UNIVERSITY OF CALIFORNIA,
IRVINE

AntMonitor: A System for Mobile Network Monitoring

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Engineering

by

Anastasia Shuba

Thesis Committee:
Associate Professor Athina Markopoulou, Chair
Associate Professor Brian Demsky
Assistant Professor Aparna Chandramowlishwaran

2016

DEDICATION

To Manila Aphay, for her continuous belief in me.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Athina Markopoulou for her guidance, patience, and open-mindedness. I could not have asked for a better advisor.

Second, a thank you is in order to the members of my committee - Brian Demsky and Aparna Chandramowlishwaran. I am especially thankful to Brian since he is the one who first introduced Android programming to me, four years ago.

Third, I would like to thank my lab mates: Janus Varmarken and Simon Langhoff for bootstrapping the client-server system, Anh Le for his continuous advice and his contributions to the mobile-only system, Minas Gjoka for his contributions to the privacy leaks and traffic classification applications, and Emmanouil Alimpertis for his work on the network measurements application.

I would not have made it this far without the support of my family and friends. I owe my gratitude to Galyna Shuba and Allen Kerry for their practical support. I am also thankful to my extended family at Ikazuchi Dojo, where I spent much of my time training and releasing my stress. Most of all, I am thankful to Karen Kim, for sharing her academic experience with me and for providing emotional support. I would also like to thank Metronome Software for providing me with a flexible internship for the past four years. Most importantly, I owe my deepest gratitude to my co-worker and friend, Manila Aphay, for her unconditional emotional support.

Finally, I am very thankful to the EECS department for providing me with a fellowship, and to NSF for funding the project with grants 1228995 and 1028394. I would also like to acknowledge the following open source libraries that I used in my project: OSMonitor, multifast, SandroProxy, and PrivacyGuard.

ABSTRACT OF THE THESIS

AntMonitor: A System for Mobile Network Monitoring

By

Anastasia Shuba

Master of Science in Computer Engineering

University of California, Irvine, 2016

Associate Professor Athina Markopoulou, Chair

In this thesis, we propose AntMonitor – a complete system for passive monitoring, collection, and analysis of fine-grained, large-scale packet measurements from mobile devices. We design AntMonitor as a VPN-based service, and we develop and compare two versions of the architecture, Client-Server and Mobile-Only. We show that the AntMonitor Mobile-Only prototype significantly outperforms the Client-Server one, as well as all comparable state-of-the-art approaches w.r.t. throughput and energy. It achieves speeds of over 90 Mbps (downlink) and 65 Mbps (uplink), which are 2x and 8x throughput of existing mobile-only approaches, and at 94% of the throughput without VPN. These speeds are achieved while using 2–12x less energy. We also argue that AntMonitor is uniquely positioned to support a number of passive monitoring applications, namely: (i) real-time detection and prevention of private information leakage from the device to the network; (ii) passive performance measurements network-wide as well as per-user; and (iii) application classification based on TCP/IP header features. Furthermore, the Mobile-Only design helps AntMonitor to scale with ease and provides enhanced privacy protection. We present results from a pilot user study at UCI and highlight the key design choices and optimizations that allow AntMonitor to achieve significant benefits.

Chapter 1

Introduction

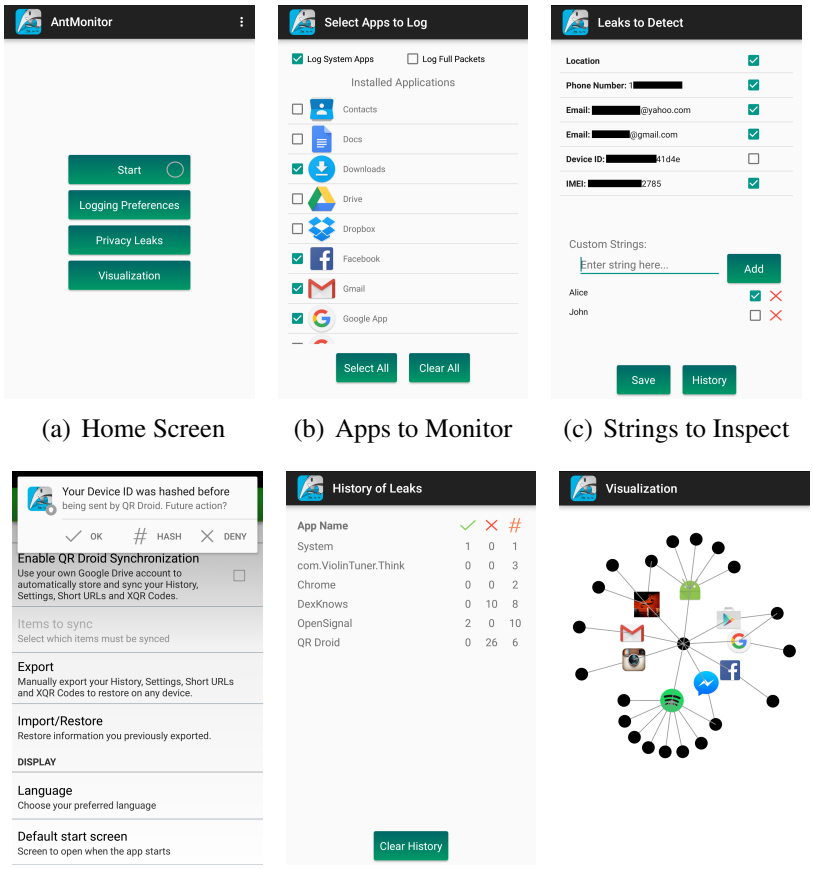
Mobile devices have become ubiquitous. The number of unique mobile users (3.6B) and the number of cellular subscribers (3B) [43] have reached half of the world population (7.2B) [54]. People spend more time on their mobile devices than on traditional desktop computers [18], and the majority of all IP traffic is generated by mobile devices, which will increase to two-thirds by 2019 [20]. Other than the sheer volume of activity generated by mobile devices, the devices themselves are increasingly personal due to their wide range of personal activities (from communication to financial transactions) and the personally identifiable information (PII) available on the devices (device and user ids, contact information, location, *etc.*). Therefore, looking at network activity from the mobile device's point of view is of interest to network operators and individual users alike.

There is a rich body of literature [63, 19, 26, 58, 37, 8, 55, 61] which studies network traffic traces, and the approaches typically fall into one of two categories: either large-scale but coarse-grained traces obtained in the middle of the network, *i.e.*, traces from Internet Service Providers (ISP) [63, 19], or fine-grained but small-scale traces from a limited set of users [26, 58]. These limitations, privacy concerns, and performance bottlenecks have hindered progress in this area.

In this thesis, we bridge that gap through the design and applications of AntMonitor: a system for collection and analysis of fine-grained, large-scale network measurements from mobile devices. AntMonitor is well positioned to become a high-performance passive monitoring tool for crowd-sourcing a range of mobile network measurements, as it combines the following desired properties: (i) it is easy to install (it does not require administrative privileges) and use (it runs as a service app in the background); (ii) it scales well; (iii) it provides users with fine-grained control of which data to monitor or log; (iv) it supports real-time analysis on the device (thus enhancing privacy) and/or on a server; and (iv) it allows for semantic-rich annotation of packet traces.

First, we design AntMonitor as a VPN-based service, which is the only way today to intercept all packets without rooting the phones. We develop and compare two versions of the architecture: Client-Server and Mobile-Only. They both use a VPN client on the device to intercept packets. While Client-Server routes them through a remote VPN server, Mobile-Only translates all connections on the device itself. In both versions, there is a separate logging server for uploading logs from the device for subsequent analyses. We show that, thanks to the efficient design and careful, extensive optimizations, AntMonitor Mobile-Only significantly outperforms all existing mobile-only approaches in terms of throughput and energy, without significantly impacting CPU and memory usage, and is therefore, the system we propose in this thesis. Specifically, AntMonitor Mobile-Only achieves 2x and 8x (the downlink and uplink) throughput of state-of-the-art mobile-only approaches, namely Privacy Guard [56] and Haystack [48], while using 2–12x less energy. The achieved throughput is also at 94% of the throughput without VPN.

Second, we argue that AntMonitor naturally lends itself as an ideal platform for a range of applications that build on top of passive monitoring, which can be of interest to individual users, network operators, and researchers. First, we use AntMonitor to detect when *leakage of private information* from the device to the network occurs. To the best of our knowledge, AntMonitor is the only app that can provide real-time detection and prevention today, together with insight into the destinations the private information leaks to. Second, we use AntMonitor for *passive*



(a) Home Screen (b) Apps to Monitor (c) Strings to Inspect

(d) Privacy Leak Alert (e) History of Leaks (f) Visualization

Figure 1.1: Screenshots of the AntMonitor prototype. (a)(b) apply to all uses of the app. (c-f) are specific to Privacy Leaks. For example, (f) shows which apps leak data to which destinations (typically trackers and ad servers).

performance measurements network-wide (*e.g.*, network performance maps) as well as per-user (usage profiles). With AntMonitor, this information comes at no additional bandwidth overhead and can provide input into network provisioning. Third, we use the packet traces collected by AntMonitor, annotated with rich contextual information, to train machine learning models for *traffic classification of flows to applications* using only TCP/IP header features. We show that we can achieve higher classification accuracy than state-of-art classification methods that use HTTP payload [47]. We present results from a pilot user study at UCI to demonstrate the capabilities of AntMonitor and its enabling potential for these applications. Fig. 1.1 shows some screenshots of AntMonitor.

The structure of the rest of the thesis is as follows. Chapter 2 presents related work. Chapter 3 describes the design and implementation of the AntMonitor system, including objectives, design, and implementation of two architectures: Client-Server and Mobile-Only. Chapter 4 presents the performance evaluation and compares the two designs to each other and to state-of-the-art approaches. Chapter 5 describes the extensions and applications of AntMonitor to three domains, namely: privacy leak detection and prevention (Chapter 5.1); passive monitoring of network performance (Chapter 5.2.1); and application classification (Chapter 5.2.2). Chapter 6 concludes the thesis and outlines directions for future work.

Chapter 2

Related Work

2.1 Other Monitoring Approaches

Work on monitoring network traffic generated by mobile devices can be roughly classified according to the vantage point and measurement approach.

OS approaches. Using a custom OS or a rooted phone one can get access to fine-grained information on the device, including passive monitoring of packet-level network traffic, typically using packet capture APIs such as `tcpdump` or `iptables-log`. Examples include `Phonelab` [8] and others [58, 26, 61]. This is a powerful approach but inherently limited to small scale-deployment as the overwhelming majority of users do not have rooted phones, and wireless providers and phone manufacturers strongly discourage rooting.

Active Measurements from Mobile Devices. There are mobile apps, developed by researchers `Netalyzr` [31], `Mobilyzer` [44] or the industry (*e.g.*, `Speedtest`, `CarrierIQ` or `Tutella`), to perform active network measurements of various metrics (throughput, latency, RSS) from the mobile device. They run at user space, without rooting the phone, and allow for accurate measurements. However, care must be put to not burden the device's resources and crowdsourcing is often used to

distribute the load (see Chapter 2.3).

Passive Monitoring inside the Network. ISPs and other organizations sometimes passively capture mobile network traffic on links in the middle of their networks, *e.g.* at an ISP's or other organization's network [15, 21, 28]. Researchers typically analyze network traces collected by others (*e.g.* large tier-1 networks [63] or from university campus WiFi networks [19]). Limitations of this approach include that (i) it only captures traffic going through the particular measurement point and (ii) it has access only to packet headers (payload is increasingly encrypted), not to ground truth or semantic-rich info. (*e.g.* apps that produced the packets).

2.2 VPN-Based Mobile Network Monitoring

The approach we follow in `AntMonitor` is passive monitoring on the device, guided by the design objectives in Chapter 3.1. In particular, the only way to intercept every packet in and out of the mobile device, while running at user space (without root privileges or custom OS), today, is to establish a VPN service on the device and run it seamlessly in the background. There are two VPN approaches: client-server and mobile-only, described below and compared on Table 2.1.

In **Client-Server VPN** approaches, packets are tunneled from the VPN client on the mobile device to a remote VPN server, where they can be processed or logged. A representative of this approach is `Meddle` [47], which builds on top of the `StrongSwan` VPN software. Additional tools have been built on top of `Meddle`: to detect content manipulation by ISPs and traffic differentiation [34], and to detect privacy leaks [49] (described in more detail in Chapter 2.3). Disadvantages of this approach include the fact that packets are routed through a middle server thus posing additional delay and privacy concerns, lack of client-side annotation (thus no ground truth available at the server), and potentially complex control mechanisms (the client has to communicate the selections of functionalities, *e.g.*, ad blocking, to the server). An advantage of the client-server VPN-based

	Discussion	Which is Better
Performance	The Mobile-Only approach has significantly higher network throughput and latency, with similar CPU, memory, and battery consumption (see Evaluation Chapter 4). Furthermore, the Client-Server performance heavily relies on the location of the VPN servers.	Mobile-Only
Scalability	The Mobile-Only approach scales better as it does not require a server component.	Mobile-Only
Privacy	With the Mobile-Only approach, all data, including sensitive ones, never traverses to a third-party server.	Mobile-Only
Routing Path	The Mobile-Only approach preserver the routing paths of the IP datagrams while the Client-Server approach alter the paths.	Mobile-Only
User Management	With the Client-Server approach, the server represents the users when requesting data: from the service provider (*e.g.*, YouTube) point of view, he is serving the server IP address. Therefore, the server becomes liable when the users abuse the service, *e.g.*, to download illegal content. This makes it challenging to deploy Client-Server approach on a large scale.	Mobile-Only
Traffic Patterns	The Mobile-Only approach breaks traffic patterns, including inter-arrival time, burstiness, latency, and datagram size. These information might be of critical importance in some application, *e.g.*, network flow classification if performed on a server (see Chapter 5.3).	Client-Server
Seamless Connectivity	The Client-Server approach can provide seamless connectivity when a user traverses between different (Wi-Fi and cellular) networks as they can maintain their connectivity with a static (server) IP.	Client-Server
Security and Other Services	The Client-Server approach enables the possibility of offering to users additional benefits, including encryption and dynamic IP location (*e.g.*, as traditional VPN services), data compression (*e.g.*, Onavo and Opera Max).	Client-Server

Table 2.1: Comparison Between Client-Server and Mobile-Only VPN Approaches

approach is that it can be combined with other VPN and proxy services (*e.g.*, encryption, private browsing) and can be attractive for ISPs to offer as an added-value service.

In **Mobile-Only VPN** approaches, the client establishes a VPN service on the phone to intercept all IP packets and does not require a VPN server for routing. It extracts the content of captured outgoing packets and sends them through newly created protected UDP/TCP sockets [12] to reach Internet hosts; and vice versa for incoming packets. This approach may have high overhead due to this layer-3 to layer-4 translation, the need to maintain state per connection and additional processing per packet. If not carefully implemented, this approach can significantly affect network throughput: for example, see the poor performance of `tPacketCapture` [10] – an application currently available on Google Play that utilizes this mobile-only approach. Therefore, careful implementation is crucial to achieve good performance.

Two state-of-the-art representatives of the mobile-only approach are `Haystack` [48, 32] and `Privacy Guard` [56]. They both focus on applying and optimizing their systems for detection of PII leaks. `Haystack` is currently in beta-testing mode, with a paper under submission [48, 32], and it is the closest baseline for comparison to `AntMonitor Mobile-Only`. It analyzes app traffic on the device, even if encrypted in user-space, and it does not require root permissions or a server in the

middle. In terms of implementation, our evaluation shows that AntMonitor Mobile-Only can achieve 2x and 8x the downlink and uplink throughput, as discussed in Chapter 4. ICSI's previous Netalyzr tool has also been adapted for mobile and used to detect private information leakage [31] via HTTP Header Enrichment [60, 57]. Privacy Guard is another recent paper that adopts the mobile-only design, albeit with some different implementation choices, which lead to inferior performance when compared to both AntMonitor and Haystack (see Chapter 4).

2.3 Applications of VPN-Based Monitoring

Privacy Leaks. Next, we review work on detecting private data leaking out of a device, which is related to our first application in Chapter 5.1. Some approaches require a custom OS or rooting the phone.[1] Another approach is to allow the user to define strings (*e.g.*, IMEI, device id, email, or any string corresponding to sensitive information that the user wants to protect) and then monitor for potential leaking of that information from the device to the network. AntMonitor as well as others Haystack, Privacy Guard, follow this approach: they monitor, on the device itself, the payload of all outgoing packets, searching for the predefined string.[2] Although the goal is the same, implementation matters: to the best of our knowledge, AntMonitor is currently the only tool that can provide prevention (*i.e.*, blocking or hashing of the private string, once the leakage is detected), in addition to detection, on the mobile-device without root privileges; AntMonitor and Privacy Guard can perform real-time detection, while Haystack does not yet.

Recon [49] also inspects packets for privacy leakage but, because it builds on top of Meddle [47], all packet processing (including privacy leaks detection) happens not on the device itself but on the Meddle server, with all the advantages and disadvantages of a client-server VPN discussed in

[1] TaintDroid [25] was one of the early tools built to identify privacy leaks in real-time. MockDroid [16] and AppFence [30] are examples of tools that dynamically intercept any permission request to certain resources. AndroidLeaks [29] and PiOS [24] are examples of tools that use static analysis to discover leaking APIs.

[2] In order to detect leaked strings in encrypted traffic, all three tools need a TLS proxy to first decrypt the traffic before string matching.

Chapter 2.2. Recon is also the first to use machine learning to identify flows that leak private data without prior knowledge of the users' PII, based on HTTP features and training on user feedback as well as on ground truth, manually obtained. This approach is also applicable to AntMonitor.

Performance Monitoring. Crowdsourcing is a powerful way to share the monitoring load among end-users and also to obtain diverse measurements from several locations. Successful such projects include Speedtest [6], OpenSignals [45], Sensorly [52], and RootMetrics [50]; these have released mobile applications that allow users to perform and report active measurements. These companies often release performance reports [46] and awards [51]) for cellular and Wi-Fi at points of interest (*e.g.*, metro areas, airports, tsports venues etc). Work in [55] analyzed crowdsourced data from Speedtest in order to compare the performance of cellular and WiFi networks in large metro areas.

Beyond informing the end-users, performance maps are of great interest to cellular and WiFi providers, who typically outsource monitoring to third party companies. Carrier IQ [17], is a well-known solution used by providers: it is embedded in the low level firmware of over 150 millions smartphones and reports network information, signal strength and the users' location; recently, there have been privacy concerns w.r.t. this platform [41]. Tutella provides a network performance *SDK*, which can be embedded in other mobile applications. Mobilyzer [44] also developed an open platform for performing controllable mobile network measurements in a principled manner, coordinated by a server. Work in [23] released an app for crowdsourcing measurements of throughput and latency over LTE and WiFi.

Learning. Several recent papers [22, 42, 63] perform *app classification* of flows by building mobile app signatures from unencrypted HTTP sessions in network traces: in [63], the HTTP User-Agent field was used to map flows into apps; in [22], HTTP header key-value pairs were used to build unique app signatures that operate on a per-flow basis; in [42], more flows could be identified by expanding the usage of tokens in the HTTP header (beyond HTTP request data) and by propagating the identification of a flow mapped to a specific app to other flows that occur at the

same time. All these methods rely on HTTP headers whereas we perform app classification using only TCP/IP headers in Chapter 5.2.2.

Early work on *behavioral analysis*, preceding mobile devices, that classified protocols based on packet headers includes: graphlets [35], profiling the end-host [36]; traffic dispersion graphs (TDGs) [33], and subflows [62]. We can go beyond protocols, and classify traffic to specific apps.

Chapter 3

The AntMonitor System

3.1 Design Rationale

Here we describe the main objectives of `AntMonitor` and the key design choices made to meet the objectives.

Objective 1: Large-Scale Measurements: We would like to use `AntMonitor` to crowdsource data from a large number of users, which poses a number of system requirements. First, the app on the mobile device must run without administrative privileges (root access). To that end, we use the public Virtual Private Network (VPN) API [12] provided by the Android OS (version 4.0+), which runs on more than 95% of Android devices [2]. Thus, the app can run on billions of Android devices today[1]. Second, in order for a large number of mobile users to adopt it, user experience must not be affected: the monitoring tool must run seamlessly in the background while the user continues to use the mobile device as usual, and the overhead on the device must be negligible in terms of network throughput, CPU, battery, and data cost. Third, the performance of any server used for data collection and analysis must scale with the number of users.

[1] VPN API has also just been released on iOS version 9.0+ since Sep. 2015; thus, our approach can be implemented on iOS as well.

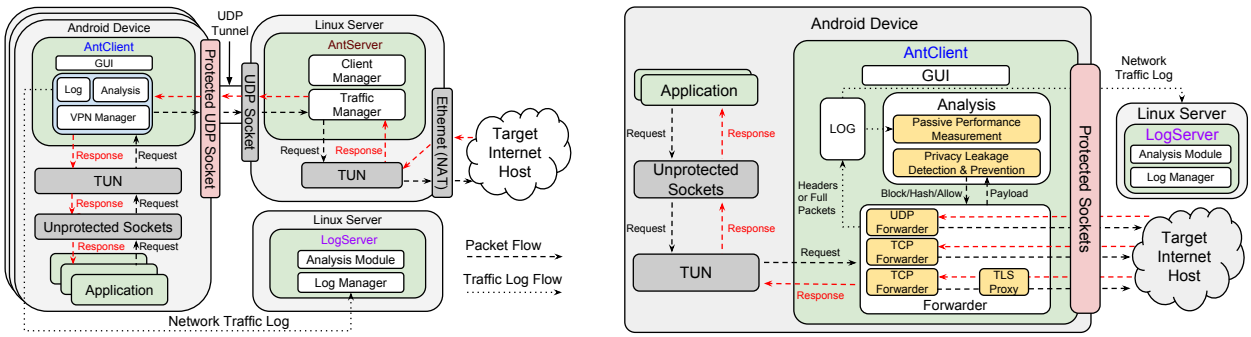
Objective 2: Making it Attractive for Users: In addition to the technical aspects of scalability, there must be incentives for users to participate. To that end, AntMonitor is designed with the capability to offer users a variety of services. The current prototype offers enhanced privacy protection (*e.g.*, preventing leakage of private information) and visualizations that help users understand where their data flows (*e.g.* see Fig. 1.1(f)). Other services could be implemented completely on the client side, such as enhanced wireless network performance (*e.g.*, increase data rates by switching among available networks; see Sec. 5.2.1). Finally, AntClient is designed to provide users with control over which data they choose to contribute to the AntMonitor logging system, *i.e.*, which applications to monitor, and whether to contribute full packets or headers only.

Objective 3: Fine-Grained Information: AntMonitor supports full-packet capture of both incoming and outgoing traffic. It collects packet traces in PCAP Next Generation format [7], which allows to append arbitrary information alongside the raw packets. This additional capability is very important because in many cases, the contextual information may only be collected accurately at the client side at the time of packet capture, and it can play a critical role in subsequent analyses. In particular, contextual information that AntMonitor can collect include names of apps that generate those packets (thus providing the ground truth for application classification), location, background apps, and information about the network used (network speed, signal strength, *etc.*).

3.2 System Design

To support the above main objectives, AntMonitor is designed to provide the following 4 main functionalities: traffic interception, routing, logging, and analysis. Here we discuss the first two functionalities and the latter two are discussed in Sec. 3.3.

Traffic Interception. The mobile app, called AntClient, establishes a VPN service on the device that runs seamlessly in the background. The service is able to *intercept all outgoing and incoming IP datagrams* by creating a virtual (layer-3) TUN interface [12] and updating the routing table so



(a) Client-Server Design (b) Mobile-Only Design

Figure 3.1: AntMonitor System Architecture

that all outgoing traffic, generated by any app on the device, is sent to the TUN interface. AntClient then routes the datagrams to their target hosts on the Internet (as described below). When a host responds, the response will be routed back to AntClient, and AntClient then sends the response packets to the apps by writing them to TUN.

Traffic Routing. To route IP datagrams generated by the mobile apps and arriving at the TUN interface, the intuitive option would be to use raw sockets. However, this option is not available on non-rooted devices. Therefore, the datagrams have to be sent out using layer-4 (UDP/TCP) sockets, which can be done in two ways:

1. Client-Server Routing: In our preliminary work [38], we used a server (AntServer), to assist with the routing of IP datagrams, as depicted on Fig. 3.1(a). This design is similar to the design of VPN services [13, 47]. In particular, in our approach, AntClient sends the datagrams out through a UDP socket to AntServer on the cloud, which further routes the datagrams towards their destinations. To avoid having the outgoing data of this socket looped back to the TUN interface, AntClient uses a protected UDP socket [12].

The main advantage of this client-server design is the simplicity of implementation: the routing is done seamlessly by the operating system at the server with IP forwarding enabled. However, as a crowdsourcing system, the requirement of AntServer faces challenges on scaling the system up

to support a large number of users. Furthermore, users may not want their traffic to change path. Therefore, we designed and implemented an alternative routing approach that can be performed entirely on the mobile device, without the need of `AntServer`, as described next.

2. Mobile-Only Routing: Routing IP datagrams to target hosts directly through layer-4 sockets requires a *translation between layer-3 datagrams and layer-4 packets*. In other words, for outgoing traffic, data of the IP datagrams has to be extracted and sent directly to the target hosts through UDP/TCP sockets. When a target host responds, its response data is read from the UDP/TCP sockets and must be wrapped in IP datagrams, which are then written to the TUN interface. To this end, we have designed and implemented a new component, called *Forwarder*, that takes care of this translation. How the Forwarder fits into the design of `AntMonitor` is shown in Fig. 3.1(b).

This new Mobile-Only design removes the dependency on `AntServer`; thus, it allows `AntClient` to be self-contained and makes `AntMonitor` easy to scale. Furthermore, this design enhances users' privacy as all data can now stay on the mobile device and is not routed through a middle-box. Nevertheless, each approach has its own merits and disadvantages. We provide a detailed comparison between Client-Server and Mobile-Only approaches in Table 2.1.

3.3 System Implementation

The Client-Server and Mobile-Only approaches share interception, logging, and analysis components, but they differ in the routing component. In the interest of space, we describe only the design of the Mobile-Only approach, which is our focus, and we refer the reader to our previous work for the implementation of the Client-Server approach [38].

3.3.1 Android Application: `AntClient`

The **Graphical User Interface** allows the user to turn the VPN service on and off and to select which applications are permitted to contribute to the data collection. Furthermore, advanced users can choose to contribute full packets or headers only. Fig. 1.1 shows screenshots of `AntClient`'s GUI.

The **Forwarder** manages the TUN interface and is in charge of routing network traffic. The Forwarder consists of two main components: UDP and TCP Forwarder (Fig. 3.1(b)).

The UDP Forwarder is the simpler component as UDP connections are stateless. When an app sends out an IP datagram containing a UDP packet, the UDP Forwarder records the mapping of the source and destination tuples, where a tuple consists of an IP address and a port number. This mapping is used for the reverse lookup later on. The Forwarder then extracts the data of the UDP packet and sends the data to the remote host through a *protected* UDP socket. When a response is read from the UDP socket, the Forwarder creates a new IP datagram, and changes the destination tuple to one that corresponds to the source tuple in the recorded mapping. The datagram is then written to TUN.

The TCP Forwarder works like a proxy server. For each TCP connection made by an app on the device, a TCP Forwarder instance is created. This instance maintains the TCP connection with the app by responding to IP datagrams read from the TUN interface with appropriately constructed IP datagrams. This entails following the states of the TCP connection (LISTEN, SYN_RECEIVED, ESTABLISHED, *etc.*) on both sides (app and TCP Forwarder) and careful construction of TCP packets with appropriate flags (SYN, ACK, RST, *etc.*), options, and sequence and acknowledgment numbers. At the same time, the TCP Forwarder creates an external TCP connection to the intended remote host through a *protected* socket to forward the data that the app sent to the server and the response data from the server to the app.

The **Log Module** writes packets (or just packet headers) to log files and uploads them to `LogServer`. This module is able to add rich contextual information to the captured packets by using the PCAP Next Generation format [7]. For instance, we currently store application names and network statistics (discussed in detail in Sec. 5.2.1) alongside the raw packets. The mapping to app names is done by looking up the packets' source and destination IPs and port numbers in the list of active connections available in `/proc/net`, which provides the UIDs of apps responsible for each connection. Given a UID, we can get the corresponding package name using Android APIs. Finally, Log Module periodically uploads the log files to `LogServer` during idle time, *i.e.*, when the device is charging and has Wi-Fi connectivity.

The **Analysis Module** can accommodate both off-line and online analyses on intercepted packets. The online capability allows it to take action on live traffic, *e.g.*, preventing private information from leaking. Since the analyses are done at the client side, private information is never leaked out of the device, setting `AntMonitor` apart from systems like `Meddle` [47], that perform leakage analysis at the VPN server. Since we require plain text in order to perform deep packet inspection, and much of the traffic is encrypted, we developed a TLS proxy that uses the SandroProxy library

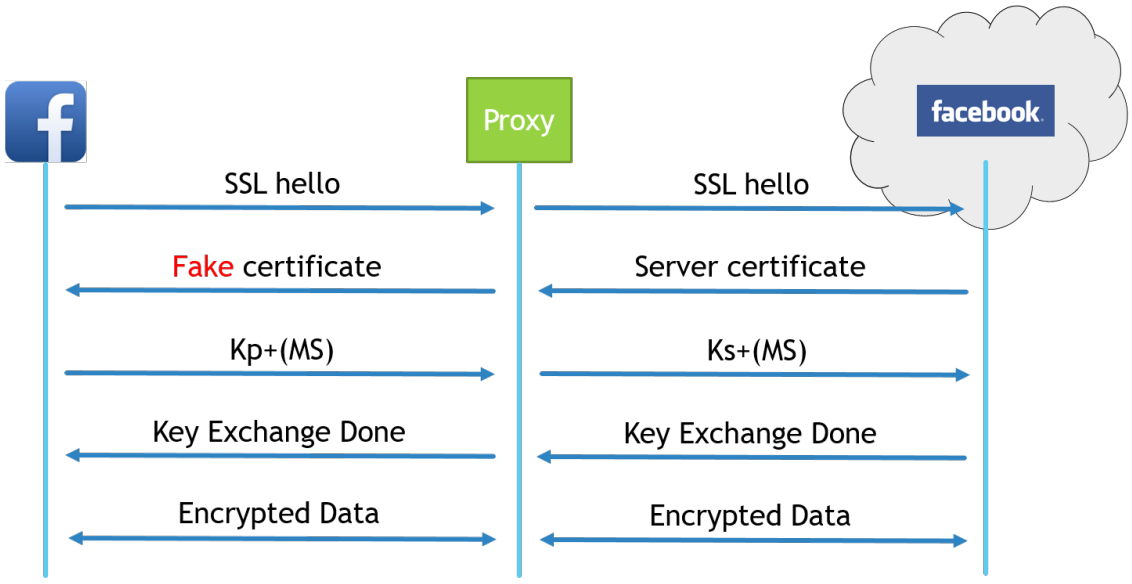


Figure 3.2: TLS Interception: Kp+ and Ks+ stand for the public key of the proxy and server, respectively.

[9], also used by Privacy Guard, to intercept secure connections. Upon install time, AntMonitor asks the user to install a root certificate. This certificate, by default, is trusted by all other apps. When an app wants to establish a secure connection, the SSL hello is intercepted by the proxy, as shown in Figure 3.2. The proxy then sends a fake certificate back to the app, which is signed by the root certificate that was installed by the user. The app and the proxy then finish negotiating keys using the fake certificate, and the proxy and the server exchange keys using the actual server certificate. Once the key exchange is complete, the proxy can successfully decrypt packets coming in from the app and from the server, and then re-encrypt them before forwarding. This method works for most apps, but it cannot intercept traffic from highly sensitive apps, such as banking apps, that use certificate pinning. These apps only trust locally stored certificates, and will not trust the installed root certificate. Due to the intrusive nature of intercepting TLS/SSL traffic, we allow users to disable this option at any time.

3.3.2 Data Collection Server: LogServer

The **Log Manager** supports uploading of files using multi-part content-type HTTPS. For each uploaded file, it checks if the file is in proper PCAPNG format. If so, for each client, the manager stores all of its files in a separate folder.

The **Analysis Module** extracts features from the log files and inserts them into a MySQL database to support various types of analyses. Compared to the Analysis Module of AntClient, this module has access to the crowdsourced data from a large number of devices, making it suitable for global large-scale analyses. For instance, it could detect global threats and outbreaks of malicious traffic.

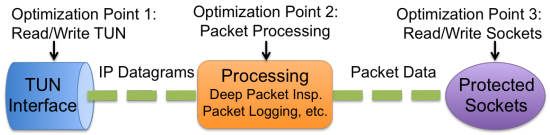


Figure 3.3: Performance Optimization

3.4 Performance Optimization

Since `AntClient` processes raw IP datagrams in the user-space, it is highly non-trivial to achieve high network performance. We have investigated the performance bottlenecks of our approaches specifically and VPN approaches in general. The bottleneck points are depicted in Fig. 3.3. We then address the bottleneck points through a combination of optimization techniques, from implementing custom native C libraries to deploying high-performance network IO patterns.

Traffic Routing (Point 1, 2, and 3). The techniques that we adopted are as follows: (i) we explicitly manage and utilize Direct ByteBuffer for IO operations with the TUN interface and the sockets, (ii) we store packet data in byte arrays, and (iii) we minimize the number of copy operations and any operations that traverse through the data byte-by-byte. These techniques are based on the following observations: Direct ByteBuffer gives the best IO performance because it eliminates copy operation when the actual IO is performed in native code. Plus, Direct ByteBuffer on the Android platform is actually backed by an array (which is not typically the case on a general Linux platform); therefore, it creates synergy with byte arrays: making a copy of the buffer to a byte array (for manipulation) can be done efficiently by performing memory block copy as opposed to iterating through the buffer byte-by-byte. (Memory copy is also used whenever a copy of the data is needed, *e.g.*, for IP datagram construction.) Finally, because the allocation of a Direct ByteBuffer is an expensive operation, we carefully manage its life cycle: for an IO operation, *i.e.*, read from TUN, we reuse the buffer for every operation instead of allocating a new one.

TUN Read/Write (Point 1). The Android public API does not provide a way to *poll* the TUN interface for available data. The official Android tutorial [3] as well as other systems [48, 56]

employed periodical sleeping time (*e.g.*, 100 ms) between read attempts. This results in wasted CPU cycles if sleeping time is small or slow read speed if the sleeping time is large, as the data may be available more frequently than the sleep time. To address this issue, we implemented a *native C library* that performs the native *poll()* operation to read data to a Direct ByteBuffer (which is then available in the Java code without costing extra copies).

It is also important to be able to read from (and write to) the TUN interface in large blocks to avoid the high overhead of crossing the Java-Native boundary and of the system calls (read() and write()). For instance, in an early implementation of our Mobile-Only approach, we observed that IP datagrams read from the TUN interface have a maximum size of 576 B (which is the minimal IPv4 datagram size). This results in the maximum read speed of about 25 Mbps on a Nexus 6 for a TCP connection, thus limiting the upload speed. We were then able to increase the datagram size by (i) increasing the MTU of the TUN interface (to a large value, *e.g.*, 16 KB) and (ii) including an appropriate Maximum Segment Size (MSS, *e.g.*, 16 KB) in the TCP Options field of SYN-ACK datagrams sent by TCP Forwarder when responding to apps' SYN datagrams. These changes effectively help to ensure that an app can acquire a high MTU (*e.g.*, 16 KB) when performing Path MTU Discovery, so that each read from TUN results in a large IP datagram (*e.g.*, 16 KB). This optimization results in the maximum read speed more than 80 Mbps on our Nexus 6. Similarly, it is also important to write to TUN in large blocks. For instance, in our Mobile Only approach, we construct large IP datagrams (*e.g.*, 16 KB) to write to TUN.

Socket Read/Write (Point 3). Similar to when interacting with the TUN interface, in order to achieve high throughput, it is important to read from (and write to) TCP sockets in large blocks. In particular, in our Mobile-Only approach, we matched the size of the buffer used for socket read (*e.g.*, 16 KB minus 40 B for TCP and IP headers) to the size of the buffer used for TUN write (*e.g.*, 16 KB). Similarly, we also matched the size of the buffer used for socket write to that of the buffer used for TUN read.

Thread Allocation (Point 2). We have fully utilized Java New I/O (NIO) with non-blocking

sockets for the implementation of the Forwarder. In particular, Forwarder is implemented as a high-performance (proxy) server, that is capable of serving hundreds of TCP connections (made by the apps) at once, while using only two threads: one thread is for reading IP datagrams from the TUN and another thread is for actual network I/O using the Java NIO Selector and for writing to TUN. Minimizing the number of threads used is critical on a resource constrained mobile platform to achieve high performance. As a baseline comparison, `Privacy Guard` creates one thread per TCP connection, which rapidly exhausts the system resources even in a benign scenario, *e.g.*, opening the CNN.com page could create about 50 TCP connections, which results in low performance (see Sec. 4).

Socket Allocation (Point 2). Since the Forwarder needs to create sockets to forward data and the Android system imposes a limit of 1024 open file descriptors per user process, sockets must be carefully managed. To this end, we minimize the number of sockets used by the Forwarder by (i) multiplexing the use of UDP sockets: we use a single UDP socket for all UDP connections, and (ii) carefully managing the life cycle of a TCP socket to reclaim it as soon as the server or the client closes the connection. For comparison, `Privacy Guard` uses 1 socket per UDP connection and 2 sockets per TCP connection.

Packet-App Mapping (Point 2). Android keeps active network connections in four separate files in the `/proc/net` directory: one each for UDP, TCP via IPv4 and IPv6. Because parsing these files is an expensive IO operation, we implemented the reading and parsing of these files in a native C library. Furthermore, to minimize the number of times we have to read and parse them, we store the mapping of app names to source/destination IPs and port numbers in a Hash Map. When the Log Module receives a new packet, it first checks the Map for the given IP and port number pair. If the mapping does not exist, the Log Module re-parses the `/proc` files and updates the Map.

Deep Packet Inspection (Point 2). Although inspecting every packet is costly, we leverage the Aho-Corasick algorithm [5] written in native C to perform real-time detection without significantly impacting throughput and resource usage (see Chapter 4.1.3). However, using the Aho-Corasick

algorithm alone is not enough. We must also minimize the number of copies we make of each packet. Although the algorithm generally operates on Strings, `AntClient` uses Direct ByteBuffers for efficient routing and creating a String out of a ByteBuffer object costs us one extra copy. Moreover, Java Strings use UTF-16 encoding and JNI Strings are in Modified UTF-8 format. This means that any String passed from Java to native C will require another copy while converting from UTF-16 to UTF-8 [11]. To avoid two extra copies, we pass the Direct ByteBuffer object and let the Aho-Corasick algorithm interpret the bytes in memory as characters. This technique enables us to perform an order of magnitude faster than Java-based approaches (Chapter 4.4).

Chapter 4

Performance Evaluation

Tool. In order to evaluate AntMonitor, we built a custom app – AntEvaluator. It transfers files and computes a number of performance metrics, including network throughput, CPU and memory usage, and power consumption. It helps us tightly control the setup and compute metrics that are not available using off-the-shelf tools, such as *Speedtest*.

Scenarios. We use AntEvaluator in three types of experiments. In Chapter 4.1, *Stress Test* performs downloads and uploads of large files so that AntMonitor has to continuously process packets. In Chapter 4.2, *Idle Test* considers an idling mobile device so that AntMonitor handles very few packets. In between the two extremes, we have also considered a *Typical Day Test*, which simulates user interaction with apps; however, due to time constraints, it was only performed on AntMonitor Client-Server.

Baselines. We report the performance of AntMonitor Mobile-Only and compare it to state-of-the-art baselines from Chapter 2.2:

- Raw Device: no VPN service running on the device; this is the ideal performance limit to compare against.

- State-of-the-art mobile-only approaches:

  `Privacy Guard` [56] v1.0 and `Haystack` [48] v1.0.0.8. (We omit the testing of `tPacketCapture` [10] since it was shown to have very poor performance in [38].)

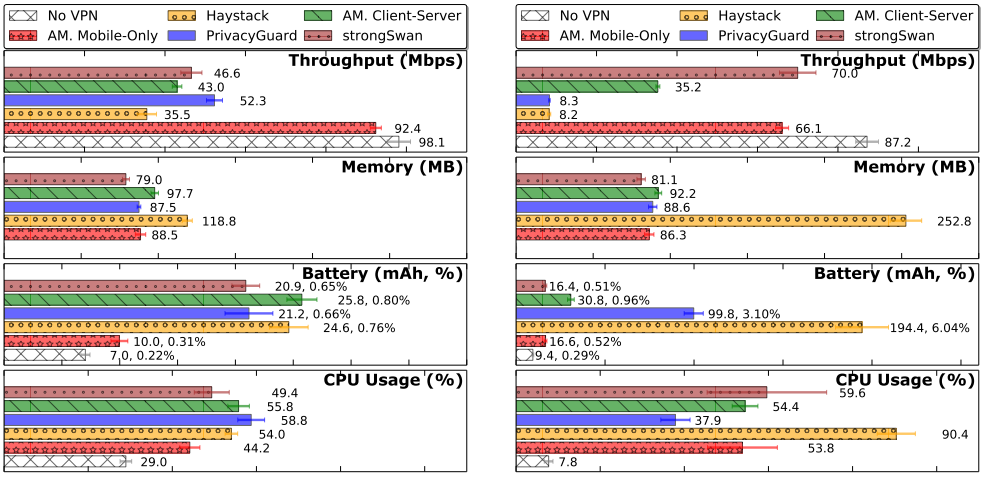
- Client-server VPN approaches: industrial grade `StrongSwan` VPN client v1.5.0 with server v5.2.1, and `AntMonitor Client-Server`. The VPN servers used by each app were hosted on the same machine.

Setup. All experiments were performed on a Nexus 6, with Android 5.0, a Quad-Core 2.7 Ghz CPU, 3 GB RAM, and 3220 mAh battery. Nexus 6 has a built-in hardware sensor, Maxim MAX17050, that allows us to measure battery consumption accurately. Throughout the experiments, the device was unplugged from power, the screen remained on, and the battery was above 30%. To minimize background traffic, we performed all experiments during late night hours in our lab to avoid interference, we did not sign into Google on the device, and we kept only pre-installed apps and the apps being tested. Unless stated otherwise, the apps being tested had TLS interception disabled and the `AntClient` was logging full packets of all applications and inspecting all outgoing packets. VPN servers ran on a Linux machine with 48-Core 800 Mhz CPU, 512 GB RAM, 1 Gbit Internet; the Wi-Fi network was 2.4Ghz 802.11ac. The files were hosted on a machine within the network of VPN servers. Each test case was repeated 10 times and we report the average.

4.1 Stress Test

4.1.1 Large File Over a Single Flow

Setup. For this set of experiments, we use `AntEvaluator` to perform downloads and uploads of a 500 MB file over a single TCP connection. In the background, `AntEvaluator` periodically measures the following metrics:



(a) Download (b) Upload

Figure 4.1: Performance of all VPN apps in Stress Test for a 500 MB file on Wi-Fi. "AM." stands for AntMonitor

A. Network Throughput: `AntEvaluator` reports the number of bytes transferred after the first 10 sec (to allow the TCP connection to reach its top speed) and the transfer duration. We use these numbers to calculate throughput.

B. Memory Usage: `AntEvaluator` uses the `top` command to sample the Resident Set Size (RSS) value.

C. Battery Usage: `AntEvaluator` uses the APIs available with the hardware power sensor Maxim MAX17050 to compute the energy consumption during each test in mAh [1].

D. CPU Usage: `AntEvaluator` uses the `top` command to measure the CPU usage.

At the end of each experiment, `AntEvaluator` reports the calculated throughput and battery usage, and the average memory and CPU (considering the sum of CPU usage of `AntEvaluator` and the VPN app) usage.

Results. Fig. 4.1(a) shows that the download throughput of `AntMonitor` Mobile-Only significantly outperforms all other approaches. It was able to achieve about *94% of the raw speed*, with throughput *2x more* than `StrongSwan` & `Privacy Guard` and *2.6x more* than `Haystack`. We

further note that all VPN apps tested have similar memory, battery, and CPU usage.[1] Fig. 4.1(b) reports the upload performance. AntMonitor Mobile-Only achieves *76% of the raw speed* while performing data logging and DPI. Most significantly, its performance is *8x faster* than both state-of-the-art mobile-only approaches.[2] StrongSwan outperforms AntMonitor as expected since, unlike with incoming packets, AntMonitor performs DPI on each outgoing packet. Nevertheless, Fig. 4.2(a) shows that AntMonitor Mobile-Only has the higher upload speed (and closest to the raw speed) if DPI is disabled. Fig. 4.1(b) also shows that all VPN apps have similar memory and CPU usage, except for Haystack, which incurs significant overhead. Since the test took longer for the slower approaches, Privacy Guard and Haystack used significantly more battery.

In general, using any VPN service roughly doubles the CPU usage during peak network activity. Although the CPU usage of 38–90% on Wi-Fi seems high, the maximum CPU usage on the quad-core Nexus 6 is 400%. In summary, this set of experiments demonstrates that among all VPN approaches, for both downlink and uplink, AntMonitor Mobile-Only has the highest throughput while having similar or lower CPU, memory, and battery consumption.

4.1.2 Small Files Over Multiple Flows

Setup. To test the efficiency of AntMonitor's thread and socket allocation, we used AntEvaluator to create 16 threads, each downloading a 50MB file. During the test AntEvaluator calculated the throughput of each flow and reported the average of all flows.

Results. The average speed of a flow (in Mbps) for each test case was the following: Raw Device: 6.82, AntMonitor Mobile-Only: 6.57, Privacy Guard: 4.75, StrongSwan: 3.73, Haystack: 3.18, and AntMonitor Client-Server: 3.06. Again, AntMonitor Mobile-Only came out on

[1] Although StrongSwan does not perform L3-L4 translation, it performs encryption and decryption, which results in about 5% higher CPU usage than AntMonitor Mobile-Only.

[2] The gains provided by the optimizations discussed in Chapter 3.4 have a greater impact on upload speeds because both Privacy Guard and Haystack favor downstream traffic since the responses from the Internet are read as streams from sockets and the responses from applications are read from TUN packet-by-packet [48].

top, achieving 96% of the raw speed.

4.1.3 Impact of Logging and DPI

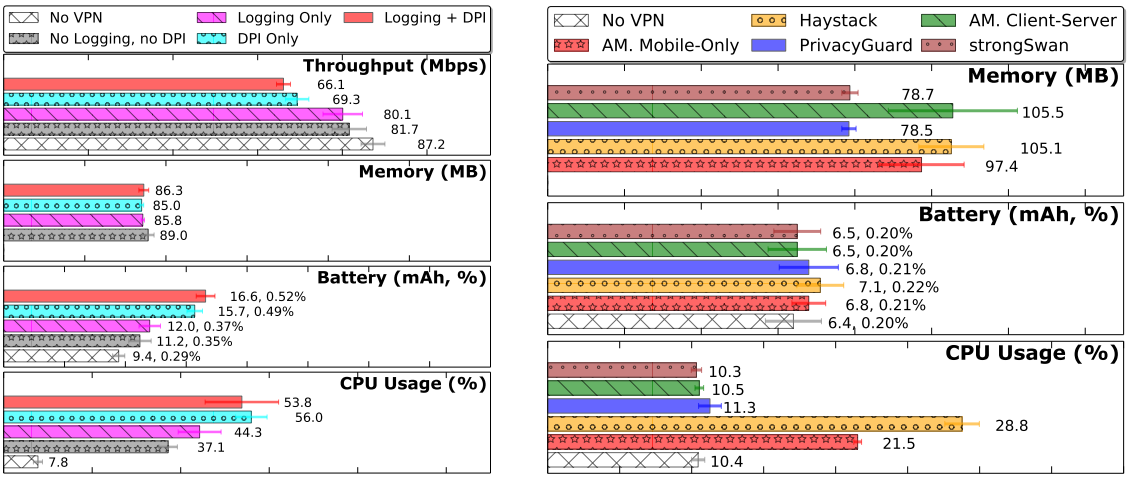
Setup. To assess the overhead caused by Logging Data and Deep Packet Inspection (DPI), we performed the single-flow upload stress test on AntMonitor Mobile-Only with all four combinations of Logging on/off and DPI on/off.

Results. First, Fig. 4.2(a) shows that logging does not have a significant impact on throughput. This is thanks to (i) the optimization of AntMonitor Mobile-Only that uses only two threads for network I/O (see Chapter 3.4) and (ii) the fact that the data collection uses two threads for storage I/O. These data logging threads do not significantly impact main network I/O threads on a quad-core Nexus 6 phone. Second, DPI is performed by one of the main network I/O threads and inflicts a 17% slow-down on upload speed. Although 17% is a significant overhead, AntMonitor Mobile-Only is still able to reach over 60 Mbps speed, which is more than enough for mobile apps. In addition, DPI causes a 28% and 33% overhead on battery and CPU, respectively. However, the CPU usage still remains 1/8 of the total possible CPU available on the Nexus 6 (of 400%), thus the overhead is acceptable. Finally, without logging and DPI, AntMonitor Mobile-Only achieves 94% of the raw speed without VPN.

4.1.4 Impact of TLS Proxy

In order to be able to inspect encrypted traffic for privacy leaks, we implemented a TLS proxy, described in Chapter 5.1.

Setup. To evaluate the performance impact of this proxy, we used AntEvaluator to download a 500MB file from a secure server over HTTPS and compared the throughput of AntMonitor-Mobile-Only to that of the Raw Device.



(a) 4 Variations of AM. Mobile-Only Upload

(b) Performance during device idle time

Figure 4.2: Performance Evaluation: 4 Variations of AM. Mobile-Only during an Upload Stress Test and performance of VPN apps during device idle time.

Results. The average throughput (in Mbps) was 77.2 and 69.1 for the `Raw Device` and `AntMonitor Mobile-Only`, respectively. As expected, the proxy causes a significant overhead since it uses an extra socket for each connection and performs one extra decryption and encryption operation per packet.

4.2 Idle Test

Setup. For this set of experiments, we kept the phone idle for 2 minutes with only background apps running. We used `AntEvaluator` to measure the battery and memory consumption of each VPN app. We also measured the aggregate CPU usage across all apps by summing the `System` and `User` % CPU Usage provided by the `top` command.

Results. Fig. 4.2(b) shows that all apps tested create very little additional overhead when the device is in idle mode. Among the mobile-only approaches, `Haystack` and `AntMonitor Mobile-Only` used more CPU than `Privacy Guard` because both of them have threads to log packets while `Privacy Guard` does not. Similarly, Logging also results in slightly higher memory usage

for Haystack and AntMonitor. (Note that StrongSwan does not log packets either, thus has lower CPU usage.) Finally, the overall memory usage of the VPN apps (˜105 MB) is acceptable; many other popular apps, *e.g.* Facebook, use as much as 200 MB of RAM.

4.3 Typical Day Test

We used the 2014 Nielsen Survey [40] to identify the 5 most popular mobile application categories and to mimic a typical day usage of a mobile user. Using the Finger Replayer application [14], we simulated a user that infrequently initiates some sessions on her mobile phone from various categories, and we made an informed decision in picking the number and length of sessions. We found that when on Wi-Fi, the simulation uses on average 13% of the battery with No VPN and 16% with an early version of AntMonitor Client-Server. These results show that in a more typical usage scenario than the stress test, AntMonitor uses only a modest amount of 3% additional battery, which will not significantly affect the average user experience. Moreover, we found no noticeable differences in response time of the phone, and no additional video buffering when running the test with AntMonitor Client-Server. Since AntMonitor Mobile-Only has shown better results in the Stress Test (Chapter 4.1) than AntMonitor Client-Server, we expect AntMonitor Mobile-Only would achieve similar, if not better, results in the typical day test.

4.4 Metrics Computed Outside AntEvaluator

Latency. We measured the latency of each VPN app by averaging over several pings to a nearby server (in the same city). In order of increasing delay, the apps rank as follows: NoVpn: 3 ms, StrongSwan: 4 ms, Haystack: 4 ms, AntMonitor Client-Server: 5 ms, AntMonitor Mobile-Only: 7 ms, and Privacy Guard: 83 ms. Compared to client-server approaches, mobile-only approaches cannot forward ICMP packets; thus, we measure latency using TCP packets. The

additional delay is due to the time required to create, send, and receive packets through TCP sockets. Compared to `Haystack`, `AntMonitor Mobile-Only` has a small additional latency as sending and receiving TCP packets involves two threads (one reads/writes the packet from/to TUN and one reads/writes the packet from/to the socket), whereas `Haystack` might have used a single thread (source code unavailable).

String Parsing. The main heavy operation required in DPI is string parsing. During real traffic conditions, our native C implementation of Aho-Corasick has a maximum run time of 25 ms. When benchmarking as a standalone library (running on the Android main thread alone), our parsing time is below 10 ms. For comparison, `Haystack` reports a 167 ms maximum run time for string parsing with Aho-Corasick.

Chapter 5

Applications

Because AntMonitor intercepts every packet in and out of the device, and it does so very efficiently, it is uniquely positioned to serve as a platform for supporting applications that build on top of this passive monitoring capability, and which may be of interest to operators as well as individual users. In this section, we consider three applications: (i) privacy leaks detection, (ii) performance monitoring, and (iii) traffic classification. To that end, we showcase results from a pilot user study at UC Irvine. The study involved 11 UCI users from our research group, who used AntClient on their phones during the period Feb. 5 – Nov. 30, 2015. AntMonitor collected the packets of apps that the volunteers selected (Fig. 1.1(b)) and logged them at LogServer.[1]

5.1 Privacy Leaks

Mobile devices today have access to personally identifiable information (PII) and they routinely leak it through the network, often to third parties without the user's knowledge. PII includes: (i) mobile phone IDs, such as IMEI (which uniquely identifies a device within a mobile network),

[1]The total volume was ˜20GB and ˜159GB for cellular and Wi-Fi data, respectively. The data contained 272 distinct apps, with a large majority of traffic being HTTP/HTTPS, and 84.3% of traffic being downstream.

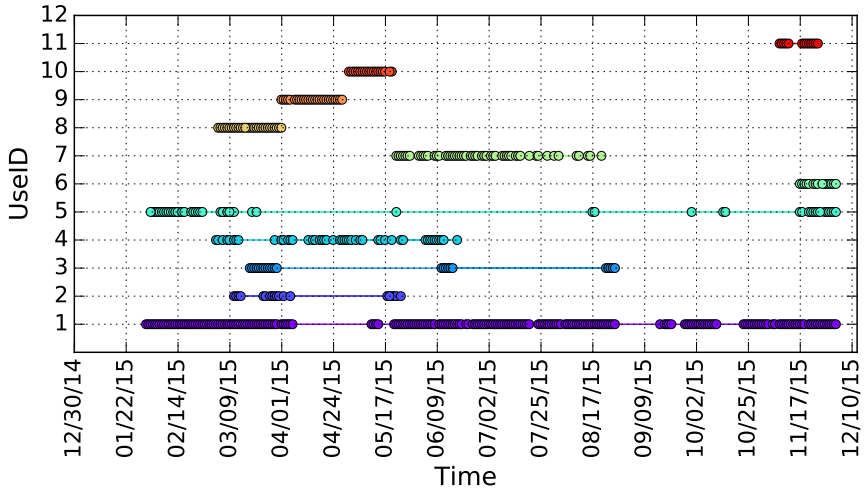


Figure 5.1: Data logged daily by different users in the study. (We have omitted some users, and we have included different devices belonging to the same individual (e.g. 7-11).)

and Android Device ID; and (ii) information that can uniquely identify the user (such as phone number, email address, or even credit card) or other information (e.g. location, demographics). The latter type of information is typically not stored on the phone; however, a user may input and send them to a friend in a previous communication, and the user wants to make sure that no other apps can sniff (*e.g.* keyboard apps) and send this information elsewhere. Sometimes sending out PII is necessary for the operation of the phone (*e.g.* a device must identify itself to connect to a wireless network) or of the apps (*e.g.* location must be obtained for location-based services). However, the leak may not serve the user (*e.g.* going to advertisers or analytics companies) or may even be malicious. Leaks in plain text can be intercepted by third parties listening *e.g.* in public WiFi networks. Although modern mobile platforms (Android, iOS, Windows) require that apps obtain permission before requesting access to certain resources, and isolate apps (execution, memory, storage) from each other, this is not sufficient to prevent information leaking out of the device, *e.g.* due to interaction between apps [27]. Even worse, users are unaware of how their data are used, [39].

Privacy Leaks Detection and Prevention Module. We extended the basic `AntMonitor` functionality with an analysis module that performs real-time DPI. The user can define strings that cor-

App Name	Leak Type	# Flows
VnExpress.net	IMEI, Phone#, Location, Email, DeviceID	33147
Zing Mp3	IMEI, DeviceID	14745
Clean Master	DeviceID	1768
WiFi Maps	IMEI, Location	1582
Relay for reddit	Location	638
System	IMEI, Location, DeviceID	301
Chrome	Location	143
ES File Expl.	DeviceID	96
MyFitnessPal	Location	76
DR Radio	DeviceID	68
Speedtest	Location	48
DexKnows	IMEI,Location	47
Skype	IMEI, DeviceID	42
Peel Smart Rmt	DeviceID	23
iWindsurf	IMEI, Location	22
WhatsApp	Phone#	20
...	...	...
All	All	52923

Domain Name	# Flows
eclick.vn.	8760
api.mp3.zing.vn.	7561
ksmobile.com.	620
zaloapp.com.	332
ngoisao.net.	209
api.staircase3.com.	150
openweathermap.org.	47
adtima.vn.	36
ads.adap.tv.	31
apps.ad-x.co.uk.	30
adkmob.com.	27
whatsapp.net.	27
mopub.com.	25
duapps.com.	25
api.dexknows.com.	24
server.radio-fm.us.	15
inmobi.com.	14
...	...
All	18020

Table 5.1: Flows Leaking PII found in the collected data. (Note: the number of flows at the left (52923 total) is higher than at the right (18020) because we count a flow that leaks multiple PIIs multiple times.)

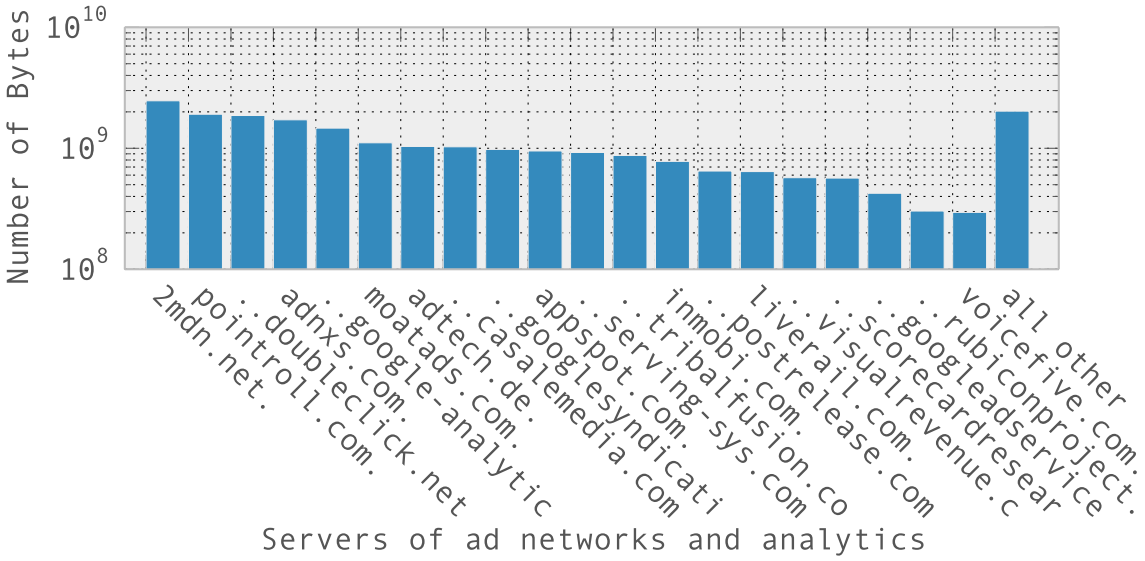


Figure 5.2: Amount of traffic sent towards ad servers and analytics services

respond to private information that should be prevented from leaking; see screenshot in Fig. 1.1(c). Before sending out a packet, the AntClient inspects it and searches for any of those strings. By default, if the string is found, AntMonitor hashes it (with a random string of the same length, so as not to alter the payload length) before sending the packet out, and asks the user what to do in the future for the given string/app combination, as shown in Fig. 1.1(d). The user can choose to allow future packets to be sent out unaltered, block them, or keep hashing the sensitive string (so that the application has a good chance to continue working but without the string being leaked). The system remembers the action to take in the future for the same app and "leak," and it will no longer bother the user with notifications. The user may also look at the history of the leaks, shown in Fig. 1.1(e). To the best of our knowledge, AntMonitor is the only tool today that provides both *real-time* detection and *prevention*, on the mobile-device (not at the server); this is possible thanks to its efficient implementation in Section 3.4, while Haystack and Privacy Guard do not achieve both; and Recon acts on the server, but it could gracefully run on top of AntMonitor as well.

Privacy Leaks Detected. We analyzed the data collected from our user study, and found a large number of privacy leaks in plain text. Table 5.1 presents the apps and destination domains with the highest number of flows leaking. The worst offender in the list was the app VnExpress.net that leaks five different types of PII up to 33,145 times towards the domain eclick.vn – an advertising network. The list of leaking apps includes very popular apps with tens of millions of downloads, such as Skype and WhatsApp, and the list of domains includes many mobile ad networks (such as mopub, inmobi, adkmob, adtima).

Prior work [59] collected data in the core of the network to show that tracking is prevalent. Using the data collected by AntMonitor at the users' mobile devices, we calculated the amount of data that is transmitted from/to ad networks, analytics and mediation services.[2] Fig. 5.2 shows that the amount of traffic transmitted towards such servers for the top 20 domains is in the order of GBs,

[2]We used the following process. First, we fetched known lists of ad servers [4, 59] that consist of hostnames that serve ads. Second, we extracted DNS queries and answers from the user data so as to get accurate mappings from hostnames to IP addresses, and we further complemented the mappings with reverse DNS results. By combining both, we were able to label individual IP addresses, found in our collected data, as being associated with ad servers.

consists of several domains, and tens of thousands of flows per domain. Fig. 1.1(f) visualizes the destinations for one device: it shows which apps leak information to which destinations. Raising awareness for the magnitude (B) of the tracking is important, and its cost in terms of data plans. AntMonitor can notify individual users about these leaks, real-time. In future work, we plan to combine this string matching-based detection with machine learning techniques proposed in [49].

5.2 Other Applications

5.2.1 Performance Measurements

By design, `AntMonitor` intercepts every packet and is thus able to passively compute performance indicators of the TCP/IP layer, such as throughput and latency. In addition, it can monitor performance at other layers (*e.g.*, the radio layer) and rich contextual information including but not limited to: (i) timestamp; (ii) geolocation in a way that achieves a low energy footprint[3]; (iii) network information (*e.g.*, WiFi or Cellular), radio access technology (RAT) and detailed cellular network information per region (*e.g.*, LTE parameters, frequency bands); (iv) received signal strength (RSS) or (v) throughput and latency measurements per app and overall. This information comes for free to `AntMonitor` (*i.e.*, without additional CPU or bandwidth overhead) and is interesting to users (*e.g.*, to manage their network access or cost) as well as to operators (*e.g.*, to assess and improve their infrastructure [21]).

For instance, in our pilot deployment of `AntMonitor`, we compared throughput measurements from a state-of-the-art *active* monitoring tool (Speedtest) vs. *passively,* (using `AntMonitor-Client-Server`). We saw that the values they computed were very close, but the passive approach did not incur any measurement overhead. In addition, we used the data collected by `AntMonitor`

[3] `AntMonitor` listens to location updates by other applications passively and only supplements that with infrequent active requests.

to build graphs of daily traffic usage both with Wi-Fi and cellular. This type of information can be useful to the end-user, *e.g.*, to make her aware of usage patterns and to give her more control. Although simple statistics of this type are currently reported by mobile devices, they are typically at a very coarse granularity (*e.g.*, total amount of data left for this month). In contrast, `AntMonitor` can report data at fine granularity (*e.g.* per app, per location, over time *etc.*) and can also monitor speed and other performance metrics. Finally, `AntMonitor` can be used to crowdsource performance measurements and to build performance maps, which can provide a comprehensive view of network-wide performance and can guide control actions. In our user study, we built a performance map of the university campus and found that LTE reception is poor on many areas and has large spatial variation. We also found that low RSRP values do not always correlate with low cellular throughput, and thus it is worth to jointly study performance at both layers.

5.2.2 Traffic Classification

`AntMonitor` can enable learning traffic profiles at different granularities using only features extracted from TCP/IP headers passively monitored. In this section, we demonstrate the capability of (i) flow classification to mobile apps they belong to, and (ii) learning user profiles from the apps they use. These can be useful building blocks for anomaly detection (at the device), traffic differentiation (by the ISP), market research, etc. It is important to be able to perform these functions using only packet headers, because payload inspection is costly and invasive; and it may not even be possible as HTTP traffic is moving towards HTTPS. Training and classification can be performed on the device (Log module in the `AntClient`) and/or at the `LogServer` (where data is contributed by multiple devices). In the rest of the section, we report results from the latter.

First, we use packet headers collected in the user study, together with app names that generated the traffic, to train models and classify flows to mobile apps. Using off-the-shelf learning tools (*e.g.* Random Forest), we are able to achieve an F1-score of up to 78%. To put this number into

perspective, Meddle reports a 64.1% precision score in classifying flows for the 92 most popular Android applications by using payload features (Host and User-Agent) [47]. For a dataset of millions of applications, the state-of-the-art approach of AppPrint [42], that also requires HTTP header data, achieves 81% flow-set coverage with 91% precision.

Second, we asked whether the users in our study group (see Fig. 5.1) can be distinguished from each other using their daily app activity. We model each user with a vector that represents their normalized activity volume over all apps in one day. One interesting fact in our dataset is that certain users have re-installed AntMonitor during the study and appear with different user ids. For example, users 7-11 in Fig. 5.1 are different devices used by the same person over different time periods. We used supervised learning in which we included users 1-7 in the training dataset and users 1-11 in testing. Users 1-7 were correctly classified as themselves, and users 8-10 were mostly classified as user 7, which is also correct. Interestingly, user 11 was classified as user 1, which also makes sense: during that period the 2 users were working on the same paper deadline and were using their phones for running similar apps for testing and performance evaluation. These results are preliminary but demonstrate AntMonitor's potential for enabling user profiling and anomaly detection, a direction we plan to explore in the future.

Chapter 6

Conclusion

In this thesis, we presented AntMonitor – a system for crowdsourcing large-scale, yet fine-grained network measurements from mobile devices. Thanks to its careful design and implementation, it significantly outperforms all state-of-the-art mobile-only approaches: it achieves 2x and 8x faster (down and uplink) speeds, and close to the raw no-VPN throughput, while using 2–12x less energy. Our pilot deployment at UCI shows that AntMonitor can enable network research and applications, including privacy leak detection and prevention, network performance measurements, and traffic classification. We plan to soon open up the beta testing on GooglePlay (to reach a wide range of users) and to eventually make it open-source (as a monitoring platform for the research community).

Bibliography

[1] Android BatteryManager API. http://developer.android.com/reference/android/os/BatteryManager.html.

[2] Android Versions. developer.android.com/about/dashboards.

[3] Google, android toyvpn example. https://android.googlesource.com/platform/development/+/master/samples/ToyVpn/src/com/example/android/toyvpn/ToyVpnService.java.

[4] List of ad server hostnames. http://pgl.yoyo.org/as/.

[5] Multifast. http://multifast.sourceforge.net/.

[6] Ookla Speedtest. https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest.

[7] PCAPNG File Format. http://goo.gl/y89d9U.

[8] PhoneLab, University at Buffalo. https://www.phone-lab.org/.

[9] Secure android proxy. https://code.google.com/archive/p/sandrop/.

[10] tPacketCapture. www.taosoftware.co.jp/en/android/packetcapture.

[11] UTF-8 and UTF-16 Strings. http://developer.android.com/training/articles/perf-jni.html#UTF_8_and_UTF_16_strings.

[12] Android VpnService. http://goo.gl/kV7ZZL, 2014.

[13] strongSwan VPN Client. https://play.google.com/store/apps/details?id=org.strongswan.android, 2014.

[14] FRep - Finger Replayer. https://play.google.com/store/apps/details?id=com.x0.strai.frep, 2015.

[15] Alcatel-Lucent. *Motive Wireless Network Guardian*, 2013. https://www.alcatel-lucent.com/products/wireless-network-guardian.

[16] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proc. of HotMobile*, 2011.

[17] Carrier IQ Inc. Mobiles' Diagnostic Analysis. http://www.carrieriq.com.

[18] M. Charts. In the US, Time Spent With Mobile Apps Now Exceeds Desktop Web Access. http://www.marketingcharts.com/online/in-the-us-time-spent-with-mobile-apps-now-exceeds-the-desktop-web-41153/, Mar. 2014.

[19] X. Chen, R. Jin, K. Suh, B. Wang, and W. Wei. Network Performance of Smart Mobile Handhelds in a University Campus WiFi Network. In *Proc. of ACM IMC*, San Diego, California, USA, Nov. 2012.

[20] Cisco. The Zettabyte Era—Trends and Analysis. www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html, May 2015.

[21] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 647–651, San Diego, California, USA, June 2003.

[22] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM, 2013 Proceedings IEEE*, pages 809–817. IEEE, 2013.

[23] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or Both?: measuring multi-homed wireless internet performance. In *Proc. of the ACM Conf. on Internet Measurement Conf. (IMC)*, pages 181–194. 2014.

[24] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[25] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS*, 2014.

[26] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A First Look at Traffic on Smartphones. In *Proc. of the 10th ACM SIGCOMM Conf. on Internet Measurement*, pages 281–287, Melbourne, Australia, Nov. 2010.

[27] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proc. of USENIX Security*, 2011.

[28] A. Gerber, J. Pang, O. Spatscheck, and S. Venkataraman. Speed Testing without Speed Tests: Estimating Achievable Download Speed from Passive Measurements. In *Proc. of the 10th ACM SIGCOMM Internet Measurement Conf. (IMC)*, pages 424–430. Nov. 2010.

[29] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the Intl. Conf. on Trust and Trustworthy Computing*, 2012.

[30] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proc. of CCS*, 2011.

[31] ICSI. Netalyzr for Android. https://play.google.com/store/apps/details?id=edu.berkeley.icsi.netalyzr.android.

[32] ICSI. Haystack: Understanding the Fate of your Private Data, Project Website and Beta Release. haystack.mobi, Oct. 2015.

[33] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 315–320. ACM, 2007.

[34] A. M. Kakhki, A. Razaghpanah, A. Li, H. Koo, R. Golani, D. Choffnes, P. Gill, and A. Mislove. Identifying traffic differentiation in mobile networks. In *IMC 2015*.

[35] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: multilevel traffic classification in the dark. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 229–240. ACM, 2005.

[36] T. Karagiannis, K. Papagiannaki, N. Taft, and M. Faloutsos. Profiling the end host. In *Passive and Active Network Measurement*, pages 186–196. Springer, 2007.

[37] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *Proc. of the 10th ACM SIGCOMM Internet Measurement Conf. (IMC)*, pages 246–259. Nov. 2010.

[38] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou. [removed] a system for monitoring from mobile devices. In *SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big Internet Data (C2BID)*, London, UK, Aug. 2015.

[39] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proc. of UbiComp*, 2012.

[40] M. LLC. Smartphones: So Many Apps, So Much Time, 2014.

[41] Lutz, Z. Carrier IQ: What it is, what it isn't, and what you need to know. http://www.engadget.com/2011/12/01/carrier-iq-what-it-is-what-it-isnt-and-what-you-need-to/.

[42] S. Miskovic, G. M. Lee, Y. Liao, and M. Baldi. Appprint: Automatic fingerprinting of mobile applications in network traffic. In *Passive and Active Measurement*, pages 57–69. Springer, 2015.

[43] I. News. Mobile subscriptions near the 7 billion mark. Does almost everyone have a phone? https://itunews.itu.int/en/3741-Mobile-subscriptions-near-the-78209billion-markbrDoes-almost-everyone-have-a-phone.note.aspx.

[44] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. Mobilyzer: An Open Platform for Controllable Mobile Network Measurements. In *Proc. of the 13th Annual Int. Conf. on Mobile Systems, Applications, and Services (MobiSys)*, pages 389–404, Florence, Italy, May 2015.

[45] Open Signal Inc. 3G and 4G LTE Cell Coverage Map. `http://www.opensignal.com`.

[46] Open Signal Inc. US Wi-Fi Report. `https://opensignal.com/reports/2014/us-wifi/`.

[47] A. Rao, A. M. Kakhki, A. Razaghpanah, A. Tang, S. Wang, J. Sherry, P. Gill, A. Krishnamurthy, A. Legout, A. Mislove, and D. Choffnes. Using the Middle to Meddle with Mobile. Technical report, Northeastern University, Dec. 2013.

[48] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson. Haystack: In Situ Mobile Traffic Analysis in User Space. *arXiv preprint arXiv:1510.01419*, Oct. 2015.

[49] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic. Technical report, Northeastern University, September 2015.

[50] Root Metrics Inc. Metro RootScore Reports. `http://www.rootmetrics.com/us`.

[51] Root Metrics Inc. Rootscore Awards and Reports. `http://www.rootmetrics.com/us/rsr/map/2015-1H`.

[52] Sensorly Inc. Unbiased Wireless Network Information. `http://www.sensorly.com`.

[53] A. Shuba, A. Le, M. Gjoka, J. Varmarken, S. Langhoff, and A. Markopoulou. Antmonitor: Network traffic monitoring and real-time prevention of privacy leaks in mobile devices. In *ACM Mobicom Demo and Short Paper (and best demo in S3)*, September 2015.

[54] W. A. Social. Digital, Social and Mobile Worldwide in 2015. `http://wearesocial.net/blog/2015/01/digital-social-mobile-worldwide-2015/`, Jan. 2015.

[55] J. Sommers and P. Barford. Cell vs. WiFi: On The Performance of Metro Area Mobile Connections. In *Proc. of ACM IMC*, pages 301–314, Boston, USA, Nov. 2012.

[56] Y. Song and U. Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proc. of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 15–26, Oct. 2015.

[57] N. Vallina-Rodriguez. Does your operator leak your private data? HTTP Header Enrichment in Mobile Networks. `http://netalyzr.icsi.berkeley.edu/blog/`.

[58] N. Vallina-Rodriguez, A. Auçinas, M. Almeida, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft. RILAnalyzer: A Comprehensive 3G Monitor on Your Phone. In *Proc. of IMC*, Barcelona, Spain, Oct. 2013.

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

[60] N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, and V. Paxson. Header Enrichment or ISP Enhancement? Emerging Privacy Threats in Mobile Networks s. In *SIGCOMM HotMiddleBox Workshop*, London, UK, Aug. 2015.

[61] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-Layer Profiling of Android Applications. In *Proc. of the 18th ACM annual int. conf. on Mobile computing and networking*, pages 137–148, Istanbul, Turkey, Aug. 2012.

[62] G. Xie, M. Iliofotou, R. Keralapura, M. Faloutsos, and A. Nucci. Subflow: towards practical flow-level traffic classification. In *INFOCOM, 2012 Proceedings IEEE*, pages 2541–2545. IEEE, 2012.

[63] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman. Identifying Diverse Usage Behaviors of Smartphone Apps. In *Proc. of ACM IMC*, Berlin, Germany, Nov. 2011.