Improving transparency and end-user control in mobile

Ashwin Rao

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS
ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE LÍNFORMATION ET DE LA COMMUNICATION

T H E S E

pour l'obtention du grade de
Docteur en Sciences
de l'Université Nice Sophia Antipolis

Mention : Informatique

présentée et soutenue par
Ashwin RAO

Improving Transparency and End-User Control in Mobile Networks

Thése dirigée par Walid DABBOUS et Arnaud LEGOUT
et préparée au sein du laboratoire INRIA, équipe DIANA
soutenue le 19 Décembre 2013

Jury

Abstract

Mobile devices are increasingly becoming the primary device to access the Internet. Despite this thriving popularity, the current mobile ecosystem is largely opaque because of the vested monetary interests of its key players: mobile OS providers, creators of mobile applications, stores for mobile applications and media content, and ISPs. This problem of opaqueness is further aggravated by the limited control end-users have over the information exchanged by their mobile devices. To address this problem of opaqueness and lack of control, we designed a user-centric platform, *Meddle*, that uses traffic indirection to diagnose mobile devices. Compared to an on-device solution, *Meddle* uses two well-known technologies, VPNs and middleboxes, and combines them to provide a solution that is agnostic to OS, ISP, and access technology. We use *Meddle* for controlled experiments and an IRB approved study, and observed that popular iOS and Android applications leak personally identifiable information in the clear and also over SSL. We then use *Meddle* to prevent further leaks using a DNS based packet filter. We also use our platform to detail the network characteristics of video streaming services, the most popular Web-service in the current Internet. We observe that the network traffic characteristics vary vastly with the device (mobile or desktop), application (native applications and also between individual desktop browsers), and container (HTML5 and Flash). This observation is important because the increased adoption of one application or streaming service, for example, an increase in the usage of mobile devices to stream videos, could have a significant impact on the network traffic.

Résumé

Les terminaux mobiles (smartphones et tablettes) sont devenus les terminaux les plus populaires pour accéder à Internet. Cependant, l'écosystème incluant les terminaux mobiles est maintenu opaque à cause des intérêts financiers des différents acteurs : les concepteurs des systèmes d'exploitation et des applications, les opérateurs des "stores", et les FAI. Cette opacité est renforcée par le peu de contrôle qu'ont les utilisateurs sur les informations échangées par leur terminal.

Pour résoudre ce problème d'opacité et de manque de contrôle, on a créé une plateforme, Meddle, qui utilise la redirection de trafic des terminaux mobiles pour analyser et modifier ce trafic. Contrairement aux solutions qui nécessitent d'être implémentées sur le terminal, Meddle combine les techniques de VPN et de "middlebox" pour offrir une solution indépendante de l'OS, du FAI et de l'accès radio.

On a utilisé Meddle pour des expérimentations contrôlées et pour une étude utilisateurs approvée par un IRB. On a observé que des applications populaires sous iOS et Android exposaient des informations personnelles dans le traffic réseau en clair et chiffré. On a ensuite exploité Meddle pour prévenir ces fuites d'informations privées.

On a également utilisé Meddle pour étudier les caractéristiques réseaux du trafic vidéo sur Internet. On a trouvé que ce trafic dépend fortement du type de terminal, de l'application utilisée pour regarder la vidéo (application native ou navigateur Web) et du contenant (HTML5, Flash, Silverlight). Ce résultat montre qu'un changement dans le terminal, l'application ou le contenant peut avoir un impact important sur le réseau.

Contents

1 Introduction

Freedom is Slavery; Ignorance is Strength. These two contradictions summarize our rights over our mobile devices: devices which we increasingly use to manage our every day life; devices that have gained seamless access to a wealth of our private information. These two phrases are part of the famous slogan—*War is Peace; Freedom is Slavery; Ignorance is Strength*—used to rule the Orwellian hell of Oceania [115]. Like Oceania, the rulers of the ecosystem inhabited by our mobile devices not only offer us limited control over our devices but also violate our privacy. And like Oceania, these rulers use security and protection against external threats as the pretext for opaqueness and lack of control.

It is important to secure and protect the data we manage with our mobile devices. This data is sensitive because mobile devices have evolved from the replacement of telephones to the replacement of personal computers.[1] This evolution has transformed our mobile devices into the primary gateway to stay connected with the world we live in—friends, family, and colleagues. As a consequence, a wealth of our private information such as contacts, emails, and photographs, is now stored on our mobile devices and managed by mobile applications. The importance of our data, and the desire to protect it, persuades us to be easily subdued by the ones who offer to secure and protect this data. This offer is currently made by the key players of the mobile ecosystem: the mobile operating system (OS) providers, the app developers, the stores for software and media distribution, and the Internet Service Providers (ISPs).

Freedom is Slavery. Mobile applications, henceforth referred to as apps, and the cloud based services that serve these apps, facilitate an on-demand access to our data. However, this flexibility comes at a cost of relinquishing control over this data to the key players that offer these services. On the one hand, mobile OSes allows apps access to our private information through coarse grained permissions, and on the other hand, these OSes impose stringent restrictions on installing customized services to protect the devices and the data from potential misbehaving apps and services. Furthermore, the warranty of mobile devices turns void if users install customized services to audit and control the flow of data in their mobile devices [82, 83, 96]. Similarly, we have limited control over the data apps exchange with the cloud based services that serve these apps, and how these cloud based services use our data. Thus, relinquishing control of our data to these players is slowly enslaving us; we are being subdued to be ruled under the slogan *Freedom is Slavery.*

Ignorance is Strength. The key players of the mobile ecosystem do not work in isolation and are connected by a web of interdependence. This interdependence exists primarily to maximize the control that each player has over this ecosystem, control that comes with its share of profits. For example, organizations responsible for mobile OSes also control the software and media distribution platforms. These platforms influence the set of apps that manage our data. Furthermore, to support apps that generate revenue from advertisements, mobile OSes support libraries that allow apps to negotiate with advertisers. These

[1]Though a hazy line separates tablets from laptops, mobile devices in the context of this dissertation is limited to smartphones and tablets.

ad libraries typically negotiate with advertisers using services provided by the organizations responsible for mobile OSes. For example, Apple's iOS supports iAd while Google's Android supports admob [60] and adsense [63]. Intuitively, the most appealing advertisements are the ones that match our likes and dislikes. To maximize our engagement with advertisements, apps leverage on our private data to send relevant advertisements. Each byte of data we store on our mobile device comes with potential monetary value for the players that rely on advertisements for generating revenue; our ignorance on the abuse of private data is important to maximize revenue for these players. Similarly, an increase in the traffic volume generated by apps can be used by ISPs to convince users to switch to plans offering higher quotas; ISPs can profit from our ignorance on the traffic characteristics of apps. Thus, our ignorance on how our data is managed and how apps interact with other devices in the Internet allows us to be ruled under the slogan, *Ignorance is Strength.*

Mobile devices will continue to be an integral part of everyday life. We will not part with our mobile devices, our gateway to stay connected to the Internet based services and the world we live in. Furthermore, billions in developing countries are expected to make a mobile device their first and only gateway to the Internet. This vision is supported in the recent International Telecommunication Union (ITU) report: *"in developing countries, mobile-broadband services cost considerably less than fixed-broadband services"* [128]. We cannot afford to revert to a disconnected life, and we do not wish to be ruled by the slogans *Freedom is Slavery; Ignorance is Strength.* We must therefore try to improve the transparency and regain control over how our data is managed by our devices. This is the goal of this dissertation.

1.1 The Mobile Ecosystem

Mobile devices are in an ecosystem whose evolution is driven by a few key players: 1) mobile OSes, 2) apps, 3) the stores for software and media content distribution, and 4) Internet service providers (ISP). These players are tied by commercial agreements among them and by their revenue models. This interdependence is the primary cause for the opaqueness and lack of end-user control that prevails in the mobile ecosystem. We address the problem of opaqueness and lack of end-user control in this dissertation.

In the following, we focus on the role of these players, their incentive to participate in this ecosystem, and the differences between their counterparts in the ecosystem of traditional personal computers.

1.1.1 The Mobile Operating System

The mobile OSes manage the various hardware resources on our mobile devices. Unlike personal computers, the hardware resources on mobile devices are limited. For example, the battery size on our mobile devices is significantly smaller that batteries that drive laptops. Furthermore, the mobile OSes need to support a large number of sensors such as accelerometers, GPS, and proximity sensors, that are not present with desktop devices. Mobile OSes are therefore fine-tuned by their developers and device manufacturers to optimize the device performance, a key difference between mobile and desktop OSes. Three mobile OSes—Android, iOS, and Window Mobile—currently dominate the current mobile ecosystem [61]; the other OSes include Blackberry, Nokia Asha, Bada and the new entrants Firefox OS and Ubuntu.

The limited resources on mobile devices demands a close coordination between OS providers and device manufacturers. This close coordination is essential to support device specific sensors, and to optimize the performance according to the hardware chosen by the manufacturers. A result of this close coordination is that OS services running on mobile devices depend on the OS providers and device manufacturers. In this dissertation, we focus on mobile OSes and use them to abstract the impact of device manufacturers on the opaqueness and lack of control in the mobile ecosystem.

Mobile OSes provide APIs to expose the resources on mobile devices. The incentive for mobile OSes to provide APIs is that they can rely on the talent of independent developers to target a wider audience of customers. Indeed, app developers have over time used these APIs to transform mobile devices from a replacement of telephones to a digital Swiss-Knife. APIs thus open mobile OSes to support a wide range of apps.

Mobile OSes enforce strict policies on the API to restrict access to the limited resources on mobile devices. Due to the critical nature of resources such as battery and sensors, the apps running on the devices need to be isolated and monitored by OS services to prevent misbehavior [6, 42]. For example, iOS limits the activities of background processes *to improve battery life and user's experience with the foreground apps* [6]. Furthermore, to prevent users and app developers from modifying the OS, the warranty of mobile devices becomes void if users modify the OS running on their devices.

In summary, mobile OSes are walled gardens built in close coordination between creators of mobile OSes and device manufacturers. The creators of mobile OSes provide APIs to expose the wide range of sensors and resources on mobile devices to application developers. To limit misbehavior and ensure optimal resource usage, these OSes rely on strict policies.

1.1.2 The Mobile Applications (Apps)

The mobile applications (apps) make the mobile ecosystem lively and dynamic. Along with apps that are a portal to Web services, such as Facebook or Twitter, app developers have used their creativity to come up with innovative uses of the wide array of sensors available on our mobile devices.

Apps are inherently different from their counterparts running on traditional personal computers because mobile OSes restrict their activity for reasons previously discussed. Mobile devices are currently shipped with a wide array of sensors including cameras, accelerometers, gyroscopes, proximity sensors, and GPS. These sensors, and the enhanced user experiences offered by apps that use these sensors, make apps superior to their desktop counterparts in many ways. For example, apps use the motion sensors to determine the best layout, portrait or landscape, while rendering content. Games also use motion sensors as an input for user actions, interactions that were previously not possible. Similarly, the proximity sensor is used to determine when the device is close to a user's face, for example, during a phone call. To limit the abuse of these sensors and other resources, mobile OSes impose restrictions on their usage. These restrictions mandate that apps explicitly demand authorization from end-users to use these sensors. These OS restrictions and heavy dependence on sensors implies that the behavior of apps depends on the devices on which they are running.

Many apps act as a gateway to cloud based services. Such apps are a portal to social networks such as Facebook, Twitter, and Google Plus, and navigation services such as Google Maps, Apple Maps, and Bing Maps. Apps for social networks typically have access to the data we use to socialize with other people, including our contacts, photos, music, and videos. For example, users can automatically back-up pictures taken by their mobile devices on social networking sites such as Google Plus, a service that is much more seamless and smooth than the photo back-up services offered for desktop computers. These apps also run as background services to receive updates on the activities of our contacts on these social networks. Such background services are not available for desktop-users who rely on Web-browsers to access these services. Thus, apps have enhanced the overall experience of network intensive services.

In summary, apps use sensors on mobile devices to enhance user experience with the aim to increase user engagement. The innovative uses of the sensors makes these apps superior to their desktop counterparts.

1.1.3 The Stores for Software and Media Content Distribution

Users can purchase apps and media content—movies, songs, and books—from online stores customized for mobile devices. However, these stores influence the choice of apps and the media content. Indeed, organizations that run the stores earn money by selling apps and media content [20].

The App Store from Apple, the Google Play Store from Google, and the Windows Phone Store from Microsoft, are the default stores for mobile devices running on Apple's iOS, Google's Android, and Microsoft's Windows Phone OS respectively. Furthermore, mobile OSes are shipped with an app which is a portal to the default store for that OS. For example, Android devices are shipped with a Google Play app while iOS devices are shipped with an App Store app. This app is responsible for the purchase, installation, upgrade, and uninstallation of other apps running on the mobile device. The mobile OS providers use this app to influence the set of apps that run on mobile devices.

The stores also perform security and performance tests on apps before making them publicly available. Such tests are performed to raise confidence on the quality of apps and media content available for download. For example, Google Play claims to use a tool called Bouncer that checks apps for malware before the apps are made available for purchase [111]. Therefore, it might be argued that these stores work towards improving the end-users experience.

However, stores restrict the availability of content on their stores based on country-specific copyright laws and code licenses. For example, the Apple App Store does not sell GPL licensed apps [132]. Furthermore, copyright laws restrict the availability of apps and media content to specific regions. For example, the Netflix app is not available in the App Store in France; similarly, songs available in France may not be available in the US due to copyright restrictions.

In summary, mobile users can purchase apps and media content on stores managed by OS providers, whose portal is installed by default on mobile devices. These stores influence the purchases made in order to maximize their profits.

1.1.4 The Internet Service Providers

The Internet service providers (ISP) enable the apps and OS services running on mobile devices to exchange data with other devices in the Internet. The network-intensive nature of mobile devices makes the ISPs a vital player in the mobile ecosystem.

Mobile devices exchange data using their wireless interfaces. Each mobile device typically come with two interfaces: one for the cellular connectivity, and one for wireless LAN (Wi-Fi); tablets that do not offer cellular connectivity are an exception to this rule. Along with these two primary communication interfaces, mobile devices may also support wireless interfaces to communicate with devices in their vicinity. Bluetooth and Near field Communication (NFC) are two such interfaces that have a limited communication range. The ISPs only serve traffic coming from the Wi-Fi and cellular interfaces of mobile devices.

A mobile device can be served by multiple ISPs. The ISP serving Wi-Fi traffic depends on device location and the Wi-Fi gateway used by the device, while the cellular interface is typically served by one ISP. The role of cellular ISP is to offer the latest wireless technologies and maximize the geographic coverage to ensure that end-users have the best Internet connectivity at all times. Unlike cellular ISPs, users are not restricted to a specific ISP when using Wi-Fi. For example, the Wi-Fi gateway at home and the Wi-Fi gateway at work can be served by different ISPs.

1.1.5 The Web of Interdependence

The mobile OS providers, the app developers, the stores for software and media content distribution, and the ISPs are the key players of the mobile ecosystem. These players depend on each other for their survival in this ecosystem, and their revenue models along with the commercial agreements between them keeps them interdependent.

The key sources of revenue in this ecosystem are as follows.

1. **Sale of mobile devices.**
   Mobile devices can be purchased from device manufacturers, and from ISPs that bundle these devices with cellular data plans. The distribution of profits depends on who sold the device. For example, when a device is sold by an ISP, the commercial agreements between the mobile OS provider, the device manufacturer, and an ISP decide the distribution of profits [107].
2. **Sale of apps and media content (music, videos, books, and magazines)**.
   Though the sale take place in the stores, the revenue is shared by app developers, sellers of media content, and the organizations managing the stores. For example, the Google Play store charges a transaction fee of 30% of the application price; the developer receives the remaining 70% [20].
3. **Subscription charges for network connectivity**.
   The ISPs charge end-users for Internet connectivity, however, this revenue might be

shared with device manufacturers if the device and network charges were bundled during the sale of the device [107].

4. **User engagement with advertisements displayed on mobile devices.**
   The revenue from mobile ads is shared by the app developers and the ad broker responsible for the ads. For example, developers earn 70% of the net revenue generated from iAd advertisements [21]. Some of the popular ad brokers are typically managed by creators of mobile OSes. For example, iAd and admob are two popular ad brokers managed by Apple and Google respectively.

These various sources of revenue make mobile devices a hen that lays golden eggs for these players. Each mobile device is an entry point to the sale of other products purchased using that device. Mobile devices are therefore shipped with a default set of apps and services tailored to maximize the revenue for players behind the sale of that device. This default set of apps includes the app for the store from which users can buy other apps and media content. Thus, the players selling mobile devices leverage their influence on other purchases made in the mobile ecosystem.

The stores influence the apps we choose to install on our mobile devices and the media content we purchase. Indeed, these stores monitor our purchases to recommend new apps and media content. The incentive to influence purchases is high for the app stores because their revenue depends on purchases made on the app store; the sales-volume depends on the recommendations made when responding to queries end-users make on these stores. Access to our private data is therefore important for the success of these stores.

Private information is also important for mobile apps because they generate revenue from targeted advertisements. The advertisement market is dominated by a few players such as iAd [64], admob [60], and adsense [63]; each player in turn has a large market share [73] that allows it to collect a lot of information on end-users for building fine grain profiles. Therefore, private information become a product that generates a lot of revenue for the app stores, the apps developers, and the ad brokers.

The key players are tightly bound in the mobile ecosystem by commercial agreements. To run these agreements, the players keep control on the mobile devices at the expense of end-users, resulting in opacity and lack of end-user control. We develop this problem of opacity and lack of end-user control in the next section.

1.2 The Problem: Lack of Transparency and Control

The mobile ecosystem suffers from a lack of transparency and end-user control. While end-users should be free to monitor and control their privacy leaks, the key players of the mobile ecosystem foster opacity and lack of control, following the slogan: *Freedom is Slavery; Ignorance is Strength.*

In this section, we first define what we mean by transparency and control. We then motivate the need for transparency and control, and describe how the key players of the mobile ecosystem compel us to compromise this demand. Finally, we have a look at the shortcomings of existing solution with a focus on how constraints by the key players make these solutions impractical.

1.2.1 Our Definition for Transparency and Control

Transparency is the awareness on *what* our mobile devices do with our information, with *whom* our mobile devices communicate, *how* our mobile devices interact with other devices on the Internet, and the impact of these interactions. While transparency enables the auditing, control empowers us to make our devices work according to our needs. In this dissertation, we focus on mobile devices because they are the only entity that end-users can monitor and control. We will now see how the closed nature of the mobile ecosystem prevents transparency and end-user control.

1.2.2 The Need for Transparency and Control

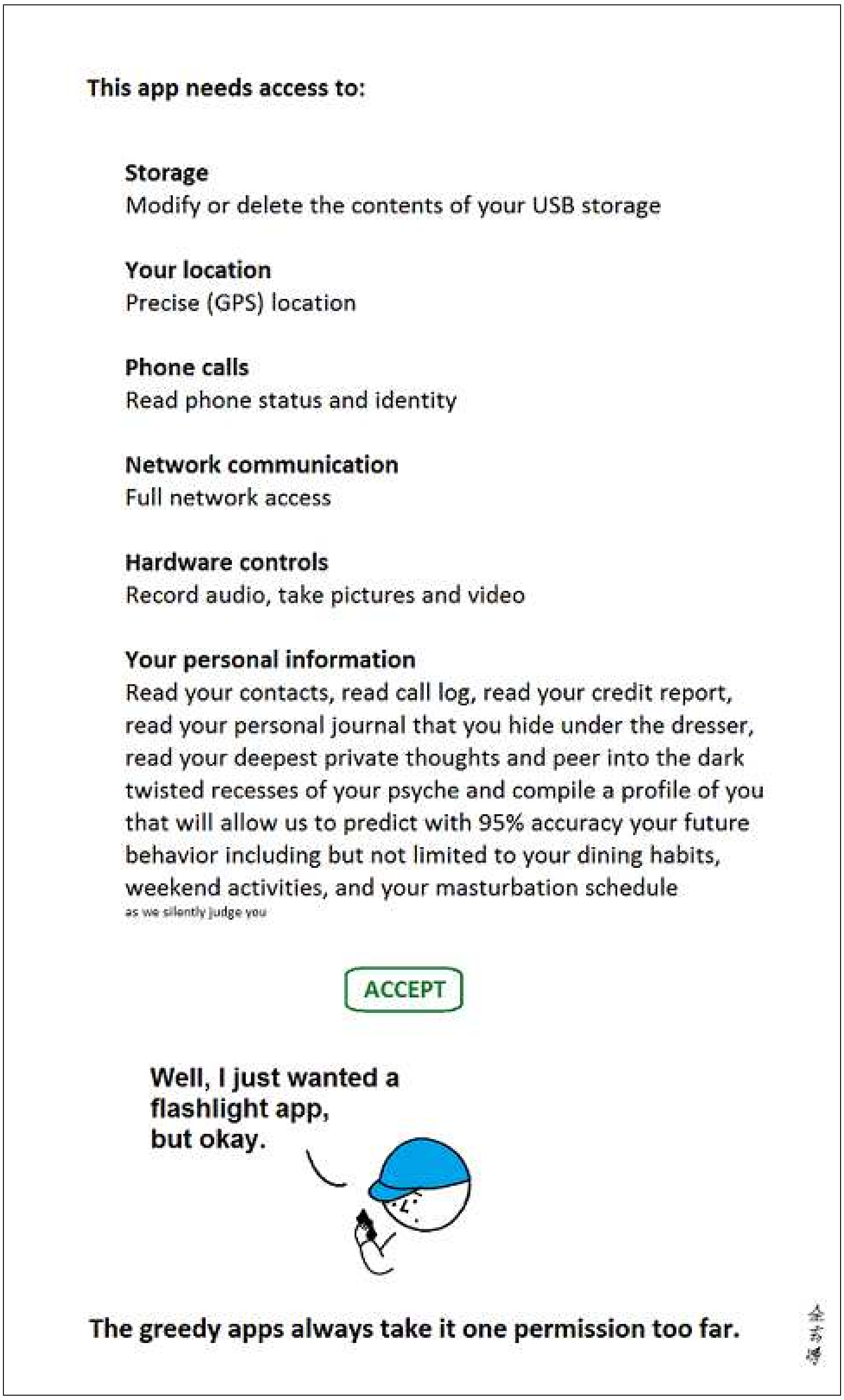
Due to the large amount of private information on mobile devices, we argue that it is fundamental to offer transparency and control on the privacy leakage to end-users. Our mobile devices act as a gateway to Internet based services. Further, cloud based services that help manage our private information periodically receive our private information such as contact details, pictures, places visited, and current location. Apps can also use the various sensors on mobile devices to monitor and manage everyday activities. For example, the marketing slogan for the Google Now application is: *Stay on top of what's happening in your life every day, including what you need to do, where you need to go, and how to get around* [19]. Sensitive apps like Google Now use coarse grained permissions to access private information. The effectiveness of such coarse grained permissions is questionable because a significant number of apps and libraries used by these apps are known to abuse their privileges and leak information without user's consent [68, 73, 82, 83, 96, 139].

The increasing usage of our mobile devices tests the limits to which the resources on these devices can be used. The ever-increasing reliance on mobile devices to manage everyday activities has resulted in an increase in the network consumption and the amount of computation performed on these devices [87, 124, 137]. The increase in network consumption stretches cellular data consumption towards the limits offered by carriers. Similarly, the increase in computation increases the power consumption which in turn decreases the battery life. Battery life and network quotas affect the availability of mobile devices. We expect mobile OS services and the apps to maximize the availability of the limited resources on mobile devices. However, we have limited knowledge on how apps use these resources [87, 137].

In summary, the importance of mobile devices and the private data managed by these devices justifies the need to monitor and control our mobile devices.

1.2.3 The Compromise We Are Compelled to Make

The key players use the argument of security and data protection to justify the opaqueness and lack of end-user control, but this argument is only partially valid. Indeed, these players have taken steps to secure and protect not only the sensitive data stored but also the limited resources available on mobile devices. For example, mobile OSes rightfully isolate apps and restrict the activities that apps can perform when running as background processes [6, 42]. Such isolation is important to increase battery life and restrict access to private data and



This app needs access to:

Storage
Modify or delete the contents of your USB storage

Your location
Precise (GPS) location

Phone calls
Read phone status and identity

Network communication
Full network access

Hardware controls
Record audio, take pictures and video

Your personal information
Read your contacts, read call log, read your credit report, read your personal journal that you hide under the dresser, read your deepest private thoughts and peer into the dark twisted recesses of your psyche and compile a profile of you that will allow us to predict with 95% accuracy your future behavior including but not limited to your dining habits, weekend activities, and your masturbation schedule
as we silently judge you

ACCEPT

Well, I just wanted a flashlight app, but okay.

The greedy apps always take it one permission too far.

Figure 1.1: Plight of end-users portrayed in the Abstruse Goose comic strip. *This image is protected under the following Creative Commons license: http://creativecommons.org/licenses/by-nc/3.0/us/.*

sensors. Similarly, the stores perform security tests on apps before they are available for purchase [111]. However, the absence of public information on these tools raises questions on their effectiveness.

We argue that though security is a valid reason to thwart misbehaving apps, it should not be the reason to stop users from installing apps that audit the behavior of OS services and apps. However, the current terms claim that such tools violate either the device warranty, the service warranty, or both.

The opaqueness prevails in the mobile ecosystem because opaqueness gives the key players a share of control over the mobile ecosystem. Increasing transparency decreases the control of the player. For example, apps that rely on advertisements would not be in favor of auditing the private information in their possession; opaqueness empowers them to build user profiles that can be sold to advertisers. Similarly, opaqueness on resource usage makes it difficult to compare not only the different devices but also the apps and services running on these devices. For example, some Android devices use this opacity to fake their performance for apps used in benchmarking tests [44].

In summary, we are compelled to blindly trust the mobile ecosystem and offer seamless control of our devices to the key players of this ecosystem – a compromise we make to stay connected with our friends, family, and colleagues.

1.3 Discussion on Related Work

Existing approaches, that tilt the balance of the transparency and control in favor of end-users, are impractical because of the constraints imposed by the key players of the mobile ecosystem. For this dissertation, we consider an approach to be practical when it can be used by off-the shelf devices regardless of the ISPs that serve these devices. Specifically, a practical approach must not violate the device warranty and should be agnostic to the mobile OSes, the ISPs, and the stores that are used to purchase apps and media content. A practical approach is desirable because it can scale to a large number of end-users, thus making the research work coming out of this approach meaningful for end-users. Existing solutions are focused on academic analysis and are not targeted for end-users. In spite of being useful for researchers, these solutions are impractical for end-users because the closed nature of the mobile ecosystem limit them to a single mobile OS, installed apps, or ISP. We are the first to propose a solution for real users. We now summarize the existing solutions based on their limitations.

1.3.1 Constrained to a Single Mobile OS

Instrumenting mobile OSes, and tracking the low level system calls, can be used to monitor and control the flow of information in our mobile devices. The seminal work in this area is Taintdroid [83], a realtime information monitoring system that sheds light on the violation of end-user privacy by instrumenting Android. In their paper, Enck *et al.* [83] report on 68 instances of potential misuse of users' private information across 20 apps and mention that *15 Android apps send users' location information to remote advertisement or analytics servers without the users' consent.* To regain control over such leaks, the creators of AppFence [96] instrument Android to implement privacy controls. These privacy controls not only substitute shadow data in place of private data but also block network transmissions of data that the user made available for on-device use only. Similarly, Pathak *et al.* [116] instrument the Android and the Window Phone OS to build Eprof, an energy profiler. With the help of Eprof, Pathak *et al.* show that the third-party advertisement and analytics modules consume up to 75% of the energy consumed by free apps. Such energy wastage severely affects the usability of the mobile devices.

However, Taintdroid, AppFence, and Eprof void the device warranty because of the stringent control exercised by the key players of the mobile ecosystem. Furthermore, instrumenting mobile OSes makes the solution specific to a given OS and cannot be applied to other OSes suffering similar issues. Instrumenting OSes thus voids the device warranty and has a scope that is limited to a subset of popular mobile OSes.

1.3.2 Constrained to Apps

App binaries can be instrumented for static and dynamic analysis to study the information flow through apps. Egele *et al.* [82], instrumented the binaries of 1400 iPhone apps and observed that more than half of these apps send the unique ID of the device to third-party sites; the third-party sites can used this information to create detailed user profiles. Similarly, AppInsight [124] instruments apps to perform dynamic analysis with the aim of identifying critical paths when the apps are in use. Such analysis sheds lights on the inner workings of apps, however, their scope is limited to the specific version of the instrumented apps and the stores from which these instrumented apps are made available. Furthermore, as in the case of OS instrumentation, the results are limited to OSes on which the instrumented apps run. For example, the different APIs available to developers on Android and iOS makes the Facebook app running on iOS to behave differently from the Facebook app on Android; this app is just one of the nearly million iOS and Android apps currently available [57, 73].

Static and dynamic analysis can also be performed without instrumenting apps. Indeed, *droidbox* [134] uses a combination of static and dynamic analysis to identify malware. Similarly, *androguard* [80] uses static analysis to identify malware and compare Android applications. However, like Taintdroid [83] and AppFence [96], *droidbox* and *androguard* cannot accurately trace native code (for example, code written in C) because its access is limited to the java code executed by Android's Dalvik virtual machine. This implies that these techniques provide an incomplete picture on app behavior.

To summarize, while static and dynamic analysis by instrumenting apps cannot scale, static and dynamic analysis without instrumenting apps cannot provide a complete picture on the behavior of apps.

1.3.3 Constrained by Access Technology

Monitoring network traffic at the gateways used by mobile devices improves transparency. However, the various access technologies—cellular and Wi-Fi—available on mobile devices create a high barrier to entry because these access technologies can be served by different

ISPs. For example, though an end-user may have a cellular plan with one ISP, they are free to use another service provider for home Wi-Fi, and to use Wi-Fi services in cafes and other public places. As a consequence, measurement studies offer a limited perspective on the network usage of mobile devices when they are based on Wi-Fi traffic measured at institution gateways [76] or traffic traces obtained by service providers [135].

1.3.4 Positioning of Our Contributions with Related Work

The constraints imposed by the key players limit the usefulness of existing solutions aimed at improving the transparency and control in the mobile ecosystem. The approaches of instrumenting the OS and application binaries, and analyzing traffic traces from service providers cannot scale to a large user participation. Such approaches are also not suitable for longitudinal studies because mobile OSes and apps can have fast release cycles [105]. Instrumenting the OS results in warranty voiding the devices, and instrumenting apps cannot scale to the vast number of apps and obsoletes the effort when new versions of the apps are released. Similarly, traffic traces from service providers do not provide a comprehensive coverage of the network usage of mobile devices. The need for a practical solution is important to ensure that users can reap the benefits of transparency and control regardless of the mobile OS, installed apps, app store, ISP, and access technologies.

The capability to monitor and control the mobile Internet traffic has the potential to improve the transparency and end-user control of the mobile ecosystem. Access to mobile Internet traffic offers a perspective that is focused on the network activity of mobile devices. Indeed, this network perspective has promising prospects because popular mobile apps are network intensive [87, 113, 135], and misbehaving apps are known to use the Internet to leak personal information [73, 82, 96, 83]. We use the network perspective on the activity of mobile devices and test the limits to which it can improve the transparency and end-user control in the mobile ecosystem.

Redirecting all the Internet traffic of a mobile device through software defined middleboxes offers the network perspective on the activity of mobile devices. A Middlebox is defined as *"any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host"* [75]. Software-defined Middleboxes come with a variety of software tools and packages to perform the desired Middlebox activities such as Firewalls, Proxies, Caches, and Packet classifiers [131]. Such Software-defined Middleboxes can be tuned to regain control over the mobile network traffic for performing activities such as monitoring the traffic and manipulating privacy invasive traffic. Offloading traffic monitoring and manipulation activities to Software-defined Middleboxes makes it possible to design and operate solutions that are independent of the mobile OSes and ISPs.

Software-defined Middleboxes can access all mobile data traffic, however this traffic can be encoded, obfuscated, or encrypted by the applications. Indeed, apps are free to transform data before transmission. This transformation includes encoding data to formats such as Base64 [101], or encrypting the data before sending it using HTTPS [126]. Therefore, the middleboxes processing data traffic are exposed to data whose encoding details are available only with the mobile application and remote hosts with whom these application communicate. For any meaningful analysis, our Software-defined Middlebox should be able to decode such traffic.

In this dissertation, we posit that proxying to redirect mobile Internet traffic through Software-defined Middleboxes can improve the transparency and end-user control in the mobile ecosystem. Specifically, we rely on VPN based proxying to tunnel mobile Internet traffic through our Software-defined Middleboxes that interpose on this traffic. Mobile devices are shipped with VPN support primarily to satisfy their enterprise clients. The native support for VPNs implies that traffic redirection does not require instrumenting the operating system and application binaries. Our approach therefore has the potential for practical improvement of transparency and end-user control. We now present our hypothesis and summarize our contributions based on this approach.

1.4 Summary of Contributions

The hypothesis of this dissertation is the following: "*The mobile Internet traffic accessed by traffic redirection can be leveraged to improve the transparency and control for end-users in the mobile ecosystem.*"

We validate this hypothesis by the following contributions.

- **Platform to improve transparency and end-user control in mobile networks.** We first demonstrate that it is feasible to redirect mobile Internet traffic through software defined middleboxes for the purpose of analysis and interposition, a solution we call *Meddle*. The key advantages of *Meddle* is that it works out of the box for Android and iOS, the two most popular mobile OSes. *Meddle* is also agnostic to ISP and access technology (*e.g.*, cellular or Wi-Fi), and users can enable and disable *Meddle* according to their convenience. Furthermore, we show that *Meddle* can be used to monitor and manipulate all Internet traffic, including SSL traffic, from real users. We show empirically that the overheads in terms of latency, power, and data consumption are reasonable for users to adopt *Meddle*. Thus, *Meddle* offers a unique vantage point allowing real users to participate in research activities without voiding their device and service warranty. We envision two scenarios in which *Meddle* can be deployed: 1) a single-user deployment on a user's home-gateways or personal servers, or 2) a multiple-user deployment on hosted servers such as Amazon EC2. *Meddle* is currently deployed using the later in private beta version and is serving users in the US, France and China. Users can sign-up for an IRB approved study through `http://meddle.mobi`, and this private beta version has served more than a 100 users.

- **Diagnosing Mobile Apps.** We then show that *Meddle* can be used to diagnose mobile applications and services. First, we use *Meddle* to perform controlled experiments to obtain a ground truth information on network flows generated by apps and OS services. We then extract signatures of apps and Web services from the protocol headers in the network flows, and used these signatures to map network flows to the apps and services that generate them. We also use our experiments to identify leaks of personally identifiable information (PII). In particular, we use *Meddle*'s ability to monitor SSL traffic to observe that misbehaving apps collude with ads and analytics libraries, and use HTTP and SSL to leak PIIs. Second, we use our technique to identify apps and web services on traffic traces that we collected in our IRB approved *in-the-wild* measurement study. Our study involved traffic traces from 117 devices belonging to users spread across US, France, and China. We use these traces to compare the device usage and improve the classification technique we built. Finally, we use our results to build a tool that allows users to visualize and block PII leaks. *Meddle* manipulates DNS responses for sites that leak PIIs, which makes it effective even for SSL traffic because DNS requests occur out of band from secure connections. To summarize, we use the research work coming from *Meddle* to create incentives to recruit users to participate in future research activities.

- **Characterizing YouTube Traffic.**
  We then characterize YouTube traffic, one of the most dominant sources of Internet traffic by volume. We present the two different streaming strategies that we identified during our measurements, synthesize the main characteristics of those strategies, and discuss their advantages and disadvantages. We show that the traffic patterns observed during streaming sessions are completely different from those observed during typical file transfers. The difference in traffic patterns is because the client side applications and the YouTube servers that stream videos explicitly control the data transfer rate. Furthermore, we observe that the traffic patterns observed when streaming YouTube videos depend on the client side application (desktop browser or mobile app) and container (Flash or HTML5). With the help of the datasets which we collected in 2011 and 2013, we show that the traffic patterns observed in 2013 are completely different from those observed in 2011. In particular, we observe that Internet Explorer is more aggressive in 2013 compared to 2011 when streaming HTML5 videos This implies that upgrading to Internet Explorer 10 can potentially waste a larger amount of bytes and network resources when users interrupt playback of HTML5 videos. Furthermore, we observe that streaming videos to mobile devices produce traffic patterns that are completely different from those observed when using desktop browsers, and that these traffic patterns change when mobile devices use Wi-Fi instead of cellular networks. This observation implies that a large scale migration from one application to another (browser to mobile app) or from Flash to HTML5 can completely change the traffic patterns observed in the backbone links. Considering the very fast changes in trends this is a real possibility, the most likely being a change from Flash over PCs to HTML5 over mobile devices.

We detail these contributions in Chapter 2, Chapter 3, and Chapter 4 respectively. We discuss the detailed related work for each of these contributions when presenting the contribution. We finally conclude by discussing some open problems in Chapter 5.

2 Meddle Architecture

We now present *Meddle*, our platform that combines VPNs and middleboxes in unintended ways to diagnose mobile devices using traffic indirection.

Meddle was built to improve the transparency and end-user control over mobile Internet traffic. This problem is not new, previous works (see Section 1.3) have attempted to address this problem for a limited set of devices or networks. We seek to avoid such limitations because these works are not suitable for large-scale deployments serving real users, and because real users cannot use existing solutions that require to either void the warranty of devices [96, 83, 116, 122], or are limited to a specific set of applications [82, 124] or ISPs [135, 123, 122, 137, 138].

In this chapter, we present *Meddle*, our user-centric approach to address this problem. First, in Section 2.1, we define our goal and detail the sub-goals that we plan to achieve with *Meddle*. Then, we detail *Meddle*'s architecture and how we achieve each of our sub-goals in Section 2.2. In Section 2.3, we present our results from controlled experiments to demonstrate that *Meddle* is practical and has minimal impact on performance and measurement fidelity. We then discuss some legal issues that need to be addressed in practical deployments in Section 2.4. Finally, we summarize the salient features of *Meddle* in Section 2.5.

2.1 Goal

The main goal of *Meddle* is *to enable all mobile Internet users to monitor and control their Internet traffic.* We use the following sub-goals to scope out our goal.

1. **Agnostic to OS, apps, ISP, and access technology.** Meddle must work regardless of the OS and apps installed on the mobile device. Furthermore, *Meddle* must monitor and interpose on traffic without explicit support from ISPs, and should work regardless of the access technology used by the device.
2. **Deployable.** *Meddle* must be easy to install, use, and configure, a feature important to support a large user-base. This sub-goal rules out OS instrumentation and similar warranty voiding techniques that are not easy to deploy.
3. **On-demand.** Once installed, users must be able to enable and disable *Meddle* on-the-fly. This ensures easy opt-in and easy opt-out, a feature essential for ease-of-use.
4. **Always-On.** Once enabled by the user, *Meddle* must automatically support switching between networks. In particular, it must not demand inputs from end-users on network state changes when users are on-the-move.
5. **Scalable.** *Meddle* must be able to scale to support a large user-base, a feature essential to ensure statistical significance for the research work based on *Meddle*.
6. **Traffic Agnostic Interposition.** *Meddle* must be able to manipulate and control all the Internet traffic to suit the needs of end-users, a feature required to ensure that *Meddle* is both a passive monitoring and an experimental platform. *Meddle* must

achieve this control for encrypted and plain-text traffic.

These sub-goals are to make *Meddle* user-friendly.

Indeed, there exists a trade-off between a user-friendly solution and a solution that offers a fine-grained control over mobile devices and the traffic they generate. Existing solutions that rely on instrumenting OSes and apps offer a fine grain control over mobile OSes and apps. This fine grained control is useful for academic research, however, the costs associated with this level of control includes warranty voiding the device or restricting to a specific set of apps, a cost that is too high for end-users. Unlike existing approaches, we take the path of building a user-friendly solution and test the limits of its usefulness. Specifically, we relinquish OS-level controls to focus on the Internet traffic generated by mobile devices and try to use this perspective to diagnose mobile applications, OS services, and the ISPs that serve these devices. We now detail how each of the above sub-goals governed *Meddle*'s architecture.

2.2 Architecture

To reach our goal, we observe that nearly all mobile devices support network traffic indirection via virtual private networks (VPNs). Therefore, we can build a system redirecting a device's Internet traffic through a middlebox that can interpose on this traffic. Importantly, we observe that this can be achieved without any additional support from OSes or ISPs. The key idea behind *Meddle* is to combine software middleboxes with VPNs to monitor and interpose on mobile Internet traffic.

We designed and implemented the architecture presented in Figure 2.1. We envision two scenarios in which *Meddle* can be deployed: 1) a single user deployment on a users' home-gateway or personal server, or 2) a multiple user deployment on hosted servers such as Amazon EC2 (shown in Figure 2.1).

We describe in the following, *Meddle*'s architecture. The devices, configured to use *Meddle*, tunnel all their Internet traffic through one of potentially many *Meddle* servers. *Meddle* maintains a per-device profile to determine the set of services that interpose on the tunneled network traffic. Users can enable and disable these services through a web-based interface. These device-specific policies are stored in the *Data Store*. The *Policy Manager* refers to these policies to manage the traffic that flows through its *Meddle* server. For example, a user may wish to only monitor mobile Internet traffic. In this scenario, the *Policy Manager* routes the traffic only through the *Traffic Monitor* but bypasses the *Traffic Manipulator*.

Though intuitive, this architecture leads to challenges that must be overcome to achieve our sub-goals. Specifically, the VPN infrastructure raises three important questions:

1. How ubiquitous is the VPN technology on mobile devices?
2. How to monitor all the Internet traffic flowing through *Meddle*?
3. How to modify traffic using *Meddle*?

Now, we present our answer to each of these questions.

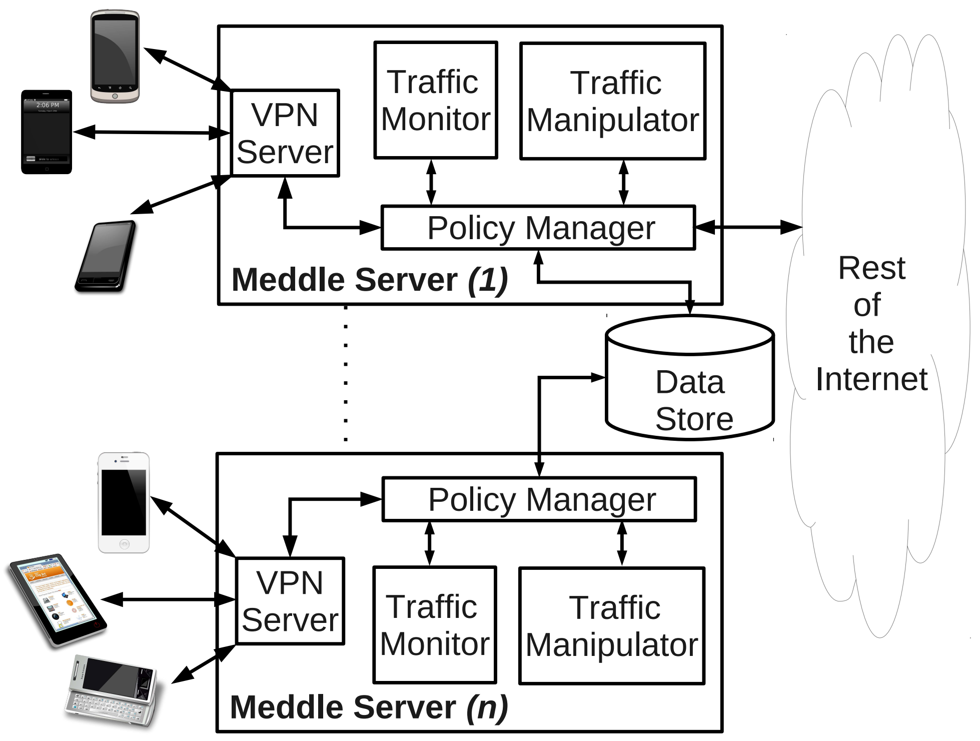


Figure 2.1: *Meddle*'s Architecture. Devices use VPN connections to tunnel all traffic to one of the potentially many *Meddle* servers. Each *Meddle* server uses a device-specific profile to determine the set of services that operate on the network traffic.

2.2.1 How ubiquitous is the VPN technology on mobile devices?

Mobile OSes and ISPs support VPNs primarily to satisfy their enterprise clients. Native support for VPNs is available on Android, BlackBerry, and iOS, three mobile OSes that represent more than 86% of the mobile devices [61]. In this dissertation, we focus on the two most popular mobile OSes: iOS and Android. These two OSes support VPN connectivity for Wi-Fi and cellular traffic—so long as the network supports IPv4. VPN tunnels on Android and iOS are transparent to the applications because traffic redirection to the VPN server is performed by the underlying OS. Thus, *Meddle* leverages on VPNs for being *agnostic to mobile OSes, ISPs, access technologies, and applications* used by the mobile device.

We now describe how we build on existing features provided by iOS and Android to provide a *deployable* system that is available *on-demand* and remains *always-on* when enabled.

Meddle on iOS Devices

All iOS devices (version 3.0 and above) support a feature called *VPN On-Demand*, which forces traffic for a specified set of domains to use VPN tunnels. This feature allows enterprises to ensure that employee's devices always use VPN tunnels when contacting specific domains, particularly those owned by the enterprise. VPN On-Demand uses suffix matching to determine which domains require a VPN connection [48]. We use each alphanumeric character (a-z, 0-9, one character per domain) as the set of domains that require a VPN connection.[1] This ensures that VPN tunnels are established before *any* network activity.

Configuring *Meddle* on iOS devices requires the user to install a single configuration file. This file contains the configurations required to drive the key exchange algorithms to establish VPN tunnels, and the patterns for the domains that require VPN tunnels. After this configuration file is installed, the iOS device uses VPN tunnels for all the Internet traffic. The user can disable *Meddle* by simply disabling the *VPN On-Demand* feature, an option exposed by iOS in the device settings screen.

The *VPN On-Demand* feature of iOS is available only for VPN tunnels that use IPsec [104] and the IKEv1 [94] key exchange protocol. This limits the options for VPN servers, for example, *Meddle* cannot use OpenVPN [35].

***Meddle* on Android Devices**

Android version 4.0 and above support VPNs, and Android version 4.2 and above support an *Always-On* VPN connection that provides the same functionality as *VPN On-Demand* for iOS. To provide the *Always ON* feature for devices running Android version 4.0 and 4.1, we use the Android API that allows applications to manage VPN tunnels. We modified the strongSwan implementation of a VPN client [40] to ensure that the VPN reconnects each time the preferred network changes, *e.g.*, when a device switches from cellular to Wi-Fi.

To configure a VPN on Android, a user needs to fill five fields. These fields are required to setup the faster IKEv2 [103] based authentication. Disabling VPN tunnels requires the users to turn off the automatic reconnect, a feature we provide in our extension to the strongSwan mobile application; a similar feature exists for *Always-On* VPN tunnels established on devices running Android 4.2 and above.

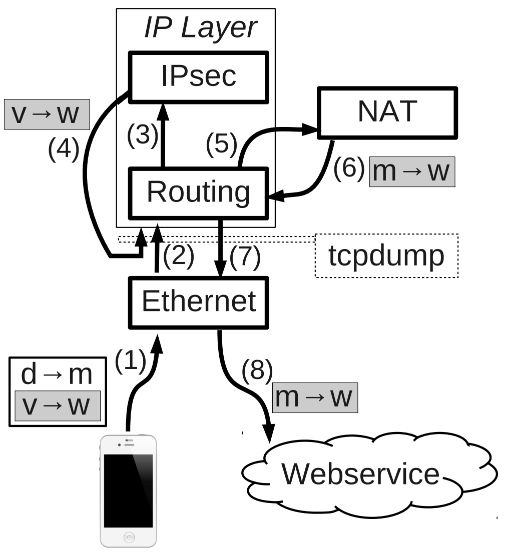
In summary, by building on the existing features provided by iOS and Android, we are able to ensure that *Meddle* is *deployable*, available *on-demand* to its clients, and *always-on* when enabled.

2.2.2 How to monitor all the Internet traffic flowing through *Meddle*?

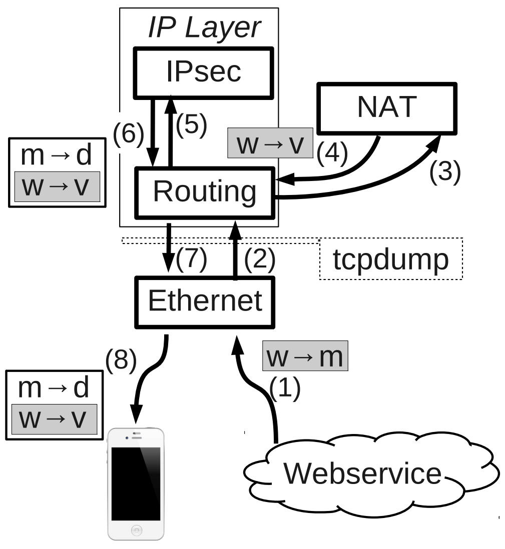
We now show how to implement a VPN proxy that supports traffic monitoring, provides an entry point to interpose on this traffic, and can be deployed on a single machine. This criteria of running on a single machine allows users the flexibility of deploying *Meddle* on their personal servers and home gateways.

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*. While the high-level design for capturing network traffic from mobile devices is straightforward, the implementation is not. In particular, the interactions between IPsec and NAT complicate our ability to map bidirectional flows to individual devices. The following paragraphs describe these challenges and how we addressed them.

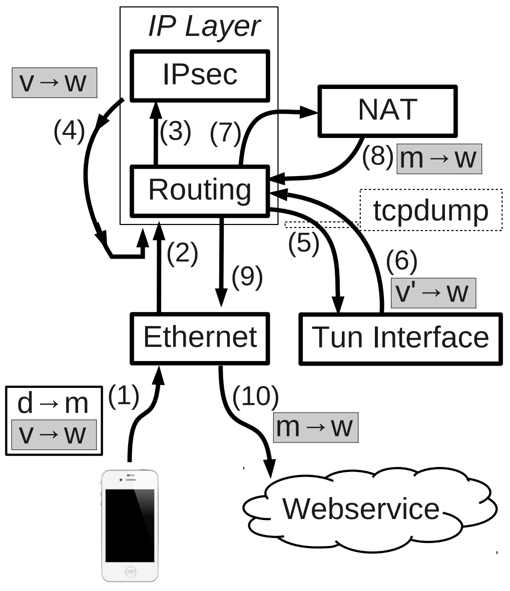
[1]We are currently working on a solution to support Internationalized domain names.



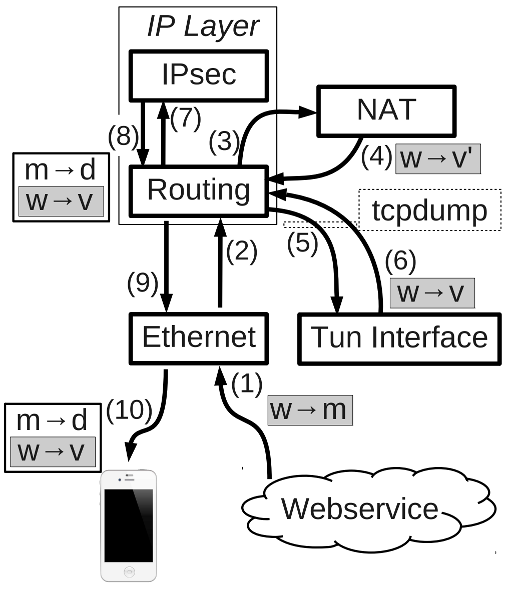
(a): Packet from mobile device. ***Tcpdump captures packets at step (2) d → m, (4) v → w, and (7) m → w.***



(b): Packet to mobile device. ***Tcpdump captures 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 captures packets at step (5) v → w, and (6) v' → w.***



(d): Packet to mobile device. ***Tcpdump captures 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 *Meddle* 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'	IP address of the mobile device for looping packets via the Tun interface.

Figure 2.2: Configuring *Meddle*'s VPN proxy to monitor IP traffic.

A VPN Proxy, apart from serving VPN tunnels, relies on NAT to proxy Internet traffic. When a mobile device establishes a VPN tunnel, the VPN server assigns it a private IP address. The mobile device therefore has two IP addresses, a private address assigned by the VPN server, and a public IP address assigned by its ISP. The VPN server maintains the mapping between the private IP address assigned to a device, its public IP address, and the unique device identifier (VPN login) in the VPN address (VPNA) table. When the VPN tunnels are established, the public IP address is used only to communicate with the VPN server while all other communication uses the private IP address. Therefore, all the traffic that would have used the public IP address when the VPN tunnel was not present, now uses the private IP address. The packets that use this private IP address are encapsulated and encrypted using IPsec and sent to the VPN server. The VPN server first decapsulates these packets and then forwards them. To forward these packets, the VPN server performs address translation because these private IP addresses cannot be used in the Internet.

We now use Figure 2.2 to show that the interactions between IPsec and NAT complicate traffic monitoring. We assume that a mobile device of public IP address d is trying to access a remote service that is located at IP address w. The packets exchanged between d and w flow through the *Meddle* server that has an IP address m. The *Meddle* server assigns a private address v to the device when the device creates the VPN tunnel, and stores this information in the VPN address (VPNA) table. This VPN address (VPNA) table maintains a mapping between the private IP address v assigned to the device and its public IP address m. In the following, we denote a packet from source s to destination d as $s \rightarrow d$.

Outbound Path: Ability to Associate a Device with its Flows

We begin with mapping flows in the forward direction ($m \rightarrow d$). Figure 2.2(a) shows the path that packets take through *Meddle*. At steps (1), (2), and (3), the encrypted datagram (in gray, $d \rightarrow m$) is passed to the IPsec module that decrypts and processes the encapsulated IP datagram ($v \rightarrow w$). After decapsulation, the kernel sees that the packet needs to be forwarded because neither the source nor the destination of the packet is its IP address, m. Forwarding decisions are taken at the IP layer, the kernel therefore sends the packet back to the IP layer, step (4). Because *Meddle* assigns private addresses to its clients, it must use NAT in step (5) to convert the private IP address v to the public IP address m. After the NAT operation, step (6), the packet is forwarded to the Internet, step (7) and step (8).

We now describe how running *tcpdump* and tracking the VPN address (VPNA) table is sufficient to sift packets based on their devices for flows in the forward direction. As shown in Figure 2.2(a), running *tcpdump* on the Ethernet interface captures packets at step (2), (4), and (7). The packet ($v \rightarrow w$) available at step (4), and the VPNA table (that contains the mapping between v and the device), are sufficient to associate the packets in the forward direction to the device from which these packets originate.

Inbound Path: The Reverse Path Mapping Problem

We now show that it is not possible to associate a mobile device with its packets in the inbound path, *i.e.*, for packets that flow to the mobile device. We refer to Figure 2.2(b), where we continue to dump packets from the Ethernet device. At step (2), with the help of *tcpdump*, we can capture the packet sent by the destination at address w to the *Meddle* server. This packet undergoes a NAT operation, step (3) and step(4), followed by IPsec encapsulation, step (5) and step (6). The packet is next seen by *tcpdump* at step (7), *i.e.*, after encapsulation. The packets captured by *tcpdump* are thus $w \rightarrow m$ (step (2)), and $m \rightarrow d$ (step (7)). If the *Meddle* server is serving more than one mobile device, then we have no way to associate a packet with a device. We need to dump the packet at step (4), but we have no access to it via the standard Linux networking stack.

To summarize, because of the complex interaction between IPsec and NAT, packets captured in the inbound path do not provide sufficient information to distinguish bidirectional flows and map them to individual devices.

Our Solution: Looping Through Tun Interface

A straightforward solution to the reverse path mapping problem is to forward traffic to a separate NAT device and dump traffic there, a solution that demands for additional hardware/VMs. This approach significantly affects scalability and limits deployability. Furthermore, users shall not be able deploy *Meddle* on their home gateways. We address this problem by virtualizing an additional network interface and routing traffic through it.

Namely, we use a Linux Tun interface and loop all packets through it. A Tun interface is a software-only interface, and unlike other network interfaces, it does not have a corresponding physical hardware component. Instead of sending traffic to the hardware components, a packet arriving at a Tun interface is sent to a userspace program that is responsible for that interface. This user-space program has complete access to the traffic flowing through the tun interface. Thus, on each *Meddle* server, we loop packets through a Tun interface for the purpose of monitoring and interposing on the network traffic flowing through the *Meddle* server.

We perform a simple NAT operation to ensure that packets do not loop indefinitely through this interface. For each mobile device, along with its private address v, *Meddle* assigns it another address v' that is internally used to loop the devices' packets through the Tun interface. For example, in the current deployment, the devices are assigned private IP addresses v from the pool 10.11.0.0/16; the v' addresses are assigned by replacing the 2nd octet in the address from 11 to 101, i.e, a device with a private address v of 10.11.11.3 shall be assigned the address 10.101.11.2 as v', a trick that avoids the need to keep another table in memory. We then use these four routing rules to enable packet forwarding through the Tun interface.[2]

1. Packets with source v are forwarded to the Tun interface after IPsec decapsulation (step(5) in Figure 2.2(c)).
2. Packets with source v' undergo NAT and are then forwarded to the Ethernet interface (step(6) to step (9) in Figure 2.2(c)).
3. Packets with destination v' are forwarded to the Tun interface (step (5) in Figure 2.2(d)).

[2]Rather than IP addresses v and v', the rules contains the pool of addresses from which v and v' are chosen.

4. Packets with destination v are forwarded to the Ethernet interface after IPsec encapsulation (step (6) to step (9) in Figure 2.2(d)).

The first two rules take care of forwarding in the outbound path ($v \rightarrow w$) while the last two rules rules take care of forwarding in the inbound path ($w \rightarrow v$).

When an inbound packet arrives at the Tun interface, our process that manages the Tun interface changes the destination address from v' to v and sends the packet to the IP layer, step (6) in Figure 2.2(d). Similarly, when an outbound packet arrives at the Tun interface, our process changes the source from v to v'. Performing *tcpdump* on the Tun interface allows us to monitor the packets $v \rightarrow w$ and $w \rightarrow v$, step (5) in Figure 2.2(c) and Figure 2.2(d). Thus, the packets captured at step (5) and the VPNA Table (mapping between v and the device) enables us to distinguish bi-directional flows and map them to individual devices.

In summary, the Tun interface provides us with an ideal vantage point to monitor and interpose on the traffic being proxied by our VPN server. The Tun interface also makes it possible to monitor and manipulate mobile Internet traffic from a single machine. In our current implementation, the *Policy Manager* in Figure 2.1 is implemented in the process managing the Tun interface. By using the Tun interface, *Meddle* can achieve its sub-goal of deployability, scalability, and capability to interpose on the traffic.

2.2.3 How to modify traffic using *Meddle*?

One of the key advantages of *Meddle* is that it allows interposing on the traffic flowing through its *Meddle* servers. As an example, we currently provide two kinds of traffic manipulation with *Meddle*.

1. Analyze the contents of SSL flows generated by mobile devices.
2. Packet filtering to block privacy invasive traffic.

Analyze SSL flows

Existing approaches that rely on ISP traces, and traffic traces collected on gateways, do not analyze the payloads of encrypted (SSL) traffic. As increasing amounts of Web traffic flows over HTTPS, we lose the ability to understand how to optimize such traffic and evaluate what private information is leaked over such encrypted tunnels. This has implications both for performance (for example, page speed optimizations) and privacy (for example, leaks of personally identifiable information (PII) over secure channels). We now describe how *Meddle* allows us to perform controlled experiments to analyze the contents of SSL flows generated by mobile devices.

First, we note that our VPN proxy, like all VPN proxies, uses a self-generated 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 [38] feature, which is essentially a man-in-the-middle operation on the secure connection.[3] As shown in Figure 2.3, 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

[3]Note that for privacy reasons we use this only for controlled experiments in the lab setting.

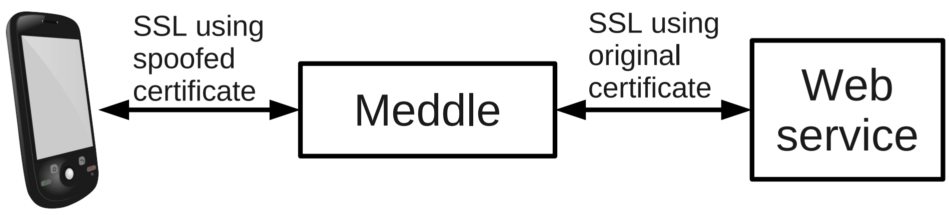


Figure 2.3: *Meddle* intercepting SSL traffic. *Meddle* can be used to perform controlled experiments that use man-in-the-middle attacks to analyze and interpose on SSL flows. During these experiments, mobile applications use the certificates issued by *Meddle* while *Meddle* uses the certificates issued by Web services.

certificate. Then the proxy establishes an SSL connection with the intended target, impersonating a mobile device. Using the traffic dumped by the *tcpdump* process and the private key generated by the squid proxy to communicate with the mobile device, we can decrypt all SSL traffic. The proxy simply forwards all non SSL traffic.

This approach fails for apps that do not trust certificates signed by unknown root authorities, a technique called pinning [5, 10]. Surprisingly, this is rarely the case. In our controlled experiments (presented in the next chapter), we observe that the Twitter and Firefox apps prevent SSL bumping by validating root certificates, while Google Chrome, Safari, Facebook, Google+, and 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 by a wide range of popular mobile applications.

Filter Personally Identifiable Information (PII) Leaks

Meddle makes it easy to implement an efficient device-wide packet-filter. We would like to point out that there exist a wide number of applications and browser plugins that offer similar filters [17, 25, 28]. However, the scope of these filters is limited to specific applications such as Web-browsers. The restrictive nature of mobile OSes require warranty voiding of the device to install device-wide traffic filters [51, 96].

Meddle currently uses a DNS-based packet filter to prevent PII leaks. Our filter builds on the past results that report on domains and services that leak PII information [96, 82, 135]. We update this list of domains based on our measurements and controlled experiments which we discuss in the next chapter. A key feature of our solution is that it works even for SSL traffic because DNS requests occur out of band from secure connections. Further, our response for the DNS request is an IP address corresponding to `localhost`, meaning that devices will generate no external network traffic when failing to resolve the ad servers. Thus, our DNS based packet filter is capable of blocking device-wide PII leaks before the information leaves the device.

Filtering misbehavior is an ongoing cat-and-mouse game, misbehaving applications and libraries that leak PII information are likely to find ways to avoid packet filters. We do not claim to have a silver-bullet to win this game, but we argue that we can follow the footsteps of ad blocking services in the desktop environment, a service that has a wide success [2, 17].

2.2.4 Architecture Summary

In this section, we showed how *Meddle* achieves the sub-goals defined in Section 2.1. To tunnel all the Internet traffic, *Meddle* uses existing features of native VPN implementations on Android and iOS to access the network perspective of mobile devices. *Meddle* servers can be deployed on a single machine, thus users have two options to deploy it: a) deploy on home-gateways, and have complete control and flexibility over personal devices, or b) use *Meddle* deployments made by researchers who can offer custom network based services.

Meddle provides a new point of control over mobile network traffic. This enables researchers to investigate what-if scenarios for the impact of new middleboxes as if they were deployed in carrier networks. Importantly, researchers and users can take advantage of these features without the support of ISPs or installing OS-specific applications.

In summary, *Meddle* provides an ideal vantage point to perform mobile traffic measurements and deploy network based services.

2.3 Discussion on Feasibility

In this section, we discuss several issues that can impact the coverage and deployability of *Meddle*. We would like to point out that these issues have a small impact on *Meddle*'s ability to monitor and interpose on mobile Internet traffic.

2.3.1 Limitations of VPN Based Traffic Redirection

Meddle relies on VPNs to redirect all Internet traffic through software defined middleboxes. The heavy reliance on VPNs implies that the restrictions imposed on VPNs affect the capabilities of *Meddle*.

1. **One Tunnel.** Currently, iOS and Android support exactly one VPN connection at a time. This allows *Meddle* to measure traffic over either the WiFi interface or the cellular interfaces, but not both at once. The vast majority of IP traffic uses only one of these interfaces, and that interface uses the VPN. An exception to this behavior is Multipath TCP (MPTCP) [70] traffic that uses more than one interface simultaneously. The iOS version 7.0 reportedly uses MPTCP to communicate with Apple servers for its SIRI service [72]. Due to the restrictions imposed by native VPN restrictions, *Meddle* cannot diagnose such traffic.
2. **Data over Voice Channels**. *Meddle* may miss some data traffic for apps and services that rely on circuit-switched channel. For example, we found evidence that iOS push notifications were being received even when IP connectivity was disabled, suggesting the use of circuit switched channel. We believe the volume of such traffic is small; however, it remains to be seen how this holds generally and over time.
3. **Proxy Location.** When traffic traverses the *Meddle* proxy, destinations will see the IP address of a *Meddle* server instead of the device's IP address. This might impact services that customize (or block access to) content according to an IP address (e.g.,

in case of localization). A local deployment of a *Meddle* server by an end-user will not have this issue.

4. **ISP Support.** We note that the incentive to allow VPN traffic is to support enterprise clients. However, ISPs might block VPN traffic, which prevents access to our current *Meddle* implementation. During our measurements, we came across only one ISP (situated in France) that blocked VPN access for mobile devices. Many ISPs deploy in-network middleboxes for traffic engineering purposes. For example, performance enhancing proxies deployed by ISPs are known to interact with TCP flows [74]. Such boxes lose the ability to implement policies of their ISP and could potentially cause mobile devices to perform sub-optimally when *Meddle* is being used.
5. **Limited ISP Characterization.** Due to its use of encrypted channels, *Meddle* cannot detect traffic differentiation or any other techniques that ISPs use to interpose on network traffic using deep packet inspection (*e.g.*, advertisement insertion [125]) or optimization (*e.g.*, downsampling content [15]). We are working on extending meddle to address this limitation, this extension is discussed in Section 5.2
6. **IPv6.** Currently, *Meddle* cannot be used on IPv6 networks, because mobile devices do not fully support IPv6. Indeed, we observe that though iOS and Android devices support IPv6, they currently do not support IPv6 traffic through VPN tunnels.
7. **Encoded Traffic.** Mobile apps and Web services are free to encode data before transmission, for example applications can exchange data using Base64 [101] encoding. Therefore, *Meddle* is exposed to data whose encoding details are available only with the mobile apps and remote hosts with whom these apps communicate. Decoding such traffic requires reverse engineering of these services. Though *Meddle* provides a vantage point to monitor and manipulate such flows, we do no automatically decode flows that use custom encoding.

2.3.2 System Overheads

Meddle uses standard and freely available software to serve mobile devices, one of its key advantages that makes it free. However, a key question is whether the system is sufficiently efficient to minimize its impact on controlled and in-the-wild experiments, and at the same time on the services offered to end-users.

We show empirically that the overheads in terms of latency, power, and data consumption are reasonable for users to adopt our systems.

Establishment delay

Mobile devices need to be authenticated by the VPN server before their traffic flows through the *Meddle* servers. This authentication is driven by the key exchange protocols of IPsec. The iOS devices use IKEv1 to manage the VPN tunnels while Android devices support both IKEv1 and IKEv2. To establish the VPN tunnel, IKEv1 requires up to 16 packets to be exchanged between the mobile device and the VPN server while IKEv2 requires 4 packets; the number of packets may vary with deployments because it depends on the encryption suites supported by the devices and the VPN server. *Meddle* uses the faster IKEv2 for

Location	**Access Technology**	**Android**		**iOS**	
		median (s)	**max. (s)**	**median (s)**	**max. (s)**
Location 1	Wifi	0.628	0.766	1.603	2.005
	Cellular (3G)	0.815	1.593	1.837	2.180
Location 2	Wifi	0.621	0.809	1.364	1.480
	Cellular (3G)	0.792	1.551	1.657	1.871

Table 2.1: Time required to establish VPN tunnels. The median and maximum values reported in this table are from performed experiments where the VPN tunnel was created 50 times from each location. *The iOS devices require more time to establish the tunnel because they rely on the slower IKEv1 protocol while the Android devices use the faster IKEv2 protocol.*

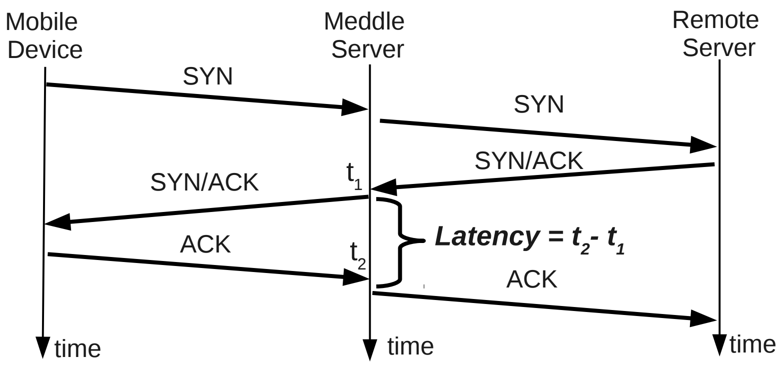
Android devices while it is forced to use the slower IKEv1 for iOS devices because iOS does not support IKEv2.

To further quantify this delay, we performed controlled experiments using one Android device (Galaxy Nexus running Android 4.2) and an iPhone 5 (running iOS 6.1). We performed our experiments from two different locations based in the same city in which our *Meddle* server was deployed. For these experiments, VPN tunnels were established for a total of 50 times during a time interval of two weeks. We present the results of our experiments in Table 2.1. The cellular experiments were performed when each device used the 3G services offered by its ISP; the same ISP served our Android device and iOS device. As expected, the iOS device requires a longer time to establish the tunnels compared to the Android device.

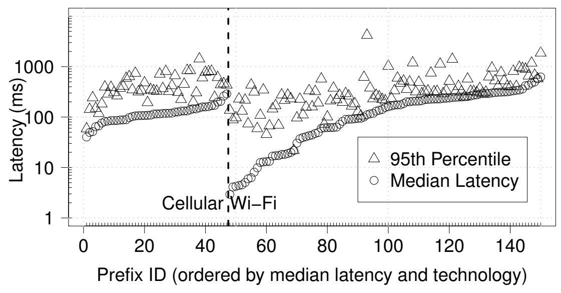
These results provide an insight on the delays that end-users might expect when using *Meddle*. Though not comprehensive, it can be used to give an estimate on the lower bound on the delay. The tunnel establishment delay can impact the performance of latency sensitive applications, however we expect the amortized cost of connecting to be small because each VPN session supports many flows.

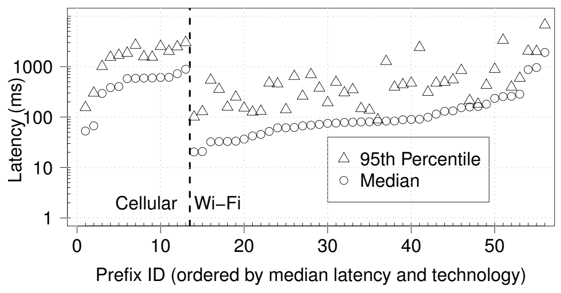
Increased Network Latency

Redirecting the traffic through a *Meddle* server may require additional hops in the path between the mobile device and the desired Web services. We performed a simple experiment to quantify the increased latency when using a deployment such as PlanetLab. For this experiment, we used data from 10 mobile phones located throughout the US and issued traceroutes from the devices to targets in Google and Facebook's networks. We then used the first non-private IP address seen from the mobile device on the path to a server. We assume that this corresponds to the first router adjacent to the mobile carrier's public Internet egress point. Note that we could not simply ping the device IP because mobile carriers filter inbound ping requests. Using this set of egress adjacencies, we determined the round-trip time from each PlanetLab site, then took the average of the nearest five sites to represent the case where a host at the nearest site is unavailable due to load or other issues. The average latency to each router was between 3 ms and 13 ms, with a median of 5 ms.



(a): Network latency computation.





(b): *mobiWest* latency. **(c): *mobiEast* latency.**

Figure 2.4: Network latency to *Meddle* servers. ***The network latency between* Meddle *servers and the devices is measured as the time between the SYN/ACK and ACK packet of a TCP three way handshake. We observe a median latency of less than 1 second across majority of prefixes through which devices tunneled their traffic.***

We also measured the latency in actual *Meddle* deployments, and observed a median latency of less than 1 second between *Meddle* servers and the mobile devices. In Figure 2.4, we present the network latency observed in *Meddle* deployments in USA, France, and China.[4] As shown in Figure 2.4(a), the network latency is computed as the time between the SYN/ACK and the ACK packets observed in the TCP handshakes. In our two datasets, *mobiWest* and *mobiEast*, we observe that the median latency between the *Meddle* servers and the mobile device is less than 1 second. Furthermore, we also observe that the network latency in cellular networks is larger than the network latency observed in Wi-Fi networks.[5] The increase in latency observed in cellular networks has various reasons which includes delays due to Radio Resource Controllers and middleboxes present in cellular networks [97, 123, 133, 136]. Thus, the network latency presented in Figure 2.4(b) and Figure 2.4(c) overestimates the redirection overhead.

In summary, when compared to RTTs of 10s or 100s of milliseconds that exist in mobile networks [97, 133], we expect a small additional latency from traversing *Meddle* servers.

[4]The two datasets, *mobiWest* and *mobiEast*, are detailed in Section 3.1.

[5]We estimate the access technology using the AS information of the prefixes. The details of our technique to estimate access technology is presented in Appendix Section A.1

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. We observed an average of 10% overhead during these 10-minute experiments. 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.

These experiments cannot capture the worst case overheads one might observe, however they do give an insight on the expected power overheads. To put the overhead in context, the iPhone 5 advertises 10 hours of browsing per charge; enabling the VPN would reduce this time to 9 hours. We use these tests to show that deploying *Meddle* does not have a significant effect on time between successive recharges.

Data Consumption

Meddle uses IPsec for datagram encryption, resulting in an encapsulation overhead for each tunneled packet. To evaluate this overhead, we use 30 days of data from 26 devices to compare encapsulated and raw packet sizes.[6] We observe a maximum encapsulation overhead of 12.8% (average approximately 10%). These overheads are negligible if *Meddle* is used to perform traffic monitoring experiments. However, in case of experiments with devices served over a limited cellular data plan, this overhead must be taken into account.

To summarize, the latency, power, and data consumption overheads are low enough to avoid significant interference with user activity. We acknowledge that the results presented in this section do not cover all possible scenarios. It is possible that end-users may face issues that we have not covered in this section. However, we believe that our results give sufficient insights on the feasibility of *Meddle* and the issues end-users might face when using it.

2.4 Legal Issues

Meddle is intended to be used by real users and it allows more than one user to share the available infrastructure. Public deployments of *Meddle* must therefore protect the people involved with *Meddle*, including the end-users. We now provide an overview on the legal issues that arise with any indirection-based deployment such as *Meddle* and how we address these issues.

Meddle is a VPN proxy for mobile devices that can be deployed on hosting services, or on home gateways. The users who deploy on their home gateway have full control over their traffic and are responsible for the *Meddle* deployment. However, when *Meddle* is deployed on a hosting service such as Amazon EC2, and the same server is shared by multiple users,

[6] These devices are part of *mobiWest* dataset, the details of which are presented in Section 3.1.

the privacy of the users whose mobile devices use *Meddle* must be protected. Similarly, the ones responsible from the *Meddle* deployment in the hosting service must be protected by any misbehaving activity performed by the mobile users.

We took the following steps to address these issues.

1. **Privacy and Trust.** *Meddle* provides a tap on network traffic that can see all the Internet traffic generated by the mobile devices, a serious risk for violating user's privacy. Users may rightfully feel uncomfortable with sending all their traffic through *Meddle*, be it in a hosting center, in the cloud, or even in their own home-gateway. To increase confidence, we are making all of our code open source so that users can inspect, modify, and extend it to suit their needs; users will have the option to run their own instance of Meddle (with their own root of trust) if they so desire.

   As discussed in the next chapter, our current deployments are part of an IRB-approved study. The IRB mandates the following steps to protect the user privacy. All the traffic captured is encrypted using Public-key cryptography before it is stored. The private key used to protect the data is not stored on the server where data is recorded. Any PII sent in the clear by applications is stripped from our datasets as soon as we identify it. Furthermore, our experiments on analyzing SSL traffic are performed on *Meddle* servers that do not serve real users.

2. **Acceptable use.** Like any proxy service, *Meddle* needs an acceptable use policy (AUP) to ensure that we are not liable for user misbehavior. We model our AUP after the one provided by EC2, one of our hosting providers. Users are informed of this AUP at the time of installation. If we are notified of an AUP violation by our ISP or hosting provider, we can block the device because each device is given a unique certificate-based credential. This makes it easy to remove offending users without disrupting compliant users.

To summarize, the users of our *Meddle* deployment are protected by an IRB, and an acceptable use policy protects us from misbehaving users.

2.5 Discussion

In this chapter, we posit that we can build a user-friendly platform to improve the transparency and end-user control in mobile networks, a solution which we call *Meddle*. The key idea behind *Meddle* is to take two well-known technologies, VPNs and middleboxes, and combine them in unintended ways for the mobile environment.

Meddle opens the mobile Internet and makes it available for measurement studies and experimentation. We showed that *Meddle* is easy to deploy on mobile devices, the overheads for *Meddle* are low, and that Meddle can scale to support a large user-base. *Meddle* thus offers a unique vantage point where researchers can offer solutions which in turn can act like incentives to recruit end-users. We now list two research directions that can benefit from our approach.

- **Peer-to-Peer (P2P) Offloading.** New web technologies such as WebRTC [67] allow client web browsers to become part of content distribution networks [143, 77]. For example, by relying on P2P technologies, Maygh [143] enable websites to distribute the cost of serving content across its visitors. These solutions and popular P2P services such as BitTorrent [7] demand multiple connections and bandwidth contributions from participating hosts, both of which are costly on mobile devices. *Meddle* can ease these costs by offloading the maintenance of P2P connections and the uploads on a *Meddle* server.

- **Privacy Preserving Middlebox.** *Meddle* has access to all the traffic from the device and can therefore be used to implement privacy preserving solutions such as Privad [92]. Furthermore, offloading the computation for preserving the privacy potentially comes with its benefits of reduced power consumption on the mobile devices.

Meddle thus enables end-users to participate in such research activities and try new mobile features without warranty voiding their devices or breaking any service agreements.

Meddle is currently in private beta with deployments in the US, France and China. User's can sign-up for an IRB approved study through `http://meddle.mobi`. In the next chapter, we discuss the results from these deployments that have served more than a 100 users.

3 Application Diagnosis

We now show that *Meddle* can be used to diagnose mobile applications and services. In particular, we show that *Meddle* can be used to not only identify apps and services that leak personally identifiable information (PII), but also block PII leaks.

We use *Meddle* to collect two classes of mobile Internet traffic: traffic from controlled experiments, and traffic from an IRB approved in-the-wild measurement. Our controlled experiments were focused on providing a ground truth information on network flows generated by apps and OS services, whereas our in-the-wild measurements help us understand the network behavior of mobile devices with real users over longer time periods. We use the traffic traces from our controlled experiments to extract signatures of apps and Web services from the protocol headers in the network flows, and we use these signatures to map network flows to the apps and services that generate them. Along with identification of apps, we use these traffic traces to identify leaks of personally identifiable information (PII). We then apply our signature based classification technique on traffic traces we collected in our *in-the-wild* measurement study. Finally, we use our results to build a tool that allow users to visualize and block PII leaks, an incentive for users to participate in future research activities based on *Meddle*.

The roadmap for this chapter is as follows. We first discuss our methodology and detail the datasets used for our analysis in Section 3.1 In our datasets, we observe HTTP and SSL to be dominant sources of Internet traffic. In Section 3.2, we focus on identifying the apps and Web services responsible for HTTP and SSL flows and present the effectiveness of our classification techniques. We then use our classification technique to identify PII leaks, and the apps and Web services responsible for these leaks in Section 3.3. In Section 3.3, we also present our tool that enables users to visualize and block PII leaks. We conclude by discussing the key takeaways from this chapter in Section 3.4.

3.1 Methodology and Dataset Description

We diagnose apps by performing offline analysis of traffic traces we collected using *Meddle*. In particular, we use tcpdump [41] to capture entire packets that traverse our *Meddle* servers. We then parse the captured traces using bro [117] and ssldump [39], and analyze these parsed traces using tools such as R [43] and MATLAB [29].

In this chapter, we use the following three datasets collected using *Meddle*.

1. ***mobiExpt.*** The *mobiExpt* dataset contains the network traffic captured when manually testing the 100 most popular iOS and Android apps in isolation. We created this dataset to map network flows to the app that generate them, and to identify PIIs in these network flows.
2. ***mobiWest.*** The *mobiWest* dataset consists of traffic traces from devices in USA and France whose users are volunteers for an ongoing IRB approved study. We gathered this dataset to understand the network behavior of devices with real users over longer time periods.

3. ***mobiEast.*** The *mobiEast* dataset consists of traffic traces from devices in China whose users are volunteers for an IRB approved study. We gathered this dataset to understand the differences between the mobile usage in China and the West.

Each of the three datasets contains the entire packets, *i.e.*, the protocol headers with their payloads. To protect the privacy of the volunteers, we used SSL-bumping (see Section 2.2.3) for the *mobiExpt* dataset only. We now detail these three datasets.

3.1.1 The *mobiExpt* Dataset

The *mobiExpt* dataset contains the traffic traces collected during controlled experiments with mobile apps. These experiments were performed to meet the following goals:

1. obtain the ground truth information on network flows generated by apps and OS services;
2. characterize the network activity for a large variety of popular apps in a lab setting;
3. detect PII leaks in the network flows and identify the popular trackers that benefit from these leaks.

The *mobiExpt* dataset contains traffic traces captured when we manually interacted with the 100 most popular iOS and Android apps in isolation. For these experiments, we use a *Meddle* server deployed in our lab to tunnel traffic from an iPhone 3Gs, running iOS 6.1.3, and a Google Nexus phone running Android 4.0.4. On this *Meddle* server, we use tcpdump to capture entire packets that flow through it. We also perform SSL-bumping (see Section 2.2.3) on this server to characterize the SSL traffic.

For each device, we begin our experiment by performing a *factory reset* on that device. This step ensures that previously installed apps do not impact the network traffic generated by the device. We then configure the mobile device with the help of dummy credentials. This unique and distinguishable set of user credentials helps us identify and extract the corresponding PII from subsequent network flows (if they are not obfuscated). We then perform the following steps.

1. We download the app from the default store (Google Play for Android and Apple App Store for iOS) and install the app. This step ensures that we are testing the latest version of the apps available at the time of conducting this experiment.
2. We then start the app and enter the required user credentials. This step is mandatory for apps that authenticate users.
3. We interact with the app for at least 10 minutes. This step allows us to characterize real user interactions with popular apps in a perfectly controlled environment.
4. Finally, we uninstall the app. This step ensures that the traffic generated by the app does not interfere with the subsequent apps being tested.

The *mobiExpt* dataset thus provide the ground-truth information for apps running in a controlled setting for a short period of time. We use the results from these experiments to analyze the traffic traces in the *mobiWest* and *mobiEast* datasets.

Parameter		***mobiWest***	***mobiEast***
Duration of measurement study	Start Date	15-Oct-2012	01-Jun-2013
	End Date	01-Sep-2013	01-Jul-2013
Number of devices	Total	26	91
	Android	11	54
	iOS	15	37
Device Activity (How many days did the devices use *Meddle*?)	minimum	5	1
	median	33	8
	maximum	315	17
Traffic Volume that devices tunneled through *Meddle*	minimum	141.5 MB	5 MB
	median	3.43 GB	96 MB
	maximum	38.08 GB	3.2 GB
	total	150.66 GB	27.8 GB

Table 3.1: Dataset Description. ***Because* Meddle *is OS agnostic, we can study a variety of devices in-the-wild. A total of 117 (26+91) devices used* Meddle *of which 65 (54+11) were Android and 52 (15+37) were iOS devices.***

3.1.2 The *mobiWest* and *mobiEast* Datasets

We collected the *mobiWest* and *mobiEast* datasets to understand the network behavior of devices with real users *in the wild* over longer time periods. We gather these datasets using four *Meddle* servers, two in the USA, one in France, and one in China. We use tcpdump on each server to capture the entire packets tunneled through it. The traffic collected on the *Meddle* servers in USA and France constitutes the *mobiWest* dataset, while the *mobiEast* dataset contains the traffic collected on the *Meddle* server deployed in China. For privacy reasons, SSL-bumping was *disabled* for all the traffic in the *mobiWest* and *mobiEast* datasets.

In Table 3.1, we summarize the *mobiWest* and *mobiEast* datasets. For the *mobiWest* dataset, we incrementally deployed two *Meddle* servers at the University of Washington, followed by one *Meddle* server at INRIA. These three servers were used to tunnel traffic from 26 devices: 11 Android devices and 15 iOS devices. The 15 iOS devices consisted of 4 iPads, 1 iPodTouch, 1 iPhone 3GS, 4 iPhone 5, and 5 iPhone 4S, while the Android devices in this dataset include the Nexus, Sony, Samsung, and Gsmart brands. For the *mobiEast* dataset, we deployed one *Meddle* server on the Aliyun [4] cloud. This server tunneled traffic from 91 devices: 54 Android devices and 37 iOS devices. Unlike the *mobiWest* dataset, the *mobiEast* dataset consists of traffic traces from Android devices manufactured by Xiaomi [53], MIUI [32], and other manufacturers that are popular among Chinese users.

Meddle can monitor traffic even when the users are asleep because *Meddle* remains *always-on* after it has been enabled by that user. For 22 of the 26 devices in the *mobiWest* dataset, *Meddle* captured at least one packet every clock hour in at least one 24 hour cycle. On the other hand, only 23 devices of 91 devices in the *mobiEast* match this criteria. One reason for this behavior is the limited cellular data plans in China; only 4 of the 91 devices claim to have a cellular data quota of more than 300 MB per month while 64 devices have a data quota of at most 150 MB per month. All the devices in the *mobiWest* dataset have a data quota of more than 1 GB per month.

We would like to point out that the *mobiExpt*, *mobiEast*, and *mobiWest* datasets have limited statistical significance. First, for the *mobiExpt* dataset, we tested only a small fraction of the apps and this test was not repeated for newer version of the apps. This limitation is important because Xu *et al.* [137] observe a heavy tail on the number of users that use a mobile app. Though our experiments covers popular apps that serve a majority of users, it cannot be used to make conclusions on the apps present in the heavy tail. Second, all the devices belonged either to students or researchers at INRIA, University of Washington, Microsoft Research Asia, or at the Peking University. Last, the current deployments served a limited number of users, most of whom used *Meddle* for a short time period. These datasets therefore have a bias towards people in Computer Science, and we cannot draw strong and generalizable conclusions based on these datasets. In spite of these limitations, we can use these datasets to gather insights on identifying apps from network traces, identifying PII leaks, and to demonstrate *Meddle*'s potential; we discuss each of these uses in the following.

3.2 Identifying Apps and Services

An important question for mobile traffic characterization is which app is responsible for the network flows. In the following, we first use the *mobiExpt*, *mobiWest*, and *mobiEast* datasets to show that apps, OS services and libraries often rely on HTTP and SSL to exchange data. This intuitive observation is not new and is supported by previous studies [86, 87, 113, 137]. We therefore focus on using the HTTP and SSL headers to identify the apps, OS services, and other services responsible for the HTTP and SSL flows. For our analysis, we use ground-truth data from the *mobiExpt* dataset 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 *mobiWest* and *mobiEast* datasets.

3.2.1 Focus on the Most Popular Protocols: HTTP and SSL

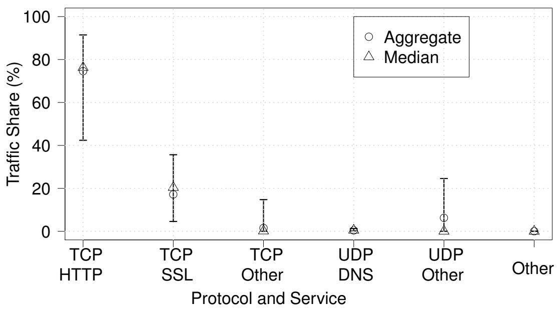
We use Bro [117] to identify the popular protocols used by mobile devices. Bro classifies flows using the protocol field in the IP header. We use this classification to label flows as either TCP, UDP, or *other*. Bro further classifies TCP and UDP flows using port numbers. For example, flows that use TCP port 80 are labeled HTTP while flows that use UDP port 53 are labeled DNS. We use this classification to label TCP flows as either HTTP, SSL, or *other*. The SSL flows include HTTPS, IMAP, and other services such as instant messaging [142] that use SSL. Similarly, we use Bro to label UDP flows as either DNS or *other*; the *other* UDP flows includes traffic from services such as Skype. This high level classification lays the foundation for identifying apps responsible for these flows.

In Figure 3.1, we summarize the results of this classification for the *mobiExpt*, *mobiWest*, and *mobiEast* datasets. There are four key take-aways from this figure.

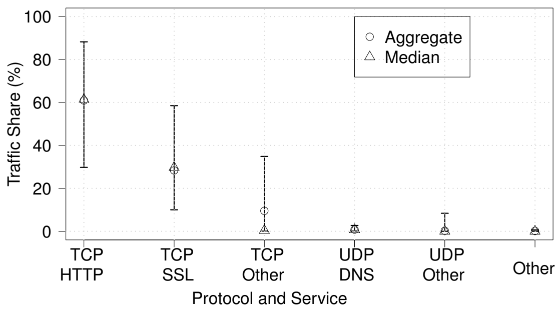
First, HTTP and SSL dominate the traffic that flows through our *Meddle* servers. This observation is important and motivates us to focus on classification of HTTP and SSL traffic.

Device	TCP			UDP		Other
	HTTP	SSL	Other	DNS	Other	
Android	88.11	11.46	0.12	0.30	<0.01	<0.01
iOS	94.17	5.73	0.04	0.05	<0.01	<0.01

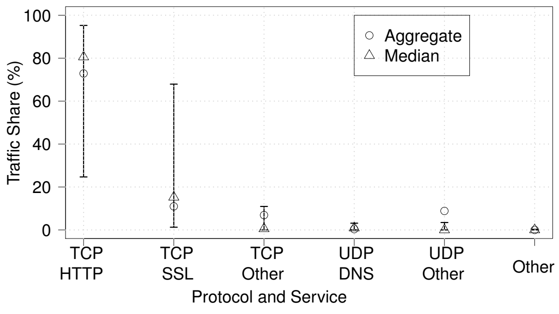
(a): ***mobiExpt. Traffic volume (percentage) when testing 100 popular iOS and Android apps.***



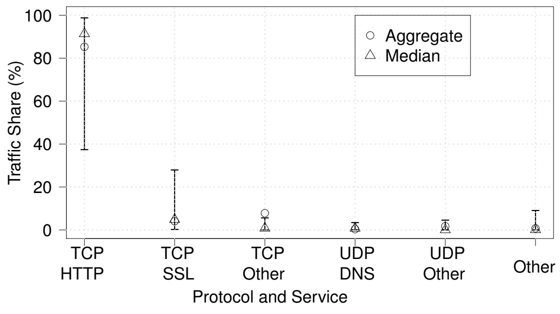
(b): ***mobiWest*** **iOS.** ***TCP other is dominated by Spotify which uses port 4070 to exchange data while UDP other is dominated by flows due to Skype.***



(c): ***mobiWest*** **Android.** ***TCP other is dominated by Spotify which uses port 4070 to exchange data.***



(d): ***mobiEast*** **iOS.** ***A large variance in HTTP and SSL traffic is because some devices are primarily used to access emails.***



(e): ***mobiEast*** **Android.** ***A higher share for HTTP traffic over SSL is because the custom ROMs such as Xiaomi and MIUI rely on HTTP to provide services such as searching for apps on their default stores.***

Figure 3.1: Traffic volume (in percentage) of popular protocols and services on Android and iOS devices. ***The error bars in figures (b)-(e) represent the 5th and 95th percentiles observed across all devices for the given protocol. Similarly, the median represents the median value for the protocol across all devices. The aggregate value for a protocol is the traffic volume that used this protocol as a fraction of the total traffic summed across all devices. We observe that TCP flows are responsible for more than 85% of aggregate traffic volume flowing through*** Meddle ***servers. HTTP and SSL are the dominant services used by Android and iOS devices in the*** mobiExpt, mobiWest, ***and*** mobiEast ***datasets.***

Second, the median and aggregate values for HTTP and SSL traffic for iOS are similar across the *mobiWest* and *mobiEast* dataset; however, the SSL traffic share for Android devices is higher in the *mobiWest* dataset in comparison to the *mobiEast* dataset. In Section 3.2.5 we show that this difference is due to the default apps in custom Android ROMs [53, 32] used by devices in the *mobiEast* dataset.

Third, for Android devices in the *mobiWest* dataset, we observe a higher share of TCP traffic labeled *other* compared to the Android devices in the *mobiEast* dataset. These flows are largely (more than 80%) due to Spotify which uses TCP Port 4070 to exchange data [22].

Finally, we observe a large variance in share of the protocols in each figure. This variance is because of the difference in device usage by the users. For example, some users use their device primarily to access emails. For such users, the share of SSL traffic will be significantly larger compared to the users that use the mobile devices to stream media content over HTTP. Similarly, users that use their devices to stream music and videos will have a larger share of HTTP traffic compared to SSL traffic. These four take-aways justify the need for a platform like *Meddle* that covers multiple OSes and can be used by real end-users.

To summarize, Figure 3.1 validates our intuition to focus on classifying HTTP and SSL flows, and identifying the apps responsible for these flows. The figure also validates the need for a platform like *Meddle* which is agnostic to the OS and apps installed on the mobile devices. We now demonstrate that previous approaches are insufficient for mapping the majority of apps to their HTTP and SSL flows, and describe several techniques to improve this mapping.

3.2.2 HTTP Traffic Classification Methodology

In this section, we describe our approach to classify HTTP traffic using the *User-Agent* and *Host* fields present in the HTTP headers, the two HTTP header fields which we show to be the most promising to identify the apps and Web-services. Previous works have used these fields in combination with other HTTP header fields to classify and analyze HTTP flows. However, their focus was to detect misbehaving sources of HTTP traffic such as bots or viruses [133, 119, 140], or to identify the category of the apps—gaming, photography,*etc.*—generating the HTTP flows [113, 137, 86, 87]. Rather that limiting ourselves to the category of apps, we now show that the HTTP headers can be used to identify the apps and Web services responsible for the HTTP flows.

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 HTTP header data.

Advantages and Disadvantages of the HTTP User-Agent

The *User-Agent* field in HTTP requests typically contains signatures of the app or the library responsible for originating the request. Our motivation to use the *User-Agent* field is based on this statement in RFC for HTTP [88]: *the User-Agent header field contains information about the user agent originating the request, which is often used by servers to help identify the scope of reported interoperability problems, to work around or tailor*

User-Agent field in HTTP header	**App Signature**
WhatsApp/2.9.3847 Android/4.1.1 Device/unknown-Full_Android_on_Crespo	WhatsApp
AppleCoreMedia/1.0.10A523 (iPad;U; CPU OS 6_0_1 like Mac OS X; en_us)	AppleCoreMedia
Dalvik/1.6.0 (Linux; U; Android 4.2.2; Nexus 4 Build/JDQ39)	NA

Table 3.2: Sample *User-Agent* strings. *The first string contains the app identifier (WhatsApp), the second hides the app and describes the OS service/library used (AppleCoreMedia), while the third does not contain any useful signature.*

***Host* field in HTTP header**	**Possible Application**
netflix348.a.nflximg.com.edgesuite.net	Netflix
r20—sn-nx57ynel.c.youtube.com	YouTube
t1-1.p-cdn.com	Pandora
itstreaming.apple.com	iTunes
cp158186-i.akamaihd.net	*Not identified*

Table 3.3: Sample *Host* strings observed for AppleCoreMedia and StageFright media libraries in the *mobiExpt* dataset. *The Host field provides hints on the possible apps behind these flows. For example, netflix348.a.nflximg.com.edgesuite.net implies a flow from Netflix, while t1-1.p-cdn.com implies a flow from Pandora. However, cp158186-i.akamaihd.net does not provide signatures of the service.*

responses to avoid particular user agent limitations, and for analytics regarding browser or operating system. Indeed, Web-services use the *User-Agent* string to customize content depending on the apps and the app versions [8]. For example, Web-services use the *User-Agent* string to recommend their native app when users access their services using mobile Web browsers. However, relying on the *User-Agent* is not sufficient to identify the app making the HTTP requests.

Indeed, the first *User-Agent* in Table 3.2 contains the information of the app, WhatsApp, while the second *User-Agent* hides the app and specifies the *AppleCoreMedia* service of iOS, and the third does not provide any useful information. Clearly, only the first *User-Agent* can be used independently to identify the apps, while the other two can only give insights on the underlying OS libraries used by the app. This limitation is the primary reason why previous works classified HTTP traffic to the granularity of the app category such as media, gaming, location services, and photography [85, 137, 113].

Advantages and Disadvantages of the HTTP Host

The *Host* field specifies the Internet host of the resource being requested. For example, the *Host* field would be *www.google.com* for a HTTP GET request made with the URL *http://www.google.fr/search?q=HTTP*. We therefore expect to find signatures of the service, and possibly the native apps, in the *Host* field. In Table 3.3 we present some of the *Host* fields we observed in the *mobiExpt* dataset for flows that contained signatures of AppleCoreMedia and StageFright streaming libraries in the *User-Agent* field. For these flows, we observe that *Host* field can provide hints on the apps and services used by the mobile devices.

In spite of its potential usefulness, the *Host* field cannot be used in isolation to identify apps. For example, 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. Thus, the *User-Agent* and *Host* fields have serious limitations when used in isolation.

Our Technique: Combination of User-Agent and Host

We rely on a combination of *User-Agent* and *Host* fields to identify apps. In particular, we give preference to the *User-Agent* field and we use the *Host* field only when the *User-Agent* cannot identify the app in isolation. Furthermore, the *Host* field may be unreliable because an app may be used to contact various Web-services. For example, the free version of TuneIn Radio app [45] communicates with radio stations selected by the user. These radio stations stream music from servers that are not managed by Tune-In Radio. We now show how we used the *User-Agent* and *Host* fields to identify apps.

We first group flows with the same *User-Agent*. For flows with the same *User-Agent*, we extract the app signatures using a set of regular expressions to filter out the auxiliary information in the *User-Agent* field. For example, the characters other than *WhatsApp* in the first *User-Agent* in Table 3.2 are not useful to identify the app. We then group flows according to their app signatures. We currently do not perform any clustering of app signatures, but we are exploring the effectiveness of the edit distance to group app signatures. As shown in Table 3.2, the app signatures extracted from *User-Agent* field may either contain 1) an app signature, 2) an OS service or library signature, or 3) no signatures. We manually group the extracted signatures in these three groups. We then use the *Host* field to identify Web services for flows that do not contain an app signature.

In particular, we use the *Host* field to identify media services. The iOS and Android devices fetch media content using the AppleCoreMedia and StageFright libraries respectively [12, 30]. We use results from our controlled experiments to extract signatures from the hostnames of servers used to stream popular media content. In particular, we used signatures for iTunes, YouTube, Netflix, Pandora, Spotify, Dailymotion, Tudou, Youkou, and Vimeo and label their flows as *Popular* media flows. The rest of the media flows are labeled as *Other* media flows.

For the rest of the HTTP flows, we search the *Host* field in the package names of Android apps used in the *mobiExpt* dataset. Android apps are written in Java and these apps typically use their reversed Internet domain name for their package names [34]. For example, the native Android app for Facebook has the package name *com.facebook.katana* while YouTube uses the package name *com.google.android.youtube*. The package name may contain some auxiliary information along with app signatures; we use a set of regular expressions to remove auxiliary information in the *Host* field. We acknowledge that Web services such as Facebook can be accessed through the mobile Web browsers. However, we believe that popular Web services are likely to be accessed through their native apps

OS	#Apps	Generates HTTP	User-Agent	Host		Combination
				Package	Organization	
iOS	100	83	79 (95.1%)	27 (32.5%)	40 (48.19%)	81 (97.5%)
And.	100	92	21 (22.8%)	27 (29.3%)	44 (47.8%)	49 (53.2%)

Table 3.4: Classification of apps based on *Host* and *User-Agent*. *A large majority of iOS apps 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 apps.*

that provide a better user experience. We now discuss the effectiveness of our approach in identifying apps and Web services.

3.2.3 Evaluation of HTTP Classification Methodology

We begin our evaluation by applying our classification on the *mobiExpt* dataset. This dataset contains the ground truth information which provides valuable insights on the effectiveness of our classification methodology. We then apply our classification on *mobiWest* and *mobiEast* datasets.

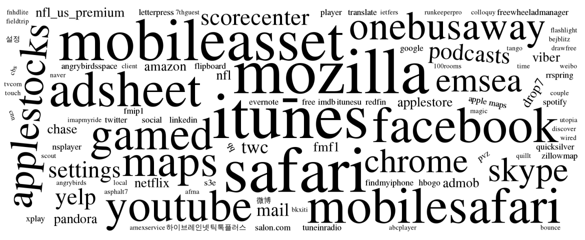
Classification of HTTP Traffic in the *mobiExpt* dataset

We use Table 3.4 to discuss the effectiveness of using the *User-Agent* and *Host* in isolation, and the added benefits of using them together to identify apps and Web services used by iOS and Android devices.

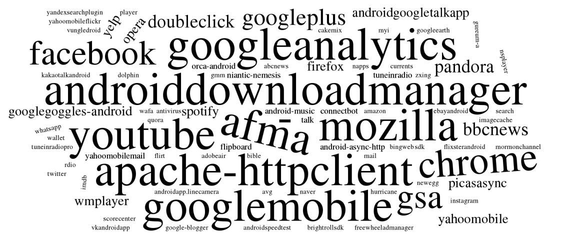
iOS Devices. First, we note that 83 of the 100 iOS apps we manually tested generated HTTP traffic.[1] Of the 83 apps, 79 apps contained signatures of the app in the *User-Agent* field. However, 55 of these 79 apps used more than one *User-Agent* field for their HTTP traffic because they use libraries such as Google Analytics and AppleCoreMedia to fetch content over HTTP. We also observe that the *Host* field uniquely identified the corresponding app for the 27 iOS apps we tested (column 5). The *Host* field can also identify the organization that released an app. For example, Zynga offers multiple games with dedicated apps that contact Zynga servers. When classifying apps according to their organization, we observe in Table 3.4 (column 6) that our classification success using only the *Host* increases to 40 iOS apps. On combining the *User-Agent* and the *Host* field, we were able to identify 81 of the 83 apps that generate HTTP traffic. However, we also observed that 79 of the 83 apps contacted other sites such as CDNs and ad sites. These hostnames of these sites did not contain any signatures of the app.

Android Devices. In Table 3.4, we observe different results for flows from Android apps released through Google Play, the default app store for Android devices. Though 92 of the 100 apps generate HTTP traffic, only 21 of the 92 apps use an app specific *User-Agent*. This number is significantly smaller than what we observed when testing iOS apps. However, on using a combination of the *User-Agent* field and the *Host* field, we were able to identify 49 apps; 89 of the 92 apps contacted other sites such as CDNs and ad sites

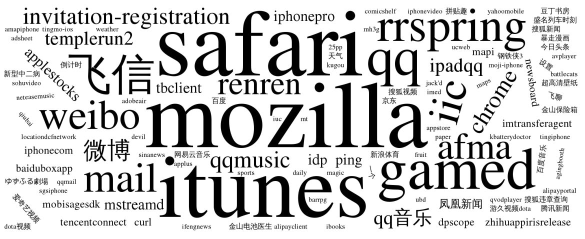
[1]The apps that do not generate HTTP traffic includes standalone apps such as Adobe Reader.



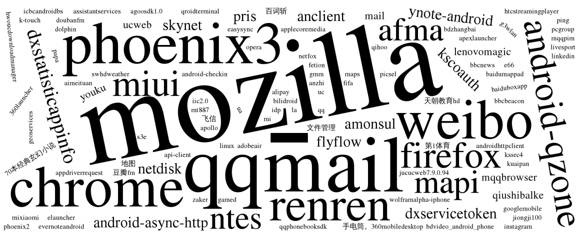
(a): *mobiWest* iOS.



(b): *mobiWest* Android.



(c): *mobiEast* iOS.



(d): *mobiEast* Android.

Figure 3.2: Word cloud of signatures in *User-Agent* field. *The size of each signature is proportional to the number of devices for which the signature was found. OS services and libraries, such as GeoServices, gamed, stagefright, and GoogleAnalytics, are some of the dominant signatures.*

which do not contain signatures of the app. 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. In Section 3.3, we show that this information is useful to identify and isolate Web sites that leak PII information.

To summarize, *User-Agent* is more effective for classifying iOS apps and *Host* is more effective for Android apps; however, neither is a complete solution when used in isolation. A topic of future work is to explore packet contents using deep packet inspection and using other HTTP header fields such as URI and Referrer. We have observed a few instances of apps identifying themselves to CDNs and ad/analytics servers in these fields and in the payload; we are working towards improving our results based on these observations. In Table 3.4, we do not present the fraction of traffic volume exchanged between the devices and remote servers because we tested each app only for 10 minutes. Instead, we discuss the effectiveness our technique for the traffic in the *mobiEast* and *mobiExpt* datasets.

Classification of HTTP Flows in the Wild (*mobiWest* and *mobiEast*)

We now discuss the effectiveness of our classification technique on the *mobiWest* and *mobiEast* datasets.

In Figure 3.2, we present the word cloud of *User-Agent* signatures extracted from HTTP flows in the *mobiWest* and *mobiEast* datasets; the size of each signature in this figure is proportional to the number of devices for which the signature was found. The key takeaway from this figure is the wide variety of apps used by devices in each dataset, and also between the iOS and Android devices within the same dataset. The figure validates the need for a platform like *Meddle* which is OS agnostic and does not require explicit support from the

Technique	**Category**	**% of iOS Traffic**		**% of Android Traffic**	
		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 (Identified)	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**

a: Classification of HTTP Traffic in the *mobiWest* dataset.

Technique	**Category**	**% of iOS Traffic**		**% of Android Traffic**	
		Bytes	**Flows**	**Bytes**	**Flows**
User-Agent	Apps	91.30	86.41	40.50	21.18
	OS Services	0.15	1.19	12.86	7.62
User-Agent + *Host*	Media (Identified)	2.11	0.84	6.87	1.16
	Media (Other)	1.81	0.49	0.31	0.01
Host	Other Apps/Web-services	0.53	2.40	28.71	42.25
Total Classified		**95.90**	**91.33**	**89.25**	**72.22**

b: Classification of HTTP Traffic in the *mobiEast* dataset.

Table 3.5: Classification of HTTP traffic in the *mobiWest* and *mobiEast* datasets. ***The strict coding guidelines enforced by Apple make the*** *User-Agent* ***field more useful in identifying iOS apps compared to Android apps in each dataset. Media traffic dominates the*** **mobiWest** ***dataset, however, the low volume of media traffic is an artifact of the small duration of the measurements. As a consequence, the iOS and Android traffic in the*** **mobiEast** ***dataset is dominated by the apps.***

OS and apps running on the mobile device.

In Figure 3.2(a), along with signatures of apps such as Facebook and YouTube, we observe signatures of OS libraries uch as *Apple Core Media* which is responsible for downloading media content. We observe that OS services such as *AppleCoreMedia*, *Android-DownloadManager*, and *Stagefright* are the most common signatures observed in the two datasets.

In each sub-figure, we observe a large number of signatures with a small font. These signatures imply that the app was used by a small number of devices in the dataset. This observation concurs with Xu *et al.* who observe that a large number of apps have a small user-base, *i.e.*, a heavy tail on the number of users that use a mobile app [137]. Xu *et al.* [137] also observe a large number of apps to be geographically dependent. The difference between the signatures in the *mobiWest* and *mobiEast* dataset concurs with this observation. We also observe that ads and analytics libraries such as Google Analytics [18] and AdSense for Mobile Applications (signature afma) [63] are popular in both datasets (see Figure 3.2(a) and Figure 3.2(c)). Similarly, the signature Mozilla, the prevalent signature in each sub-figure implies that the app fetched data using the default *User-Agent* that does not contain any app signatures. For such flows, we rely on the signatures in the *Host* field. We now discuss the effectiveness of our technique of using a combination of *User-Agent* and *Host* fields.

In Table 3.5, we observe that with our technique we were able to classify more than 91% of the iOS traffic in terms of flows and bytes, and more than 89% of the Android traffic in terms of bytes. We also observe that the *User-Agent* is more effective in identifying iOS apps compared to Android apps. We speculate that one reason for this behavior is the enforcement of stringent coding guidelines for iOS apps [11]. For the iOS devices, with the help of the *User-Agent* field, we were able to associate 85% of the HTTP flows in the *mobiWest* and 86% of HTTP flows in the *mobiEast* dataset. In comparison, only 21% of the HTTP flows from Android devices in the *mobiEast* dataset contained any app signatures in the *User-Agent* field. This observation is in line with the results of our controlled experiments.

We use the signatures of media libraries such as *AppleCoreMedia* and *Stagefright* to identify media flows, and use the *Host* field of these flows to identify the media services. Indeed, for flows with signatures of media libraries, we were able to extract signatures for popular media services such as Netflix, YouTube, Vimeo, and Pandora. By focusing on media flows, we were able to identify that more than 50% of HTTP traffic by volume in the *mobiWest* dataset using a combination of *User-Agent* and *Host*. In comparison, we do not observe a lot of media traffic for devices in the *mobiEast* dataset. We are investigating the causes for this behavior.

We use the *Host* field only for flows where the *User-Agent* field does not provide any signatures of libraries and the apps. We observe that technique is useful only for the Android devices in the *mobiEast* dataset. In contrast, we use this technique only for a small number of HTTP flows from iOS devices in the *mobiEast* dataset. As discussed previously, this is due to the enforcement of coding guidelines by Apple [11].

To summarize, the *User-Agent* field is more effective to identify HTTP flows from iOS device compared to Android devices. A combination *User-Agent* and *Host* is effective to identify media flows in iOS and Android devices. We are currently exploring the use of other HTTP header fields such as URI and Referrer and deep packet inspection to improve these results. We now present our technique to classify SSL traffic, the second largest source of Internet traffic in our datasets.

3.2.4 SSL Traffic Classification Methodology

Unlike HTTP flows, SSL flows provide limited information in plaintext that can be used to identify the apps. For the traces captured during our controlled experiments, we were able to observe HTTP requests and responses after decrypting HTTPS flows 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 *mobiWest* and *mobiEast* dataset, so we now describe how to classify SSL flows *without decryption*.

We use the TCP Port number, the SSL certificates, Server Name Identification in SSL handshakes, and DNS messages to classify SSL traffic. In particular, we use the DNS messages and subsequent SSL handshakes to determine the *hostnames* of the remote hosts

Port	*mobiWest*				*mobiEast*			
	iOS (%)		**And. (%)**		**iOS (%)**		**And. (%)**	
	Bytes	**Flows**	**Bytes**	**Flows**	**Bytes**	**Flows**	**Bytes**	**Flows**
HTTPS	92.11	79.23	96.74	90.34	89.51	78.74	87.89	65.79
Mail	4.53	7.75	0.67	0.33	9.53	16.35	6.79	8.46
Notification	2.96	10.88	2.03	6.58	0.91	4.79	2.98	19.74
Other	0.40	2.13	0.56	2.75	0.05	0.12	2.34	6.01
Total	*100*	*100*	*100*	*100*	*100*	*100*	*100*	*100*

Table 3.6: Classification of SSL Traffic based on port number. ***HTTPS is the most popular service that uses SSL, followed by Mail and Notification services.***

contacted by mobile devices. We then map these hostnames to Web services using our technique for HTTP traffic classification (see Section 3.2.2). To the best of our knowledge, we are the first to study the effectiveness of these fields in classifying SSL flows from mobile devices.

Port Number Based Classification

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 [27, 142, 16, 126]. As shown in Table 3.6, by inspecting the port numbers in SSL flows, we observe that HTTPS is the most dominant source of SSL traffic. 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. In particular, we are interested in identifying the remote hosts contacted by the mobile clients, and using their hostnames to identify the Web service. We now show how certificates, the Server Name Indication in handshakes, and DNS messages can be used to identify the hostnames of SSL sessions.

Advantages and Disadvantages of Certificates and Server Name Indication

An HTTPS session begins with a TLS handshake during which the server presents an X.509 digital certificate to the client [126]. This certificate contains the identity of the server (*e.g.*, website domain) which is digitally signed by a trusted third party. SSL certificates thus enable clients to identify and authenticate remote servers whose domain name is present in the Common Name (CN) field of the certificate. The CN field may either contain a fully qualified domain name (FQDN) such as *play.google.com* or regular expressions such as **.google.com*. A CN field with a regular expression, such as **.google.com*, hides Web services when the same domain provides multiple Web services.

The client can also specify the hostname in the Server Name Indication (SNI) extension for TLS [126]. SNI enables a server to use a single IP address to host multiple HTTPS sites. Such servers use the SNI to identify the hostname to which the client is connecting. However, the SNI cannot be used in isolation to identify the hostname because it is not widely used [81, 95].

Time	FQDN	Remote IP address	Response Index
1354557225.65	android.clients.google.com	173.194.33.4	4
1354557225.65	android.clients.google.com	173.194.33.5	5
1353279235.43	mobilemaps.clients.google.com	173.194.33.4	1
1353279235.43	mobilemaps.clients.google.com	173.194.33.5	2

Table 3.7: Sample entries in the DNS lookup table for a device in the *mobiWest* dataset. *The IP address of 173.194.33.4 is used by two hostnames, android.clients.google.com and mobilemaps.clients.google.com. The response index contains the index of the IP address in the list of IP addresses present in the DNS response. We map the remote IP address of an SSL flow to the hostname in the most recent entry with a response index of 1.*

Furthermore, SSL sessions can begin without the exchange of the hostname or domain name in the handshake. Such sessions, where the client resumes past SSL sessions using session IDs, are common because they avoid the expensive TLS handshake. SSL certificates are not exchanged during such sessions. This creates a problem when using *Meddle* to monitor traffic. In particular, the original session—where the session ID was negotiated—may not necessarily be monitored by *Meddle*. This is because users are free to enable and disable *Meddle* according to their convenience. Therefore, we have to rely on other techniques to identify the hostname for such sessions. We now discuss how we use DNS messages to overcome this issue.

DNS Classification

We identify the fully qualified domain name (FQDN) of the remote host of an SSL flow using the DNS messages between the mobile device and its DNS server. We can monitor DNS messages because *Meddle* tunnels all the Internet traffic from mobile devices. A DNS exchange consists of a DNS request, containing the FQDN to which the device wants to communicate, followed by a DNS response which contains a list of IP addresses for the requested FQDN [114]. As shown in Table 3.7, we use these DNS messages to maintain an association of the IP addresses and the FQDN. The response index contains the index of the IP address in the list of IP addresses present in the DNS response. We use this table to map the remote IP address of an SSL flow to the FQDN in the most recent entry with a response index of 1. We use a response index of 1 because in our controlled experiments we observe Android and iOS devices use the first entry in DNS response to resolve a FQDN.

This approach is similar to DN-Hunter [71]. DN-Hunter relies on the most recent FQDN that corresponds to the IP address, while we use the FQDN where the remote IP address was the first entry in the list of IP addresses. Indeed, for more than 90% of SSL flows in the *mobiWest* and *mobiEast* dataset, the latest DNS response before the TCP SYN contained the IP address as the first entry in the list of IP address. In spite of the potential usefulness of DNS messages, we give a high priority to the server-name and the certificates because DNS responses can be cached by apps, and also the DNS requests could have been made before *Meddle* was enabled.

To summarize, we use the fully qualified domain name (FQDN) of the remote servers to

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 3.8: Popular hostnames for SSL flows in *mobiExpt* dataset. *While hostnames such as imap.gmail.com, and picasaweb.google.com give clear indication of the Web services, hostnames such as www.googleapis.com hide the underlying app and Web service.*

identify the Web service. We identify the FQDN using the CN field in the SSL certificates, Server Name Identification in SSL handshakes, and DNS messages. In particular, we rely on the CN field of the certificates to identify the FQDN in SSL connections. If the fully qualified domain name is not found in the certificates, we use the Server Name Indication (SNI). We use the DNS messages only when we are not able to identify the FQDN using certificates and SNI.

Our Technique: Two Phase SSL Classification

Once we identify the hostnames, we classify the traffic in two phases. In the first phase, we use the port number and hostname to identify the service and group the traffic based on service. The five most popular groups that we found in our dataset are social network, mail, media, instant messages, and notification. For example, flows to *facebook.com*, *twitter.com*, *plus.google.com* are grouped as social networks. Traffic to well known email ports such as TCP port 993, and traffic to hosts such as *mail.google.com* are classified as mail traffic. Similarly, we use the documentation for notification services to identify the ports and hostnames used by notification services and instant messages [27, 16, 142].

In the second phase, we group hostnames that do not contain details of Web services. For example, in Table 3.8, we present some of the popular hostnames observed during our controlled experiments. While the hostname *fbcdn-photos-a.akamaihd.net* is a strong indication that the traffic is due to Facebook (due to *fbcdn*), *www.googleapis.com* hides the underlying app and Web services. We group hostnames that hide the app and Web service based on the parent organization. During manual examination of the traces, we observe three main groups: Google Services, Apple Services, and ROM services. Google Services includes flows whose remote hosts are served by Google, for example, *www.googleapis.com*. Similarly, while Apple Services includes flows to servers managed by Apple, for example, **.phobos.apple.com*. The ROM services are flows to servers managed by organizations that provide custom Android ROMs. In the *mobiEast* dataset, 21 Android devices use custom Android ROMs such as Xiaomi and MIUI. These devices used services managed by their parent organizations (Xiaomi for Xiaomi devices) instead of Google Services. We group flows to such services as ROM services. To summarize, in the second phase we label flows that do not contain details of Web services as either Google Services, Apple Services, and ROM services.

Method	*mobiWest*				*mobiEast*			
	iOS (%)		**And. (%)**		**iOS (%)**		**And. (%)**	
	Bytes	**Flows**	**Bytes**	**Flows**	**Bytes**	**Flows**	**Bytes**	**Flows**
Certificate FQDN	4.19	6.41	18.94	17.89	6.86	9.11	15.53	16.43
Certificate Regex	1.83	1.14	38.93	19.01	5.19	2.26	31.44	16.78
Server Name Indication (SNI)	4.58	4.36	14.79	9.61	13.33	11.69	6.13	3.46
DNS	92.32	88.22	93.71	90.25	90.43	79.11	91.01	93.14
Combination	95.39	90.92	98.03	92.74	96.06	92.49	95.92	94.16

Table 3.9: Fraction of SSL traffic for which the Fully Qualified Domain Name (FQDN) (or domain name for regular expressions) of the remote host was found using the Certificate, Server Name Indication (SNI), and DNS messages. ***The SSL Certificate is useful to identify less than 20% of SSL flows. Less than 20% of the SSL flows can be identified using SNI.*** **Meddle*****'s ability to monitor DNS messages fills the gap of identifying hostnames of the remote hosts in SSL flows.***

Though this classification is crude, it gives insights on the key sources of SSL traffic. We now discuss the effectiveness of our approach, and the insights obtained from this classification.

3.2.5 Evaluation of SSL Traffic Classification Methodology

We now evaluate our classification technique on the traffic traces in the *mobiWest* and *mobiEast* dataset.

We begin by discussing the effectiveness of the certificate, SNI, and DNS messages in isolation to identify the Fully Qualified Domain Name (FQDN) of the remote host. In Table 3.9, we observe that Certificates can be used to identify the remote hostname for less than 20% of SSL flows. This detection rate does not improve with regular expressions; Bermudez *et al.* [71] make similar observations when testing DN-Hunter. Similarly, while SNI is used in less than 20% of SSL flows, DNS messages are useful in identifying more than 80% of the SSL flows in each dataset. One reason why DNS messages cannot identify the FQDN for all SSL flows is because devices can cache the DNS responses made before *Meddle* was enabled. These observations support the need to rely on a combination of DNS responses, certificates, and SNI to identify remote hostnames. In Table 3.9, we observe that by using our technique we were able to identify the hostnames for more than 90% of SSL traffic by flows and bytes. We now discuss how effective the hostnames are in identifying apps and Web services responsible for these SSL flows.

As discussed previously, we first group hostnames depending on the type of app and Web service. We then group flows with ambiguous hostnames according to organizations such as Google and Apple.

In Table 3.10, 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

Phase	**Category**	**% of iOS Traffic**		**% of Android Traffic**	
		Bytes	**Flows**	**Bytes**	**Flows**
Phase 1	Social Networks	12.81	7.74	35.39	19.28
	Mail	6.11	9.26	6.46	11.02
	Media	0.94	0.25	3.66	3.62
	Instant Messages	3.70	14.09	0.21	0.48
	Notifications	4.69	17.45	2.02	6.57
	Total (A)	*28.25*	*48.79*	*47.74*	*40.97*
Phase 2	Google Services	36.32	17.56	47.31	48.27
	Apple Services	25.26	28.26	<0.01	<0.01
	Total (B)	*61.58*	*45.82*	*47.31*	*48.27*
Total (A + B)		*89.83*	*94.61*	*96.10*	*89.24*

Table 3.10: Classification of SSL traffic in the *mobiWest* dataset. *The iOS and Android SSL traffic in the* mobiWest *dataset is dominated by Google and Apple Services. The share of Social Network traffic is higher for Android devices because the default photo backup services on Android devices uses the Google Plus (and Picasa) Social Network.*

Phase	**Category**	**% of iOS Traffic**		**% of Android Traffic**	
		Bytes	**Flows**	**Bytes**	**Flows**
Phase 1	Social Networks	1.31	3.41	8.93	10.68
	Mail	10.71	16.99	7.07	8.76
	Media	0.94	0.25	3.23	5.24
	Instant Messages	1.73	24.75	1.07	1.51
	Notifications	1.95	10.03	4.12	23.02
	Total (A)	*16.64*	*55.43*	*24.42*	*49.21*
Phase 2	Google Services	1.57	4.70	58.26	29.92
	Apple Services	62.16	34.88	<0.01	<0.01
	ROM Services	0	0	2.12	3.58
	Total (B)	*63.73*	*39.58*	*60.38*	*33.50*
Total (A + B)		*80.37*	*95.01*	*84.80*	*82.71*

Table 3.11: Classification of SSL traffic in the *mobiEast* dataset. *The SSL traffic from iOS devices is dominated by Apple Services. Though Google has a large share of SSL traffic, Android devices in the* mobiEast *dataset generated significantly lower SSL traffic compared to the Android devices in the* mobiWest *dataset.*

of the App and Web service. This share does not include the traffic to Google and Apple servers that we classified as Social Network, Instant Messaging, Mail, and Media. For example, the share of Social Network traffic is higher for Android devices compared to iOS devices because the default photo backup services on Android devices uses the Google Plus (and Picasa) Social Network. Google services and Apple services are therefore the largest sources of SSL traffic in our *mobiWest* dataset.

Similarly, in Table 3.11, we observe Google and Apple to be the dominant source of SSL traffic in the *mobiEast* dataset. However, the Android devices in the *mobiEast* dataset generate significantly less SSL traffic compared to their counterparts in the *mobiWest* dataset (see Figure 3.1). This implies that Google serves fewer bytes per device over SSL for Android devices in *mobiEast* dataset. One reason for this behavior is because 21 Android devices in the *mobiEast* dataset use custom ROMs such as Xiaomi and MIUI. The default apps installed on these devices access services operated by Xiaomi and MIUI instead of Google services. Furthermore, we also observe that these services use HTTP instead of HTTPS, thus resulting in a smaller volume of SSL traffic seen in Figure 3.1.

In summary, using the certificates, SNI, and DNS messages, we were able to identify the hostname of the remote hosts for more than 90% of the SSL flows. These hostnames contain signatures of the Web services and Apps responsible for these SSL flows. We observe that Google and Apple are the dominant sources of SSL traffic in both datasets. Though Apple serves approximately the same number of bytes per device over SSL, custom ROMs such as MIUI and Xiaomi reduce the number of bytes per Android devices that Google serves over SSL.

3.2.6 Discussion

Our goal was to identify the apps and Web services responsible for the network traffic flowing through *Meddle* servers. To meet this goal, we used results from our controlled experiments to obtain the ground truth information on network flows generated by apps and OS services. We then use the a combination of *User-Agent* and *Host* field to identify apps and Web services responsible for HTTP flows. Similarly, we use certificates, SNI, and DNS messages to classify SSL flows.

We observe that the *User-Agent* field is more effective to identify HTTP flows from iOS devices compared to Android devices. One reason is the coding practices mandated by Apple for iOS apps [11]. We also observe signatures of OS libraries in HTTP flows used to exchange media content. Previous works that used the *User-Agent* were therefore limited to the granularity of app category [85, 137, 113].

For flows that contain signatures of media libraries, we use the *Host* field to identify the Web services. Though *User-Agent* and *Host* are useful to identify the apps and Web services for a majority of HTTP flows, this classification is not complete. For flows that we could not classify, we observed few apps identifying themselves in the URI and Referrer fields of HTTP headers and also in the payload of HTTP POST messages. We are working on using these fields with the help of techniques used to identify bots and viruses [133, 119, 140].

For the SSL flows, we observe that Google and Apple are the dominant sources of SSL traffic in both datasets. We identify these sources using the certificates, SNI, and DNS messages exchanged before the SSL flows begin. To the best of our knowledge, we are the first to study the effectiveness of these fields in classifying SSL flows. One key observation is the reduced number of bytes per device to Google services over SSL in the *mobiEast* dataset. This observation is important because it highlights the impact of custom ROMs such as MIUI and Xiaomi which are popular in China.

The results presented in this section cannot be used to draw general conclusions because of limited statistical significance. In particular, though our techniques can be used to identify apps and Web services, we cannot provide strong conclusions on the network traffic characteristics of these apps and services. Instead we now focus on privacy invasive apps

OS	# Apps	Email	Location	Name	Password	Device ID	Contacts	IMEI
iOS	100	8	9	4	3	4	0	0
Android	100	3	10	2	1	21	0	13

Table 3.12: Summary of personally identifiable information (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.***

we encountered in each dataset.

3.3 Diagnosing Privacy Invasive Apps

Privacy has rapidly become a critical issue for networked services. In particular, the extent of tracking of users activities by ads and analytics with the help of personally identifiable information (PII) has been highlighted by previous works [127, 108, 135]. In this section, we use *Meddle* to not only identify but also filter PII leaks. Our key contributions are as follows.

1. We conduct controlled experiments to identify how apps leak PII. We also use SSL bumping to understand how this information is leaked over secure channels (in addition to those revealed in the clear).
2. We use our results to detect PII leaks in the wild and identify trackers that use PIIs to track users.
3. We provide a tool *conVis* that allows users to monitor their devices' tracking, and the apps that facilitate this tracking. The users can also use this tool to block PII leaks on their traffic that flows through *Meddle* servers.

For our analysis, we focus on *what* PII is sent, and *to whom* is the PII sent.

3.3.1 PII leaks in *mobiExpt*

For our experiments, we create 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). Indeed, some of this information is required for normal app operation; however, such information must never travel across the network in plaintext (HTTP)

In Table 3.12, we present the different PIIs leaked by both Android and iPhone apps. We observe that the IMEI, a unique identifier tied to a phone, is the most commonly leaked PII by Android apps. The IMEI number can be used to track and correlate a user's behavior across Web services. Similarly, we observe that Android apps leak the Android ID, a unique identifier tied to an Android device. In Table 3.12, we also observe that other information like contacts, emails, and passwords are leaked in the clear. The email address,

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 3.13: 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.***

the address used to sign up for the services, was leaked in the clear by 8 iOS and 3 Android apps from our set of popular apps.

Whereas only one Android app (belonging to the *Photography* category) leaked a password in the clear, we were surprised to find that three 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). This is particularly problematic if we assume a passive eavesdropper that can sniff traffic over open Wi-Fi networks. The VPN tunnels of *Meddle* can protect this data from passive eavesdroppers in Wi-Fi networks, however, *Meddle* cannot protect from sniffers deployed between the *Meddle* server and the remote server contacted by these apps.

During our experiments, we observed that PII is also sent over HTTPS. We observed user credentials (login and password) being exchanged only with the authorized sites. For example, we observe the Facebook login and password being exchanged only with the Facebook servers. In the following, we focus on device identifiers such as the IMEI and the Android device ID. In Table 3.13, 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.[2]

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

[2]We rely on ad blockers and related work to identify ads and analytics flows [1, 2, 17, 96, 135, 127]

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, for example, via hashing or encoding. Regardless, our study showed that a significant amount of PII leaks not only in the clear but also in encrypted channels.

To summarize, previous studies identified PII leaks by either instrumenting OSes [83, 96], static or dynamic analysis of app binaries [82, 73, 84], or analyzing ISP traces [135]. We show that *Meddle* can be used to study PII leaks without warranty voiding the devices, and specific support from ISPs. Furthermore, SSL bumping also allows us to look at the PII leaks over SSL. We use the results from our experiments to develop signatures (regular expressions) to identify PII leaks. We now use these signatures in the *mobiWest* and *mobiEast* datasets.

3.3.2 PII leaks in the Wild (*mobiWest* and *mobiEast*)

We now use the *mobiWest* and *mobiEast* datasets to show that PII leaks are not limited to controlled experiments and take place in the wild. We then discuss how we use *Meddle* to mitigate this problem.

In the two datasets, we observe that the app that manages the iOS homescreen (*Spring-Board*) was responsible for more than 65% of the flows that sent the location information in the clear when fetching weather information from Yahoo servers. Though location information is obvious for passive sniffers in the local Wi-Fi network of the target, it is a serious issue if the malicious entity is present within the ISP. Weather apps such as TWC and Weather are the next largest sources of location leaks in both datasets.

In addition, we observe leaks of the device ID and IMEI number in the *mobiWest* and *mobiEast* datasets. We observe that the ads and analytics sites were the most dominant recipients of these leaks. Though the social networking sites used by volunteers for the *mobiWest* dataset did not receive the IMEI and device ID in the clear, we observe that *QQ* and *Weibo*, two popular social networking services used by volunteers in the *mobiEast* dataset, leak this information along with the device Wi-Fi MAC address in the clear.

Furthermore, we observe that *RenRen*, another popular social network, receives the list of apps installed on the device in the clear. This is a serious problem because this information can be exploited to attack the device with targeted exploits. The low data quotas in China imply that users are more likely to access social networks over Wi-Fi which may be unencrypted, making them vulnerable to such targeted exploits.

In summary, we use the results from our controlled experiments to develop signatures to identify PII leaks. We use these signatures on the *mobiWest* and *mobiEast* dataset to identify the popular trackers that use PII leaks to track users. Rather than reporting these PII leaks, we use our results on PII leaks to create filters to identify and prevent such leaks. We now discuss how we allow users, who contributed to the *mobiWest* and *mobiEast* datasets, visualize and block their PII leaks.

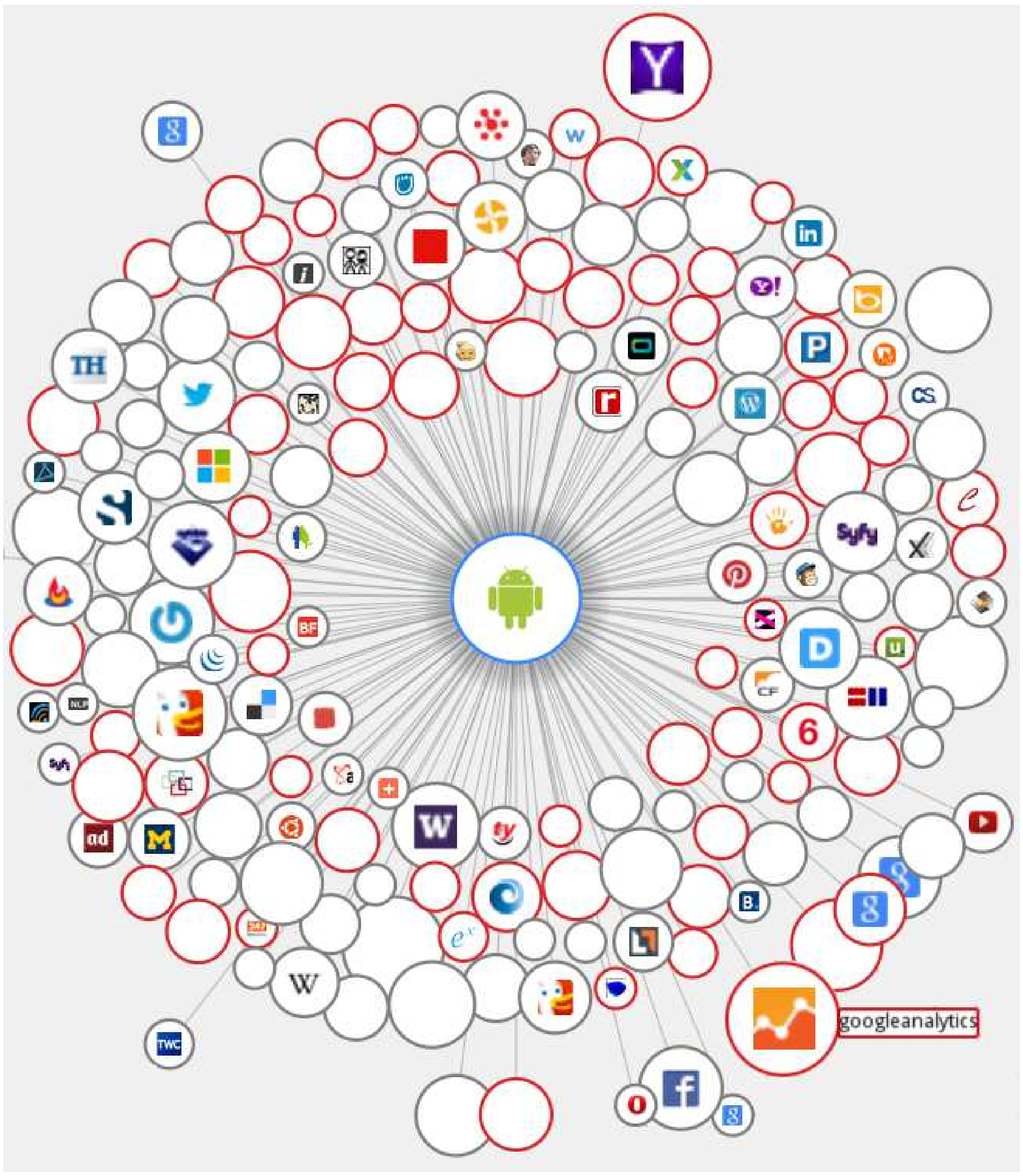


Figure 3.3: Visualizing tracking using *conVis*. ***Each circle with a shadow is an app. All other nodes are Web sites. Lines indicate the sites contacted by the app. The Red circles are Web sites such as Google Analytics that are known to track users. The size of each app is proportional to the number of flows from the app, while the size of each Web site is proportional to the number of flows to the Web site. This graph represent the flows from the default Browser app for one Android user in the*** **mobiWest** ***dataset. Some sites leaking information are shown without icons because*** *conVis* ***was unable to find their respective favicon.ico file, however we show the name of site when the circle is selected.***

3.3.3 Visualizing and Filtering PII leaks

We developed a tool, *conVis* that allows *Meddle*'s users to visualize their devices' tracking, and the apps that facilitate this tracking. *conVis* was motivated by the extensive nature of tracking and PII leaks that we observed in the *mobiWest* and *mobiEast* dataset.

The visualization of *conVis*, presented in Figure 3.3, is inspired from Mozilla Collusion [25]. Each node (circle) in the graph is either an app or a Web site contacted by the app. We use the red colored circle to represent Web sites that *potentially* leak PII. The figure represent the Web sites contacted by the default Browser app, identified using the *User-Agent*, for one Android user in the *mobiWest* dataset during a 30 day period. We observe that the user was tracked by a large number of trackers. The size of each Web site represents the number of flows between the app and the Web site. This also shows that some trackers, such as Google Analytics, are contacted more frequently than other trackers; Roesner *et al.* [127] made similar observations during their study on trackers.

Meddle also allows the user to block tracking. We use our results on PII leaks to create a list of domains that track users by leaking PIIs. *Meddle* allows users to block these sites using *Meddle*'s DNS based filter which responds to DNS requests for hosts that leak PIIs with the IP address of localhost (127.0.0.1). As discussed in Section 2.2.3, our filter is effective even from SSL traffic because DNS requests occur out of band from secure connections. However, a shortcoming of this approach is that we cannot block hostnames that are used to exchange data required for proper functioning of apps. For example, we cannot block *m.baidu.com* which offers search results and also tracks the devices' PII in the clear.

In summary, we used the results from our classification technique to allow users visualize their traffic, and also offer them the control to block tracking by filtering PII leaks. Previous works have tried to block PII leaks, however, these solutions involve instrumenting the OS to either obfuscate data or filter the access to private information [96, 118, 130]. Instead, *Meddle* allows users to participate in improving the transparency in mobile networks by offering them control over their traffic.

3.4 Discussion

The objective of building *Meddle* was to improve the transparency and end-user control over mobile devices by enabling users to monitor and interpose on the mobile Internet traffic. Interposing on network traffic requires knowledge on which app is responsible for the observed network flows, and with whom (which Web services) these apps communicate.

In this chapter, we show that it is possible to identify apps and Web services using the HTTP and SSL headers, and the DNS messages. In particular, we observe that the *User-Agent* field in the HTTP header is more useful in identifying iOS apps compared to Android apps. This observation highlights the impact of mobile OSes on the techniques used to classify mobile Internet traffic. We also compare the effectiveness of relying only on the SSL headers and show how DNS messages are useful to classify SSL flows. Our traffic classification results shows a smaller share of SSL traffic for Android devices that use custom Android ROMs, such as Xiaomi and MIUI. These ROMs are popular in China, and the Android devices that use these ROMs use fewer Google services compared to the Android devices that use the default Android ROM provided by Google. This highlights the need for a platform like *Meddle* that is independent of OSes.

We then focus on the PII leaks from popular Android and iOS apps, and use the traffic traces from our controlled experiments to built signatures for identifying PII leaks. We then used these signatures to identify PII leaks in the wild. We observe that trackers rely on HTTP and HTTPS to track users. This implies that using HTTP proxies to analyze trackers [33] will provide an incomplete picture on the tracking behavior of mobile apps. Furthermore, we observe that popular social networks in China leak PII in the clear. This makes users vulnerable to passive eavesdropping in Wi-Fi networks, a problem that can be mitigated by using *Meddle*'s VPN proxy.

Based on our results on traffic classification and identification of PII leaks, we developed a tool for end-users to visualize their Internet traffic and identify the trackers exploiting the PIIs leaked from their mobile devices. The users can also use *Meddle* to block these PII leaks. We believe this is an incentive to use *Meddle*.

The key take-away from this chapter is that we have used the research work coming from *Meddle* to create incentives to recruit users to participate in research activities. The user participation can be used to gather more insights on the behavior of mobile devices which in turn can be used to create more incentives for users. This work is part of an ongoing effort that will be continued.

4 Characterize YouTube Traffic

We now characterize YouTube traffic, one of the most dominant sources of traffic that flowed through our *Meddle* servers. Indeed, during the last decade, streaming services such as YouTube have become one of the most dominant sources of Internet traffic by volume [62, 106, 112, 37]. In spite of this popularity, the underlying strategies used by these streaming services to stream videos is largely unknown. This lack of publicly available knowledge on one of the largest sources of Internet traffic motivated us to detail the network characteristics of video streaming traffic with a focus on YouTube.

In this chapter, we show that the client side applications and the YouTube servers that stream videos control the data transfer rate during streaming sessions. This makes the traffic patterns observed during streaming sessions completely different from those observed during typical file transfers. With the help of datasets we collected in 2011 and 2013, we show that the traffic patterns observed in 2013 are completely different from those observed in 2011. Furthermore, we observe that streaming videos to mobile devices produce traffic patterns that are completely different from those observed when using desktop browsers, and that these traffic patterns change when mobile devices use Wi-Fi instead of cellular networks. We now present a generic streaming strategy that we identified during our measurements.

4.1 Streaming Strategies

In this section, we present the two different streaming strategies that we identified during our measurements. Our goal here is to synthesize the main characteristics of those strategies and present some of their advantages and disadvantages. We begin by giving an overview of a typical video streaming session. We then present the two streaming strategies and the metrics we use to characterize these strategies.

4.1.1 Phases of Data Transfer in Streaming Sessions

YouTube allows users to view videos either on personal computers (PCs), using a Web browser, or on mobile devices, using a Web browser or a native mobile app. YouTube currently supports two containers to stream videos: HTML5 [54, 93] and Adobe Flash [3]. Adobe Flash, henceforth referred to as Flash, is the default container when YouTube videos are streamed to PCs [54]. To view flash videos, PC users must install a proprietary plugin on their Web browsers. Adobe provides this plugin for PCs, however no such plugin exists for mobile devices. Because iOS and Android devices cannot stream Flash videos [59, 100], HTML5 is the default container to stream YouTube videos to mobile devices.

YouTube streams Flash and HTML5 videos over HTTP because most firewalls do not block HTTP traffic. A typical YouTube streaming session begins when a user opens a Web-page containing the video content. Along with the video content, this Web-page also contains some auxiliary information, such as the list of related videos, video ratings, and

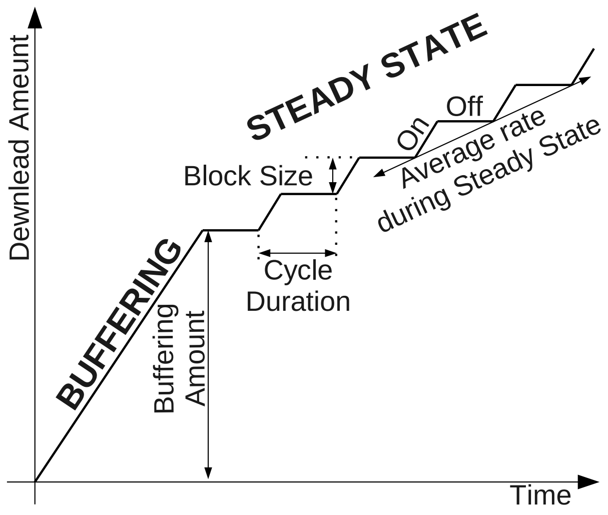


Figure 4.1: The two phases of video download. ***Video streaming begins with a buffering phase followed by a steady state phase. Cycles of ON-OFF periods in the steady state phase are used to throttle the data transfer rate.***

comments. During our measurements, we observed that the video content is transferred over HTTP while the auxiliary content is transferred over HTTP or HTTPS. We also observed that the TCP connections used to transfer the video content are different from the ones used to transfer the auxiliary content. In this chapter, we focus on the TCP connections used to transfer video content because these connections contribute to the bulk of the video streaming traffic.

In Figure 4.1, we present the time evolution of the total amount of data transferred over these TCP connections. We observe two phases: a *buffering phase* followed by a *steady state phase*.

During the buffering phase, the video content is downloaded at the end-to-end available bandwidth. The objective of the buffering phase is to ensure that the player has a sufficient amount of data to compensate for the variance in the end-to-end available bandwidth during video playback. During our measurements, we observed that the video playback *may* begin before the buffering phase ends.

During the steady state phase, the average download rate is maintained at a value that is slightly larger than the video encoding rate. This reduced transfer rate in the steady state phase ensures that the amount of video content does not overwhelm the video player while keeping a sufficient amount of data in the players buffers to compensate for variance in the end-to-end available bandwidth. The reduced rate during the steady state phase reduces the load on the streaming infrastructure, an optimization that can increase the number of videos streamed in parallel.

The steady state phase also reduces the amount of unused bytes when users' interrupt video streaming sessions due to lack of interest. Users can interrupt streaming sessions for various reasons such as poor playback quality or lack of interest in the given video. Such user interruptions are common. Gill *et al.* [91] observe that 80% of the video interruptions in a campus network are due to lack of user interest, and Finamore *et al.* [89] observe that 60% of the YouTube videos are watched for less than 20% of their duration. When a user interrupts a streaming session, the data downloaded but not used by the player is wasted. Each byte wasted implies a wastage of the network resources used to transfer that byte from the server to the users' player. Furthermore, the amount of unused bytes is also important for mobile users who rely on data plans with limited quotas.

We call the ratio of the average download rate during the steady state phase and the video encoding rate the *accumulation ratio*. An accumulation ratio close to one is desirable to prevent video playback from stopping due to empty buffers. An accumulation ratio larger than one implies that the amount of video content present in the player's buffer increases during the steady state phase, which improves the resilience to transient network congestion.

As shown in Figure 4.1, the desired accumulation ratio is achieved by periodically transferring one block of video content. These periodic transfers produce ON-OFF cycles. During each ON period, a block of data is transferred at the end-to-end available bandwidth that can be used by TCP; the TCP connection is idle during the OFF periods. The slope of the download amount during the ON periods in Figure 4.1 represents the end-to-end available bandwidth. Note that, the steady state phase will be seen only when the end-to-end available bandwidth is larger than the desired data transfer rate. A streaming session will contain only a buffering phase when the desired data transfer rate is larger than the end-to-end available bandwidth.

To summarize, video streaming applications transfer content in two phases, the *buffering phase* followed by the *steady state phase*. During the buffering phase, the video content is transferred at the end-to-end available bandwidth, while during the steady state phase, the data transfer rate is throttled to a value less than the end-to-end available bandwidth. We now use these phases to identify the two streaming strategies used to stream YouTube videos.

4.1.2 The *Crude* and *Intelligent* Streaming Strategies

The existence of a steady state phase implies that either the remote server or the client is explicitly limiting the rate of data transfer. Based on the presence or absence of a steady state phase, we identify two streaming strategies.

1. *Crude Streaming.* For this streaming strategy, the entire video content is transferred during the buffering phase. As a consequence, there is no steady state phase. The advantage of this strategy is that it requires no complex engineering at the server and the client because the video streaming session can be considered as a simple file transfer. The disadvantage is that it can overwhelm the player and cause a large amount of unused bytes when users interrupt the video playback.
2. *Intelligent Streaming.* The goal of this strategy is to ensure that the client is not overwhelmed by the amount of data sent by the server and to minimize the amount of unused bytes. The OFF periods in Figure 4.1 are observed only when the average data transfer rate is smaller than the end-to-end available bandwidth; ON-OFF cycles do not exist when the end-to-end available bandwidth is less than or equal to the average data transfer rate.

We now discuss some of the techniques that can be used to explicitly throttle the data transfer rate on TCP connections.

4.1.3 Discussion on Techniques to Throttle Data Transfer Rate

The *Intelligent* streaming strategy requires to limit the data transfer rate to less than the end-to-end available bandwidth. TCP inherently does not perform any form of rate control because it is designed to transfer data as fast as possible [121]. To achieve a goodput that is less that the TCP goodput, applications must explicitly throttle their data transfer rate. This explicit restriction of the data transfer rate by applications using TCP is commonly known as *application pacing* [90], and it can be performed either at the sender or at the receiver.

Streaming servers that stream videos can pace the data transfer by periodically sending blocks of video content. These periodic bursts can be controlled with the help of algorithms such as leaky bucket or token bucket. Furthermore, Ghobadi *et al.* [90] propose to pace TCP flows by explicitly limiting the maximum possible TCP congestion window size to a value $CW_{max} = \dfrac{RTT * E_r}{MSS}$, where, CW_{max} is the maximum congestion window size in segments, RTT is the measured round trip time, E_r is the desired rate of data transfer, and MSS is the maximum segment size.

Similarly, receivers can use the TCP receive window to pace the data transfers. For example, the applications receiving video content can periodically pull data from the TCP layer. If the application pulls data at a rate lower than the end-to-end available bandwidth, the TCP receive window will eventually become full. The event of a full window causes the TCP stack at the receiver to inform the TCP stack at the sender that the receive window is full, thus preventing the sender from sending more bytes. The receiver will continue to advertise a window size of 0 (full receive window) till the application at the receiver pulls data from its TCP stack. A pull by the application creates space in the receive window, thus enabling the TCP stack at the receiver to advertise a non-zero TCP window size to the sender. Thus, the applications that receive video content can throttle the data transfer rate by periodically pulling data from the TCP stack. Furthermore, the pacing technique proposed by Ghobadi *et al.* [90] can also be applied on TCP receivers by explicitly limiting the receive window to a value $R_{win} = E_r * RTT$.

Another technique to pace data is to use adaptive streaming techniques like HTTP live streaming (HLS) [46] or Dynamic Adaptive Streaming (DASH) [36]. HLS and DASH enable the clients to request multiple copies of the video content, each of which is encoded with a different encoding rate. These copies are downloaded in *chunks* and each chunk is requested by a separate HTTP GET request. The time between successive GET requests is determined by the video player while the server responds to each GET request by sending the data at the end-to-end available bandwidth. An advantage of this technique is that it allows the player to automatically switch between the encoding rates depending on the available end-to-end bandwidth. During our measurements, we observed that Google Chrome, Android, and iOS used adaptive streaming techniques.

To summarize, *Intelligent* streaming strategy relies on application pacing which can be performed at the sender and at the receivers. This implies that the techniques used to control the data transfer rate during the steady state phase can produce a wide range of traffic patterns. We now discuss the metrics we have used to analyze these patterns and completely characterize the streaming strategies.

4.1.4 Metrics to Characterize Streaming Strategies

The streaming sessions when using the *Crude* streaming strategy contain only the buffering phase, while the *Intelligent* streaming strategy results in sessions that contain the buffering phase and the steady state phase. We now present the metrics we used to completely characterize the streaming strategies used for YouTube videos.

1. *Buffering Amount.* The buffering amount is the amount of data downloaded in the buffering phase. It is measured as the total amount of data downloaded from the start of the streaming to the start of the first OFF period. The buffering amount is an important metric because a large buffering amount can not only overwhelm the player but also cause a large amount of unused bytes. Furthermore, the buffering amount is the size of the video when videos are streamed using the *Crude* streaming strategy.
2. *Block Size.* The block size is the amount of data transferred between consecutive OFF periods in the steady state phase. A small block size is desirable because it offers a fine grain control over the desired rate of data transfer.
3. *Accumulation ratio.* The accumulation ratio is the ratio of the average download rate during the steady state phase and the video encoding rate. An accumulation ratio that is slightly larger than one is desirable to ensure smooth playback without interruptions.

To summarize, in this section, we presented the generic streaming strategies and the metrics to detail these strategies. We now present datasets on which we used these metrics to detail the strategies used to stream YouTube videos.

4.2 Dataset Description

We used two datasets for our analysis: *you11* and *you13*, that respectively contain the traffic traces of YouTube streaming sessions from 01-Feb-2011 to 30-May-2011, and 01-Sep-2013 to 01-Oct-2013. We use these two datasets to compare the streaming strategies to PCs and mobile devices, and the changes in the streaming strategies from 2011 to 2013.

The *you11* dataset consists of 5000 Flash videos, 3000 HTML5 videos, and 2000 HD videos that were streamed to PCs, and 50 HTML5 videos that were streamed to mobile devices. The Flash videos and HD videos have encoding rates from 0.2 Mbps to 1.5 Mbps, and 0.2 Mbps to 4.8 Mbps, respectively. We extract the encoding rates of Flash videos from the header of the video file being streamed. During our measurements, we were unable to determine the exact encoding rate of the HTML5 videos. This is because the publicly available tools to parse the webm files [47]—the default format used by YouTube to stream HTML5 videos [54]—found an invalid entry for the frame rate [26]. We therefore estimate the encoding rate of HTML5 videos by dividing the `Content-Length` present in the HTTP response by the duration of the video. The encoding rate of the 3000 HTML5 videos was in the range of 0.2 Mbps to 2.5 Mbps.

The *you13* dataset consists of 300 Flash videos, 300 HTML5 videos, and 100 HD videos that were streamed to PCs, and 50 HTML5 videos that were streamed to mobile devices. The videos in the *you13* dataset are a subset of the videos in the *you11* dataset, therefore the range of encoding rates for the videos in the *you13* dataset are similar to those in the *you11* dataset. We use this dataset to see how the streaming strategies evolved from 2011 to 2013.

We performed our measurements from the following locations.

1. A 100 Mbps wired connection connected to the Internet through a 500 Mbps link. The wired link was used by PCs, while the the mobile devices used a 54 Mbps Wi-Fi connection and a cellular connection to connect to a *Meddle* server that was deployed in this network.
2. A 54 Mbps Wi-fi connection behind an ADSL router with a typical download rate of 7.7 Mbps and an upload rate of 1.2 Mbps. We performed measurements from this network to ensure that the results are not specific to the largely provisioned network we previously mentioned.

For the *you11* dataset, we used Internet Explorer 9 [14], Firefox 4.0 [24], and Google Chrome 10.0 [23] (henceforth referred to as Chrome) for streaming videos on PCs. These three browsers have a combined usage share of more than 80% [52]. For Flash videos, we installed the Flash plugin 10.2 in each of these browsers. For HTML5 videos, we installed the webM codec in Internet Explorer as YouTube uses webM [49] as the default container for HTML5 videos; Firefox and Chrome have a built-in support from webM. To study the streaming strategies used for mobile devices, we used an Android smart-phone (version 2.2) and an iPad (iOS version 4.2.1).

For the *you13* dataset, we used Internet Explorer 10 [13], Firefox 22.0, and Google Chrome 29.0 on our PCs and we used an Android smart-phone (version 4.0.4) and an iPhone (iOS version 6.1.3). For Flash videos, we installed the Flash plugin 11.7 in each of these browsers.

In each dataset, the mobile measurements were performed using a native YouTube app developed by YouTube for these mobile devices.

We capture the packets exchanged during video streaming in the following manner. When a PC is used to stream videos, we serially iterate through the list of videos in each dataset and perform the following steps for each video. We first start tcpdump, or windump depending on the operating system, to capture the packets exchanged. We then start a web browser with one URL of the dataset on the same machine to start the video streaming session. We stop the streaming session and the packet capture after 300 seconds. For mobile devices we performed the following steps for the *you11* dataset. We start the packet capture on a machine that can access the packets exchanged between the mobile device and the streaming server. We then start the video streaming. We stop the packet capture and streaming after 300 seconds. For the *you13* dataset, we used the *Meddle* server to capture the packets exchanged when mobile devices were used to stream videos.

Device	Application	*you11*			*you13*		
		Flash	**HTML5**	**HD**	**Flash**	**HTML5**	**HD**
Mobile	iOS	NA	✓	NA	NA	✓	NA
	Android	NA	✓	NA	NA	✓	NA
PC	Internet Explorer	✓	✓	X	✓	combination	X
	Google Chrome	✓	✓	X	✓	✓	X
	Firefox	✓	X	X	✓	X	X

Table 4.1: YouTube Streaming Strategies. ***✓ represents** Intelligent **streaming strategy while** X **represents the** Crude **streaming strategy. Streaming strategies vary with the container and the application used to stream videos. Internet Explorer changed its streaming strategy from** Intelligent **to** Crude **for some HTML5 videos.***

4.3 YouTube Streaming in the Wild

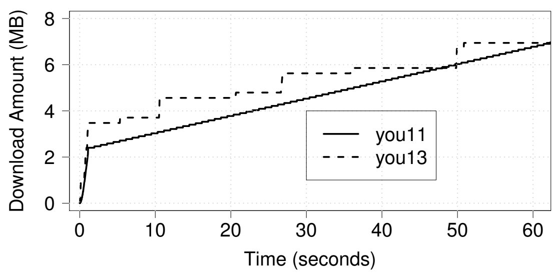
The goal of this section is to present an in depth analysis of YouTube traffic and to show that the video streaming traffic generated by YouTube can be classified by the two strategies discussed in Section 4.1.2. We also use the *you11* and *you13* datasets to detail the changes in the traffic patterns in 2013 compared to 2011. We begin by presenting possible reasons for the different implementations of streaming strategies.

Mobile devices are constrained by battery consumption, while such restrictions do not exist for PCs. This is the main reason why Flash, the default container to stream YouTube videos to PCs [54], is not supported by mobile OSes [59, 100]. Similarly, while HTML5 is the default container to stream videos to mobile devices, users must explicitly opt-in to stream HTML5 videos to PCs [54]. This observation is important and is the main reason for the different traffic patterns we discuss in this section.

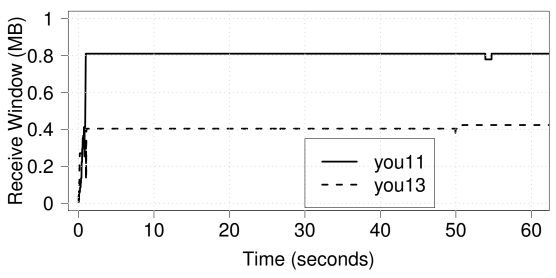
In Section 3.2.2, we observe that the native YouTube app for iOS and Android respectively use the AppleCoreMedia [12] and StageFright [30] streaming libraries. To optimize the power consumption, these streaming libraries would like control over the amount of time the radio is kept ON during the streaming session. This implies that these libraries would prefer to control the ON-OFF cycles of the steady state phase, and thus implement the streaming strategy. To satisfy mobile users, streaming services such as YouTube would prefer that the streaming strategies for HTML5 videos be implemented by these libraries. We therefore expect different implementations of the streaming strategies by iOS and Android.

For PCs, we would expect either the YouTube servers or the browsers to implement the streaming strategies. Google Chrome and YouTube belong to Google, therefore developers of Google Chrome have an inherent incentive to optimize the load on the YouTube servers. However, no such incentive exists for Firefox developers and developers of Internet Explorer. Furthermore, because Flash is the default container for PCs, and Flash is not supported by mobile devices, we would expect YouTube to implement the streaming strategy for Flash videos on their YouTube servers. We therefore expect different implementations of the streaming strategies when YouTube videos are streamed to PCs.

Indeed, Table 4.1 validates our intuition, and we observe that the streaming strategy



(a): Download Amount during a sample streaming session.



(b): Receive Window during a sample streaming session.

Figure 4.2: Representative trace for a streaming session of a Flash video. ***We observe larger blocks when streaming Flash videos in 2013 compared to 2011. We also observe that the download rate during consecutive cycles is fixed in 2011 while it changes with time in 2013.***

depends on the container and the application used to stream videos. We also observe that the streaming strategies have changed with time. When streaming Flash videos, we observe the same streaming strategy across all browsers. As we shall later see, this is because the YouTube servers explicitly throttle the data transfer rate while browsers act like regular TCP receivers. In contrast, for HTML5 videos, we observe that the streaming strategies depend on the application used. We observe different strategies because the YouTube servers do not explicitly throttle the data transfer rate, and the applications use their own techniques to throttle the data transfer rate.

We now use the metrics presented in Section 4.1.4 to detail these strategies for each application.

4.3.1 Streaming to PCs

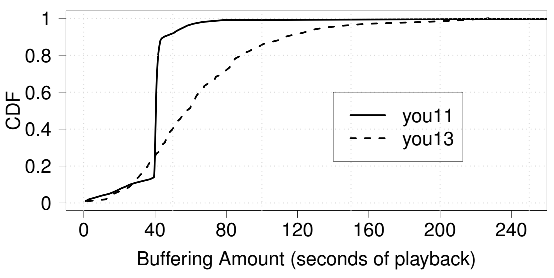
We now characterize the network traffic observed when streaming YouTube videos to PCs. In particular, we present the traffic patterns when Flash and HTML5 videos are streamed to the three most popular desktop browsers: Internet Explorer, Google Chrome, and Firefox.

Flash Videos to PC Browsers

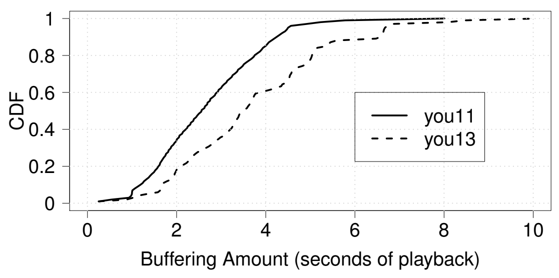
During our measurements, we observe that the traffic patterns when streaming Flash videos do not depend on the Web browsers. We now use a representative trace for one Flash video to show that the YouTube servers throttle the data transfer rate, and that the throttling technique used in 2013 is different from the one used in 2011.

In Figure 4.2(a), we see that the data is downloaded in two phases: the buffering phase followed by the steady state phase. However, during the steady state phase, we observe different step sizes for *you11* and *you13*. The step sizes represent the blocks sizes used to throttle the data transfer rate. This implies that the block sizes used to throttle data transfer rate are different in 2013 compared to 2011.

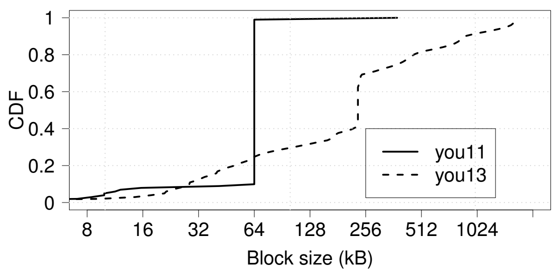
We use Figure 4.2(b), the time evolution of the advertised receive window, to show



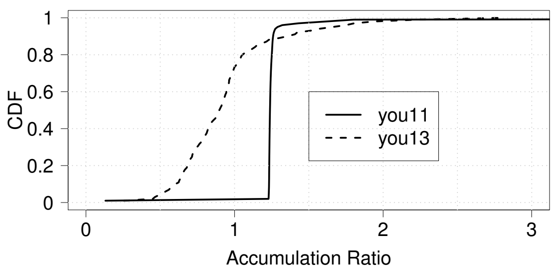
(a): Distribution of the Buffering Amount measured in units of playback time.



(b): Distribution of the Buffering Amount.



(c): Distribution of the Block Size.



(d): Distribution of the Accumulation Ratio.

Figure 4.3: Streaming Flash videos to PCs. ***More data is buffered in 2013 compared to 2011. Block sizes used when throttling the data transfer rate are larger in 2013 compared to 2011.***

that the YouTube servers throttle the data transfer rate. In this figure, we observe that the advertised receive window does not drop to zero. A non-zero receive window implies that the application receiving data (the browser) is waiting for the data to arrive from the YouTube server, which in turn implies that the YouTube servers throttle the data transfer rate.

We now detail the traffic patterns observed when streaming Flash videos and compare the differences between video streaming sessions in 2011 and 2013.

In our traces, we observe that more data is downloaded during the buffering phases in 2013 compared to 2011. In Figure 4.3(a), we observe that for 68% of the videos in the *you11* dataset, YouTube sends approximately 40 seconds worth of playback data during the buffering phase. We compute this playback time by dividing the buffering amount by the video encoding rate. We observe that the steep slope for the curve representing the *you11* dataset is not present for the *you13* dataset. This implies that the buffering amount is independent of the video encoding rate for streaming sessions in the *you13* dataset. In Figure 4.3(b), we observe that this buffering amount is not fixed and that it varies with the videos. We were unable to find any correlation of the buffering amount with the video popularity (the number of video views). We therefore speculate that this amount may be determined by the amount of time a video is seen before being interrupted by users. Regardless of these speculations, the difference in the buffering amount clearly indicates a change in the technique used to stream Flash videos in 2013 compared to 2011.

We now discuss how the steady state phase has changed in 2013 compared to 2011. In particular, in Figure 4.3(c), we observe that the block sizes used to throttle the data transfer rate are larger in 2013 compared to 2011. We observe that YouTube servers used blocks of 64 kB to throttle the data transfer rate in 2011 (labeled *you11*), while we observe that close to 40 % of the streaming sessions in 2013 use block sizes of 256 kB (labeled *you13*). These large blocks produce the large cycles which we observe in Figure 4.2(a). We now show how the YouTube servers use these blocks to reduce the amount of unused bytes and attain the desired accumulation ratio.

We compute the accumulation ratio as the slope of the line obtained by performing linear regression with time as the exploratory variable and the total amount of data downloaded as the dependent variable for samples in the steady state phase. As shown in Figure 4.3(d), for the *you11* dataset, we observe that the 64 kB blocks were used to attain an accumulation ratio of 1.25, a value that has also been reported by Ghobadi *et al.* [90]. In contrast, we observe an accumulation ratio of less than 1 for 80% of the streaming sessions in the *you13* dataset.

An accumulation ratio less than 1 and a larger buffering amount implies that the amount of data present in the players buffer decreases as the streaming session progresses. For example, in Figure 4.2(a), we observe successive cycles have a larger duration for the video in the *you13* dataset. In spite of these large cycles, we did not observe a playback freeze during our measurements.

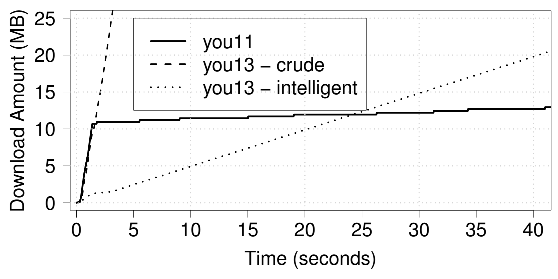
This implies that in 2013, YouTube begins by buffering a large amount of data followed by decreasing the amount of unused bytes in the players buffer as the playback progresses. This technique is completely different from the one we observed in 2011: downloading 40 seconds of video content followed by steadily accumulating the video content at 1.25 times the video encoding rate.

To summarize, YouTube servers throttle the data transfer rate for Flash videos. Though the streaming strategy of *Intelligent* streaming is used, the traffic patterns have completely changed in 2013 compared to 2011. In particular, we observe that in 2013, the amount of unused bytes decreases as playback progresses, which is completely different from what we observed in 2011. A decrease in the amount of unused bytes, and thus a potential decrease in the wastage of network resources is important because YouTube is responsible for up to 24% of the downstream Internet traffic in Europe [62]. We now show that this change in traffic patterns is not limited to Flash videos, and that similar changes are observed when streaming HTML5 videos.

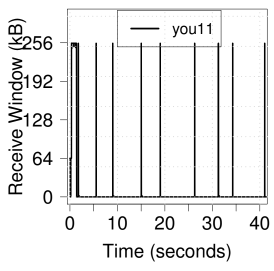
HTML5 Videos to Internet Explorer

When streaming HTML5 videos, we observe that the YouTube servers do not explicitly throttle the data transfer rate. The traffic patterns when streaming HTML5 videos therefore depend on the application used. We now detail the traffic patterns observed when streaming HTML5 videos to Internet Explorer (IE).

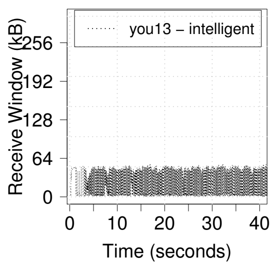
In Figure 4.4, we present a representative trace to show that IE-10 (*you13*) can use either the *Crude* streaming strategy, or the *Intelligent* streaming strategy, while IE-9 (*you11*) used



(a): Download Amount during a sample streaming session.



(b): Receive Window (IE-9).



(c): Receive Window (IE-10).

Figure 4.4: Representative trace for a streaming session of a HTML5 video with Internet Explorer (IE). ***Smaller block sizes are used in 2013 compared to 2011. We observe a combination of*** *Intelligent* ***and*** *Crude* ***streaming in 2013.***

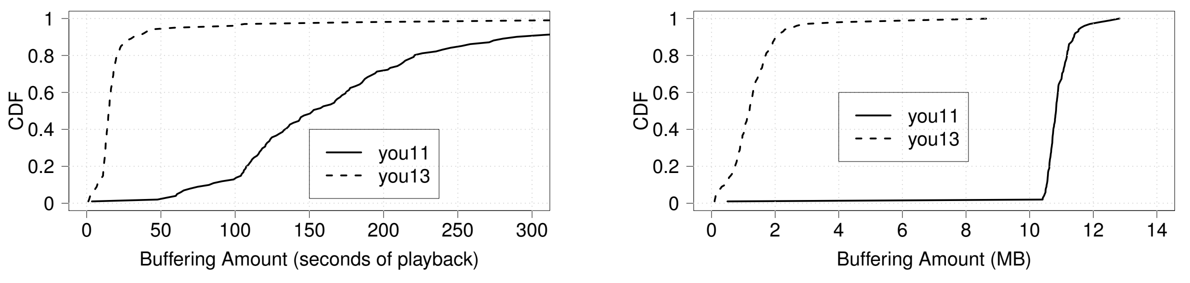
only the *Intelligent* streaming strategy. We speculate that one reason for this behavior may be the bugs associated to integrating webM with Internet Explorer [50]. Furthermore, because the videos in the *you13* dataset are a subset of the videos in the *you11* dataset, IE-10 can use the *Crude* streaming strategy for videos that would have been streamed using the *Intelligent* streaming strategy by IE-9. During our measurements, we observe that IE-10 uses the *Crude* streaming strategy for 12% of the videos in the *you13* dataset. This is a step in the wrong direction because the *Crude* streaming strategy can result in a large amount of unused bytes when users interrupt playback.

With the help of Figure 4.4(b) and Figure 4.4(c), we show that IE-9 and IE-10 uses the TCP receive window to throttle the data transfer rate in the steady state phase. The time evolution of the advertised receive window shows that the advertised receive window periodically drops to zero during the streaming session. A receive window of size zero implies that the TCP sender must wait till the receiving application has pulled the sent data from the TCP stack. This shows that Internet Explorer and not the YouTube servers is throttling the data transfer rate. In Figure 4.4(b), we observe that IE-9 periodically advertises a receive window of 256 kB. This implies that IE-9 periodically pulls 256 kB of video content from the TCP stack, thus throttling the data transfer rate using blocks of 256 kB. In contrast, in Figure 4.4(b), we observe that IE-10 periodically advertises a receive window of 50 kB. Furthermore, in Figure 4.4(a), we observe a smaller buffering amount when the *Intelligent* streaming strategy is used in 2013 compared to 2011.

Indeed, in Figure 4.5(a) and Figure 4.5(b), we observe a significantly smaller amount of data buffered during the buffering phase in 2013 compared to 2011.[1] We also observe that this buffering amount is not fixed and depends on the video. Furthermore, we observed a very weak correlation between the buffering amount and the video popularity.

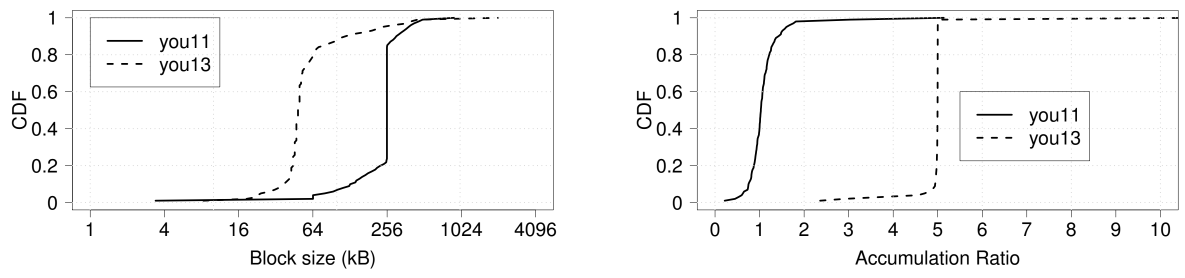
In Figure 4.5(c), we observe a block size of 50 kB to be the most common block size in the *you13* dataset compared to a block size of 256 kB that was the most commonly-used block size in the *you11* dataset. A smaller block size implies that the technique used by

[1]In Figure 4.5, we do not consider 12% of the videos in the *you13* dataset for which IE-10 used the *Crude* streaming strategy.



(a): Distribution of the Buffering Amount measured in units of playback time.

(b): Distribution of the Buffering Amount.



(c): Distribution of the Block Size.

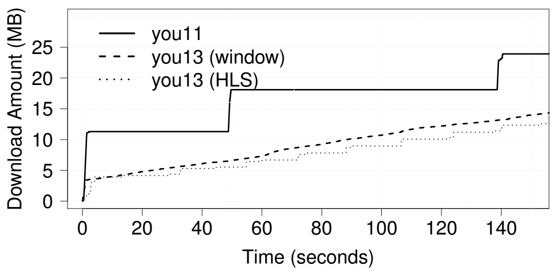
(d): Distribution of the Accumulation Ratio.

Figure 4.5: Streaming HTML5 videos to Internet Explorer (IE). ***12% of the sessions on the* you13 *dataset do not have a steady state phase. The remaining 88% of sessions use a block size of 50 kB which is smaller than the 256 kB block sizes used in 2011.***

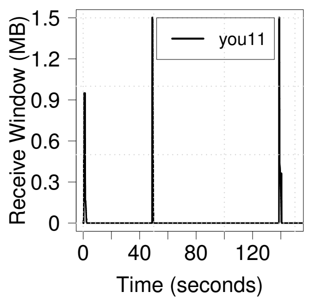
Internet Explorer to throttle the data transfer rate operates at smaller time scales in 2013 compared to 2011.

However, we observe that these small blocks are not used to reduce the amount of unused bytes. In Figure 4.5(d), we observe that the data transfer rate is throttled to attain an accumulation ratio of 5, which is much larger than the desired value of 1. An accumulation ratio of 5 implies that Internet Explorer 10 is downloading the video content at 5 times the video encoding rate; for example, a video of duration 300 seconds shall be downloaded in the first 60 seconds of streaming. In contrast, we observed an accumulation ratio close to 1 for streaming sessions in the *you11* dataset. This implies that when users interrupt videos, Internet Explorer 9 in 2011 potentially wasted a smaller amount of bytes compared to Internet Explorer 10 in 2013.

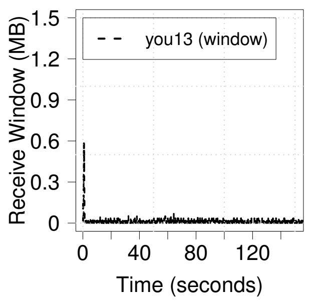
To summarize, we observe that Internet Explorer 10 operates at smaller time scales compared to Internet Explorer 9. In spite of this, Internet Explorer 10 is more aggressive in accumulating video content compared to Internet Explorer 9. This implies that the migrating from Internet Explorer 9 to Internet Explorer 10 can potentially increase the amount of unused bytes when interrupting HTML5 video streaming sessions. This observation is important because a sudden migration from Flash to HTML5 by users of Internet Explorer can increase the YouTube traffic flowing through the backbone links.



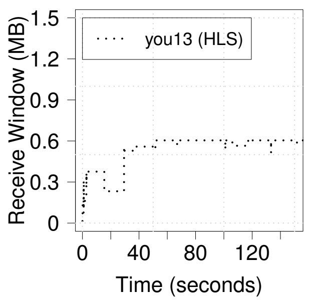
(a): Download Amount during a sample streaming session.



(b): Receive Window.



(c): Receive Window.



(d): Receive Window.

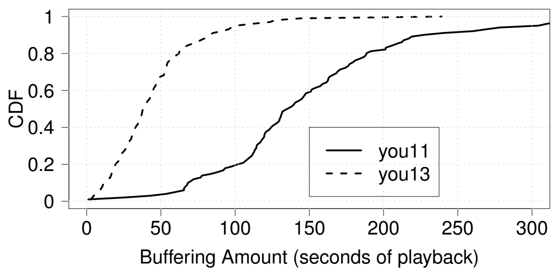
Figure 4.6: Representative trace for a streaming session of a HTML5 video with Google Chrome. ***Smaller block sizes are used in 2013 compared to 2011. We observe a combination of HLS and receive window based technique to throttle the data transfer rate in 2013.***

HTML5 Videos to Google Chrome

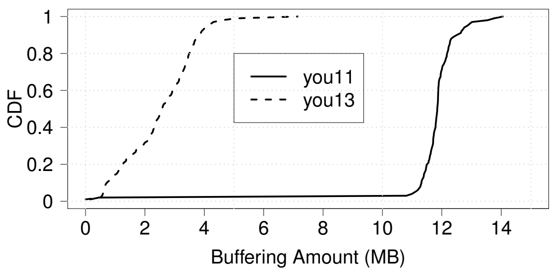
We now show that when streaming HTML5 videos in 2013, Google Chrome keeps a smaller amount of unused bytes in its buffer compared to Internet Explorer. Furthermore, the traffic patterns observed when streaming HTML5 videos to Google Chrome in 2013 are completely different from the traffic patterns observed in 2011. In particular, we observe that the technique used to throttle the data transfer rate operates at smaller time scales in 2013 compared to 2011.

In Figure 4.6, we present a representative trace to illustrate the changes between 2013 and 2011. First, we observe two different patterns for *you13* in Figure 4.6(a). This is because in 2013, Google Chrome throttles the data transfer rate by using either HTTP Live Streaming (HLS), or the TCP receive window. However, for the streaming sessions in the *you11* dataset, we observe that Google Chrome used only the TCP receive window to throttle the data transfer rate. Second, we observe larger steps for *you11* in Figure 4.6(a) compared to the steps observed for *you13*. The difference in step sizes implies that the block size used in 2013 are smaller than those used in 2011. This implies that the technique used to throttle the data transfer rate operates at smaller time scales in 2013 compared to 2011. Finally, when using HLS, we observe that the TCP receive window is not used to throttle the data transfer rate.

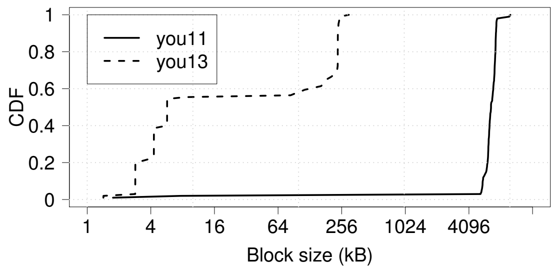
In Figure 4.7(a) and Figure 4.7(b), we observe that Google Chrome buffers smaller



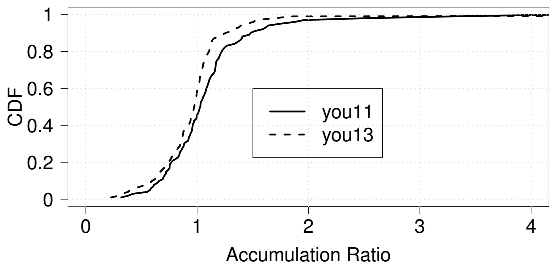
(a): Distribution of the Buffering Amount measured in units of playback time.



(b): Distribution of the Buffering Amount.



(c): Distribution of the Block Size.



(d): Distribution of the Accumulation Ratio.

Figure 4.7: Streaming HTML5 videos to Google Chrome. ***Google Chrome buffers a smaller amount of data in 2013 compared to 2011. The smaller buffering amount and an accumulation ratio close to 1 implies that Google Chrome maintains a smaller amount of unused bytes in its buffers in 2013 compared to 2011. The small block sizes in 2013 imply that the technique used to throttle the data transfer rate operates at a smaller time scale compared to 2011.***

amount of data in 2013 compared to 2011. We also observe that the buffering amount is independent of the video encoding rate. The change observed in 2013 is desirable because a small buffering amount implies that the player is not overwhelmed with video content.

In Figure 4.7(c), we observe that Google Chrome uses smaller block sizes in the steady state phase. In particular, we observe block sizes less than 64 kB when the receive window is used to throttle the data transfer rate, while block sizes of 256 kB are used when HLS is used to stream HTML5 videos. In Figure 4.7(d), we observe that these block sizes are used to ensure an accumulation ratio which is close to 1. An accumulation ratio less than 1 implies that amount of data present in the players buffer, and thus the amount of unused bytes, decreases as playback progresses.

To summarize, we observe that Google Chrome uses a smaller buffering amount and smaller block sizes in 2013 compared to 2011 when streaming HTML5 videos. The small block sizes and an accumulation ratio close to 1 implies that Google Chrome wastes fewer bytes in 2013 compared to 2011. For these reasons, using Google Chrome instead of Internet Explorer is desirable when using HTML5 to stream YouTube videos.

Crude Streaming: HTML5 Videos to Firefox and HD videos to PCs

We observe the *Crude* streaming strategy when neither the server nor the client limit the data transfer rate. The whole video is downloaded during the buffering phase; such video streaming sessions do not contain a steady state phase. We observe this strategy when streaming HTML5 videos on Firefox, and for HD videos with Flash. We do not present traces for this strategy because the traffic patterns are exactly like a TCP file transfer. For example, the time evolution of the download amount is similar to the one we observed in Figure 4.4(a) when Internet Explorer used the *Crude* streaming strategy to stream HTML5 videos.

The *Crude* streaming strategy is not desirable because it can overwhelm the player and potentially waste the network resources when users interrupt the video playback due to lack of interest. However, its primary advantage is that it uses TCP without any modification, a way in which TCP has been extensively studied and modeled [66, 99, 121]. In contrast, we shall now see how a poor implementation of the *Intelligent* streaming strategy can cause undesirable effects.

Discussion on ACK-Clocks

TCP is an ACK-clocked protocol because the TCP sender uses the acknowledgments (ACKs) as a clock to inject new packets into the network [99]. The ACK-clock is important because it is used by the TCP source to estimate the end-to-end available bandwidth.

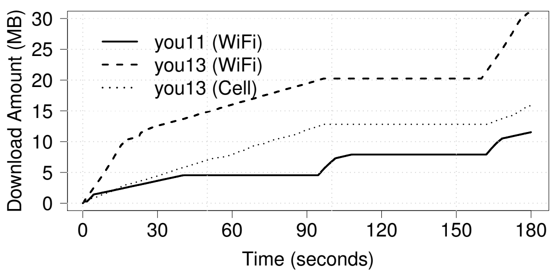
The TCP source uses this estimate of the end-to-end available bandwidth to determine the size of the TCP congestion window, the amount of bytes the source can send in one round trip time. To ensure that the TCP source does not overwhelm the network without probing the end-to-end available bandwidth, Allman *et al.* [66] recommend resetting the TCP congestion window after idle periods in the order of a retransmission timeout. Thus we expect the congestion window to decrease, and potentially reset to its initial value, after long idle periods such as the OFF periods in the ON-OFF cycles.

However, we neither observe a decrease nor a reset of the TCP congestion window during the steady state phase when streaming Flash videos in the *you11* dataset. During these streaming sessions, we observe that the entire block of 64 kB is sent in the first RTT of the ON periods. This burst of data was received even after idle periods (OFF periods) in the order of a few seconds, which implies that the TCP congestion window was not reset after the OFF periods. This observation is important because the absence of an ACK-clock can increase the loss rate in the networks. We believe the absence of ACK-clocks to be reason for the high losses that Alcock *et al.* [65] report during YouTube streaming sessions.

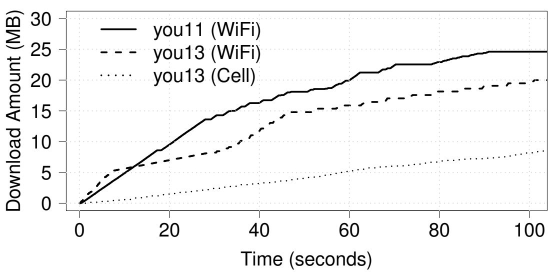
We observe ACK-clocks during streaming sessions for Flash videos in the *you13* dataset, which implies that YouTube has corrected the issue we observed in the *you11* dataset.

Summary

We observe that the traffic patterns observed when streaming YouTube videos depend on the browser and container. This observation implies that a large scale migration from one browser to another or from Flash to HTML5 can completely change the traffic patterns



(a): Sample Android streaming session.



(b): Sample iOS streaming session.

Figure 4.8: Sample streaming session to mobile devices. ***We observe iOS and Android use large blocks to download multiple copies of the video content, each of which is encoded using a different encoding rate.***

observed in the backbone links. This scenario cannot be ignored given that YouTube is responsible for a significant fraction of Internet traffic [62].

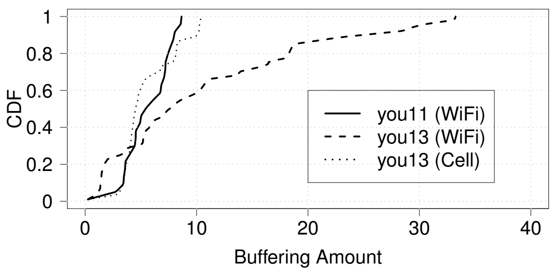
We also observe that the streaming strategies and the resulting traffic patterns have changed drastically from 2011 to 2013. In particular, we observe that Internet Explorer is more aggressive in 2013 compared to 2011. This implies that an upgrade to Internet Explorer 10 from Internet Explorer 9 can potentially waste a larger amount of bytes, and thus network resources, when users interrupt video playback. We believe this is a step in the wrong direction. Similarly, we do not observe any change in traffic patterns for HD videos streamed using Flash, and when Firefox is used to stream HTML5 videos. We have contacted the developers of Firefox, and they are currently focused on creating a generic framework to support DASH [9]. We also observe that Google Chrome throttles the data transfer rate at smaller time scales in 2013 compared to 2011. This implies that Google Chrome developers have taken steps to reduce the amount of unused bytes.

4.3.2 Streaming to Mobile Devices

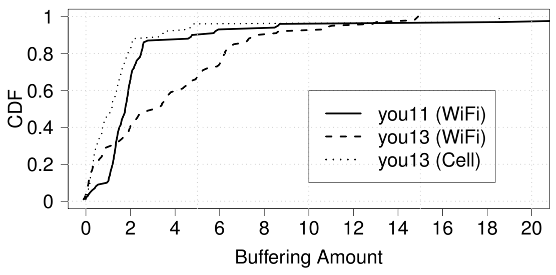
We now discuss the streaming strategies when streaming YouTube videos to mobile devices. In particular, we observe that the *Crude* streaming strategy is not deployed when streaming videos to mobile devices. However, we do observe different traffic patterns when videos are streamed to iOS and Android devices. We begin by detailing the traffic patterns observed when Android and iOS are used to stream YouTube videos followed by a comparison of these patterns with those observed when using desktop browsers.

Traffic Characteristics

In Figure 4.8, we present a sample streaming session to highlight the differences between the Android and iOS implementation of the *Intelligent* streaming strategy. In particular, we observe that the native YouTube app for Android downloads video content in larger blocks, seen as steps in the Figure 4.8(a), while we observe a smoother curve during the iOS streaming sessions. This implies that the AppleCoreMedia library for iOS and the



(a): Buffering Amount (Android).



(b): Buffering Amount (iOS).

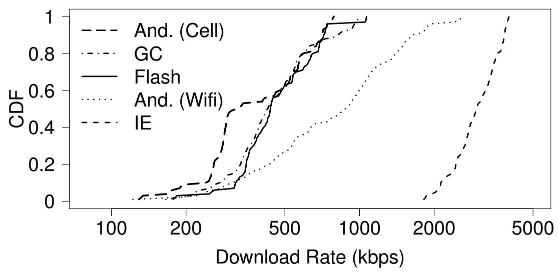
Figure 4.9: Buffering Amount. ***Multiple copies, each with a different encoding rate, is downloaded in first few seconds resulting in a large buffering amount. The YouTube app uses the buffering phase to estimate the end-to-end available bandwidth.***

StageFright library for Android use different techniques to stream HTML5 videos.

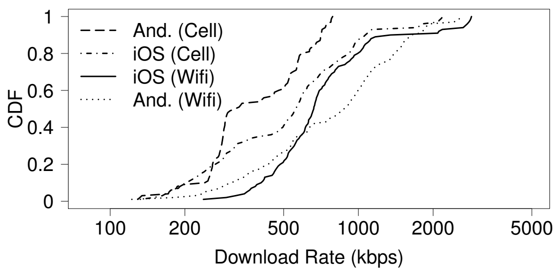
Unlike PCs, the native YouTube app for iOS and Android begins a streaming session by downloading multiple copies of a *chunk* of video content, each copy encoded using a different encoding rate. This is done to ensure playback at the highest possible encoding rate [36, 46]. Similarly, the encoding rate of each chunk downloaded after the buffering phase depends on the end-to-end available bandwidth estimated during the buffering phase [98]. This implies that the block sizes in the steady state phase depend on the estimate of the end-to-end available bandwidth, and on the different encoding rates in which the video is available for download [98]. The dependence of the buffering amount and the block sizes on a range of encoding rates implies that we cannot accurately determine the amount of playback time downloaded during the buffering phase and the accumulation ratio during the steady state phase. We therefore focus our attention on the buffering amount, *i.e.*, the amount of data downloaded while the player estimates the end-to-end available bandwidth.

In Figure 4.9, we present the distribution of the buffering amount. The buffering amount is important because a large buffering implies a large memory footprint on the resource constrained mobile devices [109, 110]. Furthermore, the players use the buffering amount to estimate the end-to-end available bandwidth and determine the best encoding rate for playback. We observe that when using Wi-Fi, Android devices download significantly more amount of data during the buffering phase in 2013, compared to Wi-Fi in 2011 and cellular in 2013. One reason for this behavior is the preference for video content at the highest encoding rate when using Wi-Fi; users have to explicitly allow high quality videos to be downloaded over cellular networks [55]. In Figure 4.9(b), we observe that iOS exhibits a similar behavior when streaming videos over Wi-Fi. The difference in the buffering amount over Wi-Fi compared to cellular networks implies that measurement studies performed over Wi-Fi networks provide an incomplete picture for the buffering amount.

To summarize, we observe that Android and iOS use the *Intelligent* streaming strategy when streaming YouTube videos. The difference in traffic patterns over Wi-Fi and cellular justify the need for a platform like *Meddle*. Because the native YouTube app for Android and iOS download content at various encoding rates, the accumulation ratio and block sizes



(a): Comparing download rates of Android and PC Browsers.



(b): Comparing Android and iOS download rates.

Figure 4.10: Comparison of download rates. ***Among desktop browsers, only Internet Explorer (IE) streaming HTML5 videos is more aggressive than Android. Android is less aggressive compared to iOS when using cellular networks because Android explicitly demands user's permission to stream high quality videos when using cellular networks.***

cannot be used to characterize the traffic. We therefore focus on comparing how aggressive these apps are compared to PC browsers.

Comparison with PCs

In Figure 4.10, we compare the average download rate observed during the first 300 seconds of the streaming session when using PC browsers and the native YouTube app. Our goal here is to see if the native YouTube app for Android and iOS is more aggressive in downloading video content compared to PC browsers. For a meaningful comparison, we consider only the download rate observed when streaming the 50 videos in the *you13* dataset to the mobile devices and PCs. For each streaming session, we compute the average download rate by dividing the total amount of data downloaded during the first 300 seconds of the streaming session by the total time required to download the data. We do not consider HTML5 videos to Flash and HD for this comparison because they do not explicitly throttle the data transfer rate and use the *Crude* streaming strategy.

In Figure 4.10(a), we observe that when using Wi-Fi, the Android app (And. Wi-Fi) is more aggressive compared to Google Chrome (GC) streaming HTML5 videos and desktop browsers streaming Flash videos. One reason for this behavior is the preference of higher encoding rates by the app when using Wi-Fi. In contrast, when using cellular networks we observe that the download rate when using Android (And. Cell) is comparable to the rate observed when streaming HTML5 videos to Google Chrome or Flash videos to desktop browsers. We also observe that Internet Explorer (IE) is the most aggressive application in downloading the video content. The reason for this behavior is the accumulation ratio of 5 which we observed in Figure 4.5(d).

In Figure 4.10(b), we observe that like Android devices, the iOS devices are more aggressive when using Wi-Fi networks than cellular networks. However, unlike the native Android app, the native iOS app does not allow the user to select lower encoding rates over cellular networks [56]. This makes the native iOS app potentially more aggressive compared to the native Android app when using cellular networks. Indeed, in Figure 4.10(b), we observe higher rates for iOS compared to Android when streaming videos over cellular networks.

To summarize, we observe that when using Wi-Fi networks, the native apps for Android and iOS are more aggressive in downloading video compared to not only Google Chrome streaming HTML5 videos but also Flash videos streamed to PCs. Furthermore, we observe that behavior over cellular networks for Android and iOS are different, primarily because the native YouTube apps for Android explicitly demands permissions from the user to stream high quality videos. These subtle differences are the cause for the vastly different traffic patterns that we observe.

4.4 Discussion

In this chapter, we presented an in-depth traffic characterization of YouTube. We identify two streaming strategies with fundamentally different traffic properties and show that implementation of these streaming strategies depend on the application and the container used. We observe that these streaming strategies produce a wide range of traffic patterns ranging from bulk TCP file transfer to non ACK-clocked traffic. These traffic patterns have been independently reported but without a detailed discussion on the factors that contribute to these patterns and the underlying streaming strategies [65, 98, 120, 129].

Unlike previous works, we characterize in detail the traffic generated by the current implementation of each streaming strategy. We also use *Meddle* to quantify the impact of access technology, and we observe that the traffic patterns are different when using Wi-Fi or cellular networks. This observation is important because studies based on Wi-Fi measurements can provide an incomplete picture on how mobile devices stream video content. We also observe that upgrading to Internet Explorer 10 from Internet Explorer 9 can potentially waste a larger amount of bytes, and thus network resources, when users interrupt video playback.

The observations we made in this chapter are important because YouTube is responsible for a large share of Internet traffic [62], and a sudden change of browser, container, or device in a large population might have a significant impact on the network traffic. Considering the very fast changes in trends, this is a real possibility, the most likely being a change from Flash over PCs to HTML5 over mobile devices.

5 Conclusions

We designed, implemented, and deployed *Meddle*, a platform to monitor and interpose on mobile Internet traffic. *Meddle* uses traffic indirection, a technique supported out-of-the-box by popular mobile OSes, to ensure participation from real users. Our reason to use crowdsourcing is in spirit of the quote, *eternal vigilance is the price of liberty*, by John Philpot Curran [79]. Indeed, liberty of all mobile users is at stake because mobile devices are capable of monitoring and influencing their activities [87, 102, 137].

In this dissertation, we demonstrate that *Meddle* can be used by researchers and end-users to improve transparency and end-user control in mobile networks. We now present the key implications of our work, followed by how *Meddle* can be used in conjunction with existing solutions to address some open research problems.

5.1 Key Implications

Our goal is *to enable all mobile users to monitor and control their Internet traffic*, and *Meddle*, our platform, is our first step in this direction. *Meddle* is built using VPNs and middleboxes to allow real users to participate in research activities without voiding their device and service warranty. In particular, *Meddle* is agnostic to the OS, ISP, and access technologies used by mobile devices. We show that *Meddle* can be used to monitor and manipulate all Internet traffic, including SSL traffic, from real users by incurring a small overhead in terms of latency, power, and data consumption. *Meddle* provides users the flexibility of deploying it on home gateways or sharing a *Meddle* deployment managed by researchers. Thus, *Meddle* offers an ideal vantage point that allows researchers to use their research activities as incentives to recruit real users. For example, peer-to-peer solutions are not widely used by mobile devices compared to desktops. One reason is the increased power and network consumption when uploading data to reciprocate for the data downloaded. *Meddle* can be used to offload the maintenance of P2P connections and the uploads.

Meddle allows access to the mobile Internet traffic, however to characterize this traffic it is necessary to identify the apps and the Web services responsible for these network flows. We used ground-truth data from controlled experiments to develop techniques to identify the apps and Web services from protocol headers. We also developed techniques to identify leaks of personally identifiable information (PII). In particular, we used *Meddle*'s ability to monitor SSL traffic and observed that misbehaving apps collude with ads and analytics libraries, and use HTTP and SSL to leak PIIs. Such analysis previously required either warranty voiding the OS [82, 83, 96], or performing static analysis on the app binaries [73, 82]. We then used our research results to build a tool that allows users to visualize and block PII leaks, an incentive to recruit users in future research activities. We thus demonstrate that the research work coming from *Meddle* can be used to create incentives to recruit users to participate in future research activities.

Considering the very fast changes in trends in Internet traffic, we envision that mobile devices will soon replace PCs to stream videos. For example, YouTube currently accounts for 20% of mobile downstream traffic in North America, Europe and Latin America [37]. We therefore characterized YouTube traffic, one of the most dominant sources of Internet traffic by volume. We observed that client side applications and the YouTube servers control the data transfer rate producing traffic patterns that are completely different from those observed during typical file transfers. Therefore, the traffic patterns observed when streaming YouTube videos depend on the client side application (desktop browser or mobile app) and container (Flash or HTML5). We observed that streaming videos to mobile devices produce traffic patterns that are completely different from those observed when using desktop browsers, and that these traffic patterns change when mobile devices use Wi-Fi instead of cellular networks. This observation implies that a large scale migration from one application to another (browser to mobile app) or from Flash to HTML5 can completely change the traffic patterns and the traffic volume on backbone links. Considering the very fast changes in trends this is a real possibility, the most likely being a change from streaming Flash videos to PCs to streaming HTML5 videos to mobile devices.

5.2 Open Problems

We used *Meddle* to touch the tip of the iceberg that represents the problems on transparency and control in mobile networks. We now discuss some of these open problems and how *Meddle* can be used in conjunction with existing solutions to address these problems.

Meddle gives access to the Internet traffic from mobile devices, however, it cannot monitor the non-IP traffic such as telephony and SMS. Indeed, malicious apps that use SMS for their operations cannot be diagnosed by using only *Meddle*. However, *Meddle* can be used to analyze the IP traffic generated by these apps. For example, malicious apps built using the Perkele malware kit [58] are known to generate IP traffic along with SMS traffic. To diagnose such apps, *Meddle* can be used to analyze the IP traffic generated by code segments that cannot be analyzed by AppFence and *droidguard*. Using techniques similar to those by Perdisci *et al.* [119] and those discussed in Section 3.2, *Meddle* can be used to identify malware from their IP traffic. Thus, *Meddle* can be used in conjunction with existing tools to diagnose apps that generate IP and non-IP traffic.

Meddle relies on VPNs to redirect Internet traffic regardless of the ISPs used by the mobile device. Mobile ISPs are known to deploy middleboxes that modify Internet traffic [15, 31]. For example, Wang *et al.* [136] discuss the various middleboxes such as firewalls that modify the traffic in flight. Though VPNs can mitigate unwanted modifications by ISPs, they cannot be used to characterize ISP modifications. We propose that by using a HTTP proxy in place of a VPN proxy can be used to study modifications of HTTP traffic. This HTTP proxy can be used to inject code such as TripWires [125] to detect packet modifications in flight. We are currently working on a prototype of this solution.

Meddle was conceptualized to provide a user-friendly solution to the problem of lack of transparency and end-user control in the mobile ecosystem. Though *Meddle* does not provide a silver-bullet to this problem, it allows end-users to participate and contribute in our on-going effort of improving transparency and end-user control in mobile networks.

Appendices

A Other Lessons from *Meddle* deployment

A.1 Diversity of ISPs and Access Technologies

AS	Description	Access Technology
12844	ASN-BOUYGTEL-MOBILE Bouygues Telecom	Cellular
5410	ASN-BOUYGTEL-ISP Bouygues Telecom S.A.	Wi-Fi
7018	ATT-INTERNET4 - AT&T Services, Inc.	Wi-Fi
20057	AT&T Wireless Service	Cellular

Table A.1: Example of AS description strings. ***We use the AS description to estimate the access technology used by the mobile device.***

Meddle gives access to all the Internet traffic from a mobile device. To infer the access technology (Wi-Fi or cellular), we use the AS description from WHOIS data for each IP address used by a mobile device. For example, in Table A.1, for AS number 12844, we observe a description ASN-BOUYGTEL-MOBILE Bouygues Telecom, a clear indication that this AS is begin used for cellular services.

Based on this classification, the *mobiWest* dataset consists of traffic from 54 distinct ASes, of which we identify 9 to be cellular ASes. Similarly, the *mobiEast* dataset consists of traffic from 23 distinct ASes of which 7 are cellular ASes. Note that, this classification fails when a cellular network is used by Wi-Fi access points to connect to the Internet. For example, the home-gateway of one user in the *mobiWest* dataset uses a cellular connection instead of a wired connection. In this case, though the mobile device is using Wi-Fi to communicate with the Internet, our classification technique would classify this traffic as cellular.

During the measurement study, each device in the *mobiWest* and *mobiEast* dataset is connected to our *Meddle* server from at most two distinct cellular ASes. A median of 4 Wi-Fi ASes were observed per device in the *mobiWest* dataset, 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 0-56% of their traffic over cellular, and the remainder over WiFi. The key take-away is that, measuring traffic from a single cellular carrier or Wi-Fi access point misses a large fraction of traffic generated by the devices.

A.2 Monitoring Evolution of Apps: The Case of Google Search

By monitoring an app behavior over time, *Meddle* can be used to gain insights on the impact of updates. In particular, *Meddle* can be used to give insights on unpublicized revelations on the network activity of apps.

Time	Bytes Downloaded	Query String	Remote Server
1353489965.97	356	a	suggestqueries.google.com
1353489966.19	321	aw	suggestqueries.google.com
1353489966.47	300	awe	suggestqueries.google.com
1353489966.88	301	awes	suggestqueries.google.com

Table A.2: Google Search in the clear.

Android	**iOS**
mercuryapps.foxnews.com	www.engadget.com
www.google.com	itunes.apple.com
www.quora.com	iadctest.qwapi.com
opml.radiotime.com	www.google.com
www.pandora.com	beacon.flipboard.com

Table A.3: Top five sites using counterproductive compression.

As an example, we show how Google searches from mobile devices have evolved with time. In the desktop environment, Google searches use HTTPS connections. In contrast, we noticed that the default browsers on iOS (version 5) and Android (version 4.0 and 4.1) send user queries in the clear. In particular, as shown in Table A.2, each letter is sent in the clear as the user types it. Interestingly, as of iOS 6 and Android 4.2, these searches are now sent using HTTPS, addressing a significant privacy vulnerability. To the best of our knowledge, this change has not been publicized.

A.3 Compressing Mobile Traffic: The Case of Counterproductive Compression

In the *mobiWest* dataset, we observed that 42.45% of the HTTP flows (1.06% by volume) have a mime-type containing "text." This content can be potentially compressed by the remote server. Importantly, our analysis reveals that 23% of HTTP flows in the *mobiWest* dataset are uncompressed, and these flows contribute only 0.5% of our users' total data. Thus, we believe that there are few meaningful opportunities for compressing content flowing through our *Meddle* servers; instead, we believe most improvements will come from transcoding media content to a lower bitrate or resolution.

Interestingly, we also observe that compression is counter-productive for 4% of the all the flows that have a mime-type containing "text"–the volume of data after compression is larger than the volume before compression. In Table A.3, we present the top five sources of traffic in the *mobiWest* dataset for which compression is counterproductive. As one example, we find that despite good intentions, Google's use of compression for search responses generally does more harm than good.

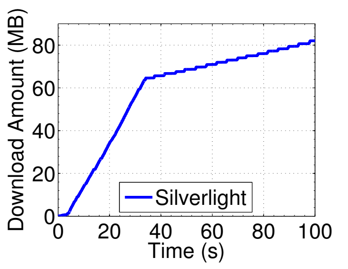
B Video Streaming Revisited

We now present an overview of the network traffic characteristics of Netflix traffic. We then derive a mathematical model to evaluate the impact of the streaming strategies on the stochastic properties of the aggregate video streaming traffic. Our model can be used to dimension the network for video streaming. In particular, it sheds light on the importance of the different video streaming parameters for traffic engineering. For example, we show that an increase in the video encoding rates will produce smoother aggregate video streaming traffic. We also present the video streaming parameters that can be adapted to minimize the amount of unused bytes on user interruptions due to lack of interest.

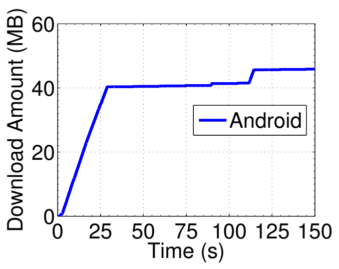
B.1 Characterize Netflix Traffic

Device	Application	Strategy (SilverLight)
Mobile	iOS	✓
	Android	✓
PC	Internet Explorer	✓
	Google Chrome	✓
	Firefox	✓

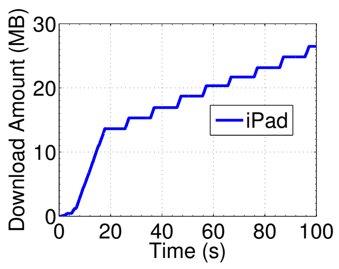
Table B.1: Netflix Streaming Strategies. ✓ *represents* Intelligent *streaming strategy while X represents the* Crude *streaming strategy.*



(a): Netflix to PCs.



(b): Netflix to Android.



(c): Netflix to iOS.

Figure B.1: Netflix Streaming Strategies

In Table B.1, we summarize the streaming strategies used to stream Netflix videos. For PCs, we streamed 200 videos that were randomly selected from the list of 11208 videos available for watching instantly as of 20-May-2011. For mobile devices, we streamed 50 videos of the 200 videos streamed to PCs.

In Figure B.1 and Table B.1, we observe that Netflix uses the *Intelligent* streaming strategy to stream videos to PCs and mobile devices. The streaming strategies for mobile devices are similar to that observed in Section 4.3.2, the cycles of ON-OFF periods when using Android have a larger duration compared to iOS devices. Netflix uses Silverlight as the container to stream videos to PCs, and like Flash videos, we observe the same streaming strategy regardless of the browser used.

B.2 Model for Aggregate Video Traffic

In Section 4.3 and Section B.1, we observe that the application and the container determine the strategy to stream videos. We now present a mathematical model to express the stochastic properties of the aggregate video streaming traffic as a function of the video parameters. Our model can be used to dimension the network and quantify the impact of migrating from one strategy to another.

We first develop our model for the case of users that do not interrupt the video download. We then study the impact of user interruption due to lack of interest on the accumulation ratio and the amount of data downloaded in the buffering phase. We then quantify the amount of bandwidth wasted when users interrupt the video download due to lack of interest.

For our model, we assume that the video streaming sessions arrive according to a homogeneous Poisson process with rate λ. We use the measurements performed by Yu *et al.* [141] for the Poisson assumption of the arrival rate.[1] Let T_n, $n \in \mathbb{Z}$, denote the arrival time of the n-th video. We assume that $n-$th video is streamed at a fixed encoding rate, e_n, and has a fixed duration (length), L_n; the size of the n-th video is $S_n = e_n L_n$. We also assume that the network is over provisioned: the end-to-end available bandwidth is larger than the video encoding rate for each video streaming session. This hypothesis is validated by our measurements presented in Section 4.3. Indeed, during our measurements we observed an accumulation ratio larger than one, which implies that the download rate, and hence the end-to-end available bandwidth, is larger than the video encoding rate.

B.2.1 Video Download without Interruptions

We now model the aggregate data rate of video streaming traffic when users do not interrupt the video download. We first examine the *Crude* strategy where the whole video is downloaded at the end-to-end available bandwidth. We assume the time required to download the n-th video is D_n. For the n-th video, the video download is *active* at time t when $T_n \leq t \leq T_n + D_n$. Let $X_n(t - T_n)$ denote the download rate of the n-th video at time t; $X_n(t) = 0$ when $t < T_n$ and $t > T_n + D_n$. Let $R(t)$ denote the aggregate data rate of the video streaming traffic at time t.

According to Barakat *et al.* [69], the mean and variance of the aggregate data rate are:

$$\mathbb{E}[R(t)] = \lambda \mathbb{E}[S_n], \tag{B.1}$$

$$V_R = \mathbb{E}[R^2(t)] - (\mathbb{E}[R(t)])^2 = \lambda \mathbb{E}[\int_0^{D_n} X_n^2(u)du], \tag{B.2}$$

respectively.

When the download rate of the n-th video is a constant G_n, substituting $D_n = \dfrac{S_n}{G_n}$,

[1]Given the fact that users watch the videos in series, it is easy to prove that the Poisson assumption is not needed at the video level. It is enough to have the Poisson assumption at the user level, which is very likely to be the case given the human nature of this activity.

Name	Description
λ	Arrival rate of videos streaming sessions.
n	number of videos.
e_n	Encoding rate of the n-th video.
L_n	Duration (or length) of the n-th video.
B_n	Buffering amount for the n-th video.
B'_n	Buffering amount for the n-th video in terms of playback time.
S_n	Size of the n-th video $S_n = e_n L_n$.
k_n	The accumulation ratio for the n-th video.
β_n	Users interrupt the n-th video after time $\beta_n L_n$.
$R(t)$	Aggregate data rate of streaming traffic at time t.
$R'(t)$	Aggregate amount of bandwidth wasted at time t when users interrupt video download due to lack of interest.

Table B.2: Variables used in the model.

$S_n = e_n L_n$, and $X_n(t) = G_n$ for $T_n \le t \le T_n + D_n$, in equations B.1 and B.2 yields:

$$\mathbb{E}[R(t)] = \lambda \mathbb{E}[e_n]\mathbb{E}[L_n], \tag{B.3}$$

$$V_R = \lambda \mathbb{E}[e_n]\mathbb{E}[L_n]\mathbb{E}[G_n]. \tag{B.4}$$

Equations B.3 and B.4 give the mean and variance of the aggregate data rate of video streaming traffic when the *Crude* streaming strategy is used to stream videos.

We now show that when users do not interrupt the video download, *the mean and variance of the data rate are independent of the streaming strategy used.* Let $D'_n(> D_n)$ denote the time required to download the video when the video contents are downloaded using either the *Intelligent* streaming strategy. For the n-th video, the download rate is G_n during the ON periods and 0 in the OFF periods. If the download rate does not change during the data transfer, then $\int_0^{D_n} X_n^2(u)du = \int_0^{D'_n} X_n^2(u)du = e_n L_n G_n$, which leads to the same variance as in Equation B.4. Using the same argument and the framework in Barakat *et al.* [69], one can extend this result to higher moments of the aggregate traffic.

Therefore, when users do not interrupt the video downloads, we conclude the following:

1. Equations B.3 and B.4 can be used to dimension the network for video streaming. A simple rule would be to set the bitrate of links carrying video streaming traffic to $E[R(t)] + \alpha\sqrt{V_r}$, where $\alpha \ge 1$ is a constraint on the tolerable bandwidth violations.
2. The mean and variance of the aggregate data rate of video streaming traffic are independent of the underlying streaming strategies used, and hence the required bandwidth. This is important as video services, where the users are expected to view the whole video and not interrupt the video download, can safely select a streaming strategy that can be optimized for other goals such as server load without overwhelming the network.
3. An increase in the video encoding rate, for example when YouTube increases the default video resolution, shall increase the aggregate rate of video traffic. However, because the variance is a *linear* function of the video encoding rate, the aggregate traffic shall be *smoother* than the aggregate traffic observed at lower encoding rates.

B.2.2 Video Download with Interruptions

Users can interrupt a streaming session due to various reasons such as poor playback quality or lack of interest in the given video. When a user interrupts the video download due to lack of interest, the data downloaded but not used by the player is wasted. The wastage of network resources can be quantified using the amount of unused bytes. The amount of unused bytes due to lack of interest is important because Gill *et al.* [91] observe that 80% of the video interruptions in a campus network are due to lack of user interest. According to Finamore *et al.* [89], 60% of the YouTube videos are watched for less than 20% of their duration.

We now present the impact of the buffering amount and the accumulation ratio on the amount of unused bytes. We assume that the user interrupts the download of the n-th video after time τ_n from the start of the video playback. We further assume that the amount downloaded in the buffering phase is B_n, $B_n \geq 0$, and the time required for downloading this amount is negligible. If G_n is the average download rate in the steady state phase, then the amount of data that can be downloaded up to time τ_n is $B_n + G_n\tau_n$. We keep denoting the encoding rate and duration of the n-th video as e_n and L_n respectively. Thus, the interruption of the n-th video shall take place before the whole video has been downloaded only if

$$e_nL_n > B_n + G_n\tau_n \geq e_n\tau_n. \tag{B.5}$$

We now assume the download rate of the n-th video is limited by the accumulation ratio $k_n = \dfrac{G_n}{e_n}$, $k_n \geq 1$. We also assume that $\tau_n = \beta_nL_n$, where β_n, $\beta_n < 1$, is the fraction of the n-th video watched before interruption. Equation B.5 can now be written as

$$e_nL_n > B_n + e_nk_n\beta_nL_n \geq e_n\beta_nL_n. \tag{B.6}$$

When $B_n = e_nB'_n$, where B'_n is the amount of playback time buffered in the buffering phase, the left hand side of Equation B.6 can be written as

$$B'_n < L_n(1 - k_n\beta_n). \tag{B.7}$$

In Section 4.3.1, we observed a buffering of 40 seconds worth of playback, and an accumulation ratio of 1.25 for Flash videos. When a user interrupts the video download after watching 20% of the video, substituting $B'_n = 40$ seconds, $k_n = 1.25$, and $\beta = 0.2$ yields $L_n = 53.3$ seconds. This implies that, assuming a fast buffering, YouTube Flash videos that have a duration smaller than 53.3 seconds will be downloaded before the viewers have seen 20% of the video.

We now use the amount of unused bytes to obtain the average bandwidth wasted due to user interruption. When the n-th user interrupts the video download at time τ_n, then the amount of bytes downloaded is $min(B_n + G_n\tau_n, e_nL_n)$. The total amount of bytes consumed by the player up to time τ_n is $e_n\tau_n$. Therefore, the amount of unused bytes is $min(B_n + G_n\tau_n, e_nL_n) - e_n\tau_n$, and the average bandwidth wasted is given by

$$\mathbb{E}[R'(t)] = \lambda\mathbb{E}[min(B_n + G_n\tau_n, e_nL_n) - e_n\tau_n]. \tag{B.8}$$

When the accumulation ratio of the n-th video is k_n and the user interrupts the video after viewing β_n fraction of the video, then substituting $B_n + G_n\tau_n = e_nB'_n + e_nL_nk_n\beta_n$ in Equation B.8 yields

$$\mathbb{E}[R'(t)] = \lambda\mathbb{E}[e_n]\mathbb{E}[min(B'_n + k_n\beta_nL_n, L_n) - \beta_nL_n]. \tag{B.9}$$

In summary, Equation B.7 provides a condition to limit the amount of unused bytes when users interrupt the video download due to lack of interest. Equations B.8 and B.9 can be used to compute the amount of bandwidth wasted due to user interruptions.

C Other Work

I now present a short description of the work I co-authored during my Ph.D. thesis, but that are not part of the present manuscript. These works were performed to create a foundation on the tools that were useful in creating *Meddle*.

1. Can Realistic BitTorrent Experiments Be Performed on Clusters?

Network latency and packet loss are considered to be an important requirement for realistic evaluation of Peer-to-Peer protocols. Dedicated clusters, such as Grid'5000, do not provide the variety of network latency and packet loss rates that can be found in the Internet. However, compared to the experiments performed on testbeds such as PlanetLab, the experiments performed on dedicated clusters are reproducible, as the computational resources are not shared. In this paper, we perform experiments to study the impact of network latency and packet loss on the time required to download a file using BitTorrent. In our experiments, we observe a less than 15% increase on the time required to download a file when we increase the round-trip time between any two peers, from 0 ms to 400 ms, and the packet loss rate, from 0% to 5%. Our main conclusion is that the underlying network latency and packet loss have a marginal impact on the time required to download a file using BitTorrent. Hence, dedicated clusters such as Grid'5000 can be safely used to perform realistic and reproducible BitTorrent experiments.

2. Floor the Ceil & Ceil the Floor: Revisiting AIMD Evaluation

Additive Increase Multiplicative Decrease (AIMD) is a widely used congestion control algorithm that is known to be fair and efficient in utilizing the network resources. In this paper, we revisit the performance of the AIMD algorithm under realistic conditions by extending the seminal model of Chui *et al.* [78]. We show that under realistic conditions the fairness and efficiency of AIMD is sensitive to changes in network conditions. Surprisingly, the root cause of this sensitivity comes from the way the congestion window is rounded during a multiplicative decrease phase. For instance, the floor function is often used to round the congestion window value because either kernel implementations or protocol restrictions mandate to use integers to maintain system variables. To solve the sensitivity issue, we provide a simple solution that is to alternatively use the floor and ceiling functions in the computation of the congestion window during a multiplicative decrease phase, when the congestion window size is an odd number. We observe that with our solution the efficiency improves and the fairness becomes one order of magnitude less sensitive to changes in network conditions.

Bibliography

[1] Ad blocking with ad server hostnames and IP addresses. *Cited in page 50*

[2] Adblock Plus: Surf the web without annoying ads! *Cited in page 23, 50*

[3] Adobe Flash Video File Format Specification. Adobe Systems Incorporated. *Cited in page 55*

[4] Aliyun. *Cited in page 33*

[5] Android developers: Security with https and ssl. *Cited in page 23*

[6] App States and Multitasking. *Cited in page 3, 7*

[7] BitTorrent - Delivering the World's Content. *Cited in page 30*

[8] Browser detection using the user agent. *Cited in page 37*

[9] Bugzilla@Mozilla: Bug 733010 - Video download should be rate limited to avoid bufferbloat. *Cited in page 70*

[10] Certificate and public key pinning - owasp. *Cited in page 23*

[11] Configuring Your Xcode Project for Distribution. *Cited in page 42, 48*

[12] Core Media Framework Reference. *Cited in page 38, 61*

[13] Download Internet Explorer 10. *Cited in page 60*

[14] Download Internet Explorer 9. *Cited in page 60*

[15] Free attaqué pour les ralentissements de youtube. *Cited in page 25, 76*

[16] GCM Architectural Overview. *Cited in page 43, 45*

[17] Ghostery: Knowledge + Control = Privacy. *Cited in page 23, 50*

[18] Google analytics: Mobile app analytics. *Cited in page 41*

[19] Google Now. The right information at just the right time. *Cited in page 7*

[20] Google play: Transaction fees. *Cited in page 4, 5*

[21] Grow your business with iad. *Cited in page 6*

[22] How do I configure my router for Spotify. *Cited in page 36*

[23] http://www.google.com/chrome/. *Cited in page 60*

[24] http://www.mozilla.com/firefox. *Cited in page 60*

[25] Introducing Collusion. Discover whoś tracking you online. *Cited in page 23, 51*

[26] Invalid video bitrate while using libwebM parser for YouTube files. *Cited in page 59*

[27] Local and Push Notification Programming Guide. *Cited in page 43, 45*

[28] MaskMe Protects Your Online Privacy. *Cited in page 23*

[29] MATLAB - The Language of Technical Computing - MathWorks. *Cited in page 31*

[30] Media — Android Developers. *Cited in page 38, 61*

[31] Mise à jour Freebox Server 1.1.9. *Cited in page 76*

[32] MIUI. *Cited in page 33, 36*

[33] Mobilescan. *Cited in page 53*

[34] Naming a Package (The Java ^{TM} Tutorials : Learning the Java Language: Packages). *Cited in page 38*

[35] Openvpn. *Cited in page 18*

[36] Overview of MPEG-DASH Standard. *Cited in page 58, 71*

[37] Sandvine Report: Apple Takes Big Bite of Streaming Video. *Cited in page 55, 76*

[38] Squid-in-the-middle SSL Bump. *Cited in page 22*

[39] ssldump home page. *Cited in page 31*

[40] StrongSwan VPN Client. *Cited in page 18*

[41] TCPDUMP/LIBPCAP public repository. *Cited in page 31*

[42] The iOS Environment. *Cited in page 3, 7*

[43] The R project for Statistical Computing. *Cited in page 31*

[44] They're (Almost) All Dirty: The State of Cheating in Android Benchmarks. *Cited in page 9*

[45] TuneIn: Listen to Online Radio, Music, and Talk Stations. *Cited in page 38*

[46] Using HTTP Live Streaming. *Cited in page 58, 71*

[47] VP8 and WebM Tools. *Cited in page 59*

[48] VPN On Demand. *Cited in page 17*

[49] WebM: an open web media project. *Cited in page 60*

[50] WebM Clearly not supported by Win8 with IE10. *Cited in page 65*

[51] What is Adblock Plus for Android? *Cited in page 23*

[52] Wikipedia: Usage share of web browsers. *Cited in page 60*

[53] Xiaomi. *Cited in page 33, 36*

[54] YouTube HTML5 Video Player. *Cited in page 55, 59, 61*

[55] YouTube on Android: Settings on Android. *Cited in page 71*

[56] YouTube on iOS: Settings on iOS. *Cited in page 72*

[57] App Store Tops 40 Billion Downloads with Almost Half in 2012, 2012. *Cited in page 10*

[58] A Closer Look: Perkele Android Malware Kit - Krebs on Security, 2013. *Cited in page 76*

[59] Adobe Flash Player 11 : Tech specs : System requirements, 2013. *Cited in page 55, 61*

[60] Build a great app business with AdMob, 2013. *Cited in page 2, 6*

[61] Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time, 2013. *Cited in page 3, 17*

[62] Global Internet Phenomena. Sandvine, 2013. *Cited in page 55, 64, 70, 73*

[63] Google: Mobile Ads, 2013. *Cited in page 2, 6, 41*

[64] iAd, 2013. *Cited in page 6*

[65] Alcock, S., and Nelson, R. Application flow control in YouTube video streams. *ACM SIGCOMM Computer Communication Review 41* (April 2011). *Cited in page 69, 73*

[66] Allman, M., Paxson, V., and Blanton, E. TCP Congestion Control. IETF Network Working Group, Request for Comments: 5681, 2009. *Cited in page 69*

[67] Alvestrand., H. Overview: Real Time Protocols for Brower-based Applications. IETF Network Working Group, Request for Comments: draft-ietf-rtcweb-overview-08, 2013. *Cited in page 29*

[68] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. PScout: Analyzing the Android Permission Specification. In *Proc. of CCS* (2012). *Cited in page 7*

[69] Barakat, C., Thiran, P., Iannaccone, G., Diot, C., and Owezarski, P. Modeling Internet backbone traffic at the flow level. *IEEE Transactions on Signal Processing 51*, 8 (2003). *Cited in page 82, 83*

[70] Barré, S., Bonaventure, O., Raiciu, C., and Handley, M. Experimenting with multipath tcp. *Proc. of the ACM SIGCOMM Conference* (2010). *Cited in page 24*

[71] 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 the Internet Measurement Conference (IMC)* (2012). *Cited in page 44, 46*

[72] Bonaventure, O. Apple seems to also believe in Multipath TCP, 2013. *Cited in page 24*

[73] Book, T., Pridgen, A., and Wallach, D. S. Longitudinal Analysis of Android Ad Library Permissions. *Proceedings of Mobile Security Technologies (MoST)* (2013). *Cited in page 6, 7, 10, 11, 51, 75*

[74] Border, J., Kojo, M., Griner, J., Montenegro, G., and Shelby, Z. Performance enhancing proxies intended to mitigate link-related degradations. IETF Network Working Group, Request for Comments: 3135, 2001. *Cited in page 25*

[75] Carpenter, B., and Brim, S. Middleboxes: Taxonomy and Issues. IETF Network Working Group, Request for Comments: 3234, 2002. *Cited in page 11*

[76] 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 the Internet Measurement Conference (IMC)* (2012). *Cited in page 11*

[77] Cheng, R., Scott, W., Krishnamurthy, A., and Anderson, T. FreeDOM: A New Baseline for the Web. In *Proc. of the Workshop on Hot Topics in Networks (HotNets)* (2012). *Cited in page 29*

[78] Chiu, D.-M., and Jain, R. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Comput. Netw. ISDN Syst.* (1989). *Cited in page 86*

[79] Davis, T. O. The speeches of the Right Honourable John Philpot Curran. *Cited in page 75*

[80] Desnos, A., and Erra, R. Descriptional Entropy: Application to Security Software Analysis. In *Advanced Infocomm Technology*, vol. 7593 of *Lecture Notes in Computer Science*. 2013. *Cited in page 10*

[81] Durumeric, Z., Kasten, J., Bailey, M., and Halderman, J. A. Analysis of the HTTPS Certificate Ecosystem. *Proc. of the Internet Measurement Conference (IMC)* (2013). *Cited in page 43*

[82] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium* (2011). *Cited in page 1, 7, 10, 11, 15, 23, 51, 75*

[83] 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 the USENIX Operating Systems Design and Implementation (OSDI)* (2010). *Cited in page 1, 7, 9, 10, 11, 15, 51, 75*

[84] Enck, W., Ongtang, M., and McDaniel, P. On Lightweight Mobile Phone Application Certification. In *Proc. of CCS* (2009). *Cited in page 51*

[85] Erman, J., Gerber, A., and Sen, S. HTTP in the Home: It is not just about PCs. *ACM SIGCOMM Computer Communication Review* (2011). *Cited in page 37, 48*

[86] Falaki, H., Lymberopoulos, D., Mahajan, R., Kandula, S., and Estrin, D. A First Look at Traffic on Smartphones. In *Proc. of the Internet Measurement Conference (IMC)*. *Cited in page 34, 36*

[87] Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. Diversity in Smartphone Usage. In *Proceedings of the International conference on Mobile systems, applications, and services (Mobisys)* (2010). *Cited in page 7, 11, 34, 36, 75*

[88] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext transfer protocol–http/1.1. IETF Network Working Group, Request for Comments: 2616, 1999. *Cited in page 36*

[89] Finamore, A., Mellia, M., Munafò, M. M., Torres, R., and Rao, S. G. YouTube Everywhere: Impact of Device and Infrastructure Synergies on User Experience. In *Proc. of the Internet Measurement Conference (IMC)* (2011). *Cited in page 56, 84*

[90] Ghobadi, M., Cheng, Y., Jain, A., and Mathis, M. Trickle: Rate limiting youtube video streaming. In *Proc. of the USENIX Annual Technical Conference* (2012). *Cited in page 58, 64*

[91] Gill, P., Arlitt, M., Li, Z., and Mahanti, A. Youtube Traffic Characterization: A View From the Edge. In *Proc. of the Internet Measurement Conference (IMC)* (2007). *Cited in page 56, 84*

[92] Guha, S., Cheng, B., and Francis, P. Privad: Practical privacy in online advertising. In *Proc. of USENIX NSDI* (2011). *Cited in page 30*

[93] Hickson, I. HTML5: A vocabulary and associated APIs for HTML and XHTML. W3C Working Draft, 2011. *Cited in page 55*

[94] Hoffman, P. Algorithms for internet key exchange version 1 (ikev1). *Cited in page 18*

[95] Holz, R., Braun, L., Kammenhuber, N., and Carle, G. The SSL Landscape - A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements. *Proc. of the Internet Measurement Conference (IMC)* (2011). *Cited in page 43*

[96] 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). *Cited in page 1, 7, 9, 10, 11, 15, 23, 50, 51, 53, 75*

[97] Huang, J., Qian, F., Guo, Y., Zhou, Y., Xu, Q., Mao, Z. M., Sen, S., and Spatscheck, O. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *Proc. of the ACM SIGCOMM Conference* (2013). *Cited in page 27*

[98] Huang, T.-Y., Handigol, N., Heller, B., McKeown, N., and Johari, R. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *Proc. of the Internet Measurement Conference (IMC)* (2012). *Cited in page 71, 73*

[99] Jacobson, V. Congestion Avoidance and Control. In *Proc. of the ACM SIGCOMM Conference* (1988). *Cited in page 69*

[100] Jobs, S. Thoughts on Flash, 2010. *Cited in page 55, 61*

[101] Josefsson, S. The Base16, Base32, and Base64 Data Encodings. IETF Network Working Group, Request for Comments: 4648, 2006. *Cited in page 11, 25*

[102] Katz, J. E. Magic in the air: Mobile communication and the transformation of social life, vol. 1. Transaction Books, 2011. *Cited in page 75*

[103] Kaufman, C., Hoffman, P., Nir, Y., and Eronen, P. Internet Key Exchange Protocol Version 2 (IKEv2). *Cited in page 18*

[104] Kent, S., and Seo, K. Security architecture for the internet protocol. IETF Network Working Group, Request for Comments: 4301, 2008. *Cited in page 18*

[105] Keybl, A. Mozilla's Heartbeat & Quarterly Firefox OS Releases. *Cited in page 11*

[106] Labovitz, C., Iekel-Johnson, S., McPherson, D., Oberheide, J., and Jahanian, F. Internet inter-domain traffic. In *Proc. of the ACM SIGCOMM Conference* (2010). *Cited in page 55*

[107] Laugesen, J., and Yuan, Y. What Factors Contributed to the Success of Apple's iPhone? In *Mobile Business and 2010 Ninth Global Mobility Roundtable (ICMB-GMR), 2010 Ninth International Conference on* (2010). *Cited in page 5, 6*

[108] 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). *Cited in page 49*

[109] Liu, Y., Li, F., Guo, L., Shen, B., and Chen, S. A Comparative Study of Android and iOS for Accessing Internet Streaming Services. In *Proc. PAM*, vol. 7799. 2013. *Cited in page 71*

[110] Liu, Y., Li, F., Guo, L., Shen, B., and Chen, S. Effectively Minimizing Redundant Internet Streaming Traffic to iOS Devices. In *Proc. of IEEE INFOCOM* (2013). *Cited in page 71*

[111] Lockheimer, H. Android and Security, Feb 2012. *Cited in page 4, 9*

[112] Maier, G., Feldmann, A., Paxson, V., and Allman, M. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proc. of the Internet Measurement Conference (IMC)* (2009). *Cited in page 55*

[113] Maier, G., Schneider, F., and Feldmann, A. A First Look at Mobile Hand-held Device Traffic. *Proc. PAM* (2010). *Cited in page 11, 34, 36, 37, 48*

[114] Mockapetris, P. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. IETF Network Working Group, Request for Comments: 1035, 1987. *Cited in page 44*

[115] Orwell, G. Nineteen Eighty-Four. eBooks @ Adelaide, 2006. *Cited in page 1*

[116] 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). *Cited in page 10, 15*

[117] Paxson, V. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks 31* (1999). *Cited in page 31, 34*

[118] Pearce, P., Felt, A. P., Nunez, G., and Wagner, D. Addroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (2012), ACM. *Cited in page 53*

[119] Perdisci, R., Lee, W., and Feamster, N. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *Proc. of the USENIX Symposium on Networked System Design and Implementation (NSDI)* (2010). *Cited in page 36, 48, 76*

[120] Plissonneau, L., En-Najjary, T., and Urvoy-Keller, G. Revisiting web traffic from a DSL provider perspective: the case of YouTube. In *Proc. of the 19th ITC Specialist Seminar* (2008). *Cited in page 73*

[121] Postel, J. Transmission Control Protocol (TCP). IETF Network Working Group, Request for Comments: 793, 1981. *Cited in page 58, 69*

[122] Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. Profiling Resource Usage for Mobile Applications: A Cross-layer Approach. In *Proceedings of the International conference on Mobile systems, applications, and services (Mobisys)* (2011). *Cited in page 15*

[123] Qian, F., Wang, Z., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O. Characterizing Radio Resource Allocation for 3G Networks. In *Proc. of IMC* (2010). *Cited in page 15, 27*

[124] Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., and Shayandeh, S. AppInsight: Mobile App Performance Monitoring in the Wild. *Proc. of the USENIX Operating Systems Design and Implementation (OSDI)* (2012). *Cited in page 7, 10, 15*

[125] Reis, C., Gribble, S. D., Kohno, T., and Weaver, N. C. Detecting In-Flight Page Changes with Web Tripwires. In *Proc. of the USENIX Symposium on Networked System Design and Implementation (NSDI)* (2008). *Cited in page 25, 76*

[126] Rescorla, E. HTTP Over TLS. IETF Network Working Group, Request for Comments: 2818. *Cited in page 11, 43*

[127] Roesner, F., Kohno, T., and Wetherall, D. Detecting and Defending Against Third-Party Tracking on the Web. *Proc. of USENIX NSDI* (2012). *Cited in page 49, 50, 53*

[128] Sanou, B. ICT Facts and Figures. Tech. rep., International Telecommunications Union, 2013. *Cited in page 2*

[129] Saxena, M., Sharan, U., and Fahmy, S. Analyzing Video Services in Web 2.0: A Global Perspective. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video* (2008). *Cited in page 73*

[130] Shekhar, S., Dietz, M., and Wallach, D. S. Adsplit: Separating smartphone advertising from applications. In *Proc. of USENIX Security Symposium* (2012). *Cited in page 53*

[131] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. of the ACM SIGCOMM Conference* (2012). *Cited in page 11*

[132] Smith, B. More about the App Store GPL Enforcement, 2010. *Cited in page 4*

[133] Sommers, J., and Barford, P. Cell vs. WiFi: On the Performance of Metro Area Mobile Connections. In *Proc. of the Internet Measurement Conference (IMC)* (2012). *Cited in page 27, 36, 48*

[134] Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., and Hoffmann, J. Mobile-sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)* (2013). *Cited in page 10*

[135] Vallina-Rodriguez, N., Shah, J., Finamore, A., Grunenberger, Y., Papagiannaki, K., Haddadi, H., and Crowcroft, J. Breaking for Commercials: Characterizing Mobile Advertising. In *Proc. of the Internet Measurement Conference (IMC)* (2012). *Cited in page 11, 15, 23, 49, 50, 51*

[136] Wang, Z., Qian, Z., Xu, Q., Mao, Z., and Zhang, M. An Untold Story of Middleboxes in Cellular Networks. In *Proc. of the ACM SIGCOMM Conference* (2011). *Cited in page 27, 76*

[137] Xu, Q., Erman, J., Gerber, A., Mao, Z., Pang, J., and Venkataraman, S. Identifying Diverse Usage Behaviors of Smartphone Apps. In *Proc. of the Internet Measurement Conference (IMC)* (2011). *Cited in page 7, 15, 34, 36, 37, 41, 48, 75*

[138] Xu, Q., Gerber, A., Mao, Z. M., and Pang, J. Acculoc: Practical localization of performance measurements in 3g networks. In *Proc. of MobiSys* (2011). *Cited in page 15*

[139] Yan, L. K., and Yin, H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proc. of USENIX Security Symposium* (2012). *Cited in page 7*

[140] Yegneswaran, V., Giffi, J. T., Barford, P., and Jha, S. An Architecture for Generating Semantics-Aware Signatures. *Proc. of USENIX Security Symposium* (2005). *Cited in page 36, 48*

[141] Yu, H., Zheng, D., Zhao, B. Y., and Zheng, W. Understanding User Behavior in Large-Scale Video-on-Demand Systems. In *Proc. of Eurosys* (2006). *Cited in page 82*

[142] Zhang, L. Building Facebook Messenger, 2011. *Cited in page 34, 43, 45*

[143] Zhang, L., Zhou, F., Mislove, A., and Sundaram, R. Maygh: Building a CDN from Client Web Browsers. Proc. of Eurosys. *Cited in page 29*

Abstract

Mobile devices are increasingly becoming the primary device to access the Internet. Despite this thriving popularity, the current mobile ecosystem is largely opaque because of the vested monetary interests of its key players: mobile OS providers, creators of mobile applications, stores for mobile applications and media content, and ISPs. This problem of opaqueness is further aggravated by the limited control end-users have over the information exchanged by their mobile devices. To address this problem of opaqueness and lack of control, we designed a user-centric platform, *Meddle*, that uses traffic indirection to diagnose mobile devices. Compared to an on-device solution, *Meddle* uses two well-known technologies, VPNs and middleboxes, and combines them to provide a solution that is agnostic to OS, ISP, and access technology. We use *Meddle* for controlled experiments and an IRB approved study, and observed that popular iOS and Android applications leak personally identifiable information in the clear and also over SSL. We then use *Meddle* to prevent further leaks using a DNS based packet filter. We also use our platform to detail the network characteristics of video streaming services, the most popular Web-service in the current Internet. We observe that the network traffic characteristics vary vastly with the device (mobile or desktop), application (native applications and also between individual desktop browsers), and container (HTML5 and Flash). This observation is important because the increased adoption of one application or streaming service, for example, an increase in the usage of mobile devices to stream videos, could have a significant impact on the network traffic.

Résumé

Les terminaux mobiles (smartphones et tablettes) sont devenus les terminaux les plus populaires pour accéder à Internet. Cependant, l'écosystème incluant les terminaux mobiles est maintenu opaque à cause des intérêts financiers des différents acteurs : les concepteurs des systèmes d'exploitation et des applications, les opérateurs des "stores", et les FAI. Cette opacité est renforcée par le peu de contrôle qu'ont les utilisateurs sur les informations échangées par leur terminal. Pour résoudre ce problème d'opacité et de manque de contrôle, on a créé une plate-forme, Meddle, qui utilise la redirection de trafic des terminaux mobiles pour analyser et modifier ce trafic. Contrairement aux solutions qui nécessitent d'être implémentées sur le terminal, Meddle combine les techniques de VPN et de "middlebox" pour offrir une solution indépendante de l'OS, du FAI et de l'accès radio. On a utilisé Meddle pour des expérimentations contrôlées et pour une étude utilisateurs approvée par un IRB. On a observé que des applications populaires sous iOS et Android exposaient des informations personnelles dans le traffic réseau en clair et chiffré. On a ensuite exploité Meddle pour prévenir ces fuites d'informations privées. On a également utilisé Meddle pour étudier les caractéristiques réseaux du trafic vidéo sur Internet. On a trouvé que ce trafic dépend fortement du type de terminal, de l'application utilisée pour regarder la vidéo (application native ou navigateur Web) et du contenant (HTML5, Flash, Silverlight). Ce résultat montre qu'un changement dans le terminal, l'application ou le contenant peut avoir un impact important sur le réseau.