MobiScope: Pervasive Mobile Internet Traffic Monitoring

Ashwin Rao
INRIA

Amy Tang
UC Berkeley

Shen Wang
University of Washington

Nick Martindell
University of Washington

Justine Sherry
UC Berkeley

Walid Dabbous
INRIA

Arvind Krishnamurthy
University of Washington

Arnaud Legout
INRIA

David Choffnes
Northeastern University

ABSTRACT

Characterizing Internet traffic naturally generated by mobile devices is an open problem because mobile devices and their OSes provide no built-in support to monitor network traffic. Existing approaches rely on privileged access to passively monitor traffic in ISPs or rooting phones to capture traffic from devices.

In this paper, we take an alternative approach: monitoring through indirection. Specifically, we exploit the fact that most mobile OSes support proxying via virtual private networks (VPNs). Sending mobile Internet traffic through a proxy server under our control enables us to monitor all flows regardless of device, OS, or access technology.

We present the architecture of *MobiScope*, a single-server software package that can monitor all mobile traffic with low overhead, and that provides a convenient plugin infrastructure to provide custom analysis not available through raw packet traces. In particular, we use a SSL bumping module that provides visibility into most SSL traffic. Then, using *MobiScope* on both controlled experiments and a 7-month IRB-approved in-the-wild study with a small set of real users, we classify application traffic and determine how those apps leak privacy-sensitive data.

1. INTRODUCTION

Today's mobile systems are walled gardens inside gated communities, *i.e.*, locked-down operating systems running on mobile devices that interact over a closed and opaque mobile network. Characterizing Internet traffic naturally generated by mobile devices thus remains an open problem.

The key challenge is that mobile devices and their OSes provide no built-in service for monitoring and reporting *all network traffic*. As a result, previous studies [26, 14, 8, 11, 27, 24] are constrained by at least one of the following: mobile OSes, access technology, device manufacturer, installed applications, and user behavior. In this work, we are the first to present an approach that compromises none of these, potentially enabling a large-scale deployment and comprehensive view of Internet traffic across carriers, mobile devices, apps, and access technologies.

This paper explores the opportunities for mobile traffic measurement through indirection. Specifically, we exploit the fact that most mobile OSes support proxying via virtual private networks (VPNs). By sending mobile Internet traffic through a proxy server (an approach we call *MobiScope*), we monitor all flows regardless of device, OS or access technology. Importantly, installing a VPN configuration does not require installing apps or rooting phones, thus facilitating large-scale deployment on unmodified OSes.

Our key contributions are as follows. We describe how we implemented a proxy-based measurement system for characterizing mobile Internet traffic for iOS and Android, and we demonstrate that it is sufficiently transparent to avoid significantly impacting measurement results. *MobiScope* captures all Internet traffic with approximately 10% power and packet overheads, and negligible additional latency. We will make the *MobiScope* software and configuration details open source and publicly available.

Next, we analyze network traffic from controlled experiments for more than 1,200 apps (iOS and Android), and from human subjects during a 7-month study, comprising 21 users and 26 devices. To the best of our knowledge, this is the first study to report a holistic view of network traffic from real user devices running iOS and Android. These users interact with networks in 54 ASes, 9 of which as cellular; their traffic strongly depends on OS and network type.

We develop new classification techniques to mapping observed network flows to mobile apps. Our approach correctly identifies 89.2% of iOS and 64.1% of Android apps and Web services. We also take a first look at classifying the traffic contained in SSL tunnels.

Using *MobiScope*, we study privacy-sensitive information leaked by mobile apps. During our experiments we observed that iOS and Android apps leak device identifiers and other sensitive information such as e-mail addresses in the clear. Specifically, we observe that one of the most popular iOS app used by US physicians sends the user's first name, last name, e-mail, password, and zip code without encryption. We further demonstrate the extent to which users in our study are tracked by mobile devices over time and across access technologies.

Last, we make available a new visualization tool for user to track and control how they are being tracked. *MobiScope*'s *ConVis* tool borrows the Mozilla project's Collusion interface, but applies it to all of a mobile device's traffic. It also facilitates specifying block lists for connections based on (app, third party) tuples. A demo of our tool is located at `http://goo.gl/A17h9`.

The rest of the paper is organized as follows. We present an overview of *MobiScope* and describe its feasibility in §2, then describe our experiments and measurement dataset in §3. In §4 we study how to classify observed network flows to the mobile apps that generated them. Last, we use the unique view from *MobiScope* to characterize how apps leak private information in the clear and to third parties §5. We discuss related work in §6 and conclude in §7.

2. MOBISCOPE OVERVIEW

In this section, we present an overview of the *MobiScope* platform[1]. The goal of *MobiScope* is straightforward: we seek to enable passive monitoring of *all the Internet traffic from and to mobile devices*. While previous work has accomplished this for a limited set of devices or networks, we seek to avoid such limitations.

1. *OS agnostic.* Monitor traffic independently of the OS run by the monitored device. In particular, we avoid the need to develop OS-specific applications, or to root or jailbreak the device.
2. *ISP agnostic.* Monitor traffic without any support from ISPs and cellular providers.
3. *Access technology agnostic.* Monitor traffic whatever the access technology used by the mobile device (Wifi, GSM, CDMA, UMTS, LTE, etc.)
4. *Continuous.* Monitor traffic continuously, even when devices switch between networks or return from being idle.
5. *Scalability. MobiScope* should be equally feasible to deploy on a single machine or using a collection of VMs in a hosted/cloud deployment.
6. *Encryption agnostic.* Achieve visibility of both encrypted and plaintext traffic.

In the following, we describe in detail the design of *MobiScope*, then we discuss the limitations of the platform.

2.1 MobiScope Design

To meet the above goals, we exploit the observation that nearly all devices support network traffic indirection via virtual private networks (VPNs). Instead of using a VPN server to access a private network, we use it as a proxy for all of a device's Internet traffic. This enables passive network monitoring regardless of OS, carrier/ISP or access technology. Further, we show that this approach is practical in that it has minimal impact on performance and measurement fidelity.

In the following sections, we describe how our measurement infrastructure achieves the goals stated above. We begin by describing how we addressed several challenges with implementing our VPN-based proxy.

[1]Our platform is currently in private beta with deployments in the US, France and China. We will make the *MobiScope* software publicly available.

2.1.1 VPNs for Mobile Devices

We now provide a detailed description of how our design achieves many of the *MobiScope* design goals. Our first goal is a measurement system that works regardless of device OS. To achieve this goal, we note that VPNs are widely supported: Android, BlackBerry, Bada, and iOS all support VPNs (primarily to satisfy their enterprise clients). In this work, we focus on the two most popular: iOS and Android.

Our second and third goals are to enable measurements regardless of ISP or access technology. iOS and Android support VPN connectivity regardless of ISP or access technology – so long as the network supports IPv4.

To meet our goal of continuous network monitoring, a VPN must always be enabled on a device. Currently, all iOS devices (version 3.0 and above) support *VPN On-Demand*, which forces traffic for a specified set of domains to use VPN tunnels. To ensure all possible destinations match this list, we exploit the fact that iOS uses suffix matching to determine which connections should be tunneled; accordingly, we specified the domain list as the set of alphanumeric characters (a-z, 0-9, one character per domain). Android version 4.2 and above supports an *Always On VPN* connection that provides the same functionality; for Android version 4.0 and above there is an app API that allows apps to manage VPN tunnels. We support both options.

Last, to support a deployment at scale we note that both iOS and Android support VPN connections using the IPSec standard. This means we can implement our VPN proxy server using free, robust, open-source code from Strongswan [25].

2.1.2 Monitoring Traffic on MobiScope

Having shown that VPNs support many of *MobiScope*'s goals, we now describe how we implement passive network monitoring using VPN proxying on a single machine (supporting our goal of scalability). While the high-level design for capturing network traffic from mobile devices is straightforward, the implementation is not. In particular, the interactions between IPSec, routing and NAT complicate our ability to map bidirectional flows to individual devices. The following paragraphs describe these challenges and how we addressed them to provide a stand-alone (*i.e.*, single server) mobile-traffic monitoring proxy.

At first glance, capturing all traffic traversing a VPN server should be as simple as running a tap on the network interface, *e.g.*using *tcpdump*.

To explain why this is insufficient, we first describe how the Linux network stack interacts with IPSec. We assume the *MobiScope* server is assigned IP address m and the mobile device's public IP address is d. When the VPN connection is established, the proxy assigns the device a private address v. Last, the device is attempting to access a service located at address w. We denote a packet from source s to destination d as $s \rightarrow d$.

Forward path. We being with mapping flows in the forward direction. Figure 1 (a) shows the path that packets take through *MobiScope*. At steps (1), (2), and (3) the encrypted datagram (in gray, $d \rightarrow m$) is passed to the IPSec module to decrypt and process the encapsulated IP datagram ($v \rightarrow w$). Because the *MobiScope* assigns private address space to clients, it must use NAT in step (5) to convert the private IP address v to the public IP address m. Last, the packet is forwarded to the Internet.

We now describe how running *tcpdump* and tracking the network address translation (NAT) table are sufficient for mapping flows in the forward direction. Running *tcpdump* on the Ethernet device captures packets at step (2), (4), and (7). The NAT table provides a map between the public IP address d of the device and its private IP address v in the tunnel. To associate packets with a device and a destination, we only need the packet captured at step (4) that associates the packet with the private address v of the device in the VPN tunnel and the destination (w), and the NAT table that gives the mapping between the device in the VPN tunnel (address v) and the public IP address d of the mobile device.

Reverse path. In the reverse direction (when packets flow from the Internet to the mobile device), it is no longer possible in general to associate a mobile device with its packets. We refer to Fig. 1 (b), where we continue to dump packets from the Ethernet device. From steps (2) and (7), we know that a packet is sent by the destination at address w to the *MobiScope* box (step (2)), but then this packet is encapsulated at the IPsec layer – address resolution is performed by the NAT without passing through *tcpdump*. So, at step (7) we have no way to know which encrypted packet is encapsulated. We need to dump the packet at step (4), but we have no access to it via the standard Linux networking stack.

Bidirectional mappings. A straightforward solution to the reverse path mapping problem is to forward traffic to a separate NAT device and dump traffic there. To avoid the need for additional hardware/VMs, we virtualize an additional network interface and route traffic through it.

Namely, we use a Linux TUN device, send packets from w to v through it and run packet captures on that TUN device (Figure 1 (d)). Indeed, when a packet is received from the Internet and arrives at NAT at step (3), the NAT resolves the public address of the *MobiScope* box m to the address of the device in the VPN tunnel v. We assign to each client an address from a space of prefix length $p - 1$. The pth bit has a specific role. By default it is set to 0; for addresses that match the prefixes assigned to the mobile devices, a process running on the TUN device changes v to v' (and also v' to v) by flipping the value of the pth bit. This facilitates routing to the TUN device all packets whose final destination is v, and forwarding all packets with a destination address v' (pth bit set to 1) to the TUN device, step (5). On the TUN device, our process changes the destination address from v' to v and sends the packet to the IP layer, step (6). Then the packet follows the path of a regular packet in a VPN tunnel.

When a packet is received from the mobile device, we perform a similar process that is described in Figure 1 (c).

2.1.3 Interposing on Traffic on MobiScope

A key limitation of previous approaches to measuring traffic from mobile devices is that the payloads of encrypted (SSL) traffic are unavailable for analysis. As increasing amounts of Web traffic flow over HTTPS, we lose the ability to understand how to optimize such traffic or to evaluate what information is exfiltrated over such tunnels. This has implications both for performance (page speed optimizations) and privacy (PII leakage over secure channels). In this section, we describe how *MobiScope* allows us to analyze the contents of SSL flows generated by mobile devices, achieving the final goal stated at the beginning of this section.

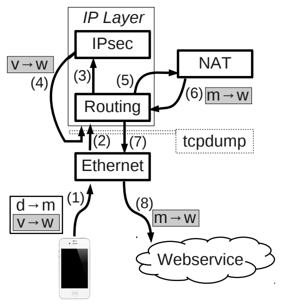
Plugin infrastructure. Figure 2 shows how *MobiScope* supports a plugin infrastructure for custom flow processing. Each plugin takes as input a network flow and outputs a network flow (potentially empty). When a packet is received at the TUN device, it is sent to a software-defined switch [4] that determines the ordered set of plugins that flows will traverse. This order is configured by a policy manager, which determines the set of plugins that should operate on each flow. After the last plugin is traversed, the network flow is sent out through the TUN device to the Internet.

Plugins support a variety of features such as ad blocking, analyzing PII leakage or page speed optimization. In the following, we describe how we use a plugin to enable SSL traffic decryption using the *MobiScope* plugin infrastructure.

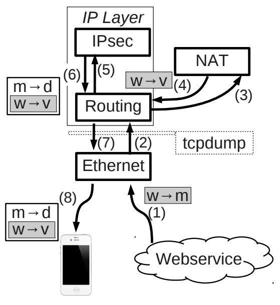
Example plugin: SSL bumping. First, we note that our VPN proxy implementation uses a self-generated *MobiScope* root certificate that is used to sign all subsequent certificates issued to participating mobile devices. This allows us to perform SSL traffic decryption using the Squid proxy's SSL bumping [5] feature, which is essentially a man-in-the-middle operation on the secure connection.[2] When the mobile device connects to a service supporting SSL, the proxy impersonates the service using a forged certificate signed with the *MobiScope* root certificate. Then the proxy establishes an SSL connection with the intended target, impersonating a mobile device. Using the traffic dumped by the *tcpdump* process as shown in Figs. 1 (c) and 1 (d), and using the private key generated by the squid proxy to communicate with the mobile device, we can decrypt all SSL traffic. We note that when traffic is not encrypted using SSL, the proxy simply forwards traffic.

This approach will fail for any app that does not trust certificates signed by unknown root authorities. Surprisingly, this is rarely the case. Whereas the Twitter and Firefox apps prevent SSL bumping by validating root certificates,

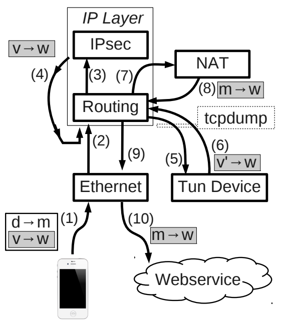
[2]Note that for privacy reasons we do not decrypt traffic generated by human subjects; rather, we use this for controlled experiments in the lab setting.



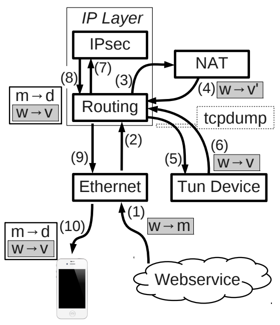
(a) Packet from mobile device. *Tcpdump can capture packets at step (2) d → m, (4) v → w, and (7) m → w.*



(b) Packet to mobile device. *Tcpdump can capture packets at step (2) w → m and (7) m → d, however it is cannot log the packet w → v.*



(c) Packet from mobile device. *Tcpdump monitoring packets on the tun device can capture packets at step (5) v → w, and (6) v' → w.*



(d) Packet to mobile device. *Tcpdump monitoring packets on the tun device can capture packets at step (5) w → v', and (6) w → v.*

Symbol	Description
d	IP address of the mobile device assigned by its ISP.
m	IP address of the *MobiScope* server.
w	IP address of the server providing the Web service.
(i)	The i-th step of packet processing.
$\boxed{a \rightarrow b}$	Packet with source IP *a* and destination IP *b*.
v	IP address of the mobile device in the VPN tunnel.
v'	Temporary IP address of the mobile device used to send the packet to the TUN device.

Figure 1: Packet monitoring in the *MobiScope* box.

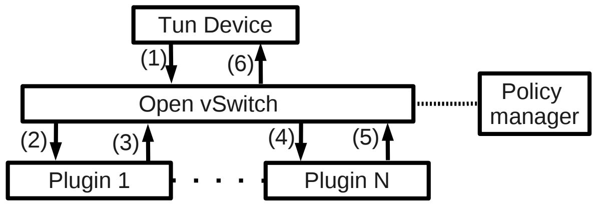


Figure 2: Plugin Infrastructure on *MobiScope*.

the Google Chrome, Safari, Facebook, and Google+ apps, as well as the default mail clients and advertisement services, do not check the validity of the root certificate. This enables our approach to provide visibility into secure channels established with a wide range of popular apps.

2.2 Limitations and Deployability

MobiScope provides a scalable way to achieve pervasive, portable and passive monitoring of network traffic from mobile devices. In this section, we discuss several issues that impact the coverage and deployability of our approach. Note that these limitations have not significantly impacted our ability to measure mobile networking traffic or to deploy our approach to users.

2.2.1 Limitations

At most one tunnel. Currently iOS and Android support exactly one VPN connection at a time. This allows *MobiScope* to measure traffic over either WiFi or cellular interfaces, but not both at once. The vast majority of traffic uses only one of these interfaces, and that interface uses the VPN.

Proxy location. When traffic traverses *MobiScope* box, destinations will see the *MobiScope* box address, not the device IP, as the source. This might impact services that customize (or block access to) content according to IP address (e.g., in case of localization). A solution to this problem is to use a *MobiScope* instance with an appropriate IP address.

ISP support. Some ISPs block VPN traffic, which prevents access to our current *MobiScope* implementation. We note that few ISPs block VPN traffic, and there is an incentive not to block VPN traffic to support enterprise clients.

Limited ISP characterization. Due to its use of encrypted channels, *MobiScope* cannot detect traffic differentiation or any other techniques that ISPs use to interpose on network traffic using deep packet inspection (*e.g.*advertisement insertion [22]) or optimization (*e.g.*downsampling content).

IPv6. *MobiScope* cannot be currently used on networks using IPv6 because IPv6 is not fully supported by mobile devices. Indeed, we observe that though iOS and Android support IPv6 they currently do not support IPv6 traffic through VPN tunnels.

2.2.2 Deployability

MobiScope uses standard and freely available software to manage and record traffic from mobile devices, making it easy to deploy to users and servers. However, a key question is whether the system is sufficiently efficient to minimize its impact on both controlled and in-the-wild experiments. We show empirically that the overheads are reasonable, and provide a brief discussion of incentives for users to adopt our system for in-the-wild measurements.

We identify the following key aspects of user-perceived inefficiency from proxying their network traffic through *MobiScope*:

- **Establishment delay.** We measured 50 VPN-connection establishment times on both iOS (iPhone 5 / iOS 6.1) and Android (Galaxy Nexus / Android 4.2), for Wi-Fi and cellular connections. For Android (using IKEv2), the maximum establishment time was 0.81 seconds on Wi-Fi and 1.59 seconds on cellular. For iOS (using IKEv1), the connection takes longer due to the older protocol version: we observe a maximum of 2 seconds on Wi-Fi and 2.18 seconds on cellular. Because each VPN session supports many flows, the amortized cost of connecting is small.

- **Encapsulation overhead – data consumption.** *MobiScope* uses IPsec for datagram encryption, thus there is an encapsulation overhead for each tunneled packet. To evaluate this overhead, we use 30 days of data from 25 devices to compare encapsulated and raw packet sizes. We observe a maximum encapsulation overhead of 12.8% (average approximately 10%). Within the scope of the traffic monitoring experiments performed with *MobiScope*, the impact of this overhead is negligible. However, in case of experiments with a limited cellular data plan, this overhead must be taken into account.
- **Encapsulation overhead – power consumption.** Mobile devices expend additional power to establish, maintain and encrypt data for a VPN tunnel. To evaluate the impact on battery, we used a power meter to measure the draw from a Galaxy Nexus running Android 4.2. We run 10-minute experiments with and without the VPN enabled. For each experiment, we used an activity script that included Web and map searches, Facebook interaction, e-mail and video streaming. The VPN leads to a 10% power overhead.

  For iOS devices, where we cannot attach a power meter directly to the battery, we conducted an experiment using video streaming to drain a fully charged battery with and without the VPN enabled. We again found approximately 10% power overhead.

In summary, the overheads are low enough to avoid significant interference with user activity. To further reduce this overhead, we are investigating using NULL encryption (*i.e.* no encryption) for users that do not need the additional privacy enabled by *MobiScope*.

Incentives. *MobiScope* simplifies measurements from controlled experiments. However, it is also important to measure traffic from real users, so we must provide an incentive for them to use the platform. We have developed a variety of user incentives that can offset the costs of *MobiScope*; here, we name a few of the most interesting ones.[3] For example, we provide users with fine-grained views of privacy leaks from their applications, and allow them to enable a *MobiScope* plugin that blocks PII leakage. In addition, we are investigating the opportunities for page speed optimization. Last, we have implemented opt-in device-wide ad-blocking, which can reduce the volume of costly cellular traffic for ads.

[3]We have implemented most of these examples, but their details are beyond the scope of this paper.

3. DATASETS DESCRIPTION

Using *MobiScope*, we collected full packet traces from Internet activity generated by mobile devices. We use this data to study how to map monitored traffic to applications, and to analyze PII leakage. Below, we describe our data-collection methodology, which consists 1) controlled experiments in a lab setting and 2) IRB-approved “in the wild” measurements gathered from real users during seven months.

3.1 Controlled Experiments

Our goal with controlled experiments is 1) to obtain ground truth information about network flows generated by apps and devices, and 2) characterize the network activity for a large variety of popular apps in a lab setting. We use this data to understand how to map network flows to the app that generated them, and how to identify PII in those network flows.

Device setup. We conducted our controlled experiments using three devices: a Galaxy Nexus running Android 4.2, a Google Nexus running Android 4.0, and an iPhone 3GS running iOS 6. We start each set of controlled experiments with a factory reset of the device to ensure that software installed by previous experiments cannot impact the network traffic generated by each device. Then we connect the device to the *MobiScope* platform, we enable the SSL-Bumping plugin, and begin the experiment.

Manual tests. We manually test the 100 most popular free Android application in the *Google Play* store and 209 iOS applications from the iOS App store on April 4, 2013. For each application, we install it, enter user credentials for the account if it is relevant, interact with it for up to 10 minutes, and uninstall it. This allows us to characterize real user interactions with popular applications in a perfectly controlled environment. Note that because we enter a unique and distinguishable set of user credentials when interacting with apps, we can easily extract the corresponding PII from network flows (if they are not obfuscated).

Automated tests. The second set of controlled experiments consist of fully-automated experiments on the most popular 908 Android applications from a free, third-party Android market, *AppsApk.com* [1]. We perform this test because Android devices can install *Third-party applications* that are not available on the *Google Play* store, without requiring the user to root the device.

Our goal is to understand how these apps differ from those in the standard *Google Play* store, as they are not subject to Google Play restrictions. We automate experiments using *adb* to install each app, connect the device to the *MobiScope* platform, and start the app. Then we use *Monkey* [6], an app-scripting tool, to perform a series of 10,000 actions that include random swipes, touches, and text entries. Finally, we use adb to uninstall the application and reboot the device to forcibly end any lingering connections. This set of experiments is limited to Android devices because iOS does not provide equivalent scripting functionality.

3.2 In The Wild Measurements

The controlled experiments in the previous section provide us with ground-truth information for a large number of apps running in a controlled setting for a short period of time. To understand the network behavior of devices with real users "in the wild" over longer time periods, we conducted an IRB-approved measurement study with a small set of subjects, from Oct. 20, 2012 to May 20, 2013.[4]

We deployed two *MobiScope* servers, one in the USA and one in France that were used by 26 devices: 10 iPhones, 4 iPads, 1 iPodTouch, and 11 Android phones. The Android devices in this dataset include the Nexus, Sony, Samsung, and Gsmart brands while the iPhone devices include one iPhone 3GS, four iPhone 5, and five iPhone 4S. These devices belongs to 21 different users, volunteers for our IRB approved study. This dataset, called *mobWild*, consists of 208 days with data; the number of days for each user varies from 5 to 205 with a median of 35 days. For privacy reasons, the SSL-Bumping plugin is *disabled* for all measurements involving real users.

Capturing all of a subject's Internet traffic raises significant privacy concerns. Our IRB-approved study entails informed consent from subjects who are interviewed in our lab, where the risks and benefits of our study are clearly explained. The incentive to use VPNs is Amazon.com gift certificates awarded by lottery. To protect the identity of information leaked in the data, we use public key cryptography to encrypt all data before storing them on disk; the private key is maintained on separate secure severs and with access limited to approved researchers. Further, subjects are free to delete their data and disable monitoring at any time. Per the terms of our IRB, we cannot make this data publicly available due to privacy concerns. We are investigating alternative data-collection techniques that provide user anonymity sufficient for sharing with other researchers.

4. TRAFFIC CLASSIFICATION

In this section, we revisit mobile traffic characterization and classification, using a comprehensive view of Internet traffic from mobile devices. Previous work uses passively gathered data to characterize such traffic, which can be useful for a variety of important topics that include traffic engineering, optimization of network-enabled apps and understanding threats to user privacy. The following sections show that 1) for the users in our study, traffic over WiFi and cellular networks are qualitatively different, and studies that focus on only one technology will miss approximately half of the traffic generated by devices; 2) previous approaches to classifying traffic fail to identify apps responsible for that traffic most of the time; 3) *MobiScope* facilitates a first look at determining which apps generate traffic over SSL connections.

4.1 Descriptive Statistics from User Study

This section highlights key features of the dataset gathered from users participating in our study (*mobWild*). Due to the relatively small number of users in our study, we cannot draw strong, generalizable conclusions; rather, we use data gathered from them to demonstrate that there is a need for *MobiScope* to obtain a comprehensive view of Internet traffic from mobile devices.

Network connectivity. First, we describe the diversity of networks that our users connected to. We infer the access technology (WiFi or cellular) using the AS description from *WHOIS* data for each IP address used by a mobile device. Based on this classification, the *mobWild* dataset consists of traffic from 54 distinct ASes, of which 9 are *cellular* ASes. During the measurement study, each device connected our *MobiScope* server from at most two distinct cellular ASes. In contrast, a median of 4 Wi-Fi ASes were observed per device and for one device we observed traffic from 25 different Wi-Fi ASes spread across 5 countries. In terms of traffic volumes, collectively our users' devices transferred 24-56% of their traffic over cellular (iOS and Andriod, respectively), and the remainder over WiFi. The key take-away is that, *for our users*, instrumenting a single cellular carrier or WiFi access point misses a large fraction of traffic generated by mobile devices. *MobiScope* avoids this limitation.

Traffic protocol classification. We begin our identification process using the classification provided Bro [20]. Bro uses the protocol field in the IP header to broadly classify the flows, and we use this classification to label flows as either TCP, UDP, or *other*. Bro further classifies TCP flows using well defined port numbers, and we use this classification to label flows as either HTTP, SSL (which includes HTTPS, IMAP, etc.) or *other* flows. Similarly, we use Bro to label UDP flows as either DNS or *other*. Table 1 presents a summary of traffic generated by user devices in our study.

There are two key additional take-aways from this table. First, Web and SSL traffic dominate traffic for users in *mobWild*, and there is significant diversity in the usage patterns for users with Android and iOS devices. For example, more than 92% of the traffic in our *mobWild* dataset is either HTTP or SSL, the fraction of total flows over cellular or Wi-Fi differ significantly for each OS. This motivates the need for a platform that covers multiple OSes and multiple access technologies. Second, a significant portion of flows occur over secure (SSL) connections that generally prevent classification using deep packet inspection. This calls into question the overall effectiveness of traffic optimization approaches that rely on middlebox technologies that interpose of plaintext traffic (*e.g.*page rewriting or downsampling media).

An important question for network characterization is which app is responsible for which network flows. As we demonstrate in the following section, previous approaches are insufficient for mapping the majority of apps to their corresponding network flows. We describe several techniques to improve this mapping, and present results for controlled ex-

[4]The measurement study is ongoing, we report the most recent subset of results.

IP Protocol	Service	Android		iOS	
		Cell.	Wi-Fi	Cell.	Wi-Fi
TCP	HTTP (%)	35.39	68.67	52.11	75.51
	SSL (%)	61.14	27.37	46.76	18.76
	other (%)	2.34	3.29	0.26	1.81
UDP	DNS (%)	0.67	0.49	0.56	0.34
	other (%)	0.32	0.11	0.27	3.57
Other	other (%)	0.14	0.07	0.04	0.01
total (%)		100.00	100.00	100.00	100.00
Traffic Volume (GB)		9.467	18.79	14.89	89.54
# Flows		851074	658331	661736	2167448

Table 1: Traffic volume (in percentage) of popular protocols and services on Android and iOS devices over cellular and Wi-Fi. *TCP flows are responsible for more than 90% of traffic volume. Traffic share of SSL over cellular networks is more than twice the traffic share of SSL over Wi-Fi.*

periments and the *mobWild* dataset.

4.2 Classification of Mobile Applications and Services

Our *mobWild* dataset suggests that apps, OS services and libraries often rely on HTTP and SSL to exchange data [17, 13, 28]. In the following analysis, we focus on identifying the apps, OS services, and other services responsible for these HTTP and SSL flows. We use ground-truth data from controlled experiments to show that the previous approach for classification fails for most popular apps; we then develop techniques to improve this mapping and apply it to our *mobWild* dataset.

4.2.1 HTTP Traffic Classification

Mapping network flows to apps is an important step for determining the origins of potentially costly network traffic, and for identifying which apps are responsible for privacy leaks. One approach for this is to use the *User-Agent* field on app-generated traffic. Previous work uses this to classify traffic, but only at the granularity of the app category [12, 28, 17]. Below we describe the limitations of this approach for a large fraction of apps, and how we improve classification using only header information (*i.e.* HTTP and lower layers).

To motivate why this is a difficult problem, consider the following scenario. When using the *Host* field in the HTTP header, a flow with *static.ak.fbcdn.net* in the *Host* field implies that it contacted one of the Facebook servers; however, the *Host* field does not tell us whether Facebook is accessed via a Web browser or through the native Facebook app. Similarly, the *User-Agent* field can specify an app name, but it is not required to do so. Below, we show the effectiveness of combining these approaches.

Controlled experiments. In table 2 we present results from our classification study using controlled experiments. To the best of our knowledge, we are the first to attempt to use ground-truth information to evaluate the effectiveness of app classification using only header data.

iOS: First, we note that 176 of the 209 iOS applications

OS	Store	Apps	Gen. HTTP	Host		User-Agent	Combi-nation
				App.	Org.		
iOS	Apple	209	176	83 (47.1%)	119 (67.6%)	149 (84.6%)	157 (89.2%)
And.	Google	100	92	41 (44.5%)	54 (58.6%)	21 (22.8%)	59 (64.1%)
And.	Other	908	494	106 (21.4%)	139 (28.1%)	32 (6.4%)	163 (32.9%)

Table 2: Classification of applications based on *Host* and *User-Agent*. *A large majority of iOS applications use dedicated User-Agent strings to fetch data over HTTP. A combination of User-Agent and Host can be used to identify the majority of Android and iOS applications.*

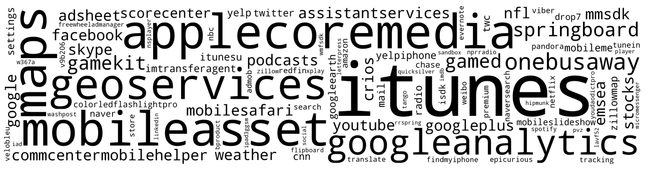
No.	User-Agent
1	YahooMobileMail/1.0 (Android Mail; 1.4.6) (crespo;samsung;Nexus S;4.1.2/JZO54K)
2	AppleCoreMedia/1.0.0.10A523 (iPad; U; CPU OS 6_0_1 like Mac OS X; en_us)
3	Dalvik/1.6.0 (Linux; U; Android 4.2.2; Nexus 4 Build/JDQ39)

Table 3: Sample *User-Agent* strings. *The first string contains the application identifier, the second hides the application and describes the OS service/library used, while the third does not contain any useful signature.*

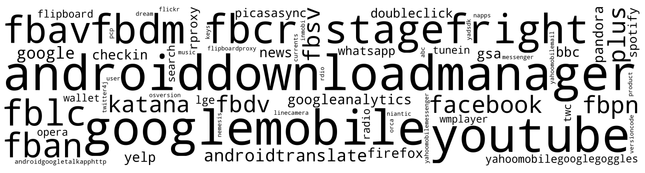
we manually tested generated HTTP traffic. We observe that the *Host* field uniquely identified the corresponding app for 47% of the iOS apps we tested (column 5). The *Host* field also can identify the provider that released an app. For example Zynga offers multiple games with dedicated apps that contact Zynga properties. When classifying apps according to their provider, we observe in table 2 (column 6) that our classification success increases to 67% of iOS applications.

Android: We observe a similar results for flows from apps released through Google Play. However, for the applications from the Third-party store we observe that the *Host* field is less effective. Primarily this is due to the fact that a majority of the applications we tested (about 77%) were stand-alone services such as games where the provider did not use its own Web services. Instead, these applications either contacted some advertisement sites or sites hosted by CDNs. While we can use the *Host* field to identify these 3rd-party sites contacted by an app, we cannot determine which app generated the traffic.

Augmenting with User-Agent: We now show that by augmenting the information in the *User-Agent* to the information gained by the *Host* field, we can classify a majority of iOS and Android applications. In our controlled experiments we observed a non-empty *User-Agent* string in more than 99.7% of the HTTP flows from iOS and 90.9%flows from Android. A *User-Agent* string may contain an application identifier and other auxiliary information such as details of the OS, manufacturer, and compatibility with Web-browsers [2]. For example, the first *User-Agent* in table 3 contains the information of the application, YahooMobile-



(a) iOS



(b) Android

Figure 3: *User-Agent* signatures in iOS and Android HTTP flows. *The font weight represents the number of users for which a particular signature was observed.*

Mail, while the second *User-Agent* hides the application and specifies the *AppleCoreMedia* service of iOS, and the third does not provide any useful information. To extract the application information, we use a set of regular expressions to filter out the auxiliary information in the *User-Agent* and further cluster these extracted tokens using the edit distance between the tokens.[5]

In Table 2 we observe that 176 of the 209 iOS applications generated HTTP traffic, 149 of these were correctly identified based on the signatures in their *User-Agent*, which we verified by manual inspection. In contrast, our *User-Agent* based identification was able to correctly identify less than 23% of the Android applications that generated HTTP traffic. For the 27 iOS applications which we failed to identify, we observed signatures for OS services and libraries, such as *Apple Core Media*, *Game Kit*, *Geo Services*, etc. and signatures of third-party libraries and services such as *Google Analytics* and *Adobe Air*. Similarly, the majority of Android HTTP traffic contained flows with the default *User-Agent* (similar to the third *User-Agent* in table 3). Further, for the applications from the third-party store, we observe that the *User-Agent* for ads and analytics libraries such as *Google Analytics* and *Adsense for Mobile* were the most common *User-Agent* after the default *User-Agent*.

To wrap up, *User-Agent* is more effective for classifying iOS apps and *Host* is more effective for Android apps; however, neither alone is a complete solution. A topic of future work is to explore packet contents using deep packet inspection – it is likely that apps are identifying themselves to CDNs and ad/analytics servers in the payload.

In the wild data. We now describe the results of our classification on data gathered from our user study. Using only the *User-Agent* on the *mobWild* dataset we were able to map flows to 243 iOS and 84 Android apps, OS libraries, and services. The *word cloud* in Table fig:http-wordcloud contains the summary of our results; the font size of a signature is proportional to the number of users for which the signature was observed. Along with signatures of applications such as iTunes and YouTube, we also observe signatures of the *Apple Core Media* and *Stagefright* services that are responsible to download media content on iOS and Android devices respectively. For the iOS devices in the *mobWild* dataset, we observe a signature of *Apple Core Media* in more than 98.45% of the content downloaded from the YouTube servers. Similarly, depending the Android version we observe either the signature for Stagefright[3] or no application or OS service signature for YouTube traffic to Android devices depending on the OS version. We observe a similar signatures for other popular media services such as Netflix, YouTube, Vimeo, Pandora, etc. Because we cannot identify the application used to access these services, we fall back to the *Host* field to identify these Web-services.

[5]We plan to release this code along with *MobiScope* package.

Category	% of iOS Traffic		% of Android Traffic	
	Bytes	**Flows**	**Bytes**	**Flows**
Media (Popular)	51.405	12.131	65.922	22.377
Application	33.987	80.758	31.353	77.498
Media (Other)	14.572	5.914	2.712	0.044
Other	0.036	1.1963	0.013	0.081
total	100 (%)	100 (%)	100 (%)	100 (%)

Table 4: Classification of HTTP Traffic. *While popular media services such as Netflix and YouTube dominated the traffic volume, the applications were responsible for more than 80% of iOS and 77% of Android HTTP flows.*

In table 4, we observe that with a combination of *User-Agent* and *Host* field in HTTP headers, we were able to classify more than 98% of the traffic in terms of flows and bytes from iOS and Android devices. We observe that media from popular hosts in our *mobWild* dataset, including Netflix, YouTube, Pandora, Spotify, and Vimeo, contribute to more than 50% of the traffic volume from iOS and Android devices. Similarly, we observe that applications identified based on the *User-Agent* were responsible for more than 77% of flows from Android and iOS devices. We also observe that media (identified based on the signatures such as *Apple Core Media* and *Stagefright* served from CDNs and others hosts from which we could not identify the webservice from other fields in the HTTP header and the DNS responses before the HTTP flows to be 14.5% of the traffic volume for the iOS devices in our dataset.

4.2.2 Classification of SSL Traffic.

Unlike HTTP flows, SSL flows provide limited information in plaintext that can be used to identify the applications. For the traces captured during our controlled experiments, we were able to observe HTTP requests and responses after decrypting with SSL bumping. We can classify such flows using the techniques described in the previous section. However, we did not perform SSL bumping for the devices in the *mobWild* dataset, so we now describe how to classify SSL flows without decryption. As we show below, we used the port number, the SSL certificate with server name identification, and DNS queries to identify the source of SSL traffic.

Mobile devices use SSL for various services including mail, notifications, instant messaging, and web browsing. Services such as mail, instant messaging, and notifications are documented to use dedicated port numbers of their traffic. On using port numbers, we observe in that more than 99% of the SSL flows observed in our controlled experiments were due to HTTPS, the rest of the flows were due to email, instant messaging, and OS notification services. We therefore focus our attention on identifying the Web-services responsible for the HTTPs flows.

Classification techniques. We first use the common name (CN) field of certificates to identify the servers that exchanged data using HTTPS. We observe that less than 25% of the HTTPS traffic from iOS and Android contains the fully qualified domain name (FQDN) in the subject of the certificate; the rest of the traffic either contains regular expressions such as *.google.com in the certificate. To further resolve the hostnames, we rely on *server name indication* used by SSL flows [9]. Servers that host multiple services use the *server name indication* to distinguish these services. For example, we observe a *server name indication* of *plus.google.com* and *s.youtube.com* in two flows that used a certificate with a CN **.google.com*. However, we observe that by using either the certificate or the *server name* we were able to identify the name of the Web-service in less than 40% of iOS and Android HTTPS traffic.

DNS mapping. For such flows we use DNS requests made by the mobile devices before starting the HTTPS flows, a technique similar to DN-Hunter [7]. DN-Hunter relies on the most recent FQDN that corresponds to the IP address, however in our controlled experiments we observe Android and iOS devices use the first entry in DNS response while resolving *hostnames*. We therefore use the latest DNS response that contains the IP address of the webservice in the first position of the DNS response. Indeed, for more than 99% of the iOS and Android HTTPS traffic that we could not classify using other fields, we observe that the latest DNS response before the flow started contained the IP address of the webservice as the first entry in the DNS response. Despite the potential usefulness of DNS responses, we give a high priority to the server-name and the certificates because we observed that for flows that contained the server name did not contain the same name in the DNS response for 9.2% of the iOS traffic and 5.6% of Android traffic.

Classification results. In table 5 we present the top five hostnames according to traffic volume – responsible for 66% of iOS and 54% of Android traffic by volume in the *mobWild* dataset. We observe that even when a hostname is specified we cannot uniquely uniquely identify the Web service for some flows. For example, *www.google-apis.com* and *clients4.google.com* offer limited information on the application or Web service that is responsible for the content; at best we can infer these flows belong to some Google service. In contrast, the hostname *fbcdn-photos-a.akamaihd.net* is a strong indication that the traffic is due to Facebook (due to "fbcdn").

iOS	Android
imap.gmail.com	picasaweb.google.com
www.google.com	www.googleapis.com
sphotos-a.xx.fbcdn.net	android.clients.google.com
itunes.apple.com	clients4.google.com
m.google.com	fbcdn-photos-a.akamaihd.net

Table 5: Popular hostnames in observed in SSL flows from iOS and Android based on traffic volume. *Hostnames such as www.googleapis.com hide the underlying application and Web-service.*

Service	% of iOS Traffic		% of Android Traffic	
	Bytes	Flows	Bytes	Flows
Mail	9.970	62.168	1.626	1.565
Social Networking	12.491	6.683	36.661	22.352
App Store	5.457	3.463	0.044	0.036
Instant Messages	0.982	7.089	1.411	3.109
Other Google Services	58.665	13.32510	45.024	46.089
total (%)	87.564	92.728	84.776	73.151

Table 6: Classification of *mobWild* SSL traffic based on names in certificate, server name identification, and DNS request.

In table 6 we present our SSL classification results. We observe that the iOS devices in our dataset generated a significant number of e-mail flows. Similarly, we were able to group 12.5% of iOS and 36.7% of Android traffic with social network services that includes *Google Plus, Facebook, and Twitter*. We speculate the increase in traffic share for Android devices is because Android devices offer services to backup photos on *Google Plus*. Similarly, we observe that 5.4% of the traffic from iOS devices was from Apple stores while we observe only 0.04% of traffic to the *Google Play* store. This low share is because Google can use hosts matching the pattern *client*.google.com* to serve different Web services. We observed a similar behavior in our controlled experiments, and we group such traffic as *other Google services*. Indeed, in table 6 we observe that Google is the largest target of SSL traffic for iOS and Android devices in our dataset. The next most popular target for SSL traffic is social networking sites.

Having developed techniques to map application flows to apps, we use this classification to study the apps responsible for leaking private information.

5. PRIVACY INVASIVE SERVICES

Privacy has rapidly become a critical issue for networked services, as large numbers of organizations develop extensive tracking infrastructure to gather information from users for ads and analytics. While the problem is well known and has been extensively studied in previous work [23, 16, 26], in this section we explore the potential to identify personals identifiable information (PII) exclusively from passively gathered app traces. Our key contributions are 1) we conduct controlled experiments to identify how PII is being leaked by apps via their network flows and 2) use SSL bumping enabled by *MobiScope* to understand how this information is being shared over secure channels (in addition to those revealed in the clear). For our analysis we focus on *what* PII is sent, *to whom* is the PII sent, and *how frequently* is PII sent.

We evaluate PII in terms of two threat models. First, we assume a passive eavesdropper that can sniff traffic over open WiFi APs or within a cellular provider. Any PII sent unencrypted is available to the attacker. Second, we assume an attack conducted by (or on) a third-party service that is able to identify a user's fine-grained locations over time.

5.1 Controlled experiments

For our experiments, we created fake user accounts with fake contact information, and fake Twitter and Facebook accounts. Our goal is to detect if any PII—email address, phone number, IMEI number—stored on the device is leaked across the network over HTTP or HTTPS (using the SSL bumping plugin). Some of this information is required for normal app operation, however; we strongly believe that such information should never travel across the network in plaintext (HTTP).

In table 7, we present the different PII leaked for both Android and iPhone apps. We observe that the IMEI, a unique identifier tied to a phone, is the most commonly leaked PII for Android applications. This IMEI can be used to track and correlate a user's behavior across Web services. Similarly, we observe that Android applications leak the Android ID, a unique identifier tied to an Android device. In table 7, we also observe that other information like contacts, emails, and passwords are leaked in the clear. The email address, the address used to sign up for the services, was leaked in the clear by 13 iOS and 3 Android applications from our set of popular applications.

While only one Android app (belonging to the *Photography* category) leaked a password in the clear, we were surprised to learn that six of the most popular iOS apps send user credentials in the clear, *including the password*. Particularly disconcerting is our observation that an app in the Medicine category – which the provider claims has "*1 million active members of which 50% are US physicians*" – sends the user's first name, last name, email, password, and zip code in the clear. Given US physician access to highly sensitive data like medical records, we believe it is particularly important for this app to protect user credentials (which are often used for multiple services).

During our experiments, we observed that PII information is also sent over HTTPS. In the following, we focus on device identifiers such as the IMEI and the Android device ID. In table 8, we present the top 10 sites ordered by the number of flows that sent the IMEI over HTTPS. We observe that four of the top 10 sites that receive this information are ads and analytics (A&A) sites.

Our observations highlight the limitations of current mobile OSes with respect to controlling access to PII via app permissions. In particular, it is unlikely that users are made aware that they are granting access to PII for A&A sites when embedded in an app that serves a different purpose. This problem is pervasive: of the 77 sites that received either the IMEI or Device ID in the clear or over HTTPS, 35 sites were third party ads and analytics sites.

Host	**IMEI**	**Device ID**	*Ads & Analytics*
chartboost.com	✓	✓	✓
tapjoyads.com	✓	-	✓
getjar.com	✓	✓	-
pocketchange.com	✓	✓	-
iheart.com	✓	✓	-
aarki.net	✓	-	✓
zynga.com	✓	-	-
droidsecurity.appspot.com	✓	-	-
google.com	-	✓	-
flurry.com	-	✓	✓
groupon.com	-	✓	-

Table 8: Top 10 hosts that receive the IMEI or Device ID over HTTPS. *Hosts are ordered by the number of flows that send the IMEI number, followed by the number of flows that send the device ID over HTTPS. Four of the top 10 hosts that receive this information are ads and analytics sites.*

We note that our observations are a conservative estimate of PII leakage. Specifically, we cannot detect PII leakage if the data is obfuscated (*e.g.*via hashing). Regardless, our study showed that a significant amount of PII leaks not only through unobfuscated channels but even unencrypted ones.

5.2 PII in the Wild

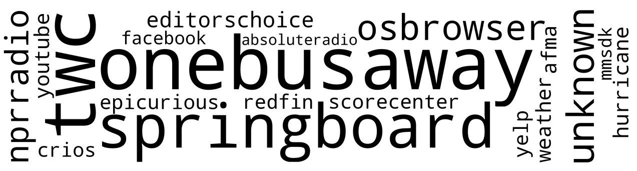
In the previous section, we focused on controlled experiments. We now analyze the *mobWild* dataset. We do not use SSL bumping on this data for privacy reasons. Instead, we focus on information is leaked in the clear.

In figure 4, we present a *word cloud* of the applications that send the location information of the devices. We observe that a bus service application (*One Bus Away*), the application that manages the iOS homescreen (*springboard*), and weather applications (*twc*, *weather*, *hurricane*) were responsible for more 78% of the flows that sent the location information in the clear. Further, SpringBoard (the app responsible for managing the home screen of iOS devices) sends location information in the clear to fetch weather information from Yahoo servers. In figure 4 (b), we observe that Springboard leaked location information for 11 devices in the *mobWild* dataset, a maximum of 14 leaks per day was observed for one device. We also observe that these devices include the iPodtouch device and all the iPhones in *mobWild*, however, we do not observe such leaks for the iPad devices in the *mobWild* dataset.

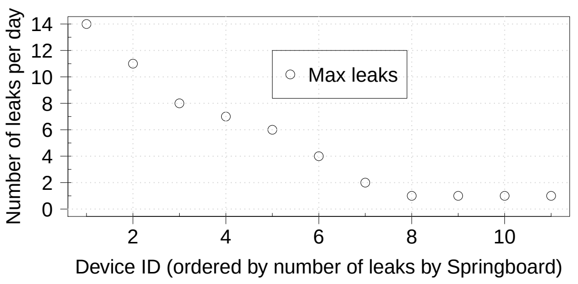
In addition, we observe that the device ID and IMEI number of devices in the *mobWild* dataset are leaked in the clear. Based on our classification methodology, we observe that the IMEI number and device ID is leaked by Web browsers; we do not observe any other app signature in non-browser

Store	Platform	# Apps	Email	Location	Name	Password	Device ID	Contacts	IMEI
App Store	iPhone	209	13 (6.2%)	20 (9.5%)	4 (1.9%)	3 (1.4%)	4 (1.9%)	0 (0%)	0 (0%)
Google Play	Android	100	3 (3%)	10 (10%)	2 (2%)	1 (1%)	21 (21%)	0 (0%)	13 (13%)
Third Party	Android	908	1 (0.1%)	32 (3.5%)	2 (0.2%)	0 (0%)	95 (10.4%)	4 (0.4%)	48 (5.3%)

Table 7: Summary of personally identifiable information leaked in plaintext (HTTP) by Android and iPhone applications. *The popular iOS applications tend to leak the location information in the clear while Android applications leak the IMEI number and Android ID in the clear.*



(a) Applications leaking location in the clear.



(b) Frequency of leaks by Springboard.

Figure 4: Applications that send the location information in the clear. *The font size in the cloud represent the number of flows that sent the location information in the clear. We observe location leaks across different version of SpringBoard and from all the iPhones and the iPodtouch devices in the* mobWild. *dataset.*

flows leaking these IDs in the *mobWild* dataset. As in the case of controlled experiments, A&A sites are the most popular destination for the IMEI number leaks. Among the 16 sites that received the IMEI number in the clear, 10 are A&A sites; the rest of the sites includes sites for games, news, and manufacturer updates.

In Table 9, we present the number of devices in the *mobWild* dataset that contact the most popular A&A sites, an activity that is receiving considerable attention [23, 16, 26]. We observe that all the devices in the *mobWild* dataset contacted doubleclick.com, an ad site, and google-analytics.com, a tracking site. Further, 66.12% of the ads and analytics traffic volume in the *mobWild* dataset was from browsers, 6.46% of the traffic contained a blank user-agent field, and 4.8% of the traffic contained a signature of *Google-Analytics* library. The rest of the traffic contained signatures of other applications such as Facebook, Pandora, and YouTube.

Given the extensive nature of tracking over mobile devices, we developed a tool, *ConVis* that allows users to visualize their devices' tracking, and the applications that facilitate this tracking, with the help of a web-based interfaces similar to that of Mozilla Collusion [18]. *ConVis* provides

Tracker	Number of devices tracked		
	Total	iOS	Android
doubleclick.net	26 *(all)*	15 *(all)*	11 *(all)*
google-analytics.com	26 *(all)*	15 *(all)*	11 *(all)*
googlesyndication.com	22	12	10
admob.com	21	11	10
scorecardresearch.com	21	11	10

Table 9: The top 5 ads and analytics sites that were contacted by the devices in our dataset. *The sites, doubleclick.net and google-analytics.com, were contacted by all the 26 devices in mobWild.*

a visual interface that allows users to specify block lists for connections based on (app, 3rd party) tuples. A demo of our tool is located at `http://goo.gl/A17h9`.

6. RELATED WORK

The network behavior of mobile systems has implications for battery life, data-plan consumption, privacy, security and performance, among others. When attempting to characterize this behavior, researchers face a number of trade-offs: compromising network coverage (limiting the number and type of ISPs measured), portability (limiting the device OSes) and/or deployability (limiting subscriber coverage). *MobiScope* compromises none of these, enabling comprehensive measurements across carriers, devices and access technologies. Table 10 puts our approach in context with previous approaches for measuring the network behavior of mobile systems.

Traces from mobile devices can inform a number of interesting analyses. Previous work uses custom OSes to investigate how devices waste energy [19], network bandwidth and leak private information [11, 15]. Similarly, AppInsight [21] and PiOS [10] can inform app performance through binary instrumentation and/or static analysis. In this work, we explore the opportunity to use network traces alone to reveal these cases without requiring any OS or app modifications.

Network traces from inside carrier networks provide a detailed view for large numbers of subscribes. For example, Vallina-Rodriguez *et al.* [26] uses this approach to characterize performance and the impact of advertising. Gerber *et al.* [14] similarly use this approach to estimate network performance for mobile devices. [17] [8] Similar to these approaches, *MobiScope* provides continuous passive monitoring of mobile network traffic; however, *MobiScope* is the first to do so across all networks to which a device connects.

Last, active measurements [27, 24] allow researchers to

	Network Coverage	Portability	Deployment model	Meas. Type
AT&T/Telefonica study [26, 14]	Single carrier	All OSes	Instrument cell infrastructure	Passive
WiFi study [8]	Single WiFi network	All OSes	Instrument WiFi network	Passive
PhoneLab/TaintDroid [11]	Multiple networks	Android	Install custom OS	Active/Passive
MobiPerf [27]/SpeedTest [24]	Multiple networks	Android	Install App	Active
MobiScope	Any network	Android / iOS	VPN configuration	Passive

Table 10: Comparison of alternative measurement approaches. *MobiScope* is the first approach to cover all access networks and most device OSes, capturing network traffic passively and with low overhead via VPN proxying.

understand network topologies and instantaneous performance at the cost of additional, synthetic traffic for probing. In contrast, *MobiScope* uses passive measurements to characterize the traffic that devices naturally generate.

7. CONCLUSION

This paper described how we use proxying to provide a comprehensive view of Internet traffic generated by mobile devices. The key observation is that most devices already support traffic indirection via VPNs; by using VPN servers as proxying and instrumenting them to record traffic flows, we passively monitor traffic from mobile devices regardless of access technology, device or OS. Using this unique view of mobile-device traffic, we conduct controlled experiments and user studies to characterize and classify apps and to identify PII leakage. As part of future work, we are investigating additional techniques to improve traffic classification and to identify PII. To understand network behavior from a much more diverse set of users and devices, we are deploying a second study that is IRB-approved for a larger set of users without requiring in-person interviews.

8. REFERENCES

[1] AppsApk.com. `http://www.appsapk.com/`.
[2] Browser detection using the user agent. `https://developer.mozilla.org/en-US/docs/Browser_detection_using_the_user_agent`.
[3] Media — Android Developers. `http://source.android.com/devices/media.html`.
[4] Open vswitch: An open virtual switch. `http://openvswitch.org/`.
[5] Squid-in-the-middle SSL Bump. `http://wiki.squid-cache.org/Features/SslBump`.
[6] UI/Application Exerciser Monkey. `https://developer.android.com/tools/help/monkey.html`.
[7] I. N. Bermudez, M. Mellia, M. M. Munafo, R. Keralapura, and A. Nucci. DNS to the Rescue: Discerning Content and Services in a Tangled Web. In *Proc. of IMC*, pages 413–426, New York, NY, USA, 2012. ACM.
[8] X. Chen, R. Jin, K. Suh, B. Wang, and W. Wei. Network performance of smart mobile handhelds in a university campus wifi network. In *Proc. of IMC*, 2012.
[9] D. Eastlake et al. Rfc 6066: Transport layer security (tls) extensions: Extension definitions, 2011.
[10] M. Egele and C. Kruegel. PiOS: Detecting privacy leaks in iOS applications. *Proc. of the NDSS Symposium*, 2011.
[11] 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 *Proc. of USENIX OSDI*, 2010.
[12] J. Erman, A. Gerber, and S. Sen. HTTP in the Home: It is not just about PCs. *ACM SIGCOMM Computer Communication Review*, 41(1):90–95, 2011.
[13] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A First Look at Traffic on Smartphones. *Proc. of IMC*, page 281, 2010.
[14] A. Gerber, J. Pang, O. Spatscheck, and S. Venkataraman. Speed testing without speed tests: estimating achievable download speed from passive measurements. In *Proc. of IMC*, 2010.
[15] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proc. of CCS*, 2011.
[16] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market. In *Proc. of Hotmobile*. ACM, 2012.
[17] G. Maier, F. Schneider, and A. Feldmann. A First Look at Mobile Hand-held Device Traffic. *Proc. PAM*, 2010.
[18] Mozilla collusion. `www.mozilla.org/en-US/collusion/`.
[19] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proc. of Eurosys*, 2012.
[20] V. Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999.
[21] L. Ravindranath and J. Padhye. AppInsight: Mobile App Performance Monitoring in the Wild. *Proc. of USENIX OSDI*, 2012.
[22] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *Proc. of USENIX NSDI*, 2008.
[23] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. *Proc. of USENIX NSDI*, 2012.
[24] J. Sommers and P. Barford. Cell vs. wifi: On the performance of metro area mobile connections. In *Proc. of IMC*, 2012.
[25] Strongswan. `www.strongswan.org`.
[26] N. Vallina-Rodriguez, J. Shah, A. Finamore, H. Haddadi, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft. Breaking for commercials: Characterizing mobile advertising. In *Proc. of IMC*, 2012.
[27] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. of ACM SIGCOMM*, 2011.
[28] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Proc. of IMC*, 2011.