Haystack: A Multi-Purpose Mobile Vantage Point in User Space

Abbas Razaghpanah
Stony Brook University

Narseo Vallina-Rodriguez
ICSI

Srikanth Sundaresan
ICSI

Christian Kreibich
ICSI / Lastline

Phillipa Gill
Stony Brook University

Mark Allman
ICSI

Vern Paxson
ICSI / UC Berkeley

Abstract Despite our growing reliance on mobile phones for a wide range of daily tasks, their operation remains largely opaque. A number of previous studies have addressed elements of this problem in a partial fashion, trading off analytic comprehensiveness and deployment scale. We overcome the barriers to large-scale deployment (*e.g.,* requiring rooted devices) and comprehensiveness of previous efforts by taking a novel approach that leverages the VPN API on mobile devices to design Haystack, an in-situ mobile measurement platform that operates exclusively on the device, providing full access to the device's network traffic and local context without requiring root access. We present the design of Haystack and its implementation in an Android app that we deploy via standard distribution channels. Using data collected from 450 users of the app, we exemplify the advantages of Haystack over the state of the art and demonstrate its seamless experience even under demanding conditions. We also demonstrate its utility to users and researchers in characterizing mobile traffic and privacy risks.

1. INTRODUCTION

Mobile phones have become indispensable aids to everyday life by offering users capabilities that rival those of general purpose computers. However, these systems remain notoriously opaque, as mobile operating systems tightly control access to system resources. While this tight control is useful in preventing unwanted application activity, it also imposes hurdles for understanding the behavior of mobile devices, especially their network activity and performance.

Despite these challenges, the research community has made steady progress in understanding mobile apps and mobile traffic over the past few years, by using two broad classes of techniques. One class is lab-oriented and uses static and dynamic analysis of app source code [22, 56], controlled execution of apps [24, 38] and dynamic analysis [68], even modifying the OS kernel to track app behavior [23]. A contrasting approach leverages network traces obtained from ISPs [27,62] or VPN tunnels that forward user traffic [52] to servers in the cloud for observation. However, each of these previous approaches faces a trade-off:

- Approaches based on static and dynamic analysis do not offer access to real-world data. Thus far, studies that have used these approaches have been constrained to analysis of source code, which is not always available, or artificial/controlled user inputs which require significant human effort to train the techniques, contextualize the results, and minimize false positives. One exception is Taintdroid [23], a modified Android version that can analyze app behavior in real-world settings. However, this technique relies on operating system modifications, which incur a significant engineering effort to catch up with new OS releases, and forces participating users to install a new firmware on their devices [65]. Consequently, the scale of analysis and app coverage they can achieve in the wild remains limited to tens of users.
- Approaches that leverage network traffic obtain visibility into real user behavior, at the cost of the richness of context that device-centric approaches can obtain. For example, while this approach can capture and analyze mobile network data, heuristics must infer which applications generated individual flows, and detouring traffic through third-party middleboxes complicates high-fidelity performance measurements due to the necessarily skewed vantage point.

In this study, we present Haystack, the first on-device mobile measurements platform that is able to passively monitor app behavior and network traffic under regular usage and network conditions, *without requiring users to root the phone*. The latter gives Haystack the potential for better scalability in deployment; users can simply install the app from Google's Play Store or similar markets. This provides us the opportunity to monitor organic mobile network activity as generated by real users in real networks using real mobile apps—all from the vantage point of the device. This combination of ease of deployability and the high-fidelity vantage point allows Haystack to hit a sweet spot in the trade-off between scalability and richness of data.

Similar to previous approaches [52], Haystack leverages Android's standard VPN interface to capture outbound packets from applications. However, rather than tunneling the packets to a remote VPN server for inspection, Haystack intercepts, inspects, and forwards the user's traffic to its intended destination. This approach gives us raw packet-level access to outbound packets as well as flow-level access to incoming traffic without modifying the network path, and without requiring permissions beyond those needed by the VPN interface. Haystack therefore has the ability to monitor network activity in the proper context by operating locally on the device. For example, a TCP connection can be associated with a specific DNS lookup and both can be coupled with the originating application. Further, we design Haystack to be extensible with new analyses and measurements added over time (*e.g.,* by adding new protocol parsers and by supporting advanced measurement methods such as reactive measurements [10]), and new features to attract and educate users (*e.g.,* ad block, malware detection, privacy leak prevention and network troubleshooting).

Haystack is publicly available for anyone to install on Google Play and has been installed by 450 users to date [40]. We discuss the design and implementation of Haystack in §3 and §4, and evaluate its performance and resource use in §5. Our tests show that Haystack delivers sufficient throughput (26–55 Mbps) at low latency overhead (2–3 ms) to drive high-performance and delay-sensitive applications such as HD video streaming and VoIP without noticeable performance degradation for the user.

While we consider our Haystack implementation prototypical in some respects (such as UI usability for non-technical users), it has already provided interesting insights into app usage in the wild: in §6 we present preliminary findings about the adoption of encryption techniques, report on local-network traffic interacting with IoT devices, study app provenance and the use of third-party tracker services, and give an outlook on potential future applications.

2. RELATED WORK

Previous studies have leveraged a variety of techniques for understanding privacy risks of mobile apps and their behavior in the network. As noted earlier, each approach made trade-offs between having access to real user behavior and device context. We classify the prior work into the following four categories.

Dynamic app analysis: This approach calls for running an app in a controlled environment such as a virtual machine [68] or an instrumented OS [23, 38]. The app is then monitored as it conducts its pre-defined set of tasks, with the results indicating precisely how the app and system behave during the test (*e.g.,* whether the app exfiltrated data). While this approach provides useful insights, the workload (which does not represent real-world operation) and difficulty of deploying custom firmware on users' phones (sacrificing scale) means that the results do not directly speak to normal users' activity. To overcome the lack of user input, studies that rely on dynamic analysis require "UI monkeys" [11, 66] to generate synthetic user-actions.

Static app analysis: This technique involves analysis of the app code, obtained by decompiling app binaries, via symbolic execution [67], analysis of control flow graphs [16, 22], by auditing third-party library use [21, 55], through inspection of the Android permissions and their associated system calls [16, 45], and analysis of app properties (*e.g.,* whether apps employ secure communications) [24, 26]. While static analysis typically provides good scale with analysis of over 10K apps in several studies [24], (modulo computational resources) this strategy does not reflect the behavior of apps in the wild, and typically requires a good amount of manual inspection. Furthermore, the analysis may under- or over-state the importance of certain code paths since it lacks a notion of how users interact with the apps in practice.

Passive traffic analysis: A number of studies rely on volunteers with rooted phones that allow their traffic to get recorded by `tcpdump` [25,39,51] or `iptables` [17,60]. These methods are challenging to deploy at scale. To obtain larger-scale data, other projects study the behavior of mobile devices by observing their network traffic either at a large ISP with millions of users [27,57,62] or by forwarding traffic through a remote VPN proxy that also modifies the network path [44, 52, 64]. As a result, these studies contain a large variety of apps and mobile platforms but they lack device context for accountability and accuracy (*e.g.,* mapping flows to originating apps). While this can be alleviated by pairing a remote VPN proxy with client-side software to provide context to the remote VPN server [44], the solution still alters the network path by rerouting traffic to the VPN server, hence providing an unrealistic view of the performance aspects of real mobile traffic. PrivacyGuard [59] uses a technique similar to the one used by Haystack to intercept user traffic to detect simple instances of private information leaks (*e.g.,* device ID and location), but it does not aim to offer the depth and versatility offered by Haystack as a measurement platform.

Active mobile network measurements: Google Play (and, on a smaller scale, the Apple Store) contains a number of tools for active mobile network measurements. Examples include Ookla's SpeedTest [34], the FCC Speed Test [28], network scanners to build network coverage maps [35], and comprehensive mea-

Approach	Scale	Real-world operation	Comprehensiveness	Local Operation	App coverage	OS compatibility
ISP traces	Large-scale	✓			All apps	All versions/platforms
Remote VPN	Crowdsourcing	✓			Crowdsourcing	All versions/platforms
Static analysis	Resource-bound		✓	✓	$\sim$ 1000 apps	Limited
Dynamic analysis	Resource-bound		✓	✓	$\sim$ 100 apps	Limited

Table 1: Comparison between different measurement approaches in the mobile environments. It should be noted that these aspects/features are not easily comparable in a binary manner and the comparison provided here is merely qualitative.

surement tools such as My Speed Test [30], Netalyzr for Android [32], NameHelp [31], and MobiPerf [29]. Such tools provide valuable insight into network performance [41, 47] and operational aspects of ISPs such as middlebox deployment [63] and traffic discrimination [42]. However, despite the fact that active measurement techniques typically provide an accurate snapshot of actual network conditions, they do not study network performance of installed apps in real-world situations.

Table 1 provides a high-level comparison of each of the measurement approaches. As we can see, none can simultaneously observe real-world operation while providing comprehensive data at scale. This trade-off has prevented the research community from exploring in detail many aspects of the mobile ecosystem. We will revisit the comparison between Haystack and state of the art techniques in §6, after we present its design, implementation, and evaluation.

3. HAYSTACK OVERVIEW

Our goal with Haystack is to help researchers avoid the trade off between accessing device context and the ability to measure real-world phone usage at scale. The crux of Haystack is its ability to observe network communication *on the mobile device itself*. Since 2011 (version 4.0), Android has provided a VPN API that enables developers to create a virtual `tun` interface and direct *all* network traffic on the phone to the interface's user-space process. To enable this functionality, the client app requests the `BIND_VPN_SERVICE` permission from the user, which, crucially, does not require a rooted device. The API typically drives VPN client applications that forward traffic to a remote VPN server [14]. Instead of relaying packets to a remote VPN server, Haystack performs two high-level operations in parallel. First, it sends a copy of the bidirectional packet stream to a background process that analyzes the traffic off-path. Second, it uses the packet headers to maintain user-space network sockets to remote hosts and relays data via these sockets.

Haystack is available in the Google Play Store [40] and has been downloaded a total of 450 times. A number of apps in Google Play leverage the `BIND_VPN_SERVICE` permission for non-VPN tasks. While we are not aware of any apps taking traffic processing to the level we realize in Haystack, tPacketCapture [37] and SSL Packet Capture [36] take advantage of the VPN API to record approximate packet traces via a user-space application. As with Haystack, these traces are approximate since the app does not have access to raw packets via Java's socket interface. A related app, NoRoot Firewall [33] allows mobile users to block traffic generated by specific apps and generate connection logs.

3.1 Ethical Considerations

Haystack's ability to observe real-world user data raises many ethical considerations [7]. We leverage the fact that Haystack runs on the user's device to do the bulk of processing on the device and only send back summary statistics (*e.g.,* domains contacted and protocols used) and by under no-circumstances user's raw traffic. We aim to minimize the amount of data sent back while maximizing it's utility. In consultation with the IRB at UC Berkeley, we developed a protocol that strikes a balance and only collects data needed for the studies at hand without uploading any personal information. This precludes certain types of detailed or longitudinal studies, which may be possible with future coordination with the IRB.

Additionally, we implement informed consent and opt-in in Haystack. First, Haystack must be explicitly installed by the user and granted permission to observe traffic. Second, we require users to opt-in a second time before we analyze encrypted traffic as described in §4.1.2. TLS interception is explained to the user in detail before they are given the option to install the CA certificate needed to intercept encrypted traffic. If the user chooses not to install the CA certificate, the TLS interception module is disabled until they explicitly choose to install the CA certificate and manually enable TLS interception in the settings. Our opt-in process aims to make the data collection as transparent as possible and provide users control over the process. While our IRB has reviewed our current approach and has deemed our work as not involving human subjects research, we maintain an active dialogue and we will seek their feedback before collecting any additional piece of information.

4. SYSTEM DESIGN

To intercept and analyze traffic on resource-constrained devices in user space, we must address several design challenges. A key issue is that the `tun` interface exposes raw IP packets to Haystack. A natural way to deal with these would be to shuttle a copy to our

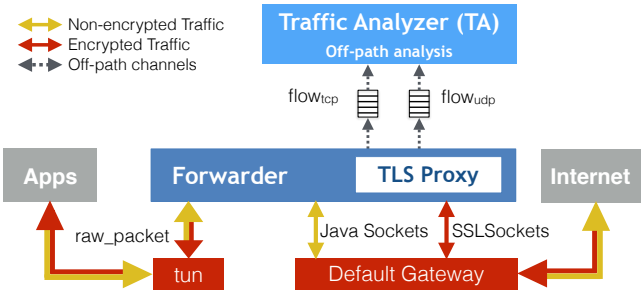


Figure 1: The Haystack architecture, highlighting system components and data forwarding channels. Solid lines represent the actual forwarding path for traffic generated by mobile apps even if encrypted (which is handled by our optional TLS proxy), while dashed lines represent the off-line path used for privacy and performance analysis.

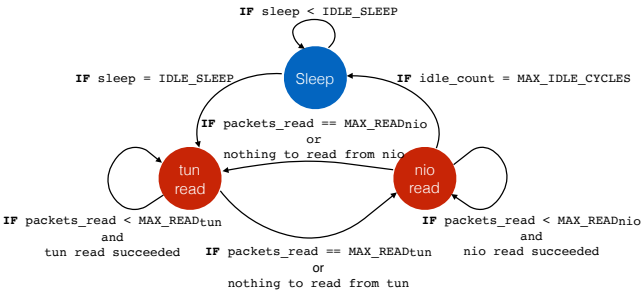


Figure 2: Haystack's Forwarder state machine. It controls read/write operations and transitions between `tun` interface, Java NIO socket, and sleep states. The idle count variable increments when both `tun` and NIO do not succeed, *i.e.,* there is nothing to read. Each read operation from the `tun` interface potentially becomes a write operation for a NIO socket and vice versa.

analysis engine and then drop the packet on the network via a raw socket. However, non-privileged apps do not have access to raw sockets and therefore we must rely on regular Java sockets to communicate with remote entities. This means that, as opposed to transparent L3 and L4 proxies that operate at a single layer of the protocol stack on both sides (with root privileges), Haystack has to bridge packet-level communication on the host (`tun`) side and flow-level interaction with the network side. Operating in mobile phones in user space requires careful design considerations to minimize Haystack's impact on device resources, battery life, app performance and user experience. Figure 1 illustrates the Haystack architecture, which includes two major components, the *Forwarder* and the *Traffic Analyzer* (TA).

4.1 The Forwarder

The Forwarder performs two key functions: (*i*) it performs transparent bridging between packets on the `tun` interface and payload data on the regular socket interface and (*ii*) it forwards traffic to the TA for analysis.

4.1.1 Flow reassembly

The Forwarder receives raw IP packets from `tun`. The Forwarder therefore acts like a layer 3/layer 4 network stack: it extracts the payload from the raw packet and sends it to its intended destination through a regular Java socket (implemented using non-blocking NIO sockets [48]). To accomplish this, the Forwarder extracts flow state from the packet headers (IP, as well as UDP or TCP) for packets arriving on the `tun` interface and maps it to a given Java socket (it creates new sockets for new flows arriving on the `tun` interface). It also maintains this state so that it can marshal data arriving from remote hosts on the sockets back into packets for transmission to the app via the `tun` interface. Haystack has dual-stack support and its routing tables correctly forward DNS and IPv6 traffic through the `tun` interface to prevent traffic leak [49].

Handling UDP and TCP: The Forwarder needs to maintain state for UDP and TCP flows. A simple flow-to-socket mapping suffices for connectionless UDP, since header reconstruction remains straightforward. Since TCP provides connection-oriented and reliable transport, we need to track the TCP state machine and maintain sequence and acknowledgment numbers for each TCP flow. We segment the data stream received from the socket and synthesize TCP headers to be able to forward the resulting packets to the `tun` interface for delivery to the app. When we read a SYN packet from the `tun` interface, we create a new socket, connect to the target and instantiate state in Haystack. After the OS establishes the socket we return a SYN/ACK via the `tun` interface. We similarly relay connection termination. We discuss Haystack's lack of support for non-TCP/UDP traffic in §4.3.

Efficient packet forwarding: The Forwarder must balance application and traffic performance with power and CPU usage on the device. This task is challenging because the `tun` interface does not expose an event-based API. We therefore implement a polling scheme that periodically checks both the `tun` interface and Java sockets for arriving data.

Figure 2 shows the state machine of the Forwarder. It reads up to max_read_{tun} packets from the `tun` interface or up to max_read_{nio} packets from the socket (NIO)[1] interface before switching to the other interface, hence preventing either operation from starving. The Forwarder immediately transitions to the other read state if it cannot read data in the current state. Each read from the `tun` interface potentially becomes a write operation for a socket and vice versa, the exception being pure TCP ACKs from the `tun` interface. We discard these, as their effect gets abstracted by the socket in-

[1]Despite the inability to count packets from socket read/write operations, we count the number of packets generated and sent back through the `tun` interface.

terface and therefore they do not require forwarding. Writes to the `tun` interface complete quickly and socket writes do not block, so we perform writes as soon as we have data to send. If it cannot read data from either interface for *max_idle_cycles* consecutive iterations, the Forwarder will sleep for *idle_sleep* ms. While this strategy reduces power consumption during idle periods, it also imposes higher latency on packets that arrive during these idle periods when polling happens at coarse intervals. We consider the tradeoff between resource conservation and performance in depth in §5.2.

4.1.2 TLS Interception

Many mobile applications have adopted TLS as the default cryptographic protocol for data communications. This is a double-edged sword, as it helps protect the integrity and privacy of users' transactions but also allows apps to conceal their network activity. With the user's consent, Haystack employs a transparent man-in-the-middle (MITM) proxy for TLS traffic [20]. At install time Haystack requests the user allow the installation of a self-signed Haystack CA certificate in the user CA certificate store. We customize the message shown to users at this time to explain why Haystack intercepts encrypted traffic.

Once equipped with a certificate, the Forwarder monitors TCP streams beginning with a TLS "Client Hello" message and forwards these flows—along with flow-level meta-information the proxy requires in order to connect to the server (*e.g.,* IP address, port, SNI)—to the TLS proxy. The proxy uses this information to connect to the remote host and reports back to the Forwarder whether the connection was successful. After successfully establishing a connection to the remote host, the proxy decrypts traffic arriving on one interface (`tun` or socket) and re-encrypts it for relay to the other while providing a clear-text version to the TA for analysis.

Dealing with failed TLS connections: As in any commercial TLS proxy, Haystack will be unable to proxy flows when the client application (*i*) uses TLS extensions not supported by Haystack [19],[2] (*ii*) bundles its own trust store, or (*iii*) implements certificate pinning. Likewise, failure occurs when the server expects to see certain TLS extensions not supported by Haystack in the "Client Hello" message or performs certificate-based client authentication. We add connections with failed TLS handshakes to a whitelist that bypasses the TLS proxy for a period of five minutes. Experience with our initial set of users indicates that apps recover gracefully from TLS failures. After five minutes we remove the app from the whitelist to account for transient failures in the handshake process. While we cannot decrypt such flows, we can still record which apps take these security measures and potentially communicate more securely for further analysis.

Security considerations: Android provides support for third-party root certificate installation. This is a feature required by enterprise networks to perform legitimate TLS interception. For increased security, Haystack generates a unique certificate and key-pair for each new installation of the app. Additionally, Haystack saves the private key to its private storage to prevent other applications from accessing it. While these precautions still permit malicious applications with root access to retrieve the key, such apps can already tap into the user's encrypted traffic without using Haystack's CA certificate (*e.g.,* by surreptitiously injecting their own CA certificate into the system's trust store).

4.2 Traffic Analyzer

The Traffic Analyzer (TA) processes flow data captured by the Forwarder. The TA operates in near real-time but off-path, *i.e.,* outside the forwarding path of network traffic. The TA augments flows with contextual information gathered from the OS for further analysis. The analyses are protocol-agnostic, and TA supports protocol parsers to parse flow contents before they are analyzed. We currently support TLS, HTTP, and DNS protocol parsers to analyze the traffic and extract relevant information, decompressing and decoding compressed and encoded data before they are searched by the DPI module for private information leakages. New protocol parsers can be added to TA in case we see new protocols getting widely adopted.

Why off-path analysis? The TA could potentially negatively affect the user experience if done in the forwarding path. Analysis of network traffic can range from simple (*e.g.,* tracking packet statistics) to quite complex (*e.g.,* parsing protocol content) and therefore can consume valuable CPU cycles and if conducted as part of traffic forwarding could increase latency. However, as we will discuss in §6, certain aspects of mobile apps and networks must be measured in-path as in the case of traffic performance analysis.

Secure and efficient IPC in Android: Unfortunately, low-latency communication between Android services can prove tricky to realize, especially in multi-threaded systems. In our implementation we use Java's thread-safe queues for communication between the Forwarder and TA modules. This allows the modules to communicate without exposing their data to other (malicious) apps as would be the case if the file system or localhost sockets were used [13]. In §5.4 we evaluate the overhead of using thread-safe queues to enable communication between the Forwarder and TA.

Application and entity mapping: One of the ba-

[2]Currently, Haystack only supports the SNI extension.

sic functions the TA provides is to map TCP and UDP flows to the corresponding apps. We do this via a two-step process: (*i*) extract the PID of the process that generated the flow from the system's `proc` directory, (*ii*) map the PID to an app name using Android's Package Manager API. Compared to network-based studies which rely on inferences—*e.g.,* using the HTTP User-Agent or destination IP address—to couple apps and flows [52, 53, 62], our approach allows highly accurate flow-to-app mappings. Since reading the PID and mapping applications requires file-system access, we cache recently read results to minimize overhead.

The TA also provides the ability to analyze protocols in depth. For example, the TA tracks DNS transactions to extract names associated with IP addresses, allowing us to map flows to target domains rather than just IP addresses. This is especially important for non-HTTP flows (*e.g.,* QUIC, HTTPS) where the hostname may not be readily available in application layer headers. Mapping IPs to their hostnames gives us the opportunity to distinguish apps sending data to their own backend as opposed to third-party ad/analytics services or CDNs, even if both reside in the same cloud service provider [18]. Further, the TA can perform traffic characterization based on domain, without analyzing application-layer headers (*e.g.,* HTTP `Host` header). We demonstrate how these capabilities in the TA can enable studies like per-app protocol usage and user tracking detection in §6.

4.3 Limitations and other considerations

Protocol support: Android limits us to only TCP and UDP sockets via Java's APIs, thus excluding protocols such as ICMP. As of today, this limitation only seems to affect a small number of network troubleshooting tools. The Forwarder provides IPv6 support, except for extended headers. We have not noticed any issues for IPv6 flows due to this limitation.

Recovery from loss of connectivity: The VPN service (and therefore the Forwarder) gets disrupted when users roam between different networks such as 3G and WiFi or different WiFi networks, or when a network disconnection occurs. Haystack identifies such events and attempts to reconnect seamlessly. Similarly, phone calls disable all data network interfaces, thus stopping the VPN service. While currently this disables Haystack, we are working on using Android APIs to identify when the calls complete to transparently restart the VPN.

Vendor-custom firmware: Many device vendors block and interfere with standard Android APIs. One case is Samsung's KNOX SDK—only available for Samsung licensed partners—which prevents third-party VPN applications from creating virtual interfaces [54]. Likewise, some vendor-locked firmwares also prevent Haystack from intercepting TLS traffic by blocking CA certificate installation. We have thus far primarily encountered this issue on Samsung phones.

DPI and arms race: Malicious agents will always have an incentive to not being identified. Against our best efforts to parse and extract information from popular protocols, inflate compressed streams, and intercept conventional TLS-encrypted flows; as well as Haystack's ability to support newer protocol (*e.g.,* QUIC and new TLS extensions) as mobile apps and the mobile ecosystem as a whole evolve, some apps will still be able to exfiltrate private information through obfuscation and encryption schemes that are not supported by Haystack. Since Haystack would fall short of studying these instances, we acknowledge that there is a possibility of an arms race between privacy-invasive and malicious apps and approaches like Haystack.

5. PERFORMANCE EVALUATION

We have implemented Haystack as a user-level Android app per the design given above. Our implementation leverages a number of external libraries for tasks such as efficient packet parsing [2], IP geo-location [5], data presentation [6], and TLS interception [20]. The Haystack codebase—excluding the external libraries and XML GUI layouts—spans 15,000 lines of code. In this section we evaluate to what extent we achieve our goal of real-time monitoring without burdening the device's resources in practice and under stress conditions.

5.1 Testbed and Measurement Apparatus

To evaluate Haystack performance in a controlled setting, we set up a testbed with a Nexus 5 phone connected to a dedicated wireless access point over a 5 GHz 802.11n link. We also connected a small server to the access point via a gigabit Ethernet link. We minimize background traffic on the phone by only including the minimal set of pre-installed apps and not signing into Google Services. We measure the latency of Haystack using simple UDP and TCP echo packets. For non-TLS throughput tests, we use a custom-built speed-test that opens three parallel TCP connections to the server for 15 seconds in order to saturate the link. We test uplink and downlink separately. For profiling TLS establishment latency and downlink speed-test, we cannot use our speed-test, as it does not employ a TLS session. Instead, we download 1 B and 20 MB objects over HTTPS from an Apache v2 web server with a self-signed x.509 certificate. We repeat each test 25 times.

While our testbed allows us to explore many parameters within the design space, Android's VPN security model precludes full automation of our experiments as it requires user interaction to enable/disable the `tun` interface. We focus on the impact of *max_idle_cycles* and *idle_sleep* and fix max_read_{tun} and max_read_{nio} to

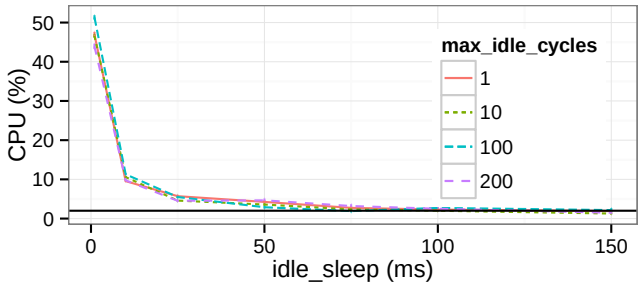


Figure 3: Haystack's CPU overhead for different *max_idle_cycles* and *idle_sleep* configurations. The horizontal line indicates the aggregated average CPU load of all apps running on the background for reference.

100 packets which favors downlink traffic.

5.2 CPU and Power Overhead

CPU usage impacts interactivity of foreground apps and as a result, the user experience. We therefore investigate the impact of how idle a device must be before starting periodic polling (for a maximum of *max_idle_cycles* cycles) and how often we poll for new traffic after a device is deemed idle (*idle_sleep* ms) on CPU load and battery life.

CPU load: Mobile phones remain idle most of the time [17, 61]. As a result, optimizing Haystack's performance in this scenario is essential to minimize its impact on limited system resources, in particular on battery life. The base CPU usage of the Nexus 5 is 2% in the absence of Haystack, when the system is idle with its screen off and normal background activity from installed apps. Figure 3 shows the impact of *max_idle_cycles* and *idle_sleep* on CPU usage when enabling Haystack. We find that *idle_sleep* has the most significant effect on CPU load, which is unsurprising as this parameter dictates how long the app sleeps and therefore does not consume CPU. With *idle_sleep* set to 1 ms, the CPU load varies between 45% and 55% for different values of *max_idle_cycles* with the Forwarder polling the interfaces at a high frequency. CPU usage drops sharply as we increase *idle_sleep*, to 10.5% and 4.6% with *idle_sleep* at 10 ms and 25 ms, respectively. In contrast to *idle_sleep*, *max_idle_cycles* shows little influence on CPU overhead, particularly at *idle_sleep* values greater than 10 ms. This is because we measure *max_idle_cycles* in loop cycles (cf. Figure 2) which take a small fraction of 1 ms each. For *idle_sleep* of 100 ms and *max_idle_cycles* of 10 or 100 cycles the overhead of Haystack is negligible, with the CPU usage close to the base CPU usage (horizontal line in Figure 3). We consider an *idle_sleep* value of 100 ms ideal for operating during idle periods (delay-tolerant) and an *idle_sleep* value of 10 ms during interactive periods. In the following subsections, we will evaluate the impact of *idle_sleep* in traffic performance.

Test Case	Power(mW) Mean/SD	Increase
Idle	1,089.6 / 125.9	+3.1%
Idle (Haystack)	1,123.8 / 150.4	
YouTube	1,755.3 / 35.5	+9.1%
YouTube (Haystack)	1,914.4 / 16.1	

Table 2: Power consumption of Haystack when *max_idle_cycles* is 100 cycles and *idle_sleep* is 1 ms in different scenarios. The percentage indicates the increase when running Haystack.

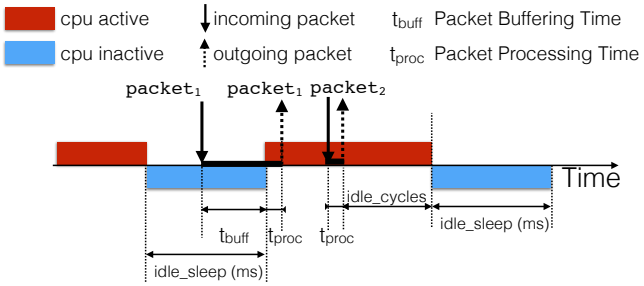


Figure 4: Latency added by *idle_sleep* and *max_idle_cycles* on packets arriving during periods of activity, or inactivity.

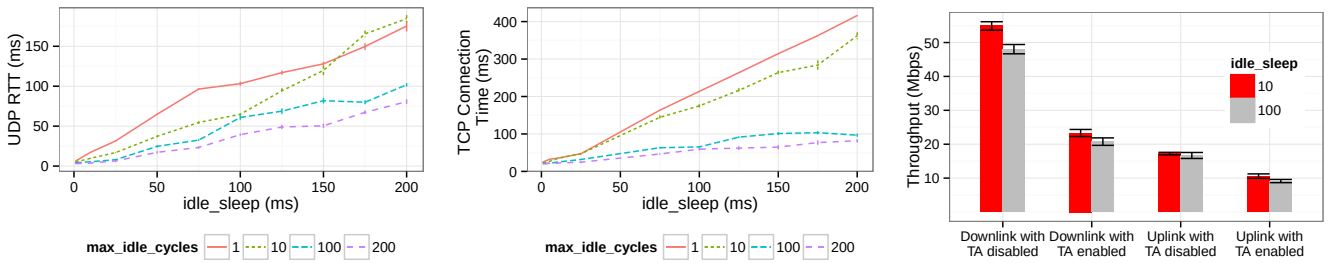
User experience during interactive periods: We next profile Haystack's overhead under heavy load. To do so we run Haystack and stream a 1080p YouTube video. This stresses packet forwarding, CPU usage, and the TLS Proxy, since YouTube delivers the video over TLS. Crucially, we do not observe delay, rebuffering events, or noticeable change in resolution during the video replay, suggesting that Haystack's performance can keep up with demanding applications.

Power consumption: We use the Monsoon Power Monitor [46] to directly measure the power consumed by Haystack on a BLU Studio X Plus phone running Android 5.0.2. [3] We removed the battery and replaced it with the power meter set to emulate the phone's standard 3.8V battery. We then record the power consumed during various situations. Table 2 summarizes the results for each scenario across 10 trials with *max_idle_cycles* set to 100 cycles and *idle_sleep* set to 1 ms. This configuration represents the worst-case (cf. Figure 3) as Haystack sleeps for only 1 ms before polling the interfaces again. Unfortunately, due to hardware limitations, we could not measure Haystack's power consumption with the screen off but, for that scenario, we can use Haystack's CPU overhead as a proxy [61]. During idle periods with the screen active, Haystack increases power consumption by 3% (similar to the CPU increase). The overhead of Haystack increases to 9% while streaming a YouTube video.

5.3 Latency Overhead

Haystack suspends polling during periods of inactivity to conserve battery. However, suspending polling

[3] We faced several instrumentation challenges that impeded measuring Haystack's power consumption on a Nexus 5.



(a) UDP latency. (b) TCP connection time. (c) TCP Throughput.

Figure 5: Haystack performance (UDP latency, TCP connection time, and TCP throughput) for different *idle_sleep* and *max_idle_cycles* configurations. For the throughput evaluation, we fix *max_idle_cycles* to 100 cycles, also showing the impact of enabling the TA. The maximum TCP throughput for this link is 73 and 83 Mbps uplink and downlink, respectively.

also increases latency for packets that arrive during loop suspension, as illustrated in Figure 4. In the figure, the packet that arrives during the first idle sleep period endures the remainder of the idle period (t_{buff}), in addition to the forwarding time (t_{proc}), which includes looking up relevant header state and translating between the layer 3 `tun` interface and layer 4 NIO sockets. However, the packet that arrives when polling is active does not experience the idle period overhead.

We now analyze the latency incurred by packets when running Haystack. Specifically, we focus on the impact of *max_idle_cycles* and *idle_sleep* and the trade-off between latency and CPU overhead. Figure 5(a) shows the results of our experiments for UDP. When *max_idle_cycles* =1 cycle the latency closely follows *idle_sleep* because Haystack's aggressive sleeping renders it more likely for packets to arrive when the system is idle, therefore delaying them for up to *idle_sleep* ms before being processed. With *max_idle_cycles* =100 cycles and *idle_sleep* =100 ms we find about 60 ms of extra delay. Reducing *idle_sleep* to 10 ms while keeping *max_idle_cycles* at 100 cycles reduces latency to as low as 3.4 ms. We find similar patterns for TCP connections. In Figure 5(b) we plot connection establishment times for the TCP echo client and server. As expected, high values of *max_idle_cycles* and coupled with low *idle_sleep* settings results in quicker connection establishment. In fact, when the RTT of the link drops below the time it takes to reach *max_idle_cycles* cycles, Haystack processes all packets in the TCP handshake without the Forwarder going into idle state.

Finally, we consider the latency incurred by a packet during processing and forwarding (t_{proc}). To get a sense of how the latter affects performance, we evaluate t_{proc} while running our speed-test app. Table 3 shows the results of our speed-test for TCP and UDP connections. Processing times for established flows are 141 μs for TCP and 76 μ s for UDP, indicating that the packet forwarding is not a bottleneck for Haystack's performance. The processing times for new connections prove larger, especially for TCP, because of the overhead of initiating state for the connection.

	Downlink	Uplink	New Flow
TCP t_{proc} (μ s)	141.6±0.5	275.6±3.5	5,647.8±998.5
UDP t_{proc} (μ s)	76.6±2.2	230.8±4.8	2,980.8±224.9

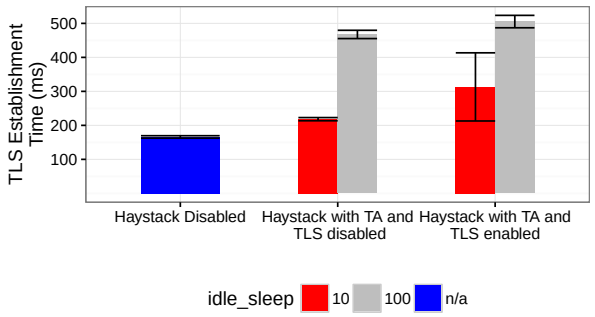
Table 3: Mean processing time (t_{proc}) and standard error of mean (SEM) for Haystack's forwarding operations for TCP and UDP flows under stress conditions. The first packet of a new flow requires a higher processing time.

5.4 Throughput of Haystack

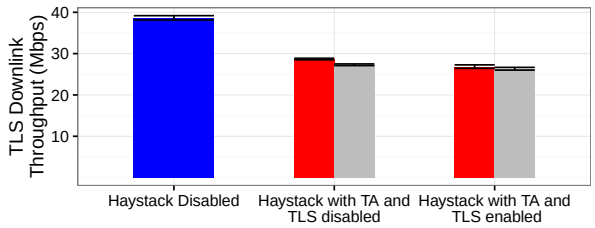
We now investigate the maximum throughput the system can achieve. We use our speed-test app to measure the throughput for non-TLS TCP and UDP flows with *idle_sleep* ={10 ms, 100 ms} and *max_idle_cycles* =100 cycles. This setting provides us with a good compromise between CPU usage and latency.

Figure 5(c) shows the maximum throughput achieved by Haystack's Forwarder. We find that Haystack can provide up to 17.2 and 54.9 Mbps uplink and downlink throughput, respectively. As expected, when *idle_sleep* increases the throughput decreases, as more packets arrive with the Forwarder in idle state, thus incurring t_{buffer} (cf. Figure 4). Haystack also has a bias towards downstream traffic, which stems from two factors. First, as we discuss in §4.1, the NIO read operation may potentially return multiple packets whereas the `tun` interface reads only a single packet at a time. Second, the operations required for upstream packets t_{proc} are more computationally expensive (see Table 3). We plan to investigate how we can adapt the max_read_{nio} and max_read_{tun} parameters to achieve more balanced throughput in future work. We note that the performance we report is still in excess of what is required for modern mobile apps.

Although the TA operates off-path, the use of thread-safe queues to enable communication between the Forwarder and the TA and its CPU intensive operations can inflict significant overhead on traffic throughput. As an example analysis task we consider string matching using the Aho-Corasick algorithm [9] on the traffic to detect tracking. Figure 5(c) shows TA's impact on throughput when performing CPU-intensive string matching on each flow. In the worst case, Haystack pro-



(a) TLS session establishment time.



(b) TLS download speeds.

Figure 6: Session establishment and throughput for TLS with Haystack for different *idle_sleep* configurations, also showing the impact of enabling both the TLS proxy and the TA. We fix *max_idle_cycles* to 100 cycles.

vides 23.3 Mbps downstream and 10.5 Mbps upstream throughput. Even when stress-testing Haystack with our speed-test, the maximum queuing time endured by packets before the string parsing engine processes them does not exceed 650 ms. This worst-case scenario arises when the queues contain a backlog of at least 1,000 packets. Even under such circumstances, the total processing time remains low enough to provide feedback to users about their traffic in less than a second (*e.g.,* exfiltrated private information).

There remains significant potential for improving the overhead imposed by the TA. In particular, we plan to investigate better means of communicating between the Forwarder and the TA (*e.g.,* via Android's IPC [12]) to make it more efficient than the thread-safe queue we currently employ.

5.5 TLS Performance in Haystack

We next turn to the overhead of dealing with encrypted communication. Figure 6 summarizes the overhead of the TLS proxy for different configurations. We first consider the baseline overhead of Haystack without the TLS proxy enabled on TLS connection establishment times, as shown in Figure 6(a). With an *idle_sleep* of 10 ms the TLS connection establishment time is 218 ms. Increasing *idle_sleep* to 100 ms has a large effect, doubling the TLS establishment time (466 ms). Using the TLS proxy further increases establishment time to 653 ms with *idle_sleep* at 10 ms, and 503 ms with *idle_sleep* at 100 ms.

We next assess the overhead of the TLS proxy on throughput, as shown in Figure 6(b). Compared to not running Haystack at all, the overhead of the TLS proxy is 26% and 29% for *idle_sleep* = 10 ms and 100 ms, respectively. Despite the decrease in throughput, overall throughput with the TLS proxy is still 26 Mbps, which (as discussed in §5.2) allows playing a 1080p YouTube video without affecting the user experience. We find *idle_sleep* has little impact on throughput since subsequent packets bring the Forwarder out of the idle state, thus avoiding t_{buffer} for the bulk of the transfer. The fact that the TLS proxy reassembles the streams for the *idle_sleep* also helps reduce the overhead.

5.6 Using Haystack to Measure Performance

Haystack's Forwarder parameters can affect Haystack's ability to accurately measure network performance. This section compares Haystack's ability to assess traffic and network performance with `tcpdump` packet-level timestamps on a rooted phone. For these experiments, we instrument a rooted mobile device with an Android app that performs 500 UDP-based DNS queries to 8.8.8.8 for `[nonce].stonybrook.edu`. The nonce ensures that all queries bypass any intermediate cache. We perform the DNS lookups in two different settings: (*i*) when the DNS traffic goes directly through the default gateway, and (*ii*) when Haystack forwards the DNS traffic. This allows us to calibrate Haystack by comparing actual performance as seen by user-space apps with passive measurements as seen by Haystack.

We use *idle_sleep* =0 ms and *max_idle_cycles* =200 cycles so that we can minimize packet wait time and to prevent blocking on a given interface at the expenses of increasing the CPU load. We send the queries sequentially and with random inter-query delays of 250 $ms + rand(0, 400)$ ms, over a stable, well-provisioned WiFi link. The random delay ensures that packets are not queued and that we are sampling the times in different polling states of Haystack (recall Figure 4). We factor out transient effects in the network by computing the difference between measurements made by Haystack, those made by the Android app, and those obtained via `tcpdump`. Figure 7 shows the difference between our user-level measurements and the reference `tcpdump` measurements. Table 4 summarizes these differences. The difference between Haystack's observation of DNS latency and the Android app is small, with mean and median values differing by less than 50 μs. We find similar results over a cellular link, which we expect because the measured overheads stem from the Java virtual machine and Haystack, not from varying network conditions. The magnitude of the differences we observe remains orders of magnitude smaller than typical mobile network delays, making Haystack suitable

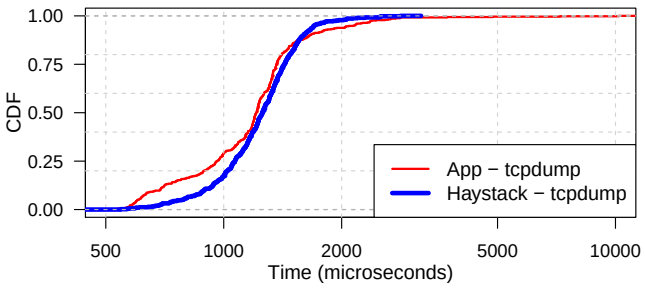


Figure 7: Difference between DNS lookup times as measured by a Java-based application (red line) and by Haystack (blue line), both compared to `tcpdump`. The cross between the red line and the blue line is likely due to instabilities in measuring from the application that is introduced by the Java VM on Android. The analysis confirms Haystack as a valid user space performance measurement platform.

	Mean	Median	StDev
Haystack-tcpdump	$1,261\mu s$	$1,254\mu s$	$303\mu s$
App-tcpdump	$1,250\mu s$	$1,211\mu s$	$658\mu s$

Table 4: Detail statistics of the distribution shown in Figure 7.

for fine-grained network performance measurements.

5.7 Haystack Adaptability

Above we demonstrate the tradeoffs between resource usage and performance of Haystack as controlled by varying *max_idle_cycles* and *idle_sleep* parameters. While we find no sweet spot that is ideal in all circumstances, our observations point to three possible ways to adapt Haystack's operation to the current device state.

Adapting parameters: Our first technique for reducing Haystack's overhead is adapting the parameters to different usage scenarios to strike a balance for the current device state. We consider two scenarios: (*i*) when the user interacts with the device or in the presence of latency-sensitive background traffic (*e.g.,* streaming audio), and (*ii*) when the phone is idle with minimal network activity (*e.g.,* push notifications), a critical scenario as phones remain idle the majority of time [17, 61]. In the first case, minimizing latency and guaranteeing the best user experience is critical ("performance" mode), whereas in the second case, traffic is delay tolerant ("low-power" mode) [17]. Based on our results we determine that *idle_sleep* = 10 ms and *max_idle_cycles* = 100 cycles gives the best tradeoff of performance and resource usage for delay-sensitive usage and *idle_sleep* = 100 ms and *max_idle_cycles* = 100 cycles gives the best tradeoff during delay-insensitive usage. Table 5 summarizes the overheads and performance for each of these settings.

Sampling: A second way to reduce the overhead of Haystack is to sample a fraction of the connections or application sessions to fully analyze. This has the usual costs and benefits of sampling: lower resource usage (*e.g.,* by not requiring the TA to do as much work) on the one hand and less coverage on the other hand. We believe that users likely care most about the apps and networks they use regularly and therefore while sampling may take longer to uncover issues, these issues will in fact be uncovered due to high usage.

Users	Apps	Total Flows	Domains
450	1,340	942,836	8,710

Table 6: Summary and scale of our user study.

Targeting: A final technique to reduce the overhead of Haystack is to allow users to instruct Haystack to only consider certain apps. For instance, this may be useful when the user installs a new app and wants to understand what information it may leak.

6. ADVANTAGES OF HAYSTACK

As we illustrate in §2, the research community has focused much attention on understanding mobile devices and networks. These previous efforts have produced many significant insights. We now turn to discussing Haystack's advantages—for both researchers and users—relative to the state of the art in mobile measurement and app profiling. We describe Haystack capabilities, as well as early results culled from data collected by the 450 users who have installed Haystack to date. We summarize our initial dataset in Table 6. We stress that these are not full-fledged measurement studies and are presented for illustrative purposes. Finally, we note that while we discuss Haystack's capabilities in isolation they can often be used together to an even greater effect.

Unprecedented View of Encrypted Traffic: Haystack's TLS proxy allows analysis—with the user's permission—of encrypted communication. This information remains opaque to other methodologies (e.g., ISP network traces) or requires trusting a third-party middleman to decrypt and correctly re-encrypt traffic while protecting the clear-text version (e.g., remote VPN endpoints). This visibility is significant as we find that in our dataset 22% of the flows are encrypted and less than 20% of apps send all their traffic in the clear. Therefore, gaining an understanding of the information flow within the mobile ecosystem critically depends on being able to cope with encrypted traffic.

Unprecedented View of Local Traffic: Haystack can naturally observe local network traffic that never traverses the wide-area network. This traffic does not appear in ISP network traces [27,62] and methodologies that rely on remote VPN tunnels [52]. This capability will only increase in importance given the emergence of Internet-Of-Things (IoT) devices in the household, typically using mobile devices for control. Our dataset includes 40 apps that generate local traffic, ranging from

Mode	idle_sleep	max_idle_cycles	Mean UDP RTT (ms)	Mean TCP Conn. Time (ms)	Mean TLS Conn. Time (ms)	Max. Throughput [Up/Down] (Mbps)	Mean CPU (%)
Performance	10	100	5.4	24.8	313.1	17.2/54.9	11.2
Low-power	100	100	60.8	65.3	505.3	16.7/48.2	2.7

Table 5: Summary of Haystack's performance for each operational mode in a 5 GHz WiFi link with 3 ms RTT.

baby monitors to media servers to smart TV remote control apps.

Representative View of Apps: When trying to understand app behavior a natural first question concerns finding a set of apps to study. Haystack answers this question quite simply by considering the natural set of apps each user executes. For other methodologies—such as static and dynamic analysis—the set of apps considered often results from a crawl of the Google Play store. However, this neglects built-in apps and apps from non-standard or alternative app stores [23], as well as behavioral changes caused by new app updates. Further, such studies frequently exclude non-free apps or code paths that stem from an in-app purchase [23, 24]. Haystack includes all of these aspects naturally.

Our initial data suggests analyzing these apps is important. We find 15% of the apps we observe did not come from the Google Play store and include (*i*) apps developed by large and small device vendors and mobile carriers, (*ii*) pre-installed third-party apps (e.g., Kineto, a Wifi calling app [4]) and (*iii*) apps downloaded from alternate or regional app stores [50]. These apps create 22% of the traffic we observe. Further, we find that 3.7% of the apps in Google Play are not free and 29% of the apps include in-app purchases. Both of these represent code paths often skipped when studying app behavior, but which Haystack considers as a matter of course. Finally, we find that apps not originating from the Google Play store do in fact leak personal information and unique identifiers (*e.g.,* to third-party tracking services such as Crashlytics).

Representative Code Paths: Related to the last point is that Haystack naturally deals with common and esoteric code paths. Dynamic analysis requires manual navigation within apps, oftentimes synthetically generated via UI Monkeys [66]. The results in turn prove sensitive to details of this test navigation. Haystack does not suffer from this problem because the interactions observed reflect the natural way that the user interacts with the app. Therefore, even though Haystack will never know that some unused code path is nefarious, it does not matter to that user because that user never invokes the problematic case. On the contrary, a dynamic analysis test case might miss some buried feature that gets exercised by real users—in which case Haystack will catch networking activity resulting from that code path.

User Involvement: Haystack's position on user devices provides the capability to engage users in novel ways that benefit both research into the mobile ecosystem, but also the users themselves. For instance, users could be given the option to opt-in to assisting researchers assessing Quality-of-Experience (QoE) of their normal traffic. Combining qualitative user feedback with quantitative measurements of the traffic could be a powerful combination that brings many insights to this space. Another case where direct interaction helps the user is in understanding the leakage of personal information from the phone. By exposing this information to users they can make more informed decisions about what apps to use or what permissions to grant to specific apps.

Novel Policy Enforcement: Haystack's position between apps and the network provides for a unique ability to implement policy before traffic leaves the mobile device. While Haystack can certainly enforce traditional network policies—*e.g.,* blocking access to specific IP addresses or hostnames, or rate-limiting certain traffic—the insight into app content means the policies can be more semantically rich (*e.g.,* blocking based on attempted leaking of a specific piece of information like the IMEI to an untrusted online service). Further, policies can be richer than simple blocking decisions. For instance, specific traffic could be sent through a VPN tunnel or an anonymization network. Or, particular sensitive elements of the contents could be obfuscated (*e.g.,* providing a random IMEI to each app to hinder cross-app tracking).

Enabling Reactive Measurement: Haystack is not beholden to naturally occurring traffic, but can trigger its own active measurements as needed. These measurements—taken from the device's natural perspective—could be proactive in an attempt to better understand the current network context. Alternatively, active measurements could be taken reactively [10] based on some observation—*e.g.,* to diagnose slow transfers or delay spikes. Finally, active measurements could explore "what if" sorts of analysis that could in turn be used to tune the device's operation for better performance. For instance, Haystack could explore whether an alternate DNS resolver would yield faster or better answers compared with the standard resolver.

Explicitness Leading To Precision: Many of the problems in measuring the mobile ecosystem stem from the need to infer or estimate various aspects of the communication. Haystack gets around many of these prob-

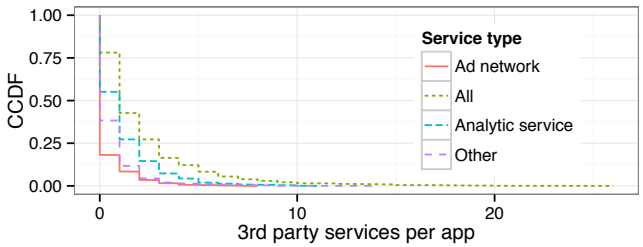


Figure 8: Distribution of the number of third-party services per app across our dataset.

lems because it operates within explicit context from the mobile operating system. For instance, instead of inferring a device's phone number, Haystack has a direct understanding of the value and therefore can directly hunt for it in the traffic instead of searching for something that "looks like" a phone number and then trying to determine if that is in fact the device's phone number or not.

As another example, Haystack is able to directly attribute traffic flows to apps rather than using heuristics or inferences. Within our dataset we use this ability to detect apps that use third-party services—for myriad reasons, including ad delivery, analytics, alternative push notifications [3]—at the network level. While a simple app-agnostic count of accesses to specific services provides some understanding of popular services, this method leaves ambiguous whether the popularity stems from broad use across apps or simply use by popular apps. Haystack can directly answer this question. Figure 8 shows the distribution of the number of third-party services used per app across our dataset. By ranking the online services by the number of apps connecting to them, we can see that Crashlytics and Flurry are the most common third-party services across our corpus of mobile apps.

Large-Scale Deployment: Haystack's operation as a normal user-level app that does not require rooting a device or a custom firmware version means the barrier to entry for using Haystack is low. This is useful for research purposes as we can coax more users to the platform than a more cumbersome tool would require. Further, the platform gives us a direct path to moving beyond research and to actually helping normal users understand the operation of their devices that simply would not be possible if users had to jump through significant hoops to install and use Haystack. These two aspects feed a virtuous circle: more users provide more data that we can leverage to increase users' understanding, which in turn provides a larger incentive to entice additional users.

While each of the above are advantages in their own right, they become even more powerful when combined. For instance, Haystack has the power to understand that a specific app is trying to leak their phone number to some IoT device in their house within an encrypted connection. Further, this understanding could be communicated to the user, as well as serving as fodder for a policy that thwarts such activity in the future. Each aspect of this example is either impossible with current techniques or at least requires inference and heuristics.

7. DISCUSSION AND FUTURE WORK

In §6 we have discussed how Haystack's features provide an unprecedented window into the mobile ecosystem. Haystack shares traits of other network monitoring/measurement platforms we maintain [8,43], increasing our confidence that Haystack's architecture provides a solid basis for future exploration. Since many enterprise and mobile device management solutions rely on the VPN permission, we also expect that Android will continue to support it.

While Haystack's implementation runs on the Android, support on other platforms proves feasible: recent iOS releases offer underlying API primitives similar to those enabling Haystack on Android [15]. In fact, given iOS's tighter technical constraints to app development (*e.g.,* preventing Taintdroid-like approaches), Haystack's approach provides a promising avenue for investigating the iOS ecosystem.

We stress that the applications of Haystack sketched in this paper serve to exemplify its abilities as a traffic inspection platform, not as a step in the network security arms race. For example, we do *not* advocate Haystack as a full-blown TLS inspector—rather, we demonstrate that the platform supports development in this direction to a point that can readily provide interesting results.

We are currently exploring ways to open up Haystack to the research and app developer communities. By further separating the Forwarder and Traffic Analyzer components we can establish access to the device's traffic streams for other apps, effectively providing a proxy to the absent "packet capture" permission on Android.[4] In doing so, we can overcome an additional constraint of Android's security model, namely that only a single VPN app can run at any given time.

We acknowledge that opening up Haystack's capabilities to third-party apps would raise grave security concerns, as malicious apps may abuse Haystack's capabilities for nefarious purposes. We defer full treatment to future work and here only mention that Android's custom permissions model [58] provides avenues for making access to these new capabilities controllable by the user.

We are planning to open-source the Haystack codebase, and will make anonymized data collected by Haystack available online via a web-based query inter-

[4]Personal communication with the Google Android team suggests that this option remains unlikely to ever materialize directly in the Android OS.

face similar to ReCon [53] or Censys.io [1].

8. SUMMARY

We have presented the design, implementation, and evaluation of Haystack, a multi-purpose mobile vantage point for Android devices built on top of Android's VPN permission. As Haystack runs completely in user-space, it enables large-scale measurements of real-world mobile network traffic from end-user devices, with organic user and network input.

Through extensive evaluation, we have demonstrated that Haystack realizes a flexible mobile measurement platform that can deliver sufficient performance with modest resource overhead and minimal impact on user activity when compared to state-of-the-art methods that rely on static and dynamic analysis.

Haystack opens a new horizon in mobile research by achieving an architectural sweet-spot that makes it easy to install on regular user phones (thus enabling large-scale deployment and benefiting from user's input) while enabling in-depth visibility into device activity and traffic (thus providing installation incentives to the user). Using a deployment to 450 users who installed the Haystack app from Google's Play Store, we demonstrated Haystack's ability to provide meaningful insights about protocol usage, its ability to identify security and privacy concerns of mobile apps, and to characterize mobile traffic performance.

9. REFERENCES

[1] Censys. https://censys.io/.

[2] jpcap. A network packet capture library for Java. http://jpcap.sourceforge.net/.

[3] JPush. http://www.jpush.cn.

[4] Kineto. http://kineto.com.

[5] Maxmind. Geo-IP Java API. https://github.com/maxmind/geoip-api-java.

[6] MPAndroidChart. https://github.com/PhilJay/MPAndroidChart.

[7] Networked system ethics. http://www.networkedsystemsethics.net.

[8] The Bro Network Security Monitor. http://www.bro.org/.

[9] Aho, A., and Corasick, M. Efficient string matching: an aid to bibliographic search. *Communications of the ACM* (1975).

[10] Allman, M., and Paxson, V. A reactive measurement framework. In *PAM* (2008).

[11] Anand, S., Naik, M., Harrold, M. J., and Yang, H. Automated concolic testing of smartphone apps. In *ACM SIGSOFT* (2012).

[12] Android Developer's Documentation. Binder IPC. http://developer.android.com/reference/android/os/Binder.html.

[13] Android Developer's Documentation. Security tips. using interprocess communications. http://developer.android.com/training/articles/security-tips.html#IPC.

[14] Android Developer's Documentation. VPN Service. http://developer.android.com/reference/android/net/VpnService.html.

[15] Apple Developer's Documentation. What's new in the network extensions and VPN. https://developer.apple.com/videos/play/wwdc2015/717/.

[16] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. PScout: Analyzing Android permission specification. In *ACM CCS* (2012).

[17] Aucinas, A., Vallina-Rodriguez, N., Grunenberger, Y., Erramilli, V., Papagiannaki, K., Crowcroft, J., and Wetherall, D. Staying online while mobile: The hidden costs. In *ACM CoNEXT* (2013).

[18] Bermudez, I. N., Mellia, M., Munafò, M. M., Keralapura, R., and Nucci, A. DNS to the rescue: discerning content and services in a tangled web. In *ACM IMC* (2012).

[19] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and Wright, T. Transport layer security (tls) extensions. Tech. rep., 2006.

[20] Boneh, D., Inguva, S., and Baker, I. SSL Man in the Middle Proxy. https://crypto.stanford.edu/ssl-mitm/.

[21] Chen, T., Ullah, I., Kaafar, M. A., and Boreli, R. Information leakage through mobile analytics services. In *ACM HotMobile* (2014).

[22] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting privacy leaks in iOS applications. In *NDSS* (2011).

[23] Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., and Sheth, A. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX OSDI* (2010).

[24] Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., and Freisleben, B. Why Eve

and Mallory love Android: An analysis of Android SSL (in) security. In *ACM CCS* (2012).

[25] Falaki, H., Lymberopoulos, D., Mahajan, R., Kandula, S., and Estrin, D. A first look at traffic on smartphones. In *ACM IMC* (2010).

[26] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *ACM CCS* (2012).

[27] Gill, P., Erramilli, V., Chaintreau, A., Krishnamurthy, B., Papagiannaki, K., and Rodriguez, P. Follow the money: Understanding economics of online aggregation and advertising. In *ACM IMC* (2013).

[28] Google Play. FCC SpeedTest. `https://play.google.com/store/apps/details?id=com.samknows.fcc`.

[29] Google Play. MobiPerf. `https://play.google.com/store/apps/details?id=com.mobiperf`.

[30] Google Play. My Speed Test. `https://play.google.com/store/apps/details?id=com.num`.

[31] Google Play. NameHelp. `https://play.google.com/store/apps/details?id=edu.northwestern.aqualab.namehelp`.

[32] Google Play. Netalyzr. `https://play.google.com/store/apps/details?id=edu.berkeley.icsi.netalyzr.android`.

[33] Google Play. Noroot firewall. `https://play.google.com/store/apps/details?id=app.greyshirts.firewall&hl=en`.

[34] Google Play. Ookla SpeedTest. `https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest`.

[35] Google Play. OpenSignal Maps. `https://play.google.com/store/apps/details?id=com.staircase3.opensignal`.

[36] Google Play. Packet Capture. `https://play.google.com/store/apps/details?id=app.greyshirts.sslcapture`.

[37] Google Play. tPacketCapture. `https://play.google.com/store/apps/details?id=jp.co.taosoftware.android.packetcapture`.

[38] 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 *ACM CCS* (2011).

[39] Huang, J., Qian, F., Mao, Z. M., Sen, S., and Spatscheck, O. Screen-off traffic characterization and optimization in 3G/4G networks. In *ACM IMC* (2012).

[40] ICSI - Google Play. Haystack app. `https://play.google.com/apps/testing/edu.berkeley.icsi.haystack`.

[41] J. Sommers and Paul Barford. Cell vs. WiFi: on the performance of metro area mobile connections. In *ACM IMC* (2012).

[42] Kakhki, A. M., Razaghpanah, A., Li, A., Koo, H., Golani, R., Choffnes, D., Gill, P., and Mislove, A. Identifying Traffic Differentiation in Mobile Networks. *ACM IMC* (2015).

[43] Kreibich, C., Weaver, N., Nechaev, B., and Paxson, V. Netalyzr: Illuminating the edge network. In *Proceedings of the ACM Internet Measurement Conference (IMC)* (Melbourne, Australia, November 2010), pp. 246–259.

[44] Le, A., Varmarken, J., Langhoff, S., Shuba, A., Gjoka, M., and Markopolou, A. AntMonitor: A System for Monitoring from Mobile Devices. In *ACM C2B1D* (2015).

[45] Leontiadis, I., Efstratiou, C., Picone, M., and Mascolo, C. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *ACM HotMobile* (2012).

[46] Monsoon Power Monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`.

[47] Nikravesh, A., Yao, H., Xu, S., Choffnes, D., and Mao, Z. M. Mobilyzer: An open platform for controllable mobile network measurements. In *ACM MobiSys* (2015).

[48] Oracle. New i/o apis. `http://docs.oracle.com/javase/1.5.0/docs/guide/nio/index.html`.

[49] Perta, V. C., Barbera, M. V., Tyson, G., Haddadi, H., and Mei, A. A glance through the vpn looking glass: Ipv6 leakage and dns hijacking in commercial vpn clients. *PETS* (2015).

[50] Petsas, T., Papadogiannakis, A., Polychronakis, M., Markatos, E., and Karagiannis, T. Rise of the planet of the apps: A systematic study of the mobile app ecosystem. In *Proceedings of ACM IMC* (2013).

[51] Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. Profiling resource usage for mobile applications: a cross-layer approach. In *ACM MobiSys* (2011).

[52] Rao, A., Sherry, J., Legout, A., Krishnamurthy, A., Dabbous, W., and Choffnes, D. Meddle: Middleboxes for Increased Transparency and Control of Mobile Traffic. In *ACM CoNEXT Student Workshop* (2012).

[53] Ren, J., Rao, A., Lindorfer, M., Legout, A., and Choffnes, D. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic . In *ACM MobiSys* (2016).

[54] Samsung. KNOX VPN SDK. `https://seap.samsung.com/sdk/knox-vpn-android`.

[55] Seneviratne, S., Kolamunna, H., and Seneviratne, A. A measurement study of tracking in paid mobile applications. In *ACM WiSec* (2015).

[56] Seneviratne, S., Seneviratne, A., Kaafar, M., Mahanti, A., and Mohapatra, P. Early detection of spam mobile apps. In *WWW* (2015).

[57] Shafiq, Z., Ji, L., Liu, A., Pang, J., Venkataraman, S., and Wang, J. A first look at cellular network performance during crowded events. In *ACM SIGMETRICS* (2013).

[58] Six, J. An in-detph introduction to the android permission model. `https://www.owasp.org/images/c/ca/ASDC12-An_InDepth_Introduction_to_the_Android_Permissions_Modeland_How_to_Secure_MultiComponent_Applications.pdf`.

[59] Song, Y., and Hengartner, U. Privacyguard: A vpn-based platform to detect information leakage on android devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices* (2015).

[60] Vallina-Rodriguez, N., Aucinas, A., Almeida, M., Grunenberger, Y., Papagiannaki, K., and Crowcroft, J. RILAnalyzer: a comprehensive 3G monitor on your phone. In *ACM IMC* (2013).

[61] Vallina-Rodriguez, N., and Crowcroft, J. Energy management techniques in modern mobile

handsets. *IEEE Communications Surveys & Tutorials* (2012).

[62] Vallina-Rodriguez, N., Shah, J., Finamore, A., Grunenberger, Y., Papagiannaki, K., Haddadi, H., and Crowcroft, J. Breaking for commercials: characterizing mobile advertising. In *ACM IMC* (2012).

[63] Vallina-Rodriguez, N., Sundaresan, S., Kreibich, C., Weaver, N., and Paxson, V. Beyond the radio: Illuminating the higher layers of mobile networks. In *ACM MobiSys* (2015).

[64] Vigneri, L., Chandrashekar, J., Pefkianakis, I., and Heen, O. Taming the Android AppStore: Lightweight Characterization of Android Applications. *ArXiv e-prints* (2015).

[65] Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., and Beznosov, K. Android permissions remystified: a field study on contextual integrity. In *USENIX Security* (2015).

[66] Wong, M. Y., and Lie, D. Intellidroid: A targeted input generator for the dynamic analysis of android malware.

[67] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *ACM CCS* (2013).

[68] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X., and Zang, B. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *ACM CCS* (2013).