International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6. IEEE, 2014.

[30] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in Android. In *Proceedings of the Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012.

[31] T. Petsas, A. Papadogiannakis, M. Polychronakis, E. P. Markatos, and T. Karagiannis. Rise of the planet of the apps: A systematic study of the mobile app ecosystem. In *Proceedings of the Internet Measurement Conference*, pages 277–290. ACM, 2013.

[32] M. Plümicke. Java type unification with wildcards. In *Applications of Declarative Programming and Knowledge Management*, pages 223–240. Springer, 2007.

[33] I. Reyes, P. Wijesekera, J. Reardon, A. E. B. On, A. Razaghpanah, N. Vallina-Rodriguez, and S. Egelman. ”won't somebody think of the children?” examining COPPA compliance at scale. *Proceedings on Privacy Enhancing Technologies*, (3):63–83, 2018.

[34] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen. Revisiting network energy efficiency of mobile apps: Performance in the wild. In *Proceedings of the Internet Measurement Conference*, pages 339–345. ACM, 2015.

[35] rovo89. Xposed Framework API. http://api.xposed.info/reference/packages.html, 2019.

[36] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[37] SensoTower. Global app revenue for q1-2019. https://sensortower.com/blog/app-revenue-and-downloads-q1-2019, 2019.

[38] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *{USENIX} Security Symposium ({USENIX} Security)*, pages 553–567, 2012.

[39] Statista. Combined global app downloads. https://www.statista.com/statistics/604343/number-of-apple-app-store-and-google-play-app-downloads-worldwide/, 2019.

[40] Statista. Number of available apps in apple app store. https://www.statista.com/statistics/779768/number-of-available-apps-in-the-apple-app-store-quarter/, 2019.

[41] Statista. Number of available apps in google-play store. https://www.statista.com/statistics/289418/number-of-available-apps-in-the-google-play-store-quarter/, 2019.

[42] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding mobile app usage patterns using in-app advertisements. In *International Conference on Passive and Active Network Measurement*, pages 63–72. Springer, 2013.

[43] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft. Breaking for commercials: characterizing mobile advertising. In *Proceedings of the Internet Measurement Conference*, pages 343–356. ACM, 2012.

[44] VirusTotal. Malware Scanner. https://www.virustotal.com, 2019.

[45] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu. Beyond Google play: A large-scale

comparative study of Chinese Android app markets. In *Proceedings of the Internet Measurement Conference*, pages 293–307. ACM, 2018.

[46] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of Android applications. In *Proceedings of the annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[47] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the conference on Internet measurement conference*, pages 329–344. ACM, 2011.

[48] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie. Automated test input generation for Android: Are we really there yet in an industrial case? In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 987–992. ACM, 2016.

[49] X. Zhang, A. Ahlawat, and W. Du. Aframe: Isolating advertisements from mobile applications in Android. In *Proceedings of the Annual Computer Security Applications Conference*, pages 9–18. ACM, 2013.

[50] O. Zungur, G. Suarez-Tangil, G. Stringhini, and M. Egele. BorderPatrol: Securing BYOD using fine-grained contextual information. In *International Conference on Dependable Systems and Networks (DSN)*, pages 460–472. IEEE/IFIP, 2019.

[51] C. Zuo and Z. Lin. Smartgen: Exposing server urls of mobile apps with selective symbolic execution. In *Proceedings of the International Conference on World Wide Web*, WWW '17, pages 867–876. International World Wide Web Conferences Steering Committee, 2017.