Using the Middle to Meddle with Mobile

Ashwin Rao[a], Arash Molavi Kakhki[b], Abbas Razaghpanah[e], Amy Tang[c], Shen Wang[d], Justine Sherry[c], Phillipa Gill[e], Arvind Krishnamurthy[d], Arnaud Legout[a], Alan Mislove[b], and David Choffnes[b]

[a]INRIA [b]Northeastern Univ. [c]UC Berkeley [d]Univ. of Washington [e]Stony Brook Univ.

Abstract

Researchers and mobile users have little visibility into the network traffic generated by mobile devices and have poor control over how, when, and where that traffic is sent and handled. This paper presents Meddle, a platform that leverages VPNs and software middleboxes to improve transparency and control for Internet traffic from mobile systems. Meddle provides a practical way to interpose on all of a device's Internet traffic, while providing clear incentives, privacy guarantees, and ease of deployment to end users. We discuss the design and implementation of our system, and evaluate its effectiveness with measurements from an IRB-approved user measurement study. We demonstrate the potential of this platform using case studies of applications built atop Meddle; namely, controlling privacy leaks and detecting ISP interference with Internet traffic.

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 networks with opaque policies. As a result, researchers and mobile-device users have little visibility into the network traffic generated by their devices, and have poor control over how, when and where that traffic is sent and handled by third parties.

This situation has negative implications for users: previous studies identified privacy [34], performance [13, 19, 30], policy [35] and security [17] issues in mobile systems. However, each of these studies is limited in terms of visibility or control. For example, passively gathered datasets from large mobile ISPs provide broad visibility but gives researchers no control over network flows (*e.g.,* to experiment with transparent proxies or malware blocking). Likewise, custom Android extensions provide strong control over network flows but measurement visibility is limited to the devices running these custom OSes or apps, often requiring warranty-voiding "jailbreaking."

To understand and address the above problems in mobile systems as they evolve over time, we need an approach that supports long-term studies of mobile Internet traffic and the ability to interpose on these flows. Ideally, such a system would be easy to deploy and use for a typical smartphone user running any operating system anywhere in the world.

This paper presents *Meddle*: a platform for measuring and interposing on all mobile-device Internet traffic (*e.g.,* from smartphones and tablets). *Meddle* combines software middleboxes with VPN proxying, enabling both visibility and control over network flows.

The vast majority of Internet-enabled mobile devices provide the ability to connect to a remote host over a virtual private network (VPN), so *Meddle* is readily deployable and usable. By redirecting all smartphone traffic over a VPN, *Meddle* provides a central vantage point for traffic monitoring. Further, software middleboxes can provide control over flows going to and from each device, and even experiment with network services for mobile devices (*e.g.,* content filtering, malware blocking, Web proxying), quickly, easily and at scale using cloud infrastructure [29]. This paper demonstrates the feasibility of our approach and explores several new opportunities for implementing applications not broadly supported in today's mobile environment.

Meddle provides useful opportunities for both users and researchers. This paper focuses on the research *Meddle* enables by providing *in situ* access to user network flows. Currently researchers with new middlebox approaches to improve the mobile user experience must test them in a lab environment or rely on ISPs or users to deploy new hardware/software — a potentially risky and costly proposition. With *Meddle*, researchers can immediately deploy new software middlebox services that interact with real users' mobile traffic (with user opt in).

To encourage users to install *Meddle*, we currently provide custom network filters (*e.g.,* device-wide ad blocking) and identification/blocking of services leaking personally identifiable information (PII). We are developing additional services as user incentives, including offloading network communication to the cloud and device-wide SPDY proxying.

Our key contributions are as follows. First, we design and implement *Meddle*, a system that provides transparency and control over all Internet traffic generated by their mobile devices. We demonstrate that it is sufficiently transparent to avoid significantly impacting measurement results. *Meddle* captures all Internet traffic with approximately 10% power and data overheads, and low additional latency. We will make the *Meddle* software and configuration details open source and publicly available.

Second, we use *Meddle* to conduct measurement studies that inform a wide range of mobile middlebox applica-

tions: identifying privacy leaks in mobile apps, detecting content manipulation and service differentiation in ISPs, and studying network activity of mobile malware. We analyze network traffic from controlled experiments with more than 1,000 apps (iOS and Android), and from human subjects during a 10-month IRB-approved 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 and connecting to a variety of cellular and Wi-Fi networks. These users interact with networks in 54 ASes, 8 of which are cellular; their traffic strongly depends on OS and network type.

Third, we implement applications atop *Meddle* that improve privacy, block unwanted traffic from ads and malware, and notify users of ISP interference. In particular, we make available a new visualization tool, *ReCon*, for users to track and control how they are being monitored by ad and analytics services. We also develop tools for detecting content manipulation by ISPs (Web Tripnets) and service differentiation (Mobile Replay) in the mobile environment, building upon previous work in these areas [14, 27]. *Meddle* has been running since October, 2012 as part of an IRB-approved study, new users can sign up at `http://meddle.mobi`.

Roadmap. The rest of the paper is organized as follows. We present the goals and implementation for *Meddle* in §2, then evaluate it in terms of overhead and effectiveness as a vantage point for mobile traffic in §3. We use this analysis to inform the design and implementation of several applications built atop *Meddle*. In §4, we describe how we detect and block privacy leaks in mobile traffic. §5 discusses how we use *Meddle* to detect ISP interference with network traffic. In §6 we discuss related work and we conclude in §7.

2. MEDDLE OVERVIEW

In this section, we present an overview of *Meddle*. We first describe the goals of the system, then discuss the *Meddle* architecture and implementation.

2.1 Goals

The goals of *Meddle* are to provide visibility into Internet traffic from mobile devices, exert control over this traffic and facilitate a large-scale deployment across multiple networks, OSes and devices (smartphones and tablets). We discuss each of these in turn.

Visibility. We aim to capture all of a mobile-device's Internet traffic, allowing us to characterize network flows and interpose on them using software middleboxes. To achieve this, we need a solution that works continuously, regardless of the mobile OS, access technology, or apps installed.

Control. Another important goal is to facilitate research into new middlebox applications for mobile traffic. It should present a simple API and policy framework for researchers and developers to block, shape, inject or otherwise modify network flows matching various criteria. It should also support applications that operate on collections of flows over time and across users (*e.g.,* Web caching/prefetching, malware detection/blocking and ISP characterization).

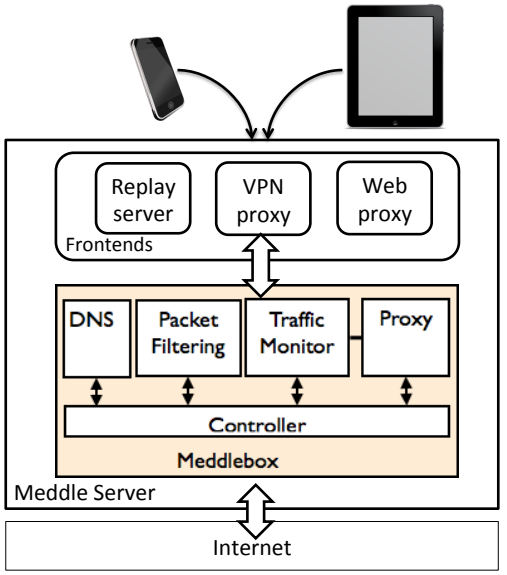


Figure 1: ***Meddle* Architecture**. *Mobile devices (top) communicate with a* Meddle *frontend (VPN proxy, Web proxy and/or traffic replay server). VPN proxy traffic is forwarded to a* meddlebox*, which provides software middlebox services that measure and/or interpose on network flows before relaying the traffic to the Internet.*

Deployability. The system must support the ability to deploy and distribute large numbers of middlebox services quickly, easily, and at scale [29] without the need to deploy hardware in homes [32] or ISPs [35]. To improve the representativeness of studies using *Meddle*, we want to recruit average (*i.e.,* non-technical) users to participate. To support this, we need a system that has incentives for users, a low barrier to adoption, is easy to deploy and that scales gracefully.

2.2 Architecture & Implementation

To achieve our goal of visibility, *Meddle* uses a VPN to direct all of a participating mobile device's Internet traffic to a proxy server (top of the Fig. 1, described in §2.2.1). To achieve our goal of controlling mobile-device traffic, the *Meddle* proxy directs traffic to a software middlebox, called a *meddlebox* (middle of Fig. 1, described in §2.2.2), that can record, block, modify and/or otherwise interact with mobile-device flows. Section 2.2.3 describes how we designed *Meddle* to have clear incentives for user adoption and a low barrier to entry to participate.

2.2.1 Visibility: VPN proxying

Meddle leverages the fact that the vast majority of mobile devices provide native VPN support, a feature typically provided to satisfy enterprise clients. We currently support VPN tunnels on iOS and Android; we anticipate being able to support the next version of Windows Phone that includes a native VPN implementation.

iPhone support. 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. This feature is originally intended to allow enterprises to ensure their employees' devices always establish a VPN connection before contacting a specified set of domains. 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). To setup this configuration, users need to install a single file, a step that needs to be performed only once.

Android support. As of Android 4.2, Android supports "always on" VPN connections that ensure all traffic is always tunneled. For Android version 4.0+ we use a modified StrongSwan implementation of a VPN client app that manages VPN tunnels. Our version ensures re-establishment of VPN tunnels on network state changes (*e.g.,* when a device switches from cellular to Wi-Fi).

Server-side implementation. *Meddle* uses the open-source Strongswan project [31] to manage the VPN tunnels on its servers. When traffic arrives at the server, *Meddle* uses *tcpdump* and *bro* to record traffic via two IRB-approved approaches. The first captures full packets, and requires subjects to be interviewed and provide written informed consent before participating. We used this study to inform the second approach, which captures only relevant packet headers necessary for the *Meddle* applications in this paper.

As we described in the previous section, encrypted flows give *Meddle* no visibility into the content of network flows and no ability to interpose on that traffic. To regain this visibility, we use SSL bumping to decrypt and access the plaintext of encrypted flows in the following way.

First, we note that *Meddle*'s VPN server, like all VPN servers, can be configured to use a self-generated root certificate 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 [7] feature. When the mobile device connects to a service supporting SSL, the proxy masquerades as the service using a forged certificate signed with the *Meddle* root certificate. Then the proxy establishes an SSL connection with the intended target, impersonating a mobile device. Using the traffic captured on *Meddle* and the private key generated by the squid proxy to communicate with the mobile device, we can decrypt all SSL traffic.

This approach fails for apps that do not trust certificates signed by unknown root authorities. For example, in our controlled experiments we observed that Firefox prevents SSL bumping by validating root certificates, while 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.

2.2.2 Control: Software Middleboxes

Once traffic arrives at a *meddlebox*, it interacts with software middleboxes that interpose on user-generated traffic and proxy services that interact with untunneled flows.

Plugin infrastructure. *Meddle* 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 arrives at the VPN proxy, *Meddle* forwards it to a software-defined switch [6] that determines the ordered set of plugins that the corresponding flow 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, *Meddle* forwards the network flow to the Internet. The same approach applies to traffic from the Internet destined for *Meddle* subscribers.

Plugins support a variety of features such as ad blocking, analyzing PII leakage or page speed optimization. Additionally, we have implemented per-connection blocking, malware analysis and DNS-based packet filters.

Frontend services. As we describe in §5, *Meddle* supports active measurements. Specifically, we inject JavaScript into Web pages to detect if the device's access network is modifying Web content in flight, and we use a companion app to test detect service differentiation within mobile ISPs. To support these, we run services on *Meddle* that are accessed via untunneled connections.

2.2.3 Deployability: Incentives and ease of use

Incentives for user adoption. *Meddle* presents a number of incentives that appeal to a wide range of users. Importantly, we do not charge users for any of these services.

Improved security. By securely tunneling all of a mobile device's traffic, users are less vulnerable to data leakage (*e.g.,* via open WiFi hotspots).

Device-wide content filters. We use *Meddle* to block content that users do not wish to access – for all apps running on a device. The most popular instance of this is device-wide ad blocking, implemented using a DNS server that returns *localhost* for requests to names for known ad servers.

Privacy revelations [36]. The *ReCon* tool (§4) allows users to see how they are being tracked by ad and analytic servers, and allows them to customize the list of trackers to block.

ISP transparency. We provide services that allow users to identify cases where ISPs are modifying HTTP content in flight, and when they provide differentiated service to traffic crossing their networks (§5).

Low barrier to entry. Configuring a VPN generally requires filling out five fields on an Android phone, and the VPN configuration can be distributed using a single file on iOS. We are hosting a cloud-based deployment that is free for users, to support large numbers of flows interacting with researcher's *meddlebox*es. We will make the *Meddle* software publicly available.

For those who want to run their own *Meddle* instance, the *Meddle* server requires only that a user can run a mod-

ern Linux OS with root privileges. This can be deployed in a single-machine instance on a home computer, dedicated server or on a VM in the cloud. *Meddle* is currently in private beta with dedicated-server, EC2 and Aliyun deployments in the US, France and China.

2.3 Discussion

Limitations. The following can impact *Meddle*'s coverage.
Trust. This paper focuses on a cloud-based *Meddle* deployment, which requires that users trust our system with their network data. We will make our code open source to build this trust. For particularly paranoid users, we will also provide a stand-alone implementation that users can run on a server of their choice.
At most one VPN tunnel. Currently iOS and Android support exactly one VPN connection at a time. This allows *Meddle* to measure traffic over either Wi-Fi or cellular interfaces, but not both at once. The vast majority of traffic uses only one of these interfaces, and VPNs can be used to tunnel that traffic.
Proxy location. When traffic traverses *Meddle*, destinations will see the *Meddle* 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 *Meddle* instance with an appropriate IP address.
ISP support. Some ISPs block VPN traffic, which prevents access to our current *Meddle* implementation. We note that few ISPs block VPN traffic, and there is an incentive not to block VPN traffic to support enterprise clients. We also note that China blocks VPN access to all but domestic destinations; thus, we use a Chinese cloud host (Aliyun) to run *Meddle* in China.
IPv6. Meddle cannot be currently used on networks using IPv6; though iOS, Android and StrongSwan support IPv6 the mobile OSes currently do not support IPv6 traffic through VPN tunnels.
Privacy. Our IRB-approved user study reports data from capturing all of a subject's Internet traffic, which raises significant privacy concerns. The study protocol 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 *Meddle* 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.

Our other IRB-approved *Meddle* study uses the same protocol except for (1) only packet headers are captured, thus reducing the privacy risks, and (2) subjects provide electronic informed consent, thus facilitating user adoption worldwide. Subjects can sign up at `http://meddle.mobi`.

Generalizability. A system for improving visibility and control over Internet traffic from mobile devices can be implemented at the endpoints (*e.g.,* in the OS), in the network (*e.g.,* a hardware middlebox), somewhere else in the middle (our approach). *Meddle* is not intended as a blanket replacement for these alternative approaches; rather it allows us to explore the opportunities for improving the state of the art in today's mobile systems by making mobile Internet traffic available to researchers.

In some cases, *Meddle* is the right location to implement new services that could be costly to deploy on devices (prefetching) or impractical to deploy in network (detecting ISP service differentiation). In other cases, *Meddle* provides a practical partial solution to a problem where the complete solution has an impractical cost. For example, identifying privacy leaks from mobile devices is reliably addressed using information flow analysis [17]. However, due to the overheads of this approach it is difficult to deploy to users and at scale. *Meddle* allows us to identify and block unobfuscated PII in network flows from arbitrary devices without requiring OS modifications or taint tracking. Regardless of the ultimate optimal solution, we can use *Meddle* today to inform the design and deployment of future functionality in OSes and for in-network devices.

3. EVALUATION

This section evaluates *Meddle* in terms of overhead, visibility into network traffic and the ability to map network traffic to the apps that generated it. We use the results in this section to inform the applications we build in §4 and §5.

3.1 Methodology

Using *Meddle*, 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 of 1) controlled experiments in a lab setting and 2) IRB-approved "in the wild" measurements gathered from real users during seven months.

3.1.1 Controlled Experiments with Apps

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 apps in a lab setting. We use this data to understand how to model apps' network behavior, 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 two Android devices (running Android 4.0 and 4.2) and an iPhone 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 *Meddle* and begin the experiment.

SSL bumping. We use SSL bumping only in controlled experiments where *no user traffic is intercepted*. We are also designing a study for IRB approval in which users can opt in to use SSL bumping for obfuscating PIIs in their SSL traffic.
Manual tests. We manually test the 100 most popular free Android apps in the *Google Play* store and 209 iOS apps from the iOS App store on April 4, 2013. For each app, we install it, enter user credentials if relevant, interact with it for up to 10 minutes, and uninstall it. This allows us to characterize real user interactions with popular apps in a controlled environment. We enter unique and distinguishable user credentials when interacting with apps to easily extract the corresponding PII from network flows (if they are not obfuscated). We use the same technique to test malware apps (§4.3).
Automated tests. The second set of controlled experiments consist of fully-automated experiments on 732 Android apps from a free, third-party Android market, *AppsApk.com* [2]. We perform this test because Android users can install *Third-party apps* without rooting their 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 *Meddle* platform, and start the app. Then we use *Monkey* [9], an app-scripting tool, to perform a series of approximately 100,000 actions that include random swipes, touches, and text entries. Finally, we use adb to uninstall the app 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.

3.1.2 In Situ Study

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. 15, 2012 to Sep. 1, 2013.[1]

Our measurement data was collected from 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 *mobUser*, consists of 318 days with data; the number of days for each user varies from 5 to 315 with a median of 35 days. For privacy reasons, the SSL-Bumping plugin is *disabled* for all measurements involving real users.

3.2 Overheads

Network Latency. We first test indirection overhead from mobile networks to a *Meddle* instance. In the US with EC2, delays from mobile-network egress points to EC2 nodes are on generally less than 10 ms. For other networks, we will achieve similarly low indirection overhead by placing instances in a cloud/hosting provider near subscribers and use DNS redirection (*e.g.,* via Amazon's Route 53) to direct clients to nearby instances.

The other source of latency is connection establishment time, incurred once per session. 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. We conduct tests in rapid succession to ensure the radio is in the high power state. The *Meddle* server was running on a university network. For Android (using IKEv2), the maximum establishment time was 0.81 seconds on Wi-Fi and 1.59 seconds on cellular. For iOS, the connection is slower because it uses IKEv1: 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.
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, we relied on the battery readings provided by iOS because we cannot attach a power meter directly to the battery. We again found an approximately 10% power overhead of using VPNs when we drained a fully charged battery while performing the operations performed during the tests for Android devices.
Traffic Volume. Meddle relies on 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 that to compare encapsulated and raw packet sizes. We observe a maximum encapsulation overhead of 12.8% (average approximately 10%). For users that have limited data plans and consume most of their quota per month, this can have a significant impact. We note that this is partially offset by *Meddle* services such as content filtering and connection blocking.
Scalability. We currently use Amazon EC2 to support users at our cost. Without exploring opportunities for economies of scale, we estimate that it will cost less than a penny ($0.0084) per user per day. At this cost, we can support up to 10,000 users with research funds. If *Meddle* were to become extraordinarily popular, it would cost each user approximately a quarter per month to pay their own way. By comparison, data plans in the US tend to cost $30-$90 per month – more than two orders of magnitude larger.

3.3 Meddle Visiblity

[1]The measurement study is ongoing, we report a subset of results.

We now use data gathered from users to motivate the need for a platform like *Meddle* that provides a comprehensive view of Internet traffic from mobile devices. Note that due to the relatively small number of users in our study, we do not attempt to draw strong and generalizable conclusions.

Observation 1: ***End-host instrumentation provides a more complete view of Internet traffic from mobile devices.*** We infer the access technology (WiFi or cellular) for each session using *WHOIS* data for each IP address used by a mobile device. Based on this classification, the *mobUser* dataset consists of traffic from 65 distinct ASes, of which 8 are cellular ASes and 7 are university networks.

We observe less diversity in cellular ASes compared to Wi-Fi ASes. During the measurement study, each device connected to our *Meddle* 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 36 different Wi-Fi ASes spread across 5 countries. In terms of traffic volumes, collectively our users with cellular connectivity transferred 24-56% of their traffic over cellular and the remainder over WiFi. The key take-away is that, for the users in the *mobUser* dataset, we would miss a large fraction of traffic generated by the mobile devices by instrumenting a single cellular carrier or WiFi access point. *Meddle* does not have this limitation.

Observation 2: Meddle *provides visibility into a wide range of traffic patterns*. We use the classification provided by Bro [25] to categorize flows as either TCP, UDP, or *other*, along with subcategories HTTP, SSL and DNS. Table 1 summarizes the traffic generated by user devices in our study.

There are three key take-aways from this table. First, Web and SSL traffic dominate the traffic for users in the *mobUser* dataset; 91.26% (137.63 GB) of the traffic volume in the *mobUser* dataset is either HTTP or SSL. Second, there is significant diversity in the usage patterns for users with Android and iOS devices; the fraction of total flows over cellular or Wi-Fi differ significantly for each OS. Third, a platform that cannot analyze SSL traffic will miss a large fraction of the traffic. A significant fraction of flows use SSL, which prevents classification using deep packet inspection. This motivates the need for a platform that not only covers multiple OSes and multiple access technologies but is also capable of intercepting all mobile Internet traffic, including SSL traffic, for the purpose of analysis and interposition.

3.4 Mapping Network Flows to Apps

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. The following sections show that *previous approaches to mapping passively gathered traffic fail to identify apps responsible for that traffic most of the time* and that *Meddle* facilitates a first look at determining which apps generate traffic over SSL connections.

Table 1 suggests that apps, OS services, and libraries often rely on HTTP and SSL to exchange data. 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 *mobUser* dataset.

IP Protocol	Service	Android		iOS	
		Cell.	Wi-Fi	Cell.	Wi-Fi
TCP	HTTP (%)	44.83	68.23	60.07	76.92
	SSL (%)	44.74	20.89	36.19	14.11
	other (%)	8.26	10.10	2.74	1.33
UDP	DNS (%)	1.31	0.58	0.64	0.38
	other (%)	0.54	0.11	0.31	7.24
Other	other (%)	0.32	0.09	0.05	0.02
total (%)		100.00	100.00	100.00	100.00
Traffic Volume (GB)		9.57	21.10	16.61	103.52
# Flows		927660	761735	730209	2796130

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.*

OS	Store	Apps	Gen.	Host		User-	Combi-
			HTTP	App.	Org.	Agent	nation
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	732	365	17 (4.6%)	79 (21.6%)	52 (14.2%)	83 (22.7%)

Table 2: **Classification of apps based on** *Host* **and** *User-Agent*. *Most iOS apps use dedicated User-Agent strings to fetch data over HTTP. A combination of User-Agent and Host identifies the majority of Android and iOS apps.*

3.4.1 Improving HTTP Traffic Classification

In *Meddle*, we need to know which app is responsible for Internet traffic using only network flow information. This section shows how to use *User-Agent* and *Host* fields to identify the apps and services responsible for HTTP flows. Previous work [22, 37] is insufficient – they use HTTP header fields to identify the *category* of corresponding apps, not the specific app.

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.

Classifying with Host: First, we note that 176 of the 209 iOS apps we manually tested generated HTTP traffic. Column 5 of Tab. 2 shows that the *Host* field uniquely identified the corresponding app for 47% of the iOS apps. Each app generated multiple flows, some of which did not contain the

app signature in the *Host* field, *e.g.,* when contacting ad sites or CDNs. Such flows comprised 2% to 85% of the traffic volume from the iOS apps used during our measurements. The *Host* field also can identify the provider that released an app. For example, we observed the name *Zynga* in the *Host* field when using *Farmville*, an app created by Zynga. When testing an app, we noted down the name of its creator as the organization, and we searched this name in the *Host* field in the HTTP flows generated by this app. In column 6 of Table 2, we see that classification by organization is effective for 67% of iOS apps.

We observe similar results for flows from apps in Google Play. However, for the apps 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 apps we tested (about 77%) were stand-alone services such as games. These apps contacted advertisement or CDN sites that do not uniquely identify the app. Along with the organizations of the apps we tested, we used the Google Play API [4] to extract the names of the creators (organizations) for the 5000 most popular Android apps on the Google Play store.

Classifying with User-Agent: 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 app identifier and other auxiliary information such as details of the OS. For example, Yahoo Mail's *User-Agent* string contains the string *YahooMobileMail/1.0*. However, some apps use more generic *User-Agent* strings such as *AppleCoreMedia* (streaming video on iOS) or *Dalvik* (generic text for Android). To extract the app information, we use regular expressions to filter the auxiliary information from the *User-Agent* and cluster the extracted tokens using the edit distance.

Table 2 shows that 84.6% of the 176 iOS apps generating HTTP traffic were correctly identified by their *User-Agent*, which we verified by manual inspection. In contrast, the *User-Agent* was useful in identifying only 23% of the Android apps generating HTTP traffic, meaning previous techniques depending solely on the *User-Agent* will fail [22, 37]. For the 27 iOS apps which we failed to identify, we observed signatures for OS services and libraries. Similarly, the majority of Android HTTP traffic contained flows with the default *User-Agent* (*e.g., Dalvik*).

Combination of User-Agent and Host: In Table 2, we observe that the *User-Agent* is more effective for mapping iOS apps while the *Host* is more effective for Android apps; however, neither alone is a complete solution. We therefore rely on a combination of *User-Agent* and *Host* to classify HTTP traffic. For our classification, we first try to classify the HTTP flow using the *User-Agent*. We use the *Host* field only if we were unable to extract any useful signature from the *User-Agent* field. In Table 2, we observe that by using a combination of the *User-Agent* and *Host* we were able to identify 64% of the Android apps and 89% of the iOS apps.

In situ data. Table 3 shows that a combination of *User-Agent* and *Host* field maps more than 92% of the traffic (flows and bytes) from iOS and Android devices. Using only the *User-Agent* on the *mobUser* dataset, we were able to identify 256 iOS and 86 Android apps, OS libraries, and services. We observe that the *User-Agent* is more effective in identifying iOS apps compared to Android apps, which agrees with what we observed in controlled experiments.

Technique	Category	iOS		Android	
		Bytes (%)	Flows (%)	Bytes (%)	Flows (%)
User-Agent	Apps	43.21	85.73	15.01	75.17
	OS Services*	0.19	3.82	17.42	0.81
User-Agent + *Host*	Media (Popular)	51.36	7.12	61.98	3.56
	Media (Other)	4.90	0.85	0.68	0.12
Host	Other Apps/Web-services	<0.01	0.49	1.53	12.98
Total Classified		**99.6**	**98.01**	**96.62**	**92.64**

Table 3: **Effectiveness of mapping HTTP traffic**. *OS services* includes services other than those used to download media content.*

Audio and video streaming apps such as Pandora and YouTube use the *Apple Core Media* and *Stagefright* services on iOS and Android respectively to download media content, and for other auxiliary content, such as list of related videos and recommendations, these apps use the *User-Agent* that contains their app signature. We therefore use a combination of *User-Agent* and *Host* field to identify such apps.

In Table 3, we observe that media from popular streaming services—Netflix, YouTube, Pandora, Spotify, and Vimeo—contribute to more than 50% of the traffic volume from iOS and Android devices in the *mobUser* dataset. We also observe that unmapped media served from CDNs and others hosts comprises less than 5% of the traffic volume for the iOS and Android devices. We also observe that the *Host* field is more useful to classify Android traffic compared to iOS traffic, which concurs with what we observed during our controlled experiments.

To summarize, by using a combination of *User-Agent* and *Host* we were able to classify more than 92% of the iOS and Android traffic by flows and bytes.

3.4.2 SSL Traffic Mapping without Decryption

SSL flows provide limited information in plaintext to identify apps. For the traces captured during our controlled experiments, we use SSL bumping to map HTTP flows using the techniques described in the previous section. However, we did not perform SSL bumping for the devices in the *mobUser* dataset, so we now describe how to map SSL flows *without decryption*.

Overview of mapping technique. Using port numbers, we observe that more than 98% of SSL flows 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

App/Org	iOS Traffic		Android Traffic	
	Bytes (%)	Flows (%)	Bytes (%)	Flows (%)
Apps	28.25	48.79	47.74	40.97
Google Services	36.32	17.56	47.31	48.27
Apple Services	25.26	28.26	<0.01	<0.01
Total	**89.83**	**94.61**	**96.10**	**89.24**

Table 4: **Mapping SSL traffic in the *mobUser* dataset.** *The SSL traffic from the iOS and Android devices in the* mobUser *dataset is dominated by Google and Apple services.*

apps responsible for the HTTPS flows. We use DNS responses and subsequent SSL handshakes to determine the *hostnames* of the remote hosts contacted by mobile devices. After identifying hostnames, we map them to apps using the technique described in the previous section.

Identifying hostname using the SSL Handshake. We first use the common name (CN) field of certificates to identify the servers that exchanged data using HTTPS. 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` or is a continuation of a previous SSL session. To further resolve the hostnames, we rely on the *Server Name Indication* (SNI) used by SSL flows [15]. Servers that host multiple services use the SNI to distinguish these services. For example, we observe an SNI of `plus.google.com` and `s.youtube.com` in two flows that used a certificate with a CN `*.google.com`. Using either the certificate or the SNI we identified the hostname for less than 40% of HTTPS traffic.

Identifying hostname using the DNS messages. For the remaining flows we use DNS messages exchanged by the mobile device with its DNS server before starting the HTTPS flows, a technique similar to DN-Hunter [11]. 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 Web service in the first position. In spite of the potential usefulness of DNS responses, we give a high priority to the SNI and the certificates because the DNS response differs from these in 9.2% of the iOS traffic and 5.6% of Android traffic. This difference is due to caching of DNS responses by the apps.

Mapping results. Table 4 shows our SSL mapping results on the SSL traffic in the *mobUser* dataset. We first group hostnames to the apps and we were able to identify the apps for more than 40% of the iOS and Android SSL flows. For flows whose hostnames are ambiguous, we group them according to organizations. During manual examination of the results, we observe that Google and Apple to be the two main organizations that contributed to the majority of the flows; we label these flows as Google Services and Apple Services.

In Table 4, we observe that 61.5% of iOS and 47.3% of Android traffic (by bytes) is respectively to Google and Apple servers where the hostname does not contain signatures of the app. This share does not include the traffic to Google and Apple servers that we classified as apps. For example, flows to `mail.google.com` were classified as GMail and are placed in the category apps, while flows to `www.googleapis.com` is categorized as Google Services. Google services and Apple services are therefore the largest sources of SSL traffic in our *mobUser* dataset.

In summary, using the certificates, SNI, and DNS messages, we were able to identify the hostname of the remote hosts for more than 89% of the SSL flows. We observe that Google and Apple are the dominant sources of SSL traffic for the Android and iOS devices in the *mobUser* dataset.

3.4.3 Summary

We use the a combination of *User-Agent* and *Host* field to identify apps responsible for HTTP flows. On applying our technique to the *mobUser* dataset, we were able to classify more than 92% of the iOS and Android traffic by flows and bytes. We observe that the *User-Agent* field is more effective to identify HTTP flows from iOS devices compared to Android devices. We speculate that this behavior is because of the strict coding practices mandated by Apple while packaging iOS apps [3].

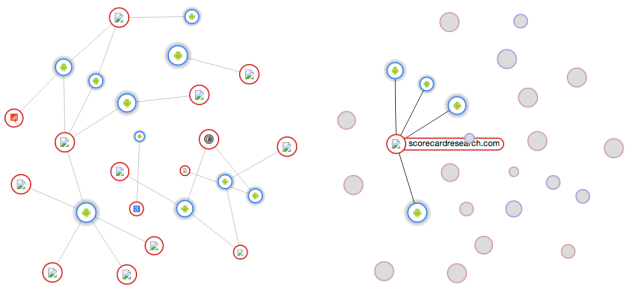
We use certificates, SNI, and DNS messages to map SSL flows, and we were able to classify more than 90% of SSL traffic in the *mobUser* dataset using our classification technique. To the best of our knowledge, we are the first to study the effectiveness of these fields in classifying SSL flows from mobile devices.

4. APPLICATION: MOBILE PRIVACY REVELATIONS

Privacy has rapidly become a critical issue for user interactions with Internet services, particularly in mobile environment where location, contact information and other PII are readily available. While the problem is well known [21, 28, 34], previous work lacks a general way to identify leaks using network flows alone, and they provide no portable way to block those activities. In this section, we use *Meddle* to identify and block these leaks. First, we describe *ReCon*, a tool that allows users to visualize and block privacy-invasive connections. Then we describe how we populate this tool with information about privacy leaks, using controlled experiments to identify how PII is being leaked by apps both in plaintext and over secure channels.

4.1 Revealing and Controlling PII Leaks

ReCon is a *Meddle* application for visualizing how users are being tracked as they use their devices, and for blocking unwanted connections. It provides a visual interface similar to Mozilla Collusion [23]; instead of visualizing only websites, our tool shows apps and the third party sites (trackers) they contact. We identify trackers using a publicly available database of tracker domains [1]; we augment this list with



(a) Default view.

(b) View showing tracking by Scorecard Research.

Figure 2: **Screen captures of the *ReCon* tool**, *allowing users to visualize how they are being tracked by apps, and install custom filters by clicking on links to "remove" them.*

the domains that leaked PII during our controlled experiments, discussed later in this section, and recent research on mobile ads [20, 21]. Figure 2 shows screen captures of our tool, with the default view of trackers contacted by apps (left) and how hovering/tapping on a tracker reveals the apps contacting the tracker (right). Similarly, hovering/tapping on an app highlights all the trackers contacted by the app.

In addition to revealing privacy leaks, *ReCon* allows users to block them. Specifically, users can click/tap on a link to "destroy" it – installing a custom (device specific) filter for the corresponding tracker. To make this approach practical, we are developing an interface to crowdsource block lists, much like how Web browser ad-blocking filters are maintained and distributed. A demo of our tool is located at `http://goo.gl/pU689F`.

4.2 PII Leaked from Popular Apps

We use the traffic traces from our controlled experiments to identify how apps leak PII. For our controlled experiments, we created dummy user accounts with fake contact information (see §3.1.1). Our goal is to detect if any PII stored on the device is leaked over HTTP/S. For our analysis we focus on the email address, location, username and password used during authentication, device ID, contact information, and the IMEI number. Some of this information is required for normal app operation; however, such information should never travel across the network in plaintext.

PII leaks in the clear. Table 5 presents PII leaked by Android and iOS apps. The IMEI, a unique identifier tied to a mobile device, and the Android ID (tied an Android installation) are most frequently leaked PII by Android apps. These can be used to track and correlate a user's behavior across Web services. Table 5 shows that other information like contacts, emails, and passwords are also leaked in the clear. The email address used to sign up for the services was leaked in the clear by 13 iOS and 3 Android apps from our set of popular apps. 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 name, email, password, and zip code in the clear. Given US physicians have access to sensitive data like medical records, we believe it is critical for this app to protect user credentials (which are often used for multiple services). Following responsible disclosure, we will notify app authors of these sensitive privacy leaks.

PII leaked from same apps on different OSes. We observed that the information leaked by an app depends on the OS. Of the top hundred apps for iOS and Android, 26 apps are available on both iOS and Android. Of these 26 apps, 17 apps leaked PII on at least one OS: 12 apps leaked PIIs only on Android, 2 apps leaked PII only on iOS, while only one app had the same data leakage in both OSes. Of the remaining two apps that leaked PII, one app leaked the android ID and IMEI in Android and username in iOS, while the other app leaked the Android ID in Android and location in iOS. The difference in the PII leaks is primarily due to the different privileges that the underlying OS provides these apps.

PII leaked over SSL. During our experiments, we observed that PII is also sent over encrypted channels. We observe that two of the top 5 sites that receive PII over SSL are trackers. 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 tracker libraries embedded in an app that serves a different purpose. This problem is pervasive: of the 77 sites that received some PII in the clear or over SSL during our controlled experiments, 35 sites were third party trackers.

We note that our observations are a conservative estimate of PII leakage because we cannot detect PII leakage using obfuscation (*e.g.,* via hashing). Regardless, our study shows that a significant PII leaks are visible from *Meddle*.

4.3 PII Leaked by Malware

Mobile malware is an increasingly important threat in mobile systems. With *Meddle*, we can monitor and detect any malware activity over IP but not circuit switched activity such as sending SMS or making phone calls. This section focuses on PII leaked by malware; we also discuss how we can our results and *ReCon* to provide malware blocking.

Meddle gives us two opportunities to detect and block malware activity over IP. First, we can detect the app binary being downloaded via a hash and block that transfer if it is identified as malware. Note that, *Meddle* can compute hashes only if the app binary is downloaded in the clear and not over a secure channel (unless SSL bumping is enabled). Second, we can use the techniques discussed in §4.2 to identify and block PII leaks by malware.

To understand malware network behavior, we use a dataset consisting of 111 confirmed malicious Android APKs gath-

Store	Platform	# Apps	Email	Location	Name	Password	Device ID	Contacts	IMEI
App Store	iPhone	209	13 (6.2%)	20 (9.5%)	4 (1.9%)	6 (2.87%)	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	732	3 (0.4%)	57 (7.8%)	3 (0.4%)	0 (0%)	85 (11.6%)	6 (0.8%)	39 (5.3%)
Malware	Android	111	1(0.9%)	20 (18.01%)	0 (0.0%)	0 (0%)	20 (18.01%)	9 (8.1%)	68 (61.2%)

Table 5: **Summary of PII leaked in plaintext (HTTP) by Android and iPhone apps.** *The popular iOS apps tend to leak the location information in the clear while Android apps leak the IMEI number and Android ID in the clear.*

Domain Name	# Malware	App Store
cooguo.com	19	✓
umeng.com	16	-
wapx.cn	11	✓
flurry.com	9	-
veegao.com	9	-

Table 6: **Top 5 sites that receive PII, and the number of malwares sending them the PII.** *Two of the top 5 sites that receive PII are third-party app stores.*

ered by the Andrubis project [10] in September, 2013. We use the approach in §3.1.1 to conduct controlled experiments on the malware. The malware consists of 46 families ranging from backdoors to spyware. Of the 111 apps, 99 (89%) apps generated network traffic; 37 of the 111 apps targeted earlier versions (below 4.0) of the Android OS and thus did not work to their full potential during our experiments.

Signature-based detection is insufficient. First, we determine whether existing malware hash registries contain signatures for the malicious apps in our dataset. Even in December 2013, 3 months after the malware was identified by Andrubis, we find that only 9 (8.1%) apps were correctly identified as malware by Cymru's Malware Hash Registry that uses 30 anti-virus software packages [8]; 7 of these 9 apps generated network traffic.

PII leaks by malware. Of the 99 apps generating network traffic, we were able to identify PII leaks from 74 apps. Similar to the Android apps previously examined, we observe that the IMEI, and Android ID are most commonly leaked PII. Of the remaining 25 apps that generated traffic and did not leak PII over HTTP and HTTPs, 17 tried to contact a remote host that was down (did not respond to a TCP SYN) or sent encoded data over HTTP and HTTPs, while 8 apps are known to exploit previous versions of Android.

In Table 6, we present the top 5 sites that receive the PII leaked when we tested the malware. We observe that 2 of the top 5 site are stores from which Android apps can be downloaded, and the most frequently contacted store, cooguo.com, was known to host malware software [5].

Blocking malware. We observe that malware apps connect to different sites, app stores and use different ports from most non-malicious traffic. For known malware stores, we can simply block access to the site via *Meddle*. For other types of malware, we are investigating the effectiveness of behavioral filtering and crowdsourcing. We can use a classifier to determine app activity that is likely malware, then ask users to validate our inference via *Recon*.

4.4 PII Leaked in User Study

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 7: **The top 5 trackers that were contacted by the devices in our dataset.** *All 26 devices in mobUser contacted doubleclick.net and google-analytics.com.*

We now analyze the PII leaks in the *mobUser* dataset using the app classification from §3.4. Note that we do not use SSL bumping on this data for privacy reasons.

Location leaks. We observe that a bus service app (*One Bus Away*), the app that manages the iOS homescreen (*SpringBoard*), and the weather apps (*TWC*, *Weather*, and *Hurricane*) were responsible for more 78% of the flows that sent location in the clear. Other apps that do not require location, such as YouTube, Epicurious and EditorsChoice, also leaked the device location. Further, *SpringBoard* leaked location information for all 11 iOS devices in the *mobUser* dataset, a maximum of 14 leaks per day was observed for one device, sufficient to expose a user's daily movements to anyone tapping Internet connections [18].

Unique ID leaks. The device ID and IMEI are frequently leaked in the clear, and as in the case of controlled experiments, trackers are the most popular destination for the IMEI leaks. Among the 16 sites that received these unique IDs in the clear, 10 are trackers; the rest includes sites for games, news, and manufacturer updates.

In Table 7, we present the top 5 trackers ordered according to the number of devices in the *mobUser* dataset that contacted them. We observe that all the devices in the *mobUser* dataset contacted doubleclick.com, an ad site, and google-analytics.com, an analytics site.

To summarize, we perform controlled experiments to identify and compare PII leaks in Android and iOS. We observe that trackers receive PIIs over HTTP and also over SSL, and that the most popular trackers were able to track all the users in our *mobUser* dataset. We build on our observations and allow users of *Meddle* to visualize and block traffic to trackers, an incentive for users to participate in our on going study.

5. APPLICATION: REVEALING ISP BEHAVIOR

This section describes how we build two applications atop

Figure 3: **Screen capture of content injection by a Chinese ISP in November, 2013.** *The highlighted region at the bottom should be an advertisement from a US company.*

Meddle to reveal the policies ISPs apply to mobile traffic traversing their networks. Previous work addressed this problem in fixed-line networks; to the best of our knowledge we are the first to provide this functionality for mobile systems.

5.1 Detecting Content Manipulation

ISPs, middleboxes and client software are known to change Web page content for a variety of reasons including performance optimization and security. In some cases, a third party can change a page for selfish reasons, *e.g.,* to insert ads that generate revenue for that party. Figure 3 depicts an example of content injection in China, where a banner ad is replaced by information about the local airport.

This problem of Web interference was first highlighted by Reis *et al.* [27]. The authors demonstrated that although a small percent of users were affected by in-flight changes, those changes tend to introduce vulnerabilities including cross-site scripting (XSS) attacks. They proposed and deployed *Web Tripwires*, Javascript code to detect in-flight page changes. The main limitation of *Web Tripwires* is that it requires each Web site to modify their content to include a tripwire.

In *Meddle*, we extended tripwires to alleviate this limitation. Namely, we use the HTTP proxy present in *Meddle* to inject a tripwire on *any* page without requiring support from Web site developers – an approach we call a *Web Tripnet*.

Implementation. With *Meddle* all traffic is tunneled, thereby preventing ISPs from modifying pages. To identify ISP content manipulation (*e.g.,* for public policy reasons and for users not protected by *Meddle*), we provide the Tripnet as an opt-in feature. Because this entails two fetches of every Web page, we also support two modes: always-on and low-rate random trials, where we insert tripwires for some small fraction of their visited sites.

The Tripnet works as follows (Fig. 4). A client requests a Web page through the *Meddle* VPN tunnel. This request is forwarded to the destination server. The response returns to the *Meddle* server, where a transparent proxy injects the

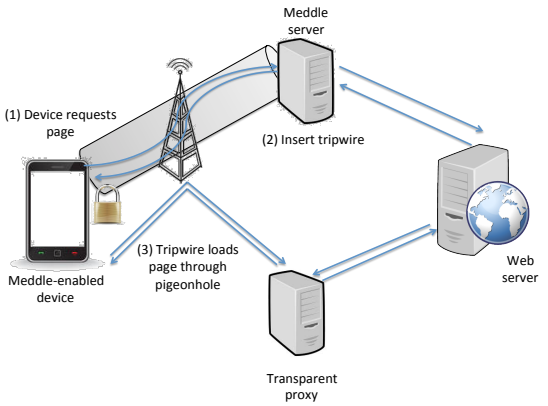


Figure 4: **Overview of the Web Tripnet.** *A Web page loads through the VPN tunnel, where a* meddlebox *inserts a Web tripwire. The tripwire causes the browser to reload the Web page using the address of a transparent proxy server that is accessed using an unencrypted connection. After the proxied version of the page is loaded, the browser (or a* meddlebox*) compare the two pages to identify manipulation.*

tripwire code.[2] The tripwire-enabled response is forwarded to the client, which executes the Javascript at page load time.

The tripwire code contains information about the page content prior to traversing the ISP. When executed, the code fetches the page again to compare with the (known) unmodified page content. To ensure that this fetch does *not* traverse the VPN connection, we use a *pigeonhole* domain whose traffic traverses an untunneled interface. For example, if the original request was for `www.facebook.com`, we send the request to `tripnet.meddle.mobi/www.facebook.com`, where we run a Web proxy.

When the request arrives at our proxy server, we could forward the request to the original target. In practice, however, doing so would return different content due to the highly dynamic nature of most Web content. Instead, we cache Web pages at the tripnet-injecting server and co-locate our Web proxy there. Thus we return exactly the same Web page that was received over the tunneled connection. Any difference in page content can only be due to ISP behavior.

Note that this does not address cases where modification is based on destination IP or when the entire page is replaced (*e.g.,* a block page). To detect this, we can set the tripwire to fetch the page from the origin server at the cost of imprecise identification due to dynamic content.

Content modification/replacement. In addition to detecting changes to text content in Web pages, we use our controlled experiments to investigate whether ISPs are manipulating media content, *e.g.,* downsampling high-resolution images to reduce bandwidth consumption from mobile devices. For this experiment, we augment our Tripnet experiments with the result of wget results from the mobile device and the proxy server. Note that this experiment requires

[2]We recognize the irony of injecting content to detect content injection, but this is done only with user consent.

an app or tethered laptop to collect the Web media objects fetched over the mobile network.

Sites and ISPs tested. We conducted controlled experiments using our Tripnet architecture, using the top 100 Web sites according to Alexa. We tested using AT&T, T-Mobile and Verizon in the US, and Sosh in France. Many sites customize content according to *User-Agent* strings, so we spoof them as coming from iOS, Android and desktop clients.

Content modification/injection. We found no cases of page modifications in the networks tested. In addition the example in Fig. 3, we found whole-page replacement in the case of T-Mobile. A service called *Web Guard* replaced the page for an adult Web site with a block page. Further, our tests allowed us to identify differences in Web content not due an ISP. Specifically, we observe that Wordpress hosts a jquery.js file using the Edgecast CDN, and the file contents differ depending on where the client is located. When accessed directly from mobile devices the script size is 256 KB; when accessed from the proxy server the script is 93 KB. The difference in size is because the latter uses a minified version; it is unclear why this is not served to all the networks.

5.2 Detecting Service Differentiation

In this section, we describe how we use *Meddle* to detect service differentiation in ISPs. We define service differentiation as any attempt to change the performance of network traffic traversing an ISP's boundaries. ISPs may implement differentiation policies for a number of reasons, including load balancing, bandwidth management or business reasons. Specifically, we focus on detecting whether certain types of network traffic receive better (or worse) performance.

Previous work [14, 33, 38] explored this problem in limited environments. Glasnost focused on BitTorrent in the desktop/laptop environment, and lacked the ability to conduct controlled experiments to provide strong evidence of differentiation. NetDiff covered a wide range of passively gathered traffic from a large ISP but likewise did not support targeted, controlled experiments. We now describe how we address these limitations with *Mobile Replay*.

Assumptions. We assume that ISPs will differentiate traffic based on hostname, IP addresses, ports, total number of connections, payload signatures, total bandwidth and time of day. Our system can diagnose nearly all of these cases.

Overview. Mobile Replay identifies service differentiation using two key components. First, it tests for differentiation by replaying real network traces generated from user interactions with apps. *Meddle* facilitates capturing this information, and we develop new strategies for replaying arbitrary app traces. Second, Mobile Replay exploits the *Meddle* VPN to conduct controlled experiments. By alternately replaying traffic over tunneled and untunneled connections multiple times in rapid succession, we control for factors that ISPs may use to differentiate traffic.

A key challenge is how to capture and replay the salient features of application traffic such that it will be subject to differentiation from middleboxes. To this end, we design a system that captures traffic generated by users' devices (via *Meddle*) and replays those flows from a *replay server*.

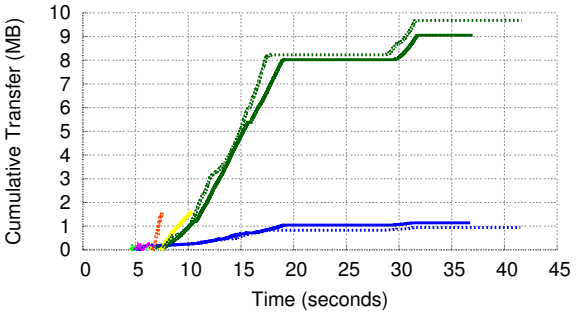


Figure 5: **Plot of record (solid) and and replay (dashed) from Netflix activity.** *The x-axis is time and y-axis is the total number of bytes transferred to that point. The figure shows that our replay closely matches the recorded trace.*

The replay system consists of a client running on the mobile device and a replay server. The client and server coordinate to replay the original flows to reproduces packet timings, sequence of bytes, ports and source IPs. Since our replay is limited to using our own replay servers, we cannot detect differentiation based on arbitrary destination IPs.

Another key challenge is how to establish ground truth as to whether the ISP is differentiating service for replay traffic. To address this, we exploit the VPN connection that *Meddle* provides as follows. When the VPN is enabled, the ISP cannot inspect flow contents and thus cannot differentiate based on the above factors except total bandwidth and time of day. We then compare this performance to the case when we send traffic untunneled. Using multiple successive trials of tunneled and untunneled replay experiments, we can determine the noise inherent in performance metrics in each type of experiment (tunneled vs not tunneled), then identify cases where there are statistically significant differences between them – indicating differentiation.

Feasibility. Figure 5 uses a sequence-number diagram to compare the behavior of original traces to those generated by our replay system, in an environment where we know differentiation is not happening. By preserving packet ordering and timing, our system produces very similar results.

Of course, a variety of factors can differ between record and replay, including network conditions and access technology. In particular, apps may change their behavior in response to network technology and available bandwidth. For example, the YouTube app did not allow HD video content for our test Android device over T-Mobile. In such cases, we must ensure that we replay traffic that was originally captured over similar network conditions. In our experiments, YouTube was the only app that exhibited such behavior.

Methodology. We detect differentiation according to the following metrics. First, we compute checksums to verify that the bytes sent/received at each endpoint during the replay are exactly the same as the original trace. If not, we

	Network Coverage	Portability	Deployment model	Meas. Type	Control?
Large ISP studies [19, 34]	Single carrier	All OSes	Instrument cell infrastructure	Passive	No
WiFi study [13]	Single WiFi network	All OSes	Instrument WiFi network	Passive	No
PhoneLab [12]/TaintDroid [17]	Multiple networks	Android	Install custom OS	Active/Passive	Yes
MobiPerf [35]/SpeedTest [30]	Multiple networks	Android	Install App	Active	Yes
Meddle	Any network	Most OSes	VPN configuration	Active/Passive	Yes

Table 8: **Comparison of related work.** Meddle *is the first approach to provide visibility and control over network traffic for all access networks and most device OSes.*

flag a case of content manipulation/blocking. Second, we compute summary statistics on throughput and loss. Unlike manipulation/blocking, there are confounding factors other than differentiation that may causes changes in these statistics between the record and replay.

To address this issue, we run multiple replay trials (10 total), alternating between using a VPN connection (R_T) and an untunneled one (R_U). By computing statistics over multiple trials of one category (R_T or R_U) we can quantify natural variations in performance that are not a result of differentiation. Having computed the variance over R_T and R_U, we can compare the summary statistics (mean/median) of R_T and R_U and use the variance in each category to determine if the differences are statistically significant.

Note that ISPs may apply differentiation to all VPN traffic, *e.g.,* by throttling. To detect this, we group all R_T samples and compare them to all R_U samples across all applications and use the analysis described above.

Results from wide-area testing. We used Mobile Replay to investigate service differentiation in AT&T, Verizon, and T-Mobile, using the following apps: YouTube (YT), Netflix (NF), Spotify (S) and Dropbox (DB). We picked these apps because they are popular and bandwidth-intensive, and thus are potential candidates for differentiation due to traffic engineering. We interact with these apps for about one minute during the record phase, then replay the traces in each measured network. Note that these apps are intended to demonstrate how our approach works; a complete treatment of service differentiation is beyond the scope of this paper.

Table 9 shows the average throughput and loss (along with standard deviations) in Verizon's 3G network in Boston. Tests with AT&T and T-Mobile were similar. The key take-away is that we observe only small differences in performance, and they are not statistically significant – indicating no service differentiation for these apps. The standard deviations also indicate that detecting differentiation in throughput can be difficult for small changes. On the other hand, small changes in packet loss should be easy to detect.

App	Throughput (KB/s)		Loss (%)	
	No VPN (avg, stdev)	VPN (avg, stdev)	No VPN (avg, stdev)	VPN (avg, stdev)
YT(DL)	(103.74, 31.16)	(99.85, 35)	(0.81, 0.06)	(0.86, 0.13)
YT(UL)	(114.52, 6.05)	(117.37, 8.78)	(0.03, 0.01)	(0.05, 0.01)
DB(DL)	(155.09, 32.42)	(148.1, 44.95)	(0.79, 0.31)	(0.88, 0.38)
DB(UL)	(115.07, 7.31)	(120.25, 5.85)	(0.07, 0.02)	(0.09, 0.01)
SPTFY	(123.91, 40.19)	(127.16, 45.96)	(0.83, 0.08)	(0.73, 0.07)
NFLX	(122.81, 28.42)	(132.26, 33.55)	(0.97, 0.03)	(0.99, 0.15)

Table 9: **Average and standard deviation for 4 apps** *(YT: YouTube, DB: Dropbox, SPTFY: Spotify, NFLX: Netflix; UL=Upload, DL=Download) on Verizon. The differences in performance are within the noise, indicating no service differentiation.*

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). *Meddle* compromises none of these, enabling visibility and control of network traffic across carriers, devices and access technologies. Table 8 puts our approach in context with related approaches regarding 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 [24], network bandwidth and leak private information [17, 20]. Similarly, AppInsight [26] and PiOS [16] 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 subscribers. For example, Vallina-Rodriguez *et al.* [34] use this approach to characterize performance and the impact of advertising. Gerber *et al.* [19] similarly use this approach to estimate network performance for mobile devices. Similar to these approaches, *Meddle* provides continuous passive monitoring of mobile network traffic; however, *Meddle* is the first to do so across all networks to which a device connects.

Active measurements [30, 35] capture network topologies and instantaneous performance at the cost of additional, synthetic traffic for probing. In contrast, *Meddle* uses passive measurements to characterize the traffic that devices naturally generate. PhoneLab [12] provides a mobile experimentation platform with low-level OS and device access; however, it does not run on unmodified device operating systems. *Meddle* does not require OS modification — facilitating large-scale, global deployment.

7. CONCLUSION

We described *Meddle*, a platform for gaining visibility and control over network flows from mobile devices. *Meddle* presents new opportunities for researchers to experiment with middlebox services on Internet traffic generated by mobile users, and provides a variety of clear incentives for users to participate in the system. We demonstrated the effectiveness of our approach based on controlled experiments, a small user study and case studies of applications built atop *Meddle*. Our ongoing work focuses on inviting more users to participate in our study (including in developing regions such as China and India), developing additional *meddlebox* services and opening the platform to other researchers.

8. REFERENCES

[1] Ad blocking with ad server hostnames and ip addresses. http://pgl.yoyo.org/adservers/.

[2] AppsApk.com. http://www.appsapk.com/.

[3] Configuring Your Xcode Project for Distribution. https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/ConfiguringYourApp/ConfiguringYourApp.html.

[4] Google Play Android Developer API. https://developers.google.com/android-publisher/libraries/.

[5] Google Safe Browsing diagnostic page for cooguo.com. http://google.com/safebrowsing/diagnostic?site=cooguo.com/.

[6] Open vswitch: An open virtual switch. http://openvswitch.org/.

[7] Squid-in-the-middle SSL Bump. http://wiki.squid-cache.org/Features/SslBump.

[8] Team Cymru - Malware Hash Registry. https://www.team-cymru.org/Services/MHR/.

[9] UI/Application Exerciser Monkey. https://developer.android.com/tools/help/monkey.html.

[10] Andrubis. http://anubis.iseclab.org/.

[11] BERMUDEZ, I. N., MELLIA, M., MUNAFO, M. M., KERALAPURA, R., AND NUCCI, A. DNS to the Rescue: Discerning Content and Services in a Tangled Web. In *Proc. of IMC* (2012).

[12] CHALLEN, G. Phonelab testbed. http://www.phone-lab.org.

[13] CHEN, X., JIN, R., SUH, K., WANG, B., AND WEI, W. Network Performance of Smart Mobile Handhelds in a University Campus WiFi Network. In *Proc. of IMC* (2012).

[14] DISCHINGER, M., MARCON, M., GUHA, S., GUMMADI, K. P., MAHAJAN, R., AND SAROIU, S. Glasnost: Enabling end users to detect traffic differentiation. In *NSDI* (2010).

[15] EASTLAKE, D., ET AL. Rfc 6066: Transport layer security (tls) extensions: Extension definitions, 2011.

[16] EGELE, M., AND KRUEGEL, C. PiOS: Detecting privacy leaks in iOS applications. In *Proc. of NDSS* (2011).

[17] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of OSDI* (2010).

[18] GELLMAN, B., AND SOLTANI, A. NSA tracking cellphone locations worldwide, Snowden documents show. *Washington Post* (December 4 2013). Retrieved from http://www.washingtonpost.com/.

[19] GERBER, A., PANG, J., SPATSCHECK, O., AND VENKATARAMAN, S. Speed Testing without Speed Tests: Estimating Achievable Download Speed from Passive Measurements. In *Proc. of IMC* (2010).

[20] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In *Proc. of CCS* (2011).

[21] LEONTIADIS, I., EFSTRATIOU, C., PICONE, M., AND MASCOLO, C. Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market. In *Proc. of Hotmobile* (2012), ACM.

[22] MAIER, G., SCHNEIDER, F., AND FELDMANN, A. A First Look at Mobile Hand-held Device Traffic. *Proc. PAM* (2010).

[23] Mozilla collusion. www.mozilla.org/en-US/collusion/.

[24] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. In *Proc. of Eurosys* (2012).

[25] PAXSON, V. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks 31*, 23-24 (1999), 2435–2463.

[26] RAVINDRANATH, L., AND PADHYE, J. AppInsight: Mobile App Performance Monitoring in the Wild. *Proc. of OSDI* (2012).

[27] REIS, C., GRIBBLE, S. D., KOHNO, T., AND WEAVER, N. C. Detecting In-Flight Page Changes with Web Tripwires. In *Proc. of USENIX NSDI* (2008).

[28] ROESNER, F., KOHNO, T., AND WETHERALL, D. Detecting and Defending Against Third-Party Tracking on the Web. *Proc. of USENIX NSDI* (2012).

[29] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making middleboxes someone else's problem: Network processing as a cloud services. In *Proc. of ACM SIGCOMM* (2012).

[30] SOMMERS, J., AND BARFORD, P. Cell vs. WiFi: On the Performance of Metro Area Mobile Connections. In *Proc. of IMC* (2012).

[31] Strongswan. www.strongswan.org.

[32] SUNDARESAN, S., DE DONATO, W., FEAMSTER, N., TEIXEIRA, R., CRAWFORD, S., AND PESCAPE, A. Broadband internet performance: A view from the gateway. In *SIGCOMM* (2011).

[33] TARIQ, M. B., MOTIWALA, M., FEAMSTER, N., AND AMMAR, M. Detecting network neutrality violations with causal inference. In *CoNEXT* (2009).

[34] VALLINA-RODRIGUEZ, N., SHAH, J., FINAMORE, A., HADDADI, H., GRUNENBERGER, Y., PAPAGIANNAKI, K., AND CROWCROFT, J. Breaking for Commercials: Characterizing Mobile Advertising. In *Proc. of IMC* (2012).

[35] WANG, Z., QIAN, Z., XU, Q., MAO, Z., AND ZHANG, M. An Untold Story of Middleboxes in Cellular Networks. In *Proc. of ACM SIGCOMM* (2011).

[36] WETHERALL, D., CHOFFNES, D., GREENSTEIN, B., HAN, S., HORNYACK, P., JUNG, J., SCHECHTER, S., AND WANG, X. Privacy revelations for web and mobile apps. In *HotOS* (2011).

[37] XU, Q., ERMAN, J., GERBER, A., MAO, Z., PANG, J., AND VENKATARAMAN, S. Identifying diverse usage behaviors of smartphone apps. In *Proc. of IMC* (2011).

[38] ZHANG, Y., MAO, Z. M., AND ZHANG, M. Detecting traffic differentiation in backbone isps with netpolice. In *SIGCOMM* (2009).