NetworkProfiler: Towards Automatic Fingerprinting of Android Apps

Shuaifu Dai*, Alok Tongaonkar†, Xiaoyin Wang*, Antonio Nucci†, and Dawn Song*
*University of California, Berkeley, USA
†Narus Inc, Sunnyvale, USA

***Abstract*—Network operators need to have a clear visibility into the applications running in their network. This is critical for both security and network management. Recent years have seen an exponential growth in the number of smart phone apps which has complicated this task. Traditional methods of traffic classification are no longer sufficient as the majority of this smart phone app traffic is carried over HTTP/HTTPS. Keeping up with the new applications that come up everyday is very challenging and time-consuming. We present a novel technique for automatically generating network profiles for identifying Android apps in the HTTP traffic. A network profile consists of fingerprints, i.e., unique characteristics of network behavior, that can be used to identify an app. To profile an Android app, we run the app automatically in an emulator and collect the network traces. We have developed a novel UI fuzzing technique for running the app such that different execution paths are exercised, which is necessary to build a comprehensive network profile. We have also developed a light-weight technique, for extracting fingerprints, that is based on identifying invariants in the generated traces. We used our technique to generate network profiles for thousands of apps. Using our network profiles we were able to detect the presence of these apps in real-world network traffic logs from a cellular provider.**

I. Introduction

A critical aspect of network management from an operator's perspective is the ability to understand all traffic that traverses the network. This ability is important for traffic engineering and billing, network planning and provisioning as well as network security. Rather than basic information about the ongoing sessions, all of the aforementioned functionalities require accurate knowledge of what is traversing the network in order to be effective [6].

In recent years, there have been dramatic changes to the way users behave, interact and utilize the network. More and more users are accessing the internet via smart devices like smart phones and tablets. According to recent statistics by Canalys [3], 488 million smartphones have been sold in the year 2011, compared to 415 million personal computers. Users of these devices typically download applications (commonly called as smartphone/mobile apps) that provide specific functionality. A majority of these apps access the internet. For example, 87% of the 90K Android apps in the Android Market [2] that we randomly sampled required permission for Internet access. The number of such devices being used is increasing rapidly in enterprise networks. An interesting trend to note is that the proportion of personal devices being used in the enterprise networks is growing rapidly. Hence, it is crucial for network operators to have clear visibility into the mobile apps that are running in their network.

Network traffic classification techniques that perform protocol identification are not sufficient for obtaining this visibility into smartphone traffic for the following reason. A majority of smartphone apps run over HTTP/HTTPS protocols. In our study of around 90K Android apps for which we could identify network calls statically, we found that more than 70K apps used HTTP/HTTPS. Operators need to be able to identify not just HTTP traffic but applications like Youtube, Pandora, and Facebook running over HTTP. This means that machine-learning techniques [14] that extract features from the network traces, and then use these features for network traffic classification are not useful as they operate at the granularity of protocol identification. On the other hand deep packet inspection (DPI) based techniques [11] will be able to identify these apps if there are signatures for the apps. Traditionally, such signatures are developed from protocol specifications or by reverse engineering the protocol from network traces [7] or application binaries [5]. However, smartphone apps generally do not have any standard specification documents and manually reverse engineering these apps does not scale due to the large number of smartphone apps available. Moreover, the number of applications is increasing rapidly due to the ease of development of smartphone apps. For example, the number of apps in the Android Market has risen from 5K in 2009 to 400K at the end of 2011.

Recent years have seen an increasing number of research works that analyze network traffic to understand usage behaviors of smartphone apps ([18], [9]). However, these papers rely on techniques for app identification which are not applicable for Android apps or rely on having access to the Android devices and monitoring the specific devices. For example, Xu et al [18] use `User-Agent` field in the HTTP header to identify the app. Apple has a guideline for iOS which requires that this field contain app identifier. However, this guideline is not strictly enforced. For Android apps the situation is even worse since developers generally put some generic string (not unique to the app but identifying the Android version and such) in this field. On the other hand the approach taken of making some users use apps on specific devices to collect network trace and profile app usage does not give real-world data ([9], [17]). Moreover, manual execution of apps suffers from the problems of incomplete app behavior coverage and scalability. The approach of using `Host` field in the HTTP header for identifying the apps works for some apps but is not sufficient because the same host may serve multiple apps. This is typically true when the same app developer such as Zynga publishes multiple apps. The increasing popularity of smart-

phone apps has led to development of many platforms, such as Facebook, which support apps from different developers. The apps which are developed on such platforms typically use the servers from the platform provider to provide their service. Hence, it is very common to see the case where multiple apps are hosted on the same third-party hosts. For instance, m.facebook.com hosts many apps such as *Pirates Mobile*, a gaming app, and *Squats*, a personal training app. Therefore, we came up with a new way of identifying Android apps from the network traffic. Just as DNA profiles of people can be used to identify individuals based on their unqiue DNA sequences, we developed the concept of a *network profile* for an Android app that uses unique characteristics of the network behavior of the app, that we call as *network fingerprints*, to identify the app. The network fingerprint for an Android app consists of two components: hosts that the app connects to and a state machine representing the patterns over the strings that occur in the HTTP header of the requests made by the app to those servers. We focus on only the HTTP traffic of apps since according to our survey of over 90K apps, most of the Android apps with Internet access are using HTTP/HTTPS, and among these apps, more than 70% do not use HTTPS. Even for the apps such as *Facebook* that use HTTPS, they often use it only in the authentication phase of the session and use HTTP for rest of the user interaction. Similar findings from 18 free and 9 paid apps led Wei et al [17] to conclude that most mobile app network traffic is unencrypted.

We developed an automatic technique for extracting these app fingerprints from the network traces obtained by running these apps in an emulator in an automated fashion. The key to generating good network profiles (i.e., which cover most of the network behaviors), is to ensure that the app is driven along most of its execution path when the network traces are being collected. For this we developed a novel technique for exercising new app behavior that is based on UI fuzzing. Even in the case that an app requires login, we are able to generate different execution runs automatically from the traces of a single execution run of the app by any user. The UI fuzzing technique works by generating events to exercise unexplored paths of the app. As a proof of concept we analyzed 10k apps and generated network profiles for those apps. The techniques presented in this paper can be used along with an infrastructure, that monitors an app market and downloads new apps, to generate network profiles for newer apps and maintain a periodically updated database of the network profiles of all Android apps. It should be noted that even though we have developed the system for Android applications, the techniques and ideas developed in this work can be applied to other mobile platforms such as iOS and Windows Mobile.

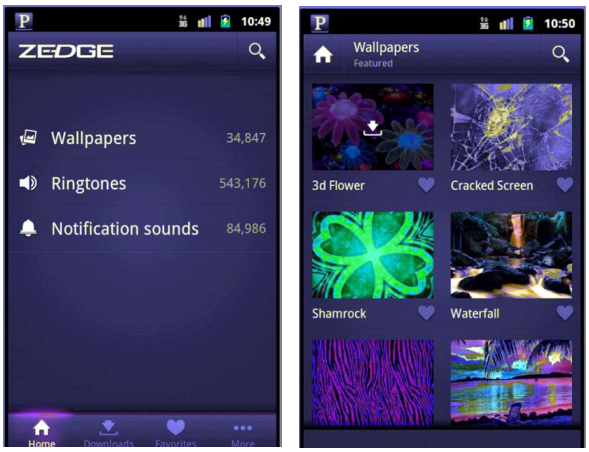
The main contributions of this work are as below.

• We built a novel system, NetworkProfiler, that is able to efficiently generate network profiles for Android apps. Compared to traditional approaches that require careful inspection and reverse-engineering of application payloads, our system requires much less time and expertise to generate the network profiles.

• Based on UI fuzzing, we developed a technique that is able to automatically identify and execute multiple paths for an Android app, which exercises different network behaviors of the app.

• We present an algorithm that automatically extracts a fingerprint for an Android app based on its network traces.

• We used NetworkProfiler to generate network profiles for Android apps, and evaluated the generated profiles on a mixture of network traffic of these apps. Evaluation results show our network profile are able to identify apps in network traffic with high precision.



(a) Main Activity (b) Wallpaper Activity

Fig. 1: The Screen Shots of *Zedge*.

The rest of this paper is organized as below. In Section II, we present the motivation for this work. In Section III, we discuss the details of the system that we have built. In Section IV, we present an evaluation of our system. We discuss related research in Section V before we conclude the paper in Section VI.

II. Motivation

Our motivation in this work is to extract *fingerprints*, i.e., patterns of string within the network traces that are unique to the app and can be used to distinguish the app from other apps. Since we consider only apps that use HTTP, the network behavior can be characterized in terms of the different HTTP requests. Note that an app can have many different network behaviors. The network behavior may be different in terms of the HTTP method, hosts contacted, URL paths or queries, and so on. To better illustrate the challenges in automatically extracting application-level fingerprints for android apps, we discuss *Zedge* [2], one of the most popular apps in the Android Market (more than 10 million downloads) as a motivating example. *Zedge* is a simple previewer and downloader for mobile phone wallpapers, ringtones, and notification sounds. The typical user interactions with the app are as follows: a user first starts the app; then she clicks on an option to choose whether she wants to download a wallpaper, a ringtone, or a notification sound (Figure 1a); after that, a table of

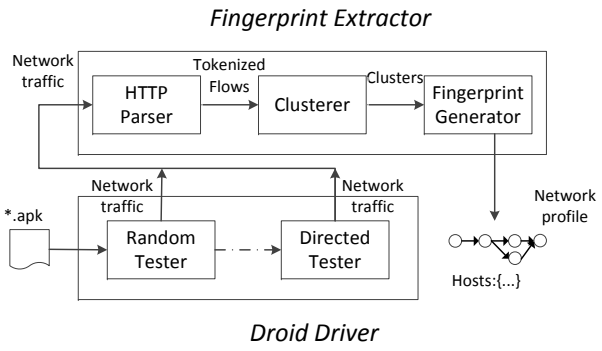


Fig. 4: NetworkProfiler overview.

googleanalytics.com. We can consider flows going to each of these servers as exhibiting different network behavior. When we collect the network traces for an app from the emulator, it contains packets going to all these servers. To generate a meaningful network profile for an app, we need to extract fingerprints for these distinct behaviors.

The straight forward approach for extracting different fingerprints is to group the flows by the `Host` field in the HTTP headers. Unfortunately, this approach results in separating out flows that exhibit similar behavior but go to different hosts. This is not good for extracting fingerprints as many apps use multiple hosts for providing the same functionality. For example, using this approach, we would separate out flows going to *fsa.zedge.net* from *fsb.zedge.net*, even though they provide the same functionality. Hence, we developed a flow grouping algorithm based on the structural similarity between the flows. Structural similarity of HTTP flows has previously been used successfully by Perdisci et al [15] to group together *malicious* flows.

We run the app being analyzed multiple times in the emulator and combine all the traces for an app. The Fingerprint Extractor first tokenizes the HTTP flows via a parser and sends the tokenized flows to the clusterer. Consider the HTTP request in Figure 5. We tokenize or breakup the request into various components. For ease of explanation we have omitted other header fields from the request. However the same techniques can be extended to cover other header fields. We can break the request into *method* (m), *page* (p), and *query* (q). Page can be further broken into a number of *page-components* (pcs) and *filename* (fn). Query can be split into *key-value* pairs (k-v). Initially we group all flows based on just the method type, i.e., all requests having the same method are grouped together. The clusterer then performs an *agglomerative clustering* of HTTP requests within each group of request by finding structural similarities between tokens. In order to capture these similarities, we define a measure of distance between two HTTP requests, say i and j, in terms of the tokens as follows -

- Distance between pages, $d_p(i,j)$: We compute the Jaccard index [4] between the $pagecomponents$ of the the pages as a measure of $similarity$. The distance is $1 - similarity$. We note that we exclude the $filenames$ in this computation.
- Distance between queries, $d_q(i,j)$: We compute the Jaccard index between the $keys$ in the queries as a measure of $similarity$. The distance is $1 - similarity$. We note that we exclude the $values$ in this computation.

Now we define the distance between two requests, i, j as $d_h(i,j) = (d_p(i,j) + d_q(i,j)/2$, If this value is above a certain threshold t_g, we group the requests into the same cluster. We start off with all requests in separate clusters of their own. We compare each request with every other request and put the requests with a distance less than the threshold, t_g, into the same cluster. We experimented with different values of threshold and found that a threshold of 0.6 gives good results in terms of the cohesiveness of the flows. Note that if a new flow is similar to flows in multiple clusters, we merge the clusters and add this flow to the new merged cluster.

Next we generate Prefix Tree Acceptor (PTA), which are trie-like state machines (i.e., no back edges), for each cluster by considering only the $method$, $page\text{-}components$ (excluding $filename$) and $query\text{-}keys$. We do not describe the construction of PTA but refer the interested readers to [13] for details. Fingerprint Extractor generates state machines on the invariant tokens for each of the clusters. The fact that we execute the app multiple times ensures that the invariant tokens do not contain terms like names of files being downloaded. We note that $query\text{-}values$ typically consist of terms that are not useful for identifying the app such as resolution of the screen or the Android version number. We discuss exceptions to this, i.e., cases when we retain the $query\text{-}value$ tokens in the state machines, in the next section. In future, we plan to run the app in emulators corresponding to different configurations with regards to Android versions, screen resolution, etc. to obtain traces where such $query\text{-}values$ are not invariant but other tokens are. Figure 6 shows the state machine for the wallpaper download behavior of *Zedge*. The states in the state machines contain implicit self-loops that allow matching of additional tokens that we have either omitted for being variable or the ones that we have not observed in the training traces. We omit these self-loops from all the figures in this paper for simplicity.

We combine fingerprints that contain the same hosts by merging the state machines. In our experiments, the cluster for downloading of wallpapers contained the hosts *fsa.zedge.net, fsb.zedge.net*, for ringtones contained *fsa.zedge.net*, and for notifications contained *fsb.zedge.net*. We combined these fingerprints as wallpapers and ringtones have *fsa.zedge.net* in common and and wallpapers and ringtones have *fsb.zedge.net* in common. The merged state machine is shown in Figure 3. Another strategy that we use is to merge fingerprints for hosts with common level two domains in the case where the state machines have a common prefix.

1) Third-party traffic: In some of the flows, the $query\text{-}values$ contain terms that have the app name embedded in them. Such tokens are very good for identifying the app. We have a simple strategy for including such tokens. Android apps are distributed in the form of `.apk` files. Each `apk` file

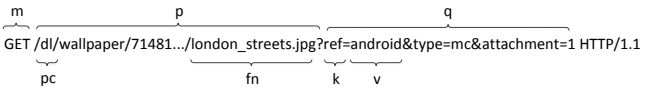


Fig. 5: HTTP request of *Zedge*.

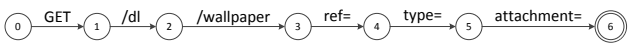


Fig. 6: *Zedge* wallpaper download state machine.

contains the app binaries as well as a *manifest* file which contains meta-data such as app name, package name, permissions required, libraries used, etc. For each app we extract a set of keywords from the manifest file that could identify the app. If any value token contains a keyword as a substring we pick up the token in the state machine. Figure 7 shows the state machine for *Youtube* that contains the term *youtube_gdata*.

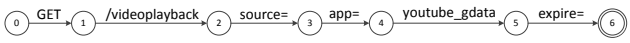


Fig. 7: State machine for *Youtube*.

The other case when $query\text{-}values$ are useful for identifying the app is in the case of third-party flows. In many cases, the third-party traffic contains certain identifiers that can be used to identify the app. For example, flows belonging to apps running on the Facebook platform contain the app identifier as the value of the *app_id* key. These identifier values are usually alphanumeric strings that are unique to the app for a specific third-party provider or the name of the app itself. Similarly, most of the free apps in Android Market contact advertisement providers to generate revenue for the developers. Many of these advertisement companies use certain identifiers in the flows to identify the apps that caused the traffic to be generated. The way that the developers use these ads is by registering with the ads provider. The ads provider generates a unique identifier for the app. The developer then makes calls to the ad libraries with this identifier. At run time, the ads library embeds this identifier when flows connect to the ad provider servers. For example, Figure 8 shows a flow to *mob.adwhirl.com*. There is a key named "appid" and the corresponding value is the app's identifier.

Third-party providers such as platform providers (e.g. Facebook) and ad providers (e.g. Admob) typically publish guidelines for developers which mention the $query\text{-}key$ that is used to identify the app. It is a one-time effort to build a mapping between hostnames and the $query\text{-}keys$, present in the flows to that hostname, that are used as unique identifiers. We can then pick the $query\text{-}values$ corresponding to these unique identifiers. For a third-party, we require that the $key\text{-}values$ picked need to satisfy *uniqueness property*, i.e., a given $key\text{-}value$ is present only in the flows belonging to an app and not in the flows belonging the other apps. This is needed since we require the fingerprint for an app to be unique, i.e., the fingerprint for an app needs to differ from the fingerprints of other apps in either the hostname or the state machine components. In such cases the fingerprint for the third-party flow can be simplified as containing the hostnames of the servers for the third-party and a simple state machine containing transitions corresponding to

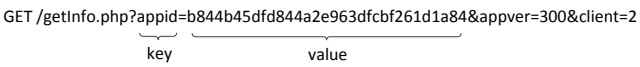


Fig. 8: HTTP request of Adwhirl in *Ringtone Maker*.

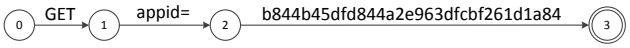


Fig. 9: State machine for *Ringtone Maker*.

just the method name, and the $key\text{-}value$ as shown in Figure 9. In case of third-party services which do not contain such unique $query\text{-}values$, different apps may have the same state machine. Hence those flows can not be used for fingerprint extraction. This is acceptable since our goal in this work is to identify different apps and not individual flow classification.

Another interesting observation is that in many cases, these unique $key\text{-}values$, satisfy another important property - *persistence*. What we mean by persistence is that in any execution of the app, there will be at least one flow that contains the given $key\text{-}value$ going to the host belonging to the third-party. In such cases, we can simplify the network profile to contain just a single fingerprint (extracted from the flows containing the persistent and unique $key\text{-}value$). This has important implications for the efficiency of Droid Driver as we will see in the next section.

B. Droid Driver

Droid Driver is the component responsible for executing the Android app and collecting the network traces. It has two main components - (i) Random Tester and (ii) Directed Tester. For any given app, the Droid Driver works in the random testing mode using the Random tester or in the directed testing mode using the Directed Tester. The need for these two distinct components becomes clear if we consider the types of traffic generated by an app. Wei et al [17] classified the traffic generated from an app into - (i) origin - traffic generated by servers belonging to the app provider, (ii) third-party - traffic going to ad providers (e.g. Admob, Google Doubleclick) and analytical services (e.g. Omniture, Google Analytics), (iii) CDN+cloud - traffic generated by Content Distribution Networks (e.g. Akamai) or cloud providers (e.g. Amazon), and (iv) rest - all other traffic. As noted earlier, for origin traffic, we can generate fingerprints consisting of just the hostnames. Similarly for third-party traffic containing unique and persistent $key\text{-}value$, we can generate fingerprints containing just hostnames and simple state machines using the $key\text{-}value$. For apps which contain the above kinds of traffic, we can run the app randomly to extract fingerprints. The Random Tester, built using Android testing tool, *monkeyrunner* [1], is responsible for running the app randomly and capturing the generated traffic for the Fingerprint Extractor. Random Testing is very efficient as the events to be sent to the app are chosen at random. In fact, in the cases where the app contacts ad providers at startup or contacts the origin server for receiving api information, just starting up an app in the emulator generates network traffic which can be used to extract the fingerprint.

TABLE II: Keys for different ads libraries.

Ads Library	Host Name	key
Admob	googleads.g.doubleclick.net	app_name
Mobclix	data.mobclix.com	a
	ads.mobclix.com	i
Adwhirl	∗.adwhirl.com	appid
Mobfox	my.mobfox.com	s
Mydas	∗.mp.mydas.mobi	apid
Adlantis	sp.ad.adlantis.jp	appIdentifier
Openx	{ox-d.ad-maker.info|u.open.net}	auid
Appsgeyser	ads.appsgeyser.com	id
Smaato	soma.smaato.{net|com}	app
Guohead	mob.guohead.com	appid
Waps	∗.waps.cn	app_id
Greystrip	∗.greystripe.com	pubappid
Adview	www.adview.cn	appid
Adsmogo	∗.adsmogo.com	appid
Admarvel	ads.admarvel.com	partner_id
I-mobile	spapi.i-mobile.co.jp	appid
Ads-svx	ads-svx.httpads.com	guid

Although, random testing is efficient, it does not suffice for all the apps. As shown in [17] many apps like *Angry Birds* and *ESPN* do not have any origin server. Also, many of the paid apps do not contain any ads. Even for apps that contain ads, the identifier may refer to the developer id and not app id, in which case it can not be used as a unique key-$value$. For such apps, we developed Directed Tester, which is responsible for driving the Android app, based on some initial paths seed, to execute those paths that generate network traffic, and collecting the network traces. We built Directed Tester system as an extension to the Android testing framework, which allows the user to communicate with a testee app via a tester app. It consists of three modules - (i) Path Recorder, (ii) Heuristic Path Generator, and (iii) Path Replayer. We describe each of these in detail in the following sections.

1) Path Recorder: Android apps consist of a number of *activities*. An activity is an application component that provides a screen with which users can interact in order to do something. The basic building block for user interface components in an activity is a *view*. A view occupies a rectangular area on the screen and is responsible for drawing and event handling. Views that are used to create interactive UI components (e.g., `Button`, `TextView`, `ImageView`) are called *widgets*. User interact with activities using events such as clicking a button, pressing a key on the keyboard, or scrolling on the screen. Path Recorder records the user events for apps running in an emulator. Path Recorder was built by modifying existing Android tools [1] - *monkeyrunner*, which provides information about the coordinates on the screen where an event such as click occurred, and *hierarchy viewer*, which keeps a mapping of the coordinates of different views present on the screen. We combine the information from both the tools to record the user events in such a way that they can be replayed at a later time or used to figure out other paths possible through the app.

2) Heuristic Path Generator: Heuristic Path Generator is the component that is responsible for generating the unexplored paths to be executed by the app. It is based on UI fuzzing technique, which is a recent advancement in mobile software testing [12]. The intuition behind the heuristic path generator is, if we perform an action on a view, such as clicking a button, that could lead to a network behavior, then there is a high probability that performing the same action on other parallel views (i.e., clicking on other buttons) will also have network behaviors. Here the parallel views means the same type of widget in a activity. For example, all the `Buttons` within an activity are parallel, and all the `ImageViews` within an activity are parallel.

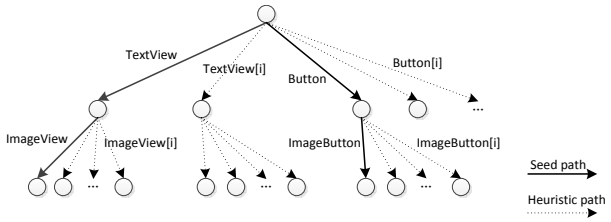


Fig. 10: Generated paths.

Considering the *Zedge* app, its main activity shows three `TextViews` corresponding to the three options (Figure 1a). If we click on *Wallpapers*, then there are six wallpapers shown on the screen as `ImageViews` (Figure 1b). No matter which wallpaper is clicked, a new download activity with a larger version of this wallpaper appears with a download button. If we only have one network trace, the fingerprint is very specific to the clicked wallpaper. But if we can obtain multiple similar network traces, then the fingerprint will be much more abstract and representative. The path of downloading the wallpaper in *Zedge* could be: (1) click wallpaper in the main activity, (2) click one wallpaper in the second activity, (3) click download button in the third activity. The paths generated by our algorithm are shown as dotted lines in Figure 10. This heuristic algorithm can not only increase the coverage of the paths with different network behaviors (which makes our profiles more representative), but also increase the number of traces with similar network behaviors (which make our fingerprints more abstract). For example, in *Zedge* app if we have only one path in the manual run corresponding to downloading the wallpaper as the seed path, then we can get other paths in step (ii), i.e., download the ringtones and notification sounds as well, since they are all parallel `TextViews`. Also, we can generate more paths corresponding to downloading other wallpapers by replacing the `ImageView` clicked in step (ii) with different `ImageViews`.

3) Path Replayer: Path Replayer is a dynamic path driven engine which forces the app to execute a given path and then captures the network trace of the app. It consists of four components: (i) View Identification Module, (ii) Event Emulation Module, (iii) System API Logging Module, and (iv) Network Traffic Capture Module, shown in Figure 11. The View Identification Module identifies the views, such as the button positions, in the current activity. The Event Emulation module takes the paths as input and perform the actions one by one. It supports different user behaviors such as click-

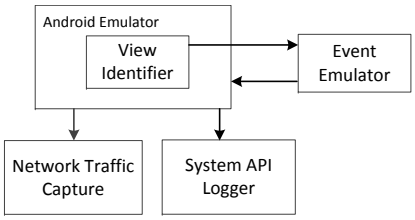


Fig. 11: Path Replayer.

ing/swiping on the screen, sending the keyboard events(such as pressing the menu button), sending broadcast events (such as sms receive notification). The Network Traffic Capture Module captures the network traffic using *tcpdump*. The System API Logging Module is used to identify which network traffic is originating from the app under observation. We ported the strace utility to Android to log each networking system call performed by the app. We identify all the threads started by the app using the process id (pid) of the app. Based on this information, we can filter out the traffic that does not origin from the app. The System API Logging Module and the Network Traffic Capture Module are used by the Random Tester as well.

IV. Evaluation

In this section we provide the details about the evaluation of our techniques which mainly focused on free apps. First, we focus on fingerprints for third-party traffic. As explained in section III-A1, flows for third-party services can be generated more efficiently. Also, if the third-party flows contain unique and persistent key-$value$, we can use the fingerprint for the identifying the apps and need not generate the more complex fingerprints for other traffic for the app.

A. Ad Traffic

As a case study of third-party services, we evaluated the fingerprint extraction algorithm for ad libraries in detail. We picked ads for our study as our experiments with two hours of mobile traffic from a national cellular provider showed that ads account for a major portion of the traffic. Michael et al [10] studied mobile in-app ads in-depth to identify private information being leaked. They identified the 100 most popular ad libraries used by 100K randomly chosen apps. These ad libraries were embedded in more than half of the apps. We used these ad libraries as our reference as the paper presents a straight-forward way for identifying these ad libraries from the manifest files. We crawled the Android market and downloaded 90K free apps. We were able to determine that 70K (i.e., 87%) of these apps asked for the `android.permission.INTERNET` in their manifest files. This permission is needed by any app which needs to access the network. For 32K of these apps we were able to identify the ads library that was being used by examining the manifest files. Figure 12 shows the number of ads library used by each app. We can see that a majority of the apps (≈25K) use only 1 ad library and only 1% of the apps use more than 5 ad libraries. This means that number of fingerprints that we need to generate is typically between 1 and 5 for these apps. This makes the use of fingerprints for a large number of apps, in large-scale network traffic data analysis, efficient and practical.

Next we tried to understand the effort required in identifying the name of the identifier key used by the ads library. We picked 10K apps randomly from the 32K apps that we knew contained ads. We ran each app in a separate emulator using Random Tester and collected the traces for it. We determined the ads used by each app using the hosts that the ad traffic connected to for 30 of the most popular ad libraries. We observe that maximum number of apps (1462) use GoogleAds, which confirms the findings of [17], although their experiments were done with a small number (18) of apps. Figure 13 shows that the top 10 ads libraries account for a majority of the apps and the rest 20 for a small fraction of the apps. The long-tail distribution suggests that we can cover a majority of the apps by handling the top, say 100-200, most popular ads library. Our findings from the dynamic analysis, i.e., from traffic traces are consistent with those in [10], where the top 100 ad libraries were being used by more than 50% of the apps, as identified by static analysis of the manifest files .

Different ad providers may use different identifier names. We examined over 3K apps that contained these 30 ad libraries to see which ad libraries have an app identifier in the manifest file. We observed that almost half of the ads libraries had identifier names in the app manifest files (Table III). Note that the identifiers listed in this table are not necessarily unique for apps. For example, Aduru uses a developer key while Wooboo and Domob may be using publisher id. We looked up ad provider documentation, and picked up the identifier names only when they are guaranteed to be unique for each app. Next we examined the traffic for each ad library from the apps to find out the $query$-key that is used. Note the $query$-key may be different from the identifier name in the manifest file. If we consider the Adwhirl library, we can see from Table III the identifier name is `ADWHIRL_KEY` while Table II shows that the identifier value is provided for $query$-key `appid`. Table II shows the $query$-$keys$ used by different ad providers as unique identifiers. The way we figure out the $query$-key for each ad library is that we run 10 different apps which contain that ad. For each app, we look for the value associated with the ad identifier in the query strings. If for all 10 apps for an ad, the $query$-$values$ corresponding to the identifier values in manifest are occurring with the same $query$-key, then we know that the given $query$-key and its $query$-$value$ are the unique tokens needed to identify the app. For the ad libraries for which the unique identifiers are present explicitly in the manifest file, we can create this mapping between the identifier name in manifest file and the $query$-key. Then for any new app that uses this ad library, we can generate the fingerprint just by extracting the identifier value from its manifest file. For the ad libraries which do not have such explicit identifiers, we need to figure out the identifier $query$-key from manual inspection of traffic and other sources such as publisher guidelines, static analysis

TABLE III: Explicit ID for Ad Libraries.

Ads Library	Key in App Manifest
Admob	ADMOB_PUBLISHER_ID
Mobclix	com.mobclix.APPLICATION_ID
Adwhirl	ADWHIRL_KEY
Waps	WAPS_ID
Wooboo	Wooboo_PID
Domob	DOMOB_PID
Admarvel	ADMARVEL_PARTNER_ID
Admogo	ADMOGO_KEY
Madvertise	madvertise_site_token
Adwo	Adwo_PID
Nexage	NEXAGE_DCN
Flurry	flurry_key
Tapjoy	tapjoy_key
Aduru	ADURU_DEVELOPER_ID

of sample apps, or even by combining all flows for the same app provider for an app and using the Fingerprint Extractor to identify the invariant tokens. The mapping between apps and their $query\text{-}value$ can then be generated in our system based on Random Testing. Lastly, for apps which contain identifier keys but which are not unique to the app, we fall back to the Directed Testing method for extracting the fingerprints.

We used the fingerprints based on the ads to identify apps in a two hour long trace for a cellular provider. We were able to identify 306 apps in the traces. Since we did not have ground truth about the apps present in the network, we manually inspected the traces to verify the accuracy of the identification. We found that 159 of those had app name as the unique key value. So we can say with certainty that these apps were identified correctly. For the flows identified as belonging to the remaining 147 apps we extracted all flows occurring within 5 minutes before and after these flows. Our assumption here is that the flows belonging to the same app must exhibit some temporal proximity. Even though this is not completely accurate, it is a heuristic which works well when devices have single apps running on them. From the flows that are close by (in time), we tried to manually verify whether the identified app was actually running at that time. We used origin traffic as well as keywords in traffic such as video, music, and games to figure out whether the app was indeed running at that time. Note that this information is not fool-proof and can not be used for doing the identification in the first place because (i) same host may serve multiple apps (ii) origin traffic may be absent in the traces. We were able to verify the presence of many (65) of the apps. We were not able to verify the presence of other apps by manual inspection which shows the inherent difficulty in app identification from network traffic.

B. Non-Ad Traffic

To evaluate the non-ad traffic fingerprints, we considered 6 popular apps - *Youtube*, *Flixster*, *ESPN Score Center*, *CNET News*, *Pandora*, and *Zedge* [2]. We manually generated a seed action path for each app. We provided this seed action path and the installation package of the app to the NetworkProfiler system. We excluded all ads traffic from the traces generated by the Directed Testing of the apps. We extracted network profiles for the remaining traffic. We used the generated network profiles for identifying apps from annotated network traces. Specifically, we manually executed each app separately. We captured the network traffic during the execution of each app and annotated the network trace with the name of the app. These annotations represent the ground truth about which app the network trace corresponds to. It should be noted that we had one group of people perform the seed-action-path generation during the network profile generation phase, and another to perform the generation of annotated network traces. To further ensure that the generated traffic is not biased towards our generated network profiles, we required the two groups to work independently without sharing any information about how they execute the apps.

We consider network fingerprints containing only the hosts and no state machines to be trivial and exclude them from the evaluation. For each app, we used all its non-trivial fingerprints to match the traffic in all the annotated network traces. We observed that the fingerprints never match traffic from any other app. This shows that we obtain high precision, i.e., low false positives. Also, we did not fail to identify any app that was present in the annotated network trace, i.e., we succeeded in identifying all apps for which we had generated the network profiles. We were also able to identify these apps in the two hour traces and verify their validity using ad flows close to the identified flows.

C. Limitations

We can not distinguish apps which use the same service and have no distinct network behavior. For example, if two apps use Google Maps service and nothing else on the network, then both contact the same hosts and have the same state machines as they are using the same api. Similarly, when different versions of the same app have no distinct network behaviors, we end up generating the same network profiles for the versions. We believe this is not a serious limitation as these apps, from a network behavior perspective are same, and thus, we can provide multiple labels (corresponding to the different apps, or different versions of the app) when a shared fingerprint matches a flow. This is true even in the case of third-party traffic, where a developer may use developer id for apps and the apps may not differ in network behavior. Indeed, we saw in our experiments that one app developer had generated multiple copies of the same wallpaper download app. So the network profiles for all these apps were similar.

Another limitation of our work is that we need a user seed path when login is involved. In future, we plan to explore automating this as well crowd-sourcing approaches for obtaining seed path. Other limitations relate to the time required to download and run apps as we need to stop and start the emulator for every app being analyzed to ensure that they are run in a clean environment. This required approximately 1 minute per app for random testing which translates to running 1440 apps a day. This means that we required 7 days for running 10K apps even in the simplest case. To improve

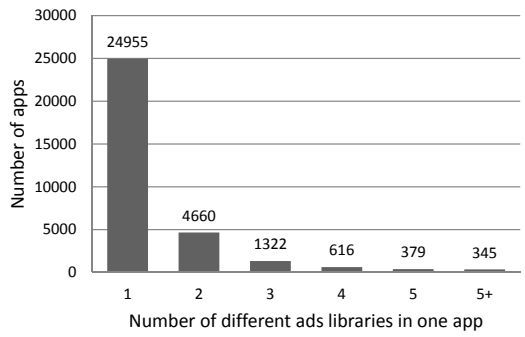


Fig. 12: Number of different ads libraries in one app.

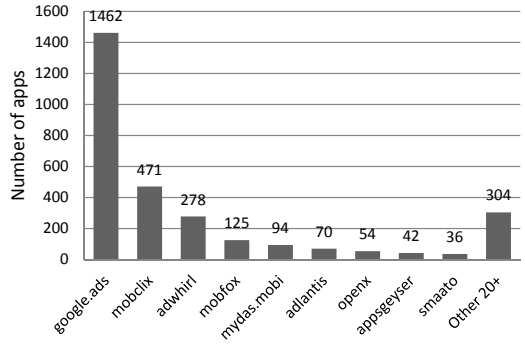


Fig. 13: Number of different ads traffic.

the scalability of the analysis, we are building a cluster for parallelizing these steps through use of virtual machines. This will also help us overcome the limits that many app markets implement on the number of apps that can be downloaded in a day from an IP.

V. Related Works

There have been a large number of efforts on generating signatures of applications from their network traffic [14] that use the statistical information, such as packet sizes, to perform a coarse-grain protocol level classification. Discoverer[7] focuses on generating application-level signatures from the network traces of the application under study. However, those techniques do not work well for the Android apps where a majority of traffic is carried within HTTP and the distinguishing features are present in the URLs. In related research, ([16], [13]) target signature generation for identifying worms.

Recently there have been many efforts that try to understand smartphone usage behavior [18], [9]. None of these papers present a systematic way for identifying Android apps in real-world traffic. [17] aims to build app profiles at multiple levels, including network, but their technique completely relies on users running apps to generate traffic. This does not scale for a large number of apps.

There has been a lot of work on analyzing Android apps for malware. But most of these target monitoring of apps [8] or static analysis of app code [10]. None of these works focus on automatically executing the different parts of an Android app.

VI. Conclusion and Future Work

In this paper, we proposed a novel system called NetworkProfiler for the automated generation of network profiles for Android apps. Our evaluation shows that we can identify the apps with high precision. In future, we want to build a comprehensive network profile library for the apps present in the Android Market. Further, we plan to combine static analysis with the dynamic analysis to improve our coverage of execution paths within Android apps.

References

[1] http://developer.android.com/tools/help/index.html.
[2] https://play.google.com/store/apps/.
[3] http://www.canalys.com/.
[4] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008.
[5] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.
[6] A. C. Callado, C. A. Kamienski, G. Szabo, B. P. Gero, J. Kelner, S. F. L. Fernandes, and D. F. H. Sadok. A survey on internet traffic identification. *IEEE Communications Surveys and Tutorials*, 11(3):37–52, 2009.
[7] W. Cui, J. Kannan, and H. J. Wang. Discoverer: automatic protocol reverse engineering from network traces. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.
[8] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.
[9] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proceedings of the 10th annual conference on Internet measurement*, 2010.
[10] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.
[11] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. Acas: automated construction of application signatures. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, 2005.
[12] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, 2011.
[13] N. James, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.
[14] T. Nguyen and G. Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys Tutorials, IEEE*, 10(4):56 –76, quarter 2008.
[15] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.
[16] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004.
[17] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, 2012.
[18] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, 2011.