UNIVERSITY OF CALIFORNIA,
IRVINE

Mobile Data Transparency and Control

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Anastasia Shuba

Dissertation Committee:
Associate Professor and Chancellor's Fellow Athina Markopoulou, Chair
Chancellor's Professor Gene Tsudik
Professor Brian Demsky

2019

DEDICATION

This one is for my mother.
For making the sacrifice of immigration and giving me the freedom to pursue my interests.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Athina Markopoulou for her guidance, patience, and open-mindedness. She has given me endless opportunties to participate in conferences, give talks, and teach. 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 Gene Tsudik. I am especially thankful to Brian for having introduced Android programming to me when I was but a teen. I also owe my gratitude to Gene Tsudik for opening up the world of security to me.

Third, I would like to thank my collaborators on all the research projects I have been involved in during my studies: Janus Varmarken, Simon Langhoff, Anh Le, Minas Gjoka, Hieu Le, Evita Bakopoulou, Zubair Shafiq, and David Choffnes.

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 patience and support. I will also always be thankful to my high school teacher, Martha Topik, for all that she has done to make my academic career possible. 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.

Finally, I am very thankful to the EECS department for providing me with a fellowship, the ARCS Foundation for providing me with a fellowship, DTL for funding AntMonitor, and NSF for funding multiple projects with grants 1228995, 1028394, and 1815666. I would also like to acknowledge the following open-source libraries that I used in my work: PrivacyGuard, SandroProxy, Frida, Droidbot, Adblock Plus, and the Aho-Corasick implementation by Christopher Gilbert.

CURRICULUM VITAE

Anastasia Shuba

EDUCATION

University of California, Irvine
Doctor of Philosophy in Computer Engineering *2019*
Master of Science in Computer Engineering *2016*
Bachelor of Science in Computer Engineering *2014*

ACADEMIC EXPERIENCE

University of California, Irvine
Graduate Research Assistant *Sept. 2014 – Jun. 2019*

TEACHING EXPERIENCE

University of California, Irvine
Teaching Assistant *Fall 2016, Fall 2017, Spring 2019*

PUBLICATIONS

Peer-Reviewed Publications

1. A. Shuba, Z. Shafiq, A. Markopoulou, "NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking," *Proceedings of the Privacy Enhancing Technologies Symposium (PoPETs), 2018*, **The Andreas Pfitzmann Best Student Paper Award**

2. A. Shuba, E. Bakopoulou, A. Markopoulou, "Privacy Leak Classification on Mobile Devices," *IEEE International Workshop on Signal Processing Advances in Wireless Communications (SPAWC), 2018*

3. A. Shuba, A. Le, M. Gjoka, J. Varmarken, S. Langhoff, A. Markopoulou, "AntMonitor: Network Traffic Monitoring and Real-Time Prevention of Privacy Leaks in Mobile Devices," *Proceedings of ACM S3 Workshop on Mobile Computing and Networking, 2015*

4. A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, A. Markopoulou, "AntMonitor: A System for Monitoring from Mobile Devices," *Proceedings of ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big Internet Data, 2015*

Demos

5. A. Shuba, A. Markopoulou, "Demo: AntWall - A System for Mobile Adblocking and Privacy Exposure Prevention," *Proceedings of the 18th ACM International Symposium on Mobile Ad Hoc Networking and Computing, 2018*

6. A. Shuba, A. Le, M. Gjoka, J. Varmarken, S. Langhoff, A. Markopoulou, "Demo: AntMonitor - A System for Mobile Traffic Monitoring and Real-Time Prevention of Privacy Leaks," *Proceedings of 21st International Conference on Mobile Computing and Networking, 2015*

7. A. Shuba, A. Le, M. Gjoka, J. Varmarken, S. Langhoff, A. Markopoulou, "Demo: AntMonitor - Network Traffic Monitoring and Real-Time Prevention of Privacy Leaks in Mobile Devices," *Proceedings of ACM S3 Workshop on Mobile Computing and Networking, 2015*, **Best Demo Award**

ABSTRACT OF THE DISSERTATION

Mobile Data Transparency and Control

By

Anastasia Shuba

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2019

Associate Professor and Chancellor's Fellow Athina Markopoulou, Chair

Mobile devices carry sensitive information that is routinely transmitted over the network and is often collected for analytics and targeted advertising purposes. Users are typically unaware of this data collection and have little control over their information. In this thesis, we develop systems and techniques that provide users with increased transparency and control over their mobile data. We develop `AntMonitor` – a user space app that can inspect and analyze all network traffic coming in and out of the device. `AntMonitor` outperforms prior art in terms of network throughput and CPU usage. Our tool enables real-time packet interception and analysis on the mobile device, including the following three applications. First, we build `AntShield`, which builds on top of `AntMonitor` to intercept packets, and uses a combination of deep packet inspection and machine learning to detect and block outgoing traffic that contains personally identifiable information. Second, we develop NoMoAds – the first mobile-specific, cross-app, machine learning-based ad-blocker, and we show that it outperforms state-of-the-art filter lists. Third, we build AutoLabel – a system for automatically labeling which packets were sent by apps and which by third-party advertisement or analytics libraries. This eliminates the need for manual labeling – which is the major bottleneck in creating filter lists and classifiers. Overall, this thesis follows a network-based approach to enhance mobile data transparency and give users control over their data.

Chapter 1

Introduction

1.1 Motivation

As of 2019, two-thirds of the world's population uses a mobile device [6]. Out of these 5.1 billion mobile users, 3.9 billion actively use the Internet from their smartphone, leading to over 8 billion mobile connections [6]. It is projected that global mobile data traffic will reach 77.5 exabytes per month by 2022 [7]. Aside from being responsible for large volumes of network traffic, mobile device also carry a wealth of personally identifiable information (PII), such as the user's location, phone number, email, *etc.* Furthermore, mobile applications (apps) offer a range of personal activities to users that range from navigation and financial transactions to diet and sleep trackers. Such apps have access to personal information that is not an explicit identifier, such as a user's sleeping and driving habits.

Mobile apps often access and transmit sensitive information over the network. Sometimes this is justified and required for the intended operation of the app, *e.g.* navigation apps require access and transmission of location data. Although apps request permissions to sensitive resources and users can choose grant or deny them, permissions have several limitations. First, they do not

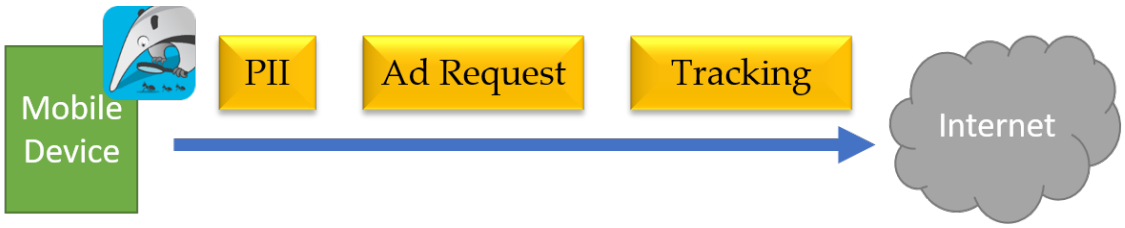


Figure 1.1: Overview of contributions of this thesis. First, we build `AntMonitor` – a system for efficient traffic interception on a mobile device (Sec. 1.2.1). Next, we apply `AntMonitor` to different tasks: (i) detecting and preventing PII exposure (Sec. 1.2.2); (ii) detecting and blocking ads (Sec. 1.2.3); and (iii) detecting and blocking third-party trackers (Sec. 1.2.4).

differentiate between access and transmission of data. Second, they do not protect against inter-process communication where one app that has the permission for a PII can transmit this PII to another app. Third, not all PII is protected by a permission, *e.g.* the Google Advertiser ID and Android Device ID require no permissions on Android. Finally, once a permission is granted to an app, any third-party library contained within is also granted the same permission. Third-party libraries are used by developers to ease development and to generate revenue by including libraries that serve advertisement (ads). These libraries often collect and transmit sensitive information to advertising and tracking (A&T) servers.

Despite increasing user concern over privacy (42% of Internet users believe their data is being misused [6]), few tools exist that provide smartphone users with transparency and control over their data. In this thesis, we aim to provide such a tool, as described in the next Section.

1.2 Contributions

In this thesis, we take a network-based approach and propose several techniques that provide mobile users with transparency and control over their data. We start by building an efficient system for inspecting all incoming and outgoing network traffic on a mobile device (Sec. 1.2.1). Next, we build several applications on top of the system, which analyze the traffic and provide additional functionality, including detecting and blocking transmission of PII (Sec. 1.2.2) and blocking com-

munication to A&T servers (Sections 1.2.3 and 1.2.4). Fig. 1.1 summarizes these contributions.

1.2.1 AntMonitor: Efficient Traffic Interception on a Mobile Device

We take a network-based approach, *i.e.* to intercept and analyze network traffic. To that end, the first step is to build an efficient mechanism for inspecting and acting upon all outgoing traffic of a mobile device. There are multiple ways to achieve this, including changing the underlying OS, described in detail in Chapter 2. Since we want our system to be used by the average user, we chose to use Android VPN APIs [8] to intercept all traffic on a mobile device without requiring rooting or changing the OS. Although our prototype is for Android, VPN APIs are also available on iOS [9]. Thus, our approach can be applied on Apple smartphones as well. Typically, VPN requires the use of a VPN client on the device talking to a VPN server in the middle of the network. However, users may not want to reroute their traffic through a 3rd party VPN server. To that end, our system eliminates the need of a VPN server. Although such an approach provides some benefits (no extra hop on the network path and better scalability), it also poses some challenges. Specifically, since the use of raw sockets requires root, we have to perform layer-3 to layer-4 translation ourselves. This requires careful system design with multiple optimizations to ensure that packet interception and inspection can be done with minimal impact on network throughput and device battery life.

Chapter 3 provides background information on our system (`AntMonitor`) and our optimizations. To enable more applications to be built on top of `AntMonitor`, we packaged it as an Android library and have made it open source at [10].

1.2.2 AntShield: Detecting and Preventing Exposures of PII

Chapter 4 describes the `AntShield` system – an application of `AntMonitor` that detects and blocks transmission of PII, referred to as *exposure*. Some PII, such as the Advertiser ID, are readily

available on the device through Android APIs. In this case, it is possible to use Deep Packet Inspection (DPI) to search outgoing packets for occurrences of such PII. However, some PII is not available via Android APIs. For example, to get PII such as username and password, apps typically present forms for users to fill out. Although the `AntShield` app could ask users to enter such PII, users may not want to share this information with `AntShield`. To that end, we propose a machine learning approach that can detect PII exposure without knowing PII values a priori. To encourage reproducibility, we made our code and dataset available at [10].

1.2.3 NoMoAds: Detecting and Blocking Ads

App developers can collect PII for legitimate reasons, but the privacy concern is greater when third-parties are involved in data collection. In the mobile ecosystem, third-parties take the form of libraries that app developers include. These libraries have the potential to track user activities across apps. Third-party libraries whose primary purpose is to provide ads have a special interest in tracking user activity in order to serve more personalized content. To that end, we propose a system for detecting requests for ads in outgoing network traffic. Our system, NoMoAds, is built on top of `AntMonitor` and can be used for ad-blocking.

In Chapter 5, we explore two ways of blocking ads. First, we match outgoing URL requests against a popular list of ad-blocking rules – EasyList [3]. We show that this list does not perform well in the mobile ecosystem since it was curated for the desktop browsing ecosystem. Second, we propose a machine learning approach for automatically detecting requests for ads. To the best of our knowledge, NoMoAds is the first mobile ad-blocker to effectively and efficiently block ads served across all apps using a machine learning approach. To encourage reproducibility and future work on mobile ad-blocking, we made our code and dataset available at [11].

1.2.4 AutoLabel: Detecting Third-Parties

The NoMoAds system presented in Chapter 5 has two limitations. First, to label ground truth NoMoAds relies on EasyList and new mobile-specific rules that were manually curated. Second, the approach described in Chapter 5 can only be applied to ads: a human can visually see an ad and examine the packet traces captured around the time the ad was shown. However, stateless tracking (tracking that does not require PII) is invisible, thus it is unclear how to manually label such packets.

To that end, in Chapter 6, we present AutoLabel – a system that can automatically label packets that are generated by any third-party library or category thereof. We are particularly interested in A&T libraries. To that end, we use the Frida dynamic instrumentation framework to hook Android networking APIs and collect outgoing network requests along with the stack trace leading to each request. By examining package names in the stack trace, we can identify whether a third-party library generated the request or the app itself. To automatically identify package names belonging to libraries, we utilize advances in static analysis techniques. This eliminates the need to maintain a list of package names and also provides a solution for package name obfuscation. The only list required in the AutoLabel approach is one that maps which package names belong to A&T libraries specifically. We use AutoLabel to collect a new dataset and extend the machine learning framework of NoMoAds to trackers. To enable future research on mobile tracking, we will make AutoLabel open source and we will also release our labeled dataset.

1.3 Definitions

In this chapter we have defined various terms that we will use throughout this thesis. For convenience, these terms are summarized in Table 1.1.

Term	Definition
Apps	Mobile applications
Ads	Advertisements
A&T	Advertising and Tracking
PII	Personally identifiable information, *i.e.* explicit identifiers such as Advertiser ID, *etc.*
Exposure	The transmission of PII from the device to the network. This may be for any (legitimate, tracking, or malicious) reason
Leak	The transmission of PII from the device to the network for a nefarious reason (tracking, exfiltration, lack of permission, *etc.*) as opposed to a legitimate use of PII (for the functionality of the app and with user permission)
DPI	Deep Packet Inspection
DT	Decision Tree classifier
AntMonitor	A system for efficient traffic interception on mobile devices (Chapter 3)
AntShield	A system for detecting and preventing exposures of PII on mobile devices (Chapter 4)
NoMoAds	A system for detecting and blocking mobile ads (Chapter 5)
AutoLabel	A system for detecting third-parties in mobile apps (Chapter 5)

Table 1.1: Definitions

1.4 Thesis Outline

The structure of the rest of this thesis is as follows. Chapter 2 discusses related work. Chapter 3 provides background on `AntMonitor` – a system for packet interception. Chapters 4, 5, 6 describe the following applications of `AntMonitor`: (i) `AntShield` – detecting and preventing transmission of PII, (ii) NoMoAds – ad-blocking based on filter lists and machine learning, and (iii) AutoLabel – automatically labeling packets generated by third party libraries, and using it to train classifiers for detecting A&T, respectively. Chapter 7 concludes the thesis.

Chapter 2

Related Work

Various approaches have been proposed for providing mobile users with increased transparency and control over their data. The approaches can be roughly split into three categories, as described next.

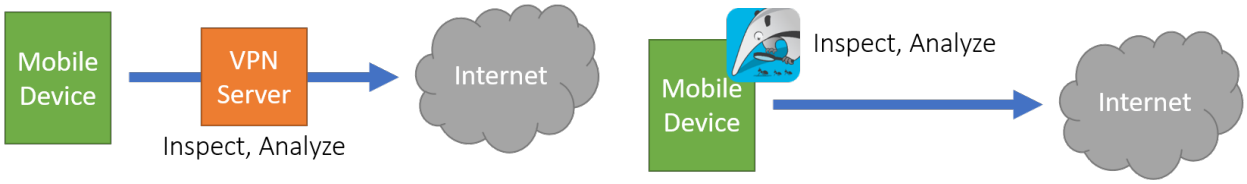
2.1 Custom OS/Rooted Devices

Multiple studies have proposed privacy enhancing solutions that require either a custom OS or a rooted device. For example, TaintDroid [12] was the first to propose taint tracking for mobile privacy. Specifically, TaintDroid adapted the Android OS to track how a taint (PII) propagates from its sink (an API call that provides the PII) to its source (*e.g.* network APIs). Since TaintDroid only considered unencrypted traffic as a sink, AppFence [13] extended the system to also consider SSL traffic. However, both approaches could not deal with PII leaks sent over native code. Similarly, ProtectMyPrivacy (PmP) [14] uses the Xposed framework [15] to dynamically hook into PII-accessing APIs and record the stack trace leading to the API call. From the stack trace, they are able to tell whether the PII is being accessed by the app or a third-party library. However, PmP

was not able hook into networking APIs to trace which part of the app is sending out information. In general, rooting a device or installing a custom OS can be difficult for the average user, and certain phone manufacturers make it altogether impossible. In our work, we use a rooted device for data collection only: our proposed solution for the end-user requires no custom OS and no rooting. Specifically, we combine and improve on the above approaches: we use the Frida framework [16] to hook into networking APIs, including native APIs, and we collect the stack traces that have led to each function call (Sec. 6.3.1.1).

2.2 Static Analysis

Android apps are packaged into APK (Android Package) files, which are essentially ZIP files that can be extracted and analyzed. A separate body of work has proposed extracting and decompiling these APK files to statically analyze the bytecode contained within. This type of analysis can be used to identify potential privacy leaks or to detect presence of third-party libraries. For example, FlowDroid [17], DroidSafe [18], and AndroidLeaks [19] use static taint tracking to detect potential leakage of PII. In another line of work, LibRadar [20] provides a tool for detecting third-party libraries contained within an APK. AdRisk [21] identifies presence of ad libraries within apps and analyzes them for potential privacy and security violations. PEDAL [22] goes a step further: after identifying ad libraries and what sensitive resources they access, PEDAL rewrites the application and blocks accesses to resources based on a user's choice. Unfortunately, static analysis and application rewriting can prove inaccurate and they cannot deal with native code [22], reflection, and dynamically loaded code [23]. In this work, we use existing static analysis tools only to help us identify package names of third-party libraries (see Sec. 6.2).



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

Figure 2.1: Client-Server vs. Mobile-Only VPN Approach: The former requires a VPN server in the middle, while the latter does all inspection on the device.

2.3 Traffic Interception

Another approach to mobile privacy is traffic interception, since by definition, exposure of personal information is transmitted over the network. There are several ways to perform network monitoring on mobile devices: via tools such as `tcpdump`, at the ISP, or at a proxy server. Unfortunately, using `tcpdump` requires root and it can only log traffic – not act on it in real-time. Additionally, within the context of an app that any user can use, monitoring at the ISP is not a viable option either. Therefore, the only way to intercept traffic without requiring root is to use VPN APIs. VPN APIs are widely used: a 2016 study has examined 283 VPN-enabled apps from the Google Play Store [24]. There are two VPN approaches: client-server and mobile-only, visualized in Fig. 2.1, described in Sections 2.3.1 and 2.3.2 respectively, and compared in Table 2.1. Once traffic is intercepted with either method, multiple approaches can be used to block privacy-invasive traffic (Sec. 2.3.3).

2.3.1 Client-Server VPN

In Client-Server VPN approaches (Fig. 2.1(a)), 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` [25], which builds on top of the `StrongSwan` VPN software. 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 avail-

able 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 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.

2.3.2 Mobile-Only VPN

In Mobile-Only VPN approaches (Fig. 2.1(b)), 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 to reach Internet hosts; and vice versa for incoming packets. This approach may have high overhead due to the layer-3 to layer-4 translation, the need to maintain state per connection, and additional required processing per packet. If not carefully implemented, this approach can significantly affect network throughput: for example, [26] has shown the poor performance of `tPacketCapture` [27] – 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` (now renamed to "`Lumen`") [28] and `Privacy Guard` [29]. In Chapter 3 we show that our proposed `AntMonitor` system outperforms both of these approaches.

2.3.3 Analyzing and Acting on Traffic

Once traffic interception is in place, packets can be analyzed via DPI, and encrypted traffic can be decrypted via a man-in-the-middle proxy (*e.g.* see Sec. 3.2.1.1). From there, various privacy-enhancing tools can be built. In literature, two approaches have been proposed: (i) blocking PII and (ii) blocking connections to A&T servers.

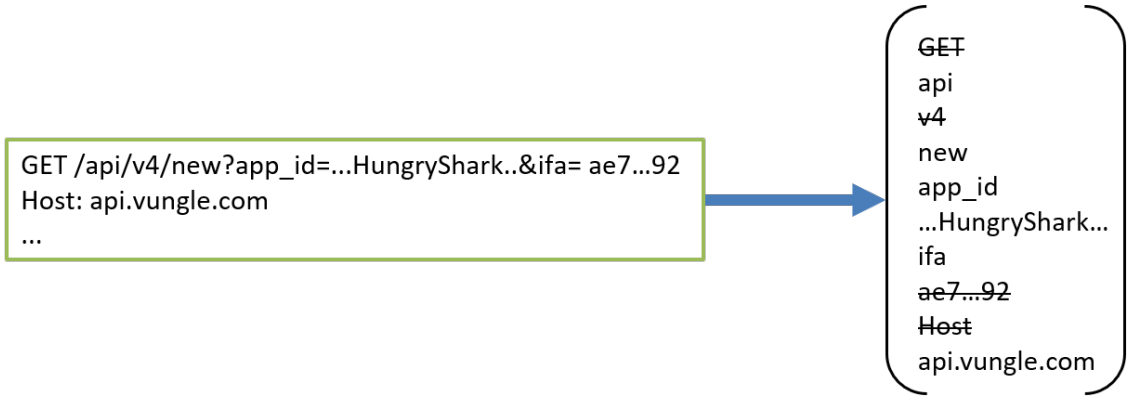


Figure 2.2: Extracting features from packets: a packet is broken into words based on delimiters (*e.g.* '?', '=', ':'). Frequently occurring words, such as common HTTP headers, are removed. In the case of PII prediction, PII is also removed.

2.3.3.1 Blocking PII

`Privacy Guard` [29] utilized the Mobile-Only VPN approach to detect and block PII exposures in real-time based on a predefined list of strings. However, such an approach is unable to detect exposure of information that changes dynamically or is not part of the list. To remedy this, `ReCon` [2] used machine learning to predict PII exposure and block the offending packets at a VPN server, using the Client-Server VPN approach. Specifically, they broke packets into words based on delimiters (*e.g.* '?', '=', ':') and then used these words as features in classification. Various methods were used to ensure that the PII themselves and strings that occur too often or too infrequently are not part of the feature list, see [2] for details. This method of extracting features from packets is summarized in Fig. 2.2. To decide whether or not a packet contains PII (a binary classification problem), `ReCon` used the Java Weka library's [30] C4.5 Decision Tree (DT), and then heuristics for extracting the type of PII. To improve classification accuracy `ReCon` built specialized classifiers for each destination domain that received enough data to train such a classifier. For the rest of the domains, a *general* classifier was built. For the heuristic step, `ReCon` maintained a list of probabilities that a particular key-word corresponds to a PII value. For each PII type, the probability was calculated by taking the number of times the key was present in a packet with the given PII, and dividing it by the number of times the key appeared in all packets. During PII extraction,

ReCon looked for keys with probability higher than an empirically computed threshold.

In this thesis, we build and improve on the ReCon approach for PII detection (Chapter 4) and we also extend it to blocking ads (Chapter 5) and trackers (Chapter 6).

2.3.3.2 Blocking A&T Connections

Applying Filter Lists. Traditional ad-blocking for desktop browsers and mobile ad-blockers, such as DNS66 [31] and AdGuard [32], rely on manually curated filter lists to math outgoing URL requests. Unfortunately, many of such lists, *e.g.* EasyList [3], were created for the web browsing ecosystem, and it was shown in [33, 34, 35] that they do not translate well to the mobile ecosystem. Furthermore, [35] has shown that all popular filter lists that they tested failed to block stateless fingerprinting. Thus, an automation framework is needed to at least assist in filter list creation.

Beyond Filter Lists. Due to the limitations of filter lists, various heuristics-based and machine learning-based approaches emerged. For instance, PrivacyBadger [36] marks hosts as potential trackers if they appear across multiple web pages. If a host is multi-purposed (*i.e.* both functional and tracking), then PrivacyBadger only blocks cookies. However, since users can be tracked via PII and fingerprinting, blocking cookies is not enough. Going a step further, Safari [37] uses machine learning to identify trackers and provides only a simplified system profile to prevent fingerprinting. Unfortunately, both PrivacyBadger and Safari provide protection within a web browser only. In literature, multiple works have proposed machine learning approaches to detect A&T connections, as discussed next. Bhagavatula et al. [38] trained a machine learning classifier on older versions EasyList to detect previously undetected ads. More specifically, they extracted URL features (*e.g.* ad-related keywords and query parameters) to train a k-nearest neighbor classifier for detecting ads reported in the updated EasyList with 97.5% accuracy. Bau et al. [39] also used machine learning to identify tracking domains within the web ecosystem. Later, Gugelmann et al. [40] trained classifiers for complementing filter lists (EasyList and EasyPrivacy) used by popular ad-blockers.

They extracted flow-level features (*e.g.* number of bytes and HTTP requests, bytes per request) to train Naive Bayes, logistic regression, SVM, and tree classifiers for detecting advertising and tracking services with 84% accuracy. Razaghpanah et al. [34] leveraged graph analysis to discover 219 mobile ad and tracking services that were unreported by EasyList. They identified third-party hosts by noting which ones are contacted by more than one app, and then inspected each third party host's landing page for certain keywords that would mark it as an ad or tracking service. Recently, Iqbal et al. [41] proposed an approach that relies on multiple layers of the web stack (HTML, HTTP, and JavaScript) to train machine learning classifiers that detect ads and trackers.

Compared to prior work, our approach trains per-packet classifiers (thus maintaining less state than per-flow) to detect A&T packets in mobile traffic (Chapters 5 and 6). Furthermore, regardless of being mobile-specific or not, all of the proposed machine learning-based approaches suffer from not having an automatic way to obtain ground truth. Specifically, to seed ground truth, prior art relies on either filter lists [39, 38, 34, 41], manually labeled data [40], or a combination of both [33]. In Chapter 6, we build a system that can be used to label data and provide ground truth for any machine learning-based approach. To the best of our knowledge, prior research is lacking an effective approach to automatically detect A&T traffic directly on the mobile device.

	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

Chapter 3

Background: Efficient Traffic Interception on a Mobile Device with AntMonitor

3.1 Overview

In this chapter, we present background information on the design and evaluation of AntMonitor – a system for on-device packet interception and inspection. We note that an earlier version of AntMonitor appeared in [42] and [43]. Compared to the earlier version of AntMonitor, the AntMonitor system presented in this thesis contains notable extensions, including a modularized design, new performance measurements, and a new Graphical User Interface (GUI). We start by outlining the main objectives of AntMonitor and the key design choices made to meet them.

Objective 1: Capturing Packets and Contextual Information: The main purpose of AntMonitor is to intercept, inspect, and collect mobile network traffic. To that end, AntMonitor uses the public Virtual Private Network (VPN) API [8] provided by the Android OS (version 4.0+) to capture all incoming and outgoing IP datagrams. In order to facilitate analysis, AntMonitor saves each datagram in PCAP Next Generation (PCAPNG) format [44], which allows to append

arbitrary information alongside the raw packets. Examples of useful contextual information to append include names of apps that generate the packets and whether or not the packet contains any PII. In many cases, such contextual information may only be collected accurately on the device at the time of packet capture, and can play a critical role in subsequent analyses.

Objective 2: Modularized Design: `AntMonitor` is intended to be a tool that the community can easily build on. For example, if a researcher wants to save packets in JSON format instead of PCAPNG, she should be able to easily make that change by modifying only a few Java classes of `AntMonitor`. Similarly, if a researcher wants to build a tool that blocks all packets containing PII, she should only have to change a few files. To that end, we build `AntMonitor` as an Android library that other researchers can use or modify as needed. We expose several APIs that allow developers to customize packet filtering and capture while hiding all the low-level details of optimized traffic interception.

Objective 3: Ease-of-Use: Aside from being a research tool, `AntMonitor` is also meant to be released to the general public. For example, other researchers may want to provide a privacy-enhancing app for the end-user or to crowdsource data from a large number of users. This objective poses a number of system requirements. First, the app on the mobile device must run without administrative privileges (root access). Thus, we cannot simply use `tcpdump` to capture traffic since it requires a rooted device. In contrast, the VPN APIs do not require root and run on Android versions of 4.0 and above – 99% of devices today [45]. Second, in order for users to adopt `AntMonitor`, user experience must not be affected: monitoring must occur 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 must scale with the number of users. For this reason, we design `AntMonitor` to run completely on the device, without needing to re-route traffic to a VPN server.

The rest of the chapter is organized as follows: Sec. 3.2 describes the implementation of `AntMonitor` and Sec. 3.3 evaluates our system in terms of network throughput, latency, and resource

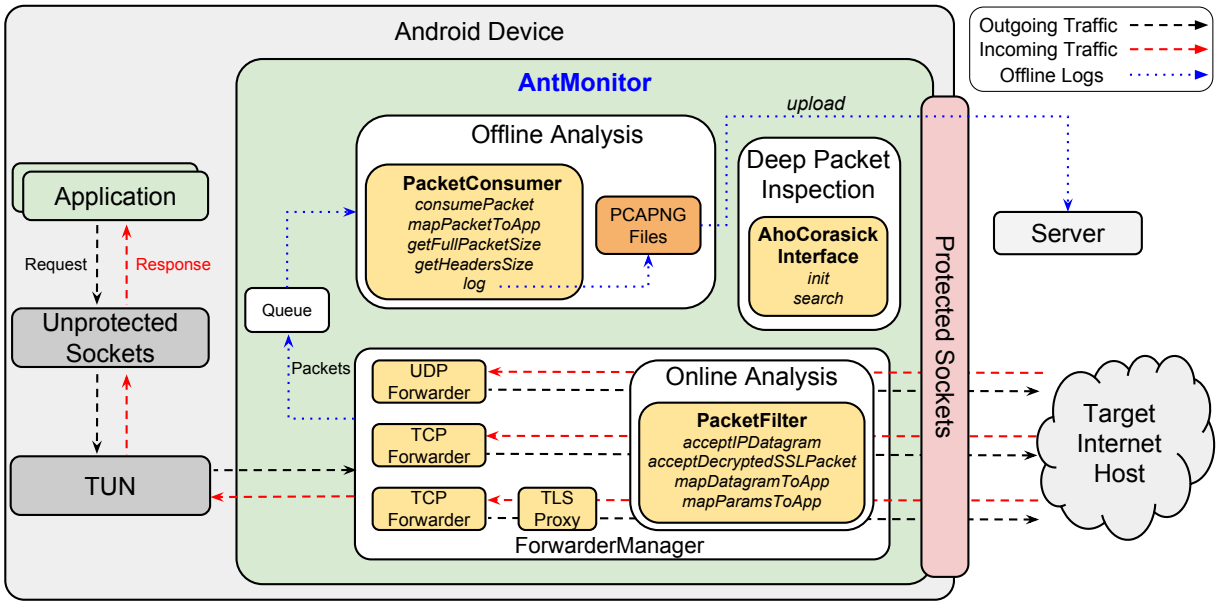


Figure 3.1: The AntMonitor Architecture. Packet interception is performed via the VPN TUN interface. Three modules provide APIs for use by other researchers: Online Analysis (allows packet filtering), Offline Analysis (allows analysis to be performed after packets are sent), and Deep Packet Inspection (provides the ability to inspect packets in real-time).

usage. Sec. 3.4 concludes the chapter.

3.2 System Implementation

As shown in Fig. 3.1, the AntMonitor system consists of four components. The Forwarder is responsible for traffic interception and routing (Sec. 3.2.1). Details of the Forwarder are hidden from AntMonitor's users, who are only provided with high-level APIs within the other three modules, namely the Online Analysis, Offline Analysis, and Deep Packet Inspection modules (Sec. 3.2.2).

3.2.1 Traffic Interception and Routing

To intercept traffic, AntMonitor establishes a VPN service on the device that runs seamlessly in the background. The VPN service is available on 99% of Android devices today [45] and has also been released for iOS versions 8.0 and above [9]. Thus, our approach can also be implemented on iOS. Using a VPN service allows us to intercept all outgoing and incoming IP datagrams by creating a virtual (layer-3) TUN interface [8] and updating the routing table so that all outgoing traffic, generated by any app on the device, is sent to the TUN interface. Once a datagram is captured from the TUN interface, AntMonitor must route it to the target host on the Internet. When the host responds, the response will be routed back to AntMonitor, which then forwards the response to the intended app by writing to TUN.

To route IP datagrams generated by the mobile apps and arriving at the TUN interface, the intuitive solution would be to use raw sockets, which unfortunately 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 one of the following two ways:

1. Client-Server Routing: This follows the design of a typical VPN service: all traffic is routed through a VPN server [46, 25, 26]. The main advantage is the simplicity of implementation: the routing is done seamlessly by the operating system at the server with IP forwarding enabled. However, this approach has several limitations. First, if AntMonitor is to be used as a crowdsourcing system, requiring a VPN server poses scalability challenges. Second, adding an extra hop to the network path may negatively impact network throughput and user experience. Finally, redirecting users' traffic to a server expands the trust base: users not only have to trust the app running on their phone, but also the server. Therefore, we use an alternative routing approach that can work entirely on the mobile device, without the need for a VPN server, as described next.

2. Mobile-Only Routing: Routing IP datagrams to target hosts through layer-4 sockets requires a *translation between layer-3 datagrams and layer-4 packets*. 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. The Mobile-Only design removes the dependency on a VPN server, thus making `AntMonitor` self-contained and easy to scale. Furthermore, this design enhances user privacy as all data can now stay on the mobile device and is not routed through a middlebox. If a researcher needs to crowdsource data, packets can be temporarily logged on the device then uploaded to a remote server, as described in the next section.

3.2.1.1 Implementation Details

The **Forwarder** manages the TUN interface and is responsible for the translation between layer-3 datagrams and layer-4 packets. The current implementation of `AntMonitor` supports only IPv4 traffic, but it can be easily extended to support IPv6. As depicted in Fig. 3.1, the Forwarder consists of three main components: the UDP and TCP Forwarders and a TLS Proxy, as described below.

The **UDP Forwarder** is simpler, since 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 (a tuple consists of an IP address and a port number), to be used later for reverse lookup. 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, changes the destination tuple to the corresponding source tuple in the recorded mapping, and writes the datagram 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 a 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 **TLS proxy** allows us to decrypt TLS traffic in real-time so that analysis can be performed on plain text. This is necessary since much of the traffic is encrypted and can be used to fetch ads or leak PII. We based our TLS Proxy implementation on `Privacy Guard`, which uses the SandroProxy library [47] 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.

3.2.2 APIs Provided

Next, we describe the APIs provided by `AntMonitor` in the Online Analysis, Offline Analysis, and Deep Packet Inspection modules (see Fig. 3.1).

The **Online Analysis** module provides the capability to take action on live traffic, *e.g.*, preventing private information from leaking. Since this analysis is done on the device, private information is

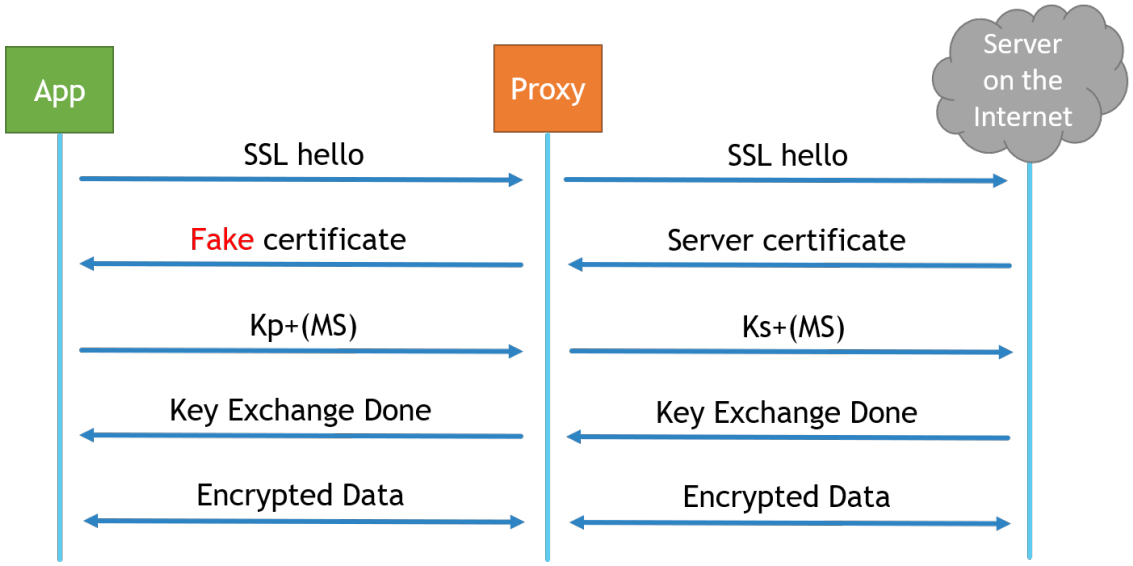


Figure 3.2: TLS Interception: Kp+ and Ks+ stand for the public key of the proxy and server, respectively.

never leaked out, setting `AntMonitor` apart from systems like `Meddle` [25], that perform leakage analysis at a VPN server. In addition, this module provides APIs for mapping intercepted datagrams to the names of the apps that generated them. 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 UIDs of apps responsible for each connection. Given a UID, we get the corresponding package name using Android APIs. This can be useful if users want to take different actions for different apps. For instance, *Google Maps* may be allowed to sent location data, but a flashlight app may not.

The **Offline Analysis** module allows users to take action on traffic after it has been sent. For example, if a researcher wants to crowdsource data, she can save captured datagrams. `AntMonitor` provides APIs for saving datagrams in PCAPNG format, so developers can also append various comments to each packet. For instance, they can use the mapping of datagrams to apps (as in the Online Analysis module) to save the name of the app responsible for generating the datagram. In general, this module can be used for any heavy processing of datagrams that cannot be performed on live traffic without impacting performance. Finally, this module also provides the ability to

upload captured PCAPNG files to a server specified by the developer.

The **Deep Packet Inspection** module provides developers with the ability to match strings within live traffic. Thanks to various optimizations (see Sec. 3.2.3), this module can inspect traffic in real-time without causing a significant impact on performance. Users specify the strings that they are interested in using the initialization API call, and can then inspect every datagram for those strings using the search API call. For example, researchers may be interested in searching for PII.

3.2.3 Optimizations

Since `AntMonitor` processes raw IP datagrams in user-space, it is highly non-trivial to achieve good performance. We investigated the performance bottlenecks of our approach specifically and VPN approaches in general. We also provide a detailed comparison of our design to `PrivacyGuard` [29] (whose source code is publicly available); in contrast, `Haystack`'s (now renamed to "`Lumen`") source code is unavailable, therefore we qualitatively compare a subset of techniques that we could infer from `Haystack`'s description [28] and our observations.

3.2.3.1 Minimizing Resource Usage

First, to minimize the impact of `AntMonitor` on the mobile device, we strive to use the minimum number of resources while achieving high performance. Specifically, we use the minimum number of: (i) threads (two) for network routing, and (ii) sockets, as described next.

Thread Allocation. 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 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 so that `AntMonitor` (which runs in the background) does not interfere with other apps that run in the foreground. For 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. 3.3).

Socket Allocation. 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. Using a single socket per UDP connection poses a challenge when routing DNS packets. This is because when we read data from our UDP socket, we are only returned the IP and port number of the server sending us UDP data. In most cases, we can keep a mapping of this tuple (server IP and port number) to the intended destination port number on the device (IP address of the device remains the same during a VPN session). However, in the case of DNS, the server IP address (*e.g.*, `8.8.8.8` – Google's DNS server) and port number (53) are always the same. Thus, when multiple DNS requests are issued in close proximity to each other, we cannot keep an accurate mapping of the server tuple to the port number on the device. To handle this special case, we use the DNS transaction ID to identify DNS connections and reconstruct DNS responses with correct destination port numbers. For comparison, `Privacy Guard` uses 1 socket per UDP connection and 2 sockets per TCP connection; `Haystack` uses 1 socket per UDP connection and 1 socket per TCP connection.

3.2.3.2 Achieving High Performance

Aside from using the minimum amount of resources, we also address multiple bottlenecks that arise at various points of VPN-based packet inspection – see Fig. 3.3. We address these bottle-

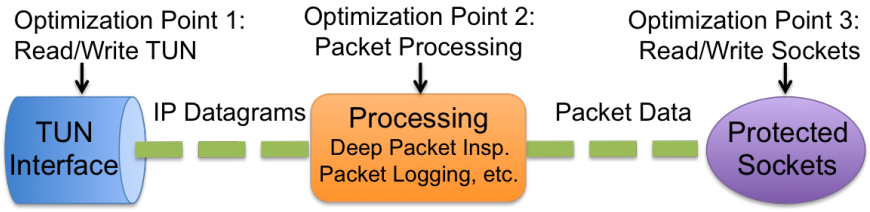


Figure 3.3: Performance Optimization Points.

necks through a combination of approaches: from typical optimization methods (including implementing custom native C libraries and deploying high-performance network I/O patterns) to highly customized techniques that we specifically devised for the VPN-based architecture.

Traffic Routing (Points One, Two, and Three). The techniques that we adopted are as follows: (i) we explicitly manage and utilize a Direct ByteBuffer for I/O 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 I/O performance because it eliminates copy operations when I/O is performed in native code. Plus, Direct ByteBuffer on Android 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 (*e.g.* for manipulation or packet logging) can be done efficiently by performing a 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 I/O operation, *i.e.*, read from TUN, we reuse the buffer for every operation instead of allocating a new one.

TUN Read/Write (Point One). The Android API does not provide a way to *poll* the TUN interface for available data. The official Android tutorial [48], as well as `Privacy Guard` and `Haystack` [29, 28], employ periodic sleeping (*e.g.*, 100 ms) between read attempts. This results in wasted CPU cycles if sleeping time is small, or in slow read speeds 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()* to read data to a Direct ByteBuffer (which is then available in the Java code without extra copies). It is also important 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()). In early implementations, 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 a maximum read speed of about 25 Mbps on a Nexus 6 for a TCP connection, thus limiting the upload speed. We were able to increase the datagram size by (i) increasing the Maximum Transmission Unit (MTU) of the TUN interface to a large value, *e.g.*, 16 KB and (ii) including an appropriate Maximum Segment Size (MSS) in the TCP Options field of SYN-ACKs sent by the TCP Forwarder when responding to apps' SYN datagrams. These changes help ensure that an app can acquire a high MTU when performing Path MTU Discovery, so that each read from TUN results in a large IP datagram. This results in the maximum read speed, *i.e.*, more than 80 Mbps on our Nexus 6. Similarly, it is important to write to TUN in large blocks. We have experimented with several large block values (e.g., 8K, 32K) and found that a datagram size of 16 KB achieves the highest throughput on a Nexus 6.

Socket Read/Write (Point Three). 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, we match 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. For comparison, `Privacy Guard` does not implement this matching. Although sending a large IP datagram read from TUN might through a UDP or TCP socket may cause fragmentation, it is handled efficiently by the kernel.

Packet-to-App Mapping (Point Two). 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 I/O operation, we minimize the number of times we have to read and parse

them by storing the mapping of app names to the source port numbers in a `HashMap`. This way, when the API call for packet-to-app mapping in invoked, we first check the `Map` for the given source port. If the mapping does not exist, we re-parse the `/proc` files and update the `Map`. We note that it is also possible to do file parsing in native C, but based on our experiments this technique does not provide benefits in this case. This is because only `Direct ByteBuffers` can be shared across the native C and Java boundary, while other structures, such as `Maps`, require an additional copy. In addition, we made the choice to key our `Map` by source port alone, because adding IP addresses can cause a high overhead when key comparison is performed during an item fetch. Specifically, in earlier versions of `AntMonitor`, we keyed our `Map` by IP destination, destination port, and the source port, all concatenated into a `String`. However, this implementation could not be used in real-time: we were only able to reach a throughput of one Mbps when testing with *Speedtest*. Switching to source port as a key allowed us to do real-time packet-to-app mapping while achieving network speeds close to regular device operation speeds. We note that it is also possible to store a 32-bit IPv4 address and both port numbers (16 bits each) in a 64-bit primitive type, such as a `long`, which can be used for fast lookup in a `HashMap`. However, this technique cannot extend to 128-bit IPv6 addresses. Thus, we keep the source port as the key to our `Map`. Since each TCP and UDP connection must have a unique source port, using it as the only key is acceptable.

DPI: Deep Packet Inspection (Point Two). A researcher may want to inspect every packet for a list of strings, which can be a costly operation. To allow such analysis to happen in real-time without significantly impacting throughput and resource usage (see Sec. 3.3.1.3), we leverage the Aho-Corasick algorithm [49] written in native C [50]. However, this alone is not enough: we must also minimize the number of copies of each packet. Although the algorithm generally operates on Strings, `AntMonitor` uses Direct ByteBuffers for 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. Therefore, any String passed from Java to native C requires another copy while converting from UTF-16 to UTF-8 [51]. 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 enables us to perform an order of magnitude faster than Java-based approaches, such as `Privacy Guard`'s Java-based string matching and `Haystack`'s Java-based Aho-Corasick implementation (Sec. 3.3.3).

3.3 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 two types of experiments. In Sec. 3.3.1, *Stress Test* performs downloads and uploads of large files so that `AntMonitor` has to continuously process packets. In Sec. 3.3.2, *Idle Test* considers an idling mobile device so that `AntMonitor` handles very few packets.

Baselines for Comparison. We report the performance of `AntMonitor` v0.0.1 and compare it to state-of-the-art baselines from Sec. 2.3:

- `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` [29] v1.0 and `Haystack` [28] v1.0.0.8. (We omit the testing of `tPacketCapture` [27] since it was shown to have very poor performance in [26].)
- Client-server VPN approaches: industrial grade `StrongSwan` VPN client v1.5.0 with server v5.2.1, and an `AntMonitor Client-Server` implementation based on [26]. Note, that in the

labels of the performance figures, we refer to the latter as "AM. Client-Server," to distinguish it from the actual proposed `AntMonitor`, referred to as "AM. Mobile-Only". The VPN servers used by each app were hosted on the same machine.

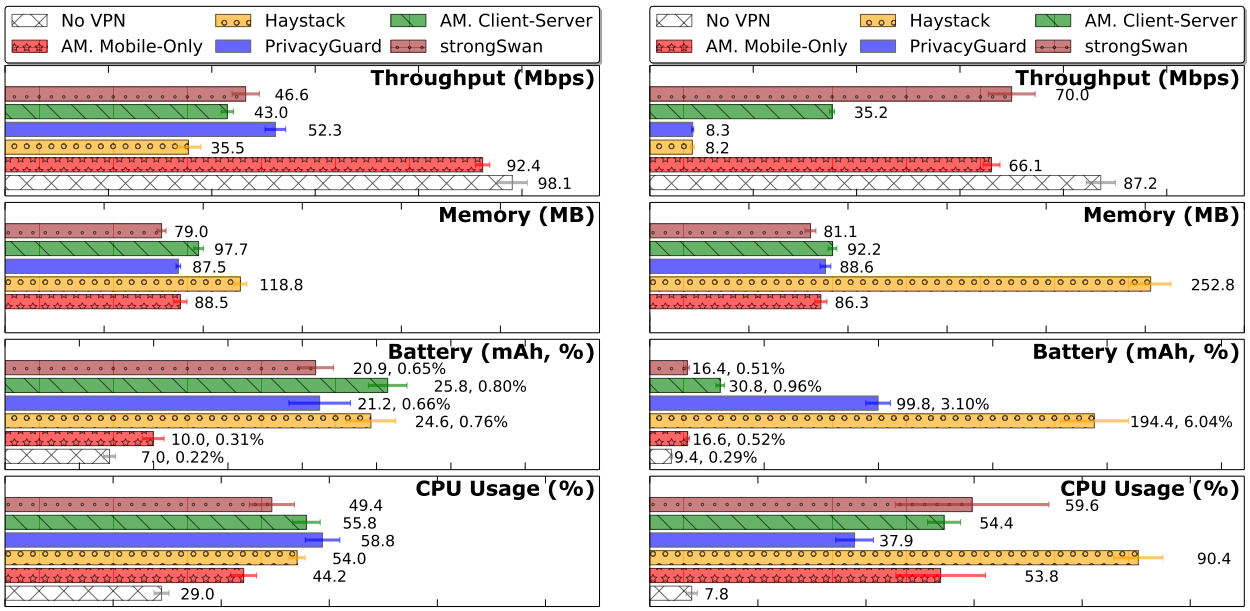
Setup. All experiments were performed on a Nexus 6, with Android 5.0, a Quad-Core 2.7 Ghz CPU, 3 GB RAM, and 3220 mAh battery. Nexus 6 has a built-in hardware sensor, Maxim MAX17050, that allows us to measure battery consumption accurately. Throughout the experiments, the device was unplugged from power, 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 `AntMonitor` was logging full packets of all applications and inspecting all outgoing packets. In terms of versions, all tests were done with `AntMonitor` v0.0.1 and `Haystack` v1.0.0.8, unless indicated otherwise (Sec. 3.3.1.1). 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. Each test case was repeated 10 times and we report the average.

3.3.1 Stress Test

3.3.1.1 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:

- *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.



(a) Download (b) Upload

Figure 3.4: Performance of all VPN apps when downloading or uploading a 500 MB file on Wi-Fi. "AM." stands for AntMonitor. "AM. Mobile-Only" stands for AntMonitor proposed in this thesis. "AM. Client-Server" is only used as a baseline for comparison.

- *Memory Usage*: AntEvaluator uses the top command to sample the Resident Set Size (RSS) value.
- *Battery Usage*: AntEvaluator uses the APIs available with the hardware power sensor Maxim MAX17050 to compute the energy consumption during each test in mAh [52].
- *CPU Usage*: AntEvaluator uses the top command to measure the CPU usage.

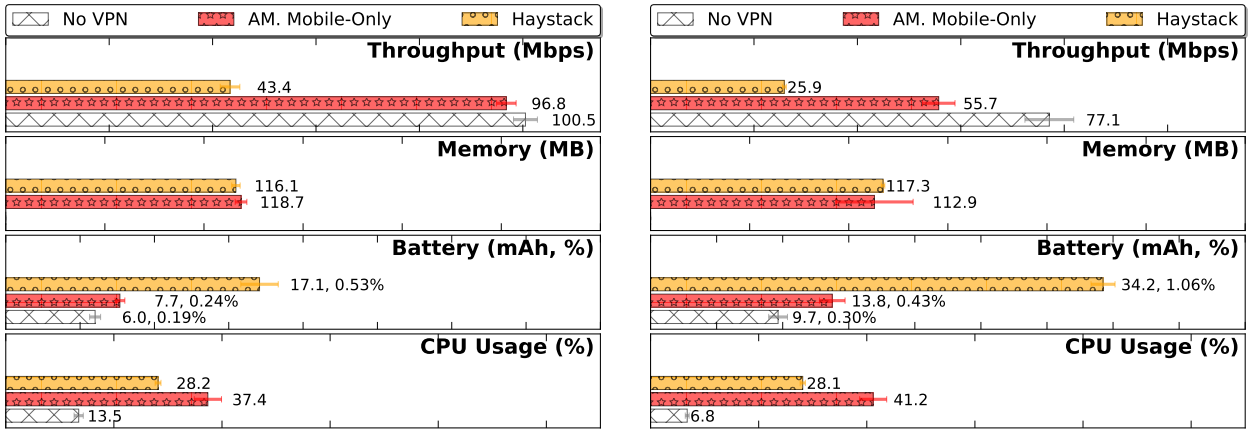
At the end of each experiment, AntEvaluator reports the calculated throughput and battery usage, and the average memory and CPU (considering the sum of CPU usage of AntEvaluator and the VPN app) usage.

Results for AntMonitor v0.0.1 in Dec. 2015. Fig. 3.4(a) shows that the download throughput of AntMonitor 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. Although StrongSwan does not perform L3-L4 translation, it performs encryption and decryption, which results in about 5% higher CPU usage than AntMonitor. Fig. 3.4(b) reports the upload performance. AntMonitor 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. The gains provided by the optimizations discussed in Sec. 3.2.3 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 [28]. StrongSwan outperforms AntMonitor as expected since, unlike with incoming packets, AntMonitor performs DPI on each outgoing packet. Nevertheless, Fig. 3.6(a) shows that AntMonitor has the higher upload speed (and closest to the raw speed) if DPI is disabled. Fig. 3.4(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 has the highest throughput while having similar or lower CPU, memory, and battery consumption.

Results for AntMonitor v0.1.5 in Feb. 2017. Since the time we performed the above evaluation (in Dec. 2015), both AntMonitor and Haystack (now renamed to "Lumen") were updated on GooglePlay, to versions 0.1.5 and 1.1.2, respectively. The main change for AntMonitor from version 0.0.1 to 0.1.5 was improving (and making more complex) the GUI and disabling packet logging (in order to avoid dealing with PII of the general public participating in the open-beta); the underlying design (including key optimizations, such as the use of two threads and minimum number of sockets, and the optimized reading/writing from the TUN) remains the same. The internal changes in Haystack beyond the GUI are not available to us but the design of the system appears to



(a) Download (b) Upload

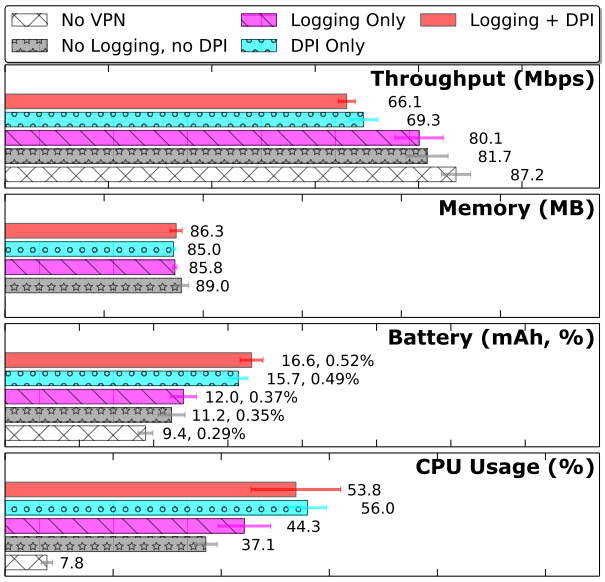
Figure 3.5: Performance of updated VPN apps (in Feb. 2017) when downloading or uploading a 500 MB file over Wi-Fi.

remain the same, according to [53]. To see how the updates affected the performance of both apps, we repeated a set of stress tests and we report these results separately in Fig. 3.5(a-b). The network conditions have changed since the original tests were performed: `NoVpn` throughput is now 100 Mbps and 78 Mbps on the downlink and uplink, respectively. `Haystack` v1.1.2 has improved its download throughput to 43 Mbps, and `AntMonitor` reached 96% of the raw speed with a 96 Mbps throughput. The newer `Haystack` app also has improved its upload throughput to 26 Mbps. The latest `AntMonitor` has stayed in the 70% range of the raw upload speed with a 56 Mbps throughput. This confirms that the design and optimization techniques applied to `AntMonitor` still result in significant performance benefits, despite the added complexity of the updated GUI.

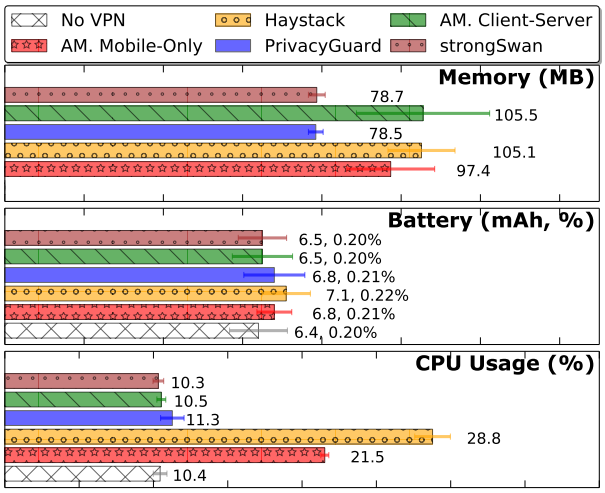
3.3.1.2 Small Files Over Multiple Flows

Setup. To test the efficiency of `AntMonitor`'s thread and socket allocation, we used `AntEvaluator` to create 16 threads, each downloading a 50 MB 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`:



(a) Impact of logging packets and performing DPI with AntMonitor when uploading a 500 MB file.



(b) Impact of VPN apps during device idle time.

Figure 3.6: Performance of VPN apps and variations of AntMonitor under various conditions.

6.82, AntMonitor: 6.57, Privacy Guard: 4.75, StrongSwan: 3.73, Haystack: 3.18, and AntMonitor Client-Server: 3.06. Again, AntMonitor came out on top, achieving 96% of the raw speed.

3.3.1.3 Impact of Logging and DPI

Setup. To assess the overhead caused by DPI and logging packets in PCAPNG format, we performed the single-flow upload stress test on AntMonitor with all four combinations of logging on/off and DPI on/off.

Results. First, Fig. 3.6(a) shows that logging does not have a significant impact on throughput. This is thanks to (i) the optimization of AntMonitor that uses only two threads for network I/O (see Sec. 3.2.3) 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 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 achieves 94% of the raw speed without VPN.

3.3.1.4 Impact of TLS Proxy

Setup. To evaluate the performance impact the TLS Proxy (Sec. 3.2.1), we used AntEvaluator to download a 500 MB file from a secure server over HTTPS and compared the throughput of AntMonitor to that of the Raw Device.

Results. The average throughput (in Mbps) was 77.2 and 69.1 for the Raw Device and AntMonitor, 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.

3.3.2 Idle Test

Setup. For this set of experiments, we kept the phone idle for two 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. 3.6(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 used more CPU than Privacy Guard because both of them have threads to log packets while PrivacyGuard 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.

3.3.3 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`: 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` 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).

DPI. 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 less than one ms. For comparison, `Haystack` reports a 167 ms maximum run time for string parsing with Aho-Corasick [28].

3.4 Summary

In this chapter, we presented the `AntMonitor` system for intercepting and inspecting network traffic on mobile devices. `AntMonitor` uses VPN for traffic interception, without requiring a VPN server or rooting of the device. Thanks to a number of optimizations, `AntMonitor` outperforms previous state-of-the-art mobile-only approaches, namely `Privacy Guard` [29] and `Haystack` [28]. Specifically, it achieves 2x and 8x faster (down and uplink) speeds, while using 2–12x

less energy. This significant performance benefit allows other researchers to build a multitude of applications on top of AntMonitor. These applications can then be easily adopted by users since AntMonitor does not slow down the mobile device and does not drain its battery. The next three chapters showcase some of the possible applications of AntMonitor. To enable further study of mobile network traffic, we packaged AntMonitor as an Android library and made it open-source at [10] for the community to maintain and evolve.

Chapter 4

AntShield: Detecting and Preventing Exposures of PII

4.1 Overview

In this chapter, we present `AntShield` – a system built on top of `AntMonitor` for detecting and blocking exposures of personally identifiable information (PII). We refer to a PII *exposure* as the transmission of PII from the device to the network. This transmission may be for any reason, including tracking and legitimate usage of PII needed for the functionality of the app. In contrast, a PII *leak* is the transmission of PII from the device to the network for a nefarious reason, such as tracking and exfiltration. In this Chapter, we are interested in detecting exposures only, and leave it up to the user to decide when an exposure is a leak.

To detect PII we use a combination of string matching and machine learning. Our approach runs 100% on the device, without routing traffic through a remote VPN server, and in real-time – around one millisecond. This is enabled by careful system design and multiple optimizations discussed in Sec. 4.2.4. Our *multi-label* classification methodology achieves significantly higher F-scores

(7-49% improvement) and lower variance (a factor of two to six) compared to state-of-the-art. We also design and advocate for *per-app*, instead of *per-domain*, classifiers: they achieve similar classification accuracy, but allow faster and more scalable operation while covering more traffic. In order to demonstrate the effectiveness of our approach, we collect a new dataset of privacy exposures on mobile devices, which we made available to the community at [54].

4.2 System Design & Implementation

4.2.1 Goals and Design Rationale

Mobile devices have access to a wealth of resources and information, much of which is personal and potentially sensitive. We will refer to such personally identifiable information as *PII*. Examples include:

- Device Identifiers: IMEI, AndroidID, phone number, serial number, ICCID, MAC Address.
- User identifiers: credentials (per app, usually transmitted over HTTPS), advertiserID, email.
- User demographic: first/last name, gender, zipcode, city, *etc.* - unavailable through Android APIs.
- Location: available through Android APIs.
- User-defined: the user can also define any custom string that should be monitored (*e.g.* see GUI in Fig. 4.3(a)), such as digits of her credit card.

A key insight of our design is the distinction on whether PII of interest is known to the device or not. We refer to PII that consists of strings known a priori on the device (*e.g.* via Android APIs, or defined by the user) as *predefined*. We refer to PII that is not known to our system (*e.g.* hidden

from apps) as *unknown*. Our system employs different techniques to detect the transmission of *predefined* PII (`String Matching`) and *unknown* PII (classification). We refer to the transmission of a packet from the device to the network, containing at least one PII, as a *privacy exposure*. This transmission may be intended to collect information about the user; or can be benign, *e.g.* necessary for the operation of the app, acceptable to the user, or more of the honest-but-curious nature. On the other hand, a PII *leak* is the transmission of PII from the device to the network for a nefarious reason, such as tracking and exfiltration. Distinguishing between privacy *exposure* and an actual privacy *leak* is out of the scope of this thesis, and we leave it up to the user to decide when an exposure is a leak. Our goal is to detect privacy exposures *on the device* with low overhead, accurately and in real-time.

To that end, we build `AntShield` on top of the `AntMonitor Library` described in Chapter 3. This provides us with all the advantages of `AntMonitor`: our solution can run completely on the device and can be used by a non-sophisticated end-user. In addition, using `AntMonitor` provides us with access to app names and *predefined* PII. To detect *predefined* PII (through `String Matching`), we utilize the DPI API provided by `AntMonitor` (see Sec. 3.2.2). To detect the remaining *unknown* PII, we build machine learning classifiers defined in Sec. 4.2.3. This hybrid approach allows us to accurately detect a comprehensive range of PII in real-time.

4.2.2 AntShield Architecture

The overview of the `AntShield` architecture is depicted in Fig. 4.1. It consists of a mobile app and an optional server. A brief overview of each component is provided next.

Online Exposure Detection. This is the core functionality of our PII exposure detection. As shown in Fig. 4.1, `AntShield` leverages calls to `AntMonitor Library` (`acceptIPDatagram` and `acceptDecryptedSSLPacket`) to intercept packets (in clear text or decrypted SSL, respectively). Each outgoing intercepted packet is analyzed with DPI for *predefined* exposures and

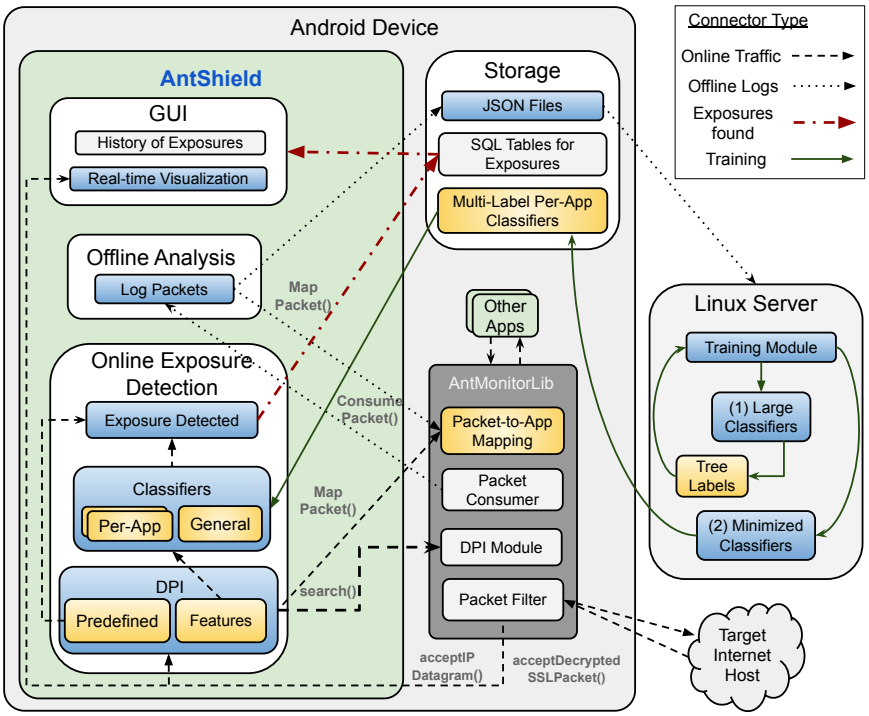


Figure 4.1: `AntShield` Architecture. It consists of a mobile app on the device and an (optional) server. Real-time classification consists of the following steps: each packet is intercepted by `AntMonitor Library`, mapped to an app, and analyzed for multiple *predefined* and *unknown* exposures; detection occurs before the packet is forwarded towards its remote destination (and an action may be taken to block the exposure). Offline operations include loading and (re)training the classifiers, and (if the user agrees) uploading any logs.

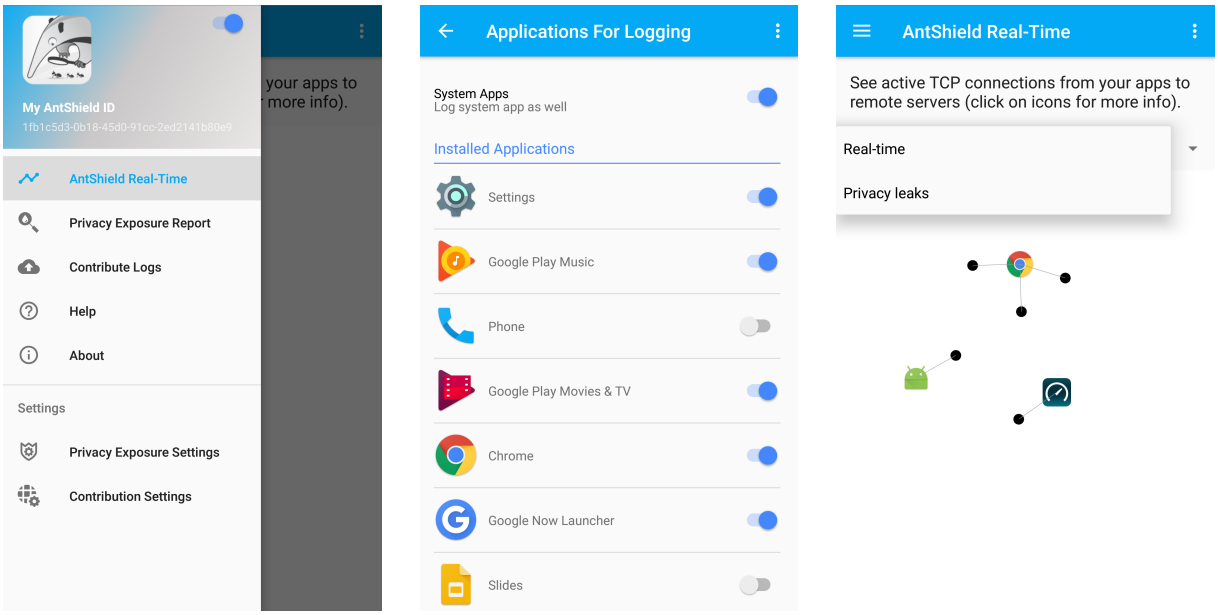
for features. The features are then passed to classifiers to detect *unknown* exposures (described in detail in Sec. 4.2.3). Either way, if a PII exposure is detected, the user is notified and the exposure is logged. If the user chooses to, the offending packet can be blocked, or it can be allowed to continue towards its remote destination. In the case of a *predefined* exposure, the user can also choose the replace the PII with a randomly generated string.

Offline Analysis. This module can be used when heavier processing is required. For example, to generate logs on the device, which require I/O operations, we use `AntMonitor Library`'s `consumePacket()` API from this module. This module can be used to generate ground truth on the device; and in the future, it can be extended to re-train classifiers on the device without sending data to a central server.

Storage. The AntShield app comes pre-loaded with classifiers trained on our existing dataset (Sec. 4.3), so that users have no need to contact any server and can use the system as-is. Only if the user chooses to do so, logs (*e.g.* packet traces) can be maintained on the device and occasionally be uploaded to a server. The user has full control of these logs: by default, logs do not need be collected or leave the device. In our work, we use the Offline Analysis module to generate the dataset used in our evaluation: each captured packet is labeled with the type of PII that it is exposing and the app name that generated the packet, the PII itself is replaced, and the packet is converted to JSON format for easier processing at the machine learning training stage (see Sec. 4.3 for details). In general, this feature is useful for other researchers to generate their own datasets manually or from user studies. Data can either be uploaded to a server or pulled from the device using ADB (Android Debug Bridge).

Sever. The use of the server is optional: the user may choose this option for offline operations such as storing data, sharing information with other users to get the benefits of crowdsourcing, and training and retraining classifiers. In our work, we use a server for (i) data collection, (ii) training/re-training on arriving ground truth, and (iii) keeping classifier files for users to download upon app installation.

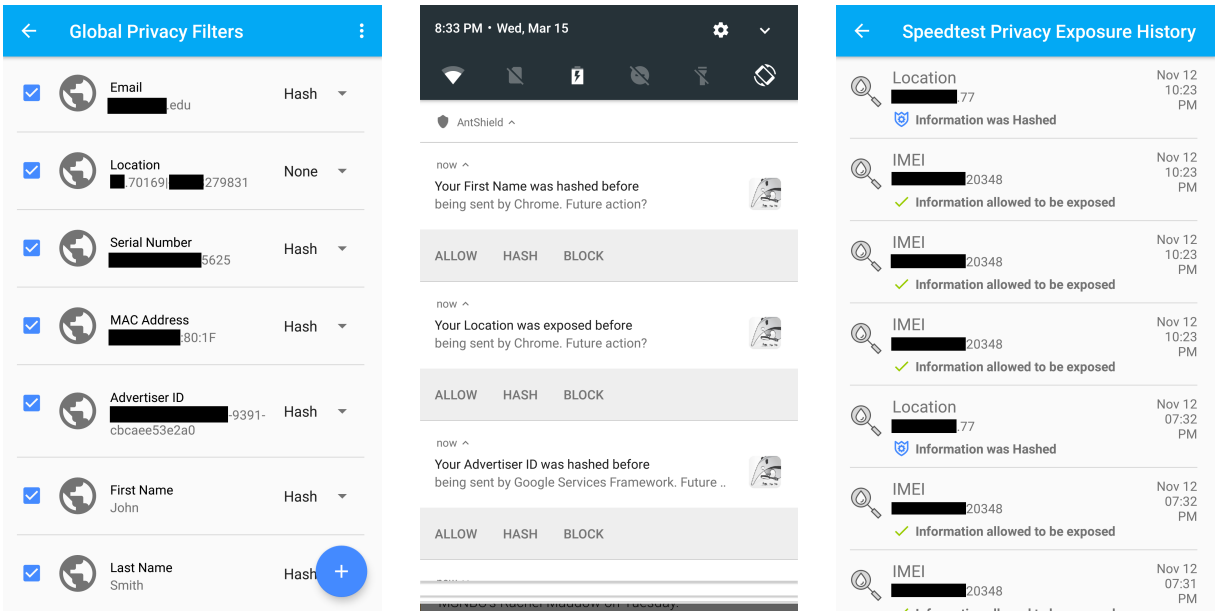
GUI. The GUI provides researchers and users with a multitude of options. Fig. 4.2(a) shows the main menu of the AntShield app, from which users can explore the app and can turn the packet interception on and off. Users can also choose which apps' packet traces to log – see Fig. 4.2(b). This can be useful for researchers who may only be interested in collecting data of one app at a time. In the case of a user study, users may choose to not share data of sensitive apps, such as banking apps. AntShield also takes a step towards data transparency: Fig. 4.2(c) showcases a visualization of which apps talk to which remote servers. The visualization can either show real-time TCP connections, or a history of which apps have exposed PII to which servers. AntShield also provides users with the ability to specify which *predefined* PII to monitor, as shown in Fig. 4.3(a). By default, these include PII available to AntShield through Android APIs. Users that trust



(a) (b) (c)

Figure 4.2: Screenshots of the `AntShield` Android app for general use cases: (a) Main menu from which users can explore the app and turn the packet interception on and off; (b) Users can select which apps' packet traces to log; (c) Visualization – can show real-time TCP connections between apps and servers or a summary of PII exposed.

`AntShield` can also opt-in and predefine additional PII, such as name and gender or any string (*e.g.* digits of a credit card). When a PII is exposed, `AntShield`'s GUI notifies the user in real-time – see Fig. 4.3(b). From here, users can decide to allow the exposure to happen, replace the *predefined* PII with a random string of the same length (so as not to alter the payload size), or block the packet completely. Whatever action the user selects, it is remembered for future occurrences of the same PII/app combination. As shown in Fig. 4.3(c), users can view a history of exposures at any time, and they can alter the action taken for specific PII/app combinations. In the future, these actions can be pre-filled for the user as recommendations based on research studies.



(a) (b) (c)

Figure 4.3: Screenshots of the `AntShield` Android app that are specific to privacy: (a) *predefined* PII, possible actions, and custom filters (name); (b) Privacy Exposure Notification; (c) History of Exposures.

4.2.3 PII Detection Methodology

At the heart of `AntShield` lies the online inspection of network packets to detect if they contain PII. As described next, we utilize three novel ideas to improve on existing PII detection approaches.

4.2.3.1 Hybrid Classification Methodology

We use a *hybrid* `String Matching`-classification methodology. As described in Sec. 4.2.1, a key insight is that PII can be split into two categories: *predefined* and *unknown*, depending on whether they are known a priori or not. This is an inherent advantage of operating on the device: `AntShield` has access to all the *predefined* strings and can use DPI to search for them; we refer to this method as `String Matching`. This not only gives us 100% accuracy on finding *predefined* exposures, if they are not obfuscated, but also reduces the set of PII that classifiers must learn, thereby improving

the accuracy of finding *unknown* exposures and reducing variance (see Sec. 4.4).

4.2.3.2 PII: A Multi-Label Problem

We treat PII detection as a `Multi-Label` problem, since a packet may contain zero, one, or multiple PII. Our classifiers decide, in one step, if any PII are contained in a packet, and if so - what type. More specifically, we use Mulan [55] to perform multi-label classification using the Binary Relevance (BR) transformation method [56]. The idea is to train a separate binary classifier for each label. Since the C4.5 DTs worked well for classifying exposure vs. non-exposure, we use them as independent classifiers in BR. For feature extraction, we follow the same methodology as in `ReCon` (Sec. 2.3.3.1), but instead of randomizing PII values during training, we completely exclude them from the set of features. This is possible because we replace all PII values with specific strings. For example, each occurrence of the IMEI number is replaced by "AntShield_IMEI" during the off-line data collection. Thus, all values starting with "AntShield_" can be excluded from the feature set.

4.2.3.3 Per-App Classifiers

Instead of building a separate classifier for each second level domain (SLD), we build classifiers *per-app*. This is possible thanks to `AntShield` running on the device: it can accurately map a packet to the app that generated it. From a classification point of view, per-app classifiers perform similarly to per-domain classifiers, as shown in Sec. 4.4. However, per-app classifiers have important system advantages.

First, they allow for easy setup and scalability. Consider the scenario when a new user wants to sign up for our system – she will need to download our classifiers. In order to reduce impact on RAM and disk space usage of her mobile device, she will want to download the minimal amount. Since it is impossible to know in advance which SLDs will be contacted, the user will need to

download classifiers belonging to all possible SLDs, of which there are millions. As of February 25th, 2019, the `.com` TLD (top level domain) alone consists of over 140 million SLDs [57]. In contrast, according to a 2017 report by AppAnnie [58], U.S. users have less than 100 apps installed on their phones, on average. The numbers are similar for other countries examined in the report. Even if we could predict which SLDs apps will contact, the number of SLDs contacted will usually be higher than the number of apps installed on a user's phone. For example, in our dataset of 307 apps, 699 SLDs were contacted. With per-app classifiers, a new user to our system would need to download less than 100 classifiers. Furthermore, when a user installs a new app, the appropriate classifier can be downloaded, without the need to guess which SLDs the new app will contact.

Second, they apply to all TCP and UDP traffic, not just to HTTP(S) traffic. `ReCon` parsed HTTP(S) packets to extract the hostname and then used a Public Suffix List (*e.g.* [59]) to get the corresponding domain and apply the relevant classifier. This process is not only costly in terms of CPU and memory, but is also not applicable to non-HTTP(S) packets, such as plain TCP and UDP packets that do not contain a `Host` field. One possible solution is to do reverse-DNS lookup to map (not only HTTP(S)) packets to their intended hosts. However, many companies opt-in to use third-party web service providers (such as Amazon AWS), and for them, reverse-DNS returns host names that are not very useful (*e.g.* `ec2-54-164-159-29.compute1.amazonaws.com`). As a work-around, it may be possible to implement a reverse-DNS cache on the device by keeping track of all the DNS requests. Unfortunately, we have seen many cases where the same IP maps to multiple host names (again, due to third-party web service usage).

One problem with both per-app and per-domain classifiers is that sometimes there is not enough training data. For apps that do not contain enough training samples, we train and use a *general classifier*, which is trained on all apps' data.

4.2.4 Real-time Implementation on the Mobile Device

The classifiers described in the previous section have value on their own right. However, it is highly non-trivial to apply them in real-time on a mobile device, with limited CPU and RAM. `AntShield` is the first system to achieve this goal thanks to the following system optimizations.

4.2.4.1 Real-Time Prediction

Our hybrid approach relies on `String Matching` to search for *predefined* exposures and on classification methodology to detect *unknown* ones. The former benefits from the good performance of `AntMonitor Library`'s efficient DPI module. The latter needs to parse packets to extract words that are used as features of the classifiers. With off-the-shelf `ReCon`, to extract words from a packet, several invocations of Java string parsing methods would be required, which are extremely slow on a mobile device. We were able to extract features from the traffic while completely avoiding parsing by exploiting the following observation: we only need to find the features that appear in our resultant decision trees. Most of our decision trees are one-level deep and only a third of the trees have a depth greater than two. Only the general classifier requires 500 or more features. We propose using DPI to search for the features that appear in the classifiers. Since the Aho-Corasick algorithm used in the `AntMonitor Library` can search for many strings in one pass of the packet, having these extra words to search for does not affect performance.

Extracting words that appear in the decision tree nodes and using DPI to search for them works well in most cases. However, in some cases the words are too small and can actually be part of a longer word. In this case, our DPI search would mark a feature as existent, when in fact it's part of a different word, causing an incorrect prediction. As an example, *hulu* was receiving the word '`profile`' in the packets that also contained the user's first name. However, many packets that did not contain any exposures, contained the word '`video_profile`.' To avoid these DPI-based false positives, we decided to keep the delimiters surrounding each word during feature selection. So,

in the case of *hulu*, we used '`/profile?`' as the feature. This trick allowed us to extract the same words with DPI as with Java parsing.

Evaluation in Sec. 4.5 shows that our methodology makes a crucial difference for running in real-time. Java parsing often takes over 30 millisecond, while the Aho-Corasick search stays below one millisecond. 30 milliseconds is too long of a delay for network packets, as certain TCP connections have lower timeout values. Based on our experiments in Sec. 4.5, the classification itself takes about 1 millisecond, which is an acceptable overhead for real-time performance. In `ReCon`'s methodology there is also the extra step of PII extraction, which also requires Java parsing. Since this steps occurs only for packets that are classified as containing PII, this delay may be tolerable. However, with the `Multi-Label` method, we avoid this second step and completely avoid Java parsing for all packets.

4.2.4.2 Minimizing Classifiers to Load in RAM

With limited RAM, care must be taken when loading machine learning models from disk to memory. To minimize the impact on RAM, we: (i) load per-app models only for those apps that are installed on the device; and (ii) perform a two-step training method to reduce the general classifier feature set (see Fig. 4.1). Specifically, the general classifier has a feature set size of over 12k, and during prediction needs the allocation of a double array with size 12k+. While this is a small size for a server, on the mobile device it causes major issues. Specifically, if one were to load the full general classifier, most web pages and applications would not load. This is because each time a packet that does not belong to a per-app classifier, it has to be predicted by the general one, and the memory allocation becomes so large that a *blocking garbage collection call* has to be executed by the Android OS after every prediction. This blocked our main networking thread, caused connections to time-out, and prevented pages from loading. We were able to reduce the feature set by exploiting the existing classifier tree: we re-trained the general classifier using only the words that appear in the tree nodes as features. This resulted in a feature set size of only 509, *i.e.* a 24x

	ReCon dataset(s)		**AntShield dataset(s)**	
	Auto	**Manual**	**Auto**	**Manual**
# of Apps	564	91	414	149
# of packets	16761	13079	21887	25189
# of destination domains	450	368	597	379
# of exposures detected	1566	1755	4760	3819
# of *unknown* exposures	4	78	483	516
# of exposures in encrypted traffic	-	-	1513	1526
# of packets with **multiple exposures**	50	224	1506	790
# of background exposures	-	-	2289	639
# of HTTP packets	16761	13079	13694	13648
# of HTTPS packets	-	-	6830	8103
# of TCP packets	-	-	867	2264
# of exposures in TCP (other ports)	-	-	38	7
# of UDP packets	-	-	496	1174
# of exposures in UDP	-	-	17	12

Table 4.1: Summary of Datasets. ReCon is the previous state-of-the-art, collected in the middle of the network [2]. AntShield's Manual and Automated datasets were collected on the device.

reduction for the general classifier, which in itself allowed AntShield to run in real-time. Overall, AntShield's memory usage is around 100 MB, which is acceptable: many popular apps, *e.g.* *Facebook*, use as much as 200 MB RAM.

4.3 The AntShield Datasets

In order to evaluate the effectiveness of our methodology in detecting private information exposure, we collected and analyzed two AntShield datasets. We logged all packets generated by different apps on a test device (Nexus 6) and used the JSON format to save any relevant fields, such as destination IP address and port number, HTTP method, HTTP headers, and *etc.* Each packet was also annotated with any PII exposures (see 4.2.1 for a list) that it contained. We collected two different datasets, depending on how we interacted with apps, described next.

Manual Testing. First, in order to assess PII exposures during typical user behavior, we tested 100 most popular and free Android apps, based on rankings in *AppAnnie* [60]. Since `AntShield` can map each packet to the app that generated it, we tested apps in batches to catch packets sent by apps when they are in the foreground and in the background. Specifically, we installed five apps on the test device and used `AntShield` to intercept and log packets while interacting with each app for five minutes. This way, while another app is being tested, the other applications on the phone can send background traffic. After all applications in the batch were tested, we switched off the screen on our test device and waited for five minutes for any additional background traffic of the app that was tested last. Finally, we uninstalled each application in the order they were installed and turned off `AntShield`.

Automatic Testing. We also used the *UI/Application Exerciser Monkey* [61] to automatically interact with apps. This does not capture typical user behavior but enables extensive and stress testing of more apps. We installed four batches of 100 applications each, and had *Monkey* perform 1,000 random actions in each tested app while `AntShield` logged the generated traffic. Due to random actions of Monkey, we skipped apps that can make a payments or require a real phone number. For apps that required a login, we first logged in manually (with test credentials) and then ran *Monkey*. At the end of each batch, we switched off the screen of the test device and waited for ten minutes to capture background traffic.

Advantages. Using `AntShield` to capture packets on the device has several advantages compared to previous datasets collected in the middle of the network. First, `AntMonitor` provides accurate packet-to-app mapping. This obviates the need to infer the application through ML or heuristics and reduces restrictions in testing - multiple apps can be installed at once. Second, we are able to collect contextual information on the phone, such as which app is in the foreground. Such information can provide insights into the mobile ecosystem and, in the future, it can be used as a feature in machine learning. Third, we gained insight into TLS, UDP, and regular TCP traffic, in addition to HTTP. Finally, scrubbing PII and labeling packets with the type of PII they expose

was fully automated: AntShield already provides *predefined* strings, and we entered the *unknown* strings (*e.g.* fake test account credentials) as custom filters (as in Fig. 4.3(a)).

Summary. For the purposes of training and testing classifiers, we merged the Automatic and Manual datasets into one, referred to as the AntShield dataset. Our dataset is summarized in Table 4.1, next to the prior state-of-the-art PII datasets collected by ReCon [2]. The AntShield dataset contains more and richer information about exposures than before. We note that some advantages are inherent to running on the device (*i.e.* the ability to capture contextual information, including the app names). Other differences are due to changes in app versions and exposure behavior over time. Therefore, in addition to being used to evaluate our methodology (Section 4.4), our datasets have value on their own and we have made them available to the community at [54].

4.4 Classification Evaluation

4.4.1 Setup

Classification Schemes under Comparison. In this section, we use our datasets to compare the classification accuracy of the proposed AntShield approach (Sec. 4.2.3) to the previous state-of-the-art ReCon approach (Sec. 2.3.3.1). Since our proposed method combines several ideas, we also report results from the evaluation of individual ideas, to help assess which idea brings the most benefit:

1. Complete ReCon approach as per Sec. 2.3.3.1: classify all (*predefined* and *unknown*) exposures, using binary classifiers first to detect exposure, then heuristics to determine the type of PII.[1]

2. ReCon classifying *unknown* exposures only.

[1]We use the ReCon code available here: https://github.com/Eyasics/recon

3. `String Matching` on *predefined* exposures, `ReCon` trained on *unknown*; testing done on all exposures.

4. `Multi-Label` classification trained and tested on *predefined* and *unknown* exposures.

5. `Multi-Label` classification trained and tested only on *unknown*.

6. Complete `AntShield` as per Section 4.2.3: `String Matching` for *predefined* and `Multi-Label` classification for *unknown* exposures; `Multi-Label` trained on *unknown* only, testing done on all exposures.

Per-app vs. Per-domain classifiers. In Section 4.2.3, we discussed the system advantages of using per-app instead of per-domain classifiers. In this section, we show that their classification performance is similar. For each method, we compare how well the per-domain, per-app, and general classifiers perform. We train specialized classifiers for those domains and apps that contain at least one positive sample (packet with an exposure), and one negative sample (packet with no exposure). In that sense, we find that per-app classifiers are able to cover more data than the per-domain classifiers. In particular, we obtain the following numbers for packets "covered" by a classifier:

- All PII, per-app classifiers: 211 (93.3% of traffic, 99.5% of packets with PII)
- All PII, per-domain classifiers: 182 (63.6% of traffic, 95.0% of packets with PII)
- *unknown* PII, per-app classifiers: 47 (54.4% of traffic, 99.5% of packets with *unknown* PII)
- *unknown* PII, per-domain classifiers: 49 (24.5% of traffic, 87.4% of packets with *unknown* PII)

This is expected since apps generally exhibit more diverse behavior by connecting to various domains, some of which collect PII and some of which do not. Thus, we are more likely to find apps

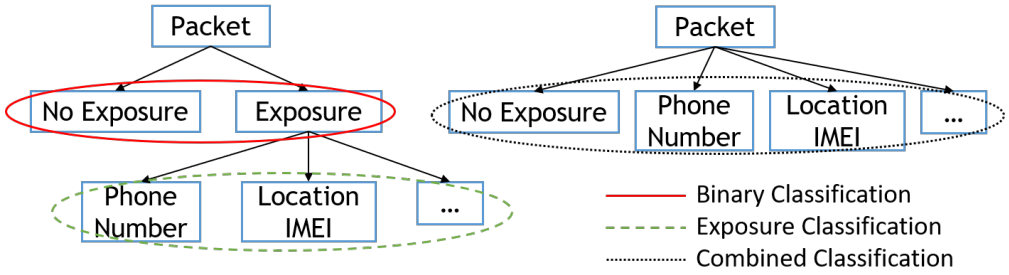


Figure 4.4: Evaluation approaches: (1) Binary Classification: we assess how well we identify whether or not a packet contains a PII (Sec. 4.4.2); (2) Exposure Classification: we assess how well we infer the PII type from packets that already contain a PII, ignoring packets without PII (Sec. 4.4.3); (3) Combined Classification - assess how well we identify the PII type *and* the No Exposure label, considering all packets (Sec. 4.4.4).

that have sent at least one packet containing PII and one packet without PII, as opposed to domains that receive packets with and without PII.

Evaluation Approaches and Metrics. After classifying a packet, either an exposure is detected with a particular PII type, or No Exposure is detected. Depending on how one summarizes these numbers over all packets classified, we may have different assessments. In particular, whether or not we consider packets that do not contain PII, affects the numbers, since this is the majority of the packets. We considered three evaluation schemes, summarized in Fig. 4.4:

1. *Binary Classification*: this approach evaluates how well the applicable algorithms classify a packet as containing an exposure or not (Sec. 4.4.2).

2. *Exposure Classification*: this approach evaluates how well each algorithm distinguishes PII types in packets that contain an exposure (Sec. 4.4.3), *i.e.* packets without a PII are not taken into account.

3. *Combined Classification*: this approach evaluates how well each algorithms distinguishes among PII types and "no exposure" (Sec. 4.4.4), *i.e.* packets without a PII are taken into account.

For each approach, we perform five-fold cross-validation on the given model (unless otherwise

specified), and calculate the average and the standard deviation across the trained specialized classifiers. Since ReCon's second (non-binary) step and String Matching are both heuristic, we did not perform cross-validation on these methods when evaluating exposure and combined classification, but simply ran the algorithms on the entire applicable dataset (columns one through three and six) in the Tables.

Because a packet can expose more than one PII type, for the latter two approaches, we use evaluation metrics specific to multi-labeling problems [62]. We report precision, recall, and F-score using their standard definitions:

- *Precision*: the number of correct labels, divided by the number of predicted labels.
- *Recall*: the number of correct labels, divided by the number of true labels.
- *F-score*: two times the product of precision and recall, divided by the sum of precision and recall.

4.4.2 Binary Classification

We report the binary classification results in Table 4.2 for the two machine learning algorithms under consideration: ReCon's DT, and our Multi-Label BR. We report the standard metrics for binary classification: F-score, specificity, and recall. The first column is consistent with ReCon's own reports in [2] - the model achieves high F-scores and low false positives/negatives. The second column shows that there is little benefit in focusing on *unknown* exposures only. This makes sense, since in this binary step, we only want to see whether or not a packet contains PII, and not to extract what type of PII it is (see Fig. 4.4). The third and fourth columns also show little benefit from our Multi-Label approach since within the BR, we still use a similar decision tree to classify exposure vs. non-exposure. We also note that the standard deviation is higher when focusing on *unknown* exposures only (columns two and four). This is expected since there is now less data to

		Method			
		ReCon on All PII	ReCon on *unknown*	Multi-Label on All PII	Multi-Label on *unknown*
Per-Domain Average	F-score	98.0% ± 6.80	97.2% ± 14.2	97.1% ± 9.32	97.2% ± 11.3
	Specificity	97.5% ± 8.10	98.5% ± 4.35	98.4 ± 5.75	98.9% ± 5.72
	Recall	98.5% ± 6.78	97.9% ± 14.1	97.1% ± 9.38	97.3% ± 9.46
Per-App Average	F-score	97.0% ± 7.99	96.4% ± 14.9	96.2% ± 7.25	96.4% ± 12.0
	Specificity	98.1% ± 4.24	96.8% ± 11.3	96.4% ± 7.30	98.3 ± 8.87
	Recall	96.4% ± 8.95	97.6% ± 14.5	97.4% ± 6.48	95.9% ± 12.1
General	F-score	97.5%	94.9%	95.5%	99.5%
	Specificity	98.9%	99.8%	95.6%	91.8%
	Recall	95.8%	91.9%	98.4%	99.8%

Table 4.2: Binary Classification Results (Sec. 4.4.2)

work with and some domains send *unknown* PII only once in a while. Furthermore, in the case of binary classification, the general classifiers perform close to the specialized ones. However, we are interested in improving the accuracy on the type of PII classification, and as we show in the next two subsections, our approaches and the specialized classifiers bring benefit there.

4.4.3 Exposure Classification

The results are shown in Table 4.3. First, standard deviation is high because certain domains are easy to learn and get near 100%, while a small set of domains are difficult (some even have 0% F-score). `ReCon`'s heuristic scores low when attempting to extract the PII type (column one); see Sec. 2.3.3.1 and [2] for a description of the heuristic. Second, when we reduce the set of PII types to look for (column two), the heuristic performs slightly worse, probably due to not having enough samples of *unknown* exposures. Third, as expected, `String Matching` can find *predefined* exposures with 100% accuracy, thus the overall F-score improves by ~20% (column three vs. column one), and standard deviation decreases. Fourth, the `Multi-Label` approach shows significant improvement when compared to `ReCon`'s heuristic (column four vs. column one, and column five vs. column two); this is expected, since we do not need to estimate probabilities or calculate out thresholds. Fifth, the complete `AntShield` achieves near perfect performance, and

		(1) ReCon on All PII	(2) ReCon on *unknown*	(3) String Matching & ReCon on *unknown*	(4) Multi-Label on All PII	(5) Multi-Label on *unknown*	(6) String Matching & Multi-Label
Per-Domain Average	F-score	**72.7% ± 39.7**	69.5% ± 45.5	95.9% ± 18.4	99.2% ± 1.90	99.3% ± 2.88	**98.5% ± 11.0**
	Precision	**74.8% ± 39.3**	69.5% ± 45.5	96.2% ± 18.1	99.3% ± 1.95	99.3% ± 3.21	**98.5% ± 11.0**
	Recall	**73.5% ± 39.6**	69.5% ± 45.5	95.9% ± 18.4	99.3% ± 1.79	99.5% ± 2.11	**98.9% ± 10.4**
Per-App Average	F-score	**73.2% ± 31.1**	69.0% ± 42.7	97.6% ± 13.1	98.8% ± 2.24	98.9% ± 3.23	**99.4% ± 4.58**
	Precision	**76.7% ± 30.4**	69.0% ± 42.7	98.0% ± 12.8	98.9% ± 2.20	99.0% ± 3.29	**99.4% ± 4.58**
	Recall	**73.5% ± 31.0**	69.1% ± 42.8	97.6% ± 13.1	98.9% ± 2.18	99.1% ± 2.40	**100% ± 0.06**
General	F-score	**49.9%**	50.2%	97.1%	77.4%	81.8%	**99.3%**
	Precision	**58.2%**	50.3%	97.6%	79.6%	84.7%	**99.5%**
	Recall	**53.3%**	50.3%	97.1%	75.9%	79.4%	**99.7%**

Table 4.3: Exposure Classification Results (Sec. 4.4.3)

decreases the standard deviation (column six vs. columns one through three). Finally, in all cases: (i) the specialized classifiers outperform the general ones, and (ii) the per-app classifiers achieve higher F-scores and lower standard deviation in our final method (column six).

4.4.4 Combined Classification

The results for combined classification are shown in Table 4.4 and the difference between the performance of different classification methods is less pronounced than before. This is because the majority of packets do not contain PII, the binary classifiers work well (see Sec. 4.4.2) and classify the ”no exposure” packets correctly, making the results look deceivingly good. This is why we also report the Exposure Classification performance (Sec. 4.4.3), as it provides deeper insight into the classifiers' performance. We note that in this case, the `Multi-Label` general classifiers appears to do worse than the `ReCon` ones; but this is because the results reported for columns four and five are based on cross-validation, so the general classifiers don't get to see all the training data and do worse on some folds.

		Method					
		(1) ReCon on All PII	(2) ReCon on *unknown*	(3) String Matching & ReCon on *unknown*	(4) Multi-Label on All PII	(5) Multi-Label on *unknown*	(6) String Matching & Multi-Label
Per-Domain Average	F-score	**89.1% ± 22.1**	91.8% ± 17.7	98.5% ± 7.81	99.2% ± 2.02	99.3% ± 2.54	**99.5% ± 3.99**
	Precision	**90.0% ± 21.5**	91.8% ± 17.7	98.7% ± 7.55	99.2% ± 2.10	99.3% ± 2.82	**99.5% ± 3.99**
	Recall	**89.2% ± 22.0**	91.8% ± 17.7	98.5% ± 7.80	99.2% ± 1.83	99.5% ± 1.87	**99.8% ± 1.60**
Per-App Average	F-score	**91.3% ± 15.2**	95.0% ± 12.6	99.5% ± 3.09	98.7% ± 2.31	98.9% ± 2.83	**99.1% ± 7.35**
	Precision	**92.7% ± 14.1**	95.0% ± 12.6	99.5% ± 2.96	98.7% ± 2.24	99.0% ± 2.89	**99.1% ± 7.35**
	Recall	**91.4% ± 15.2**	95.0% ± 12.6	99.5% ± 3.06	98.7% ± 2.26	99.1% ± 2.11	**99.4% ± 6.91**
General	F-score	**89.9%**	99.1%	99.3%	78.5%	76.5%	**99.8%**
	Precision	**91.3%**	99.1%	99.4%	80.6%	79.1%	**99.8%**
	Recall	**90.4%**	99.1%	99.4%	77.0%	74.4%	**99.9%**

Table 4.4: Combined Classification Results (Sec. 4.4.4)

4.5 Prediction and Training Time

In order to run privacy exposure detection in real-time on the device, performance is key. Thus, we evaluate the two feature extraction approaches: (1) `ReCon`'s Java string parsing, and (2) `AntMonitor Library`'s Aho-Corasick search for features and *predefined* PII. We also compare: (1) `ReCon`'s binary classification, and (2) `AntShield`'s `Multi-Label` classification. We find that our classifiers have negligible impact on battery and can run in real-time. This is mainly thanks to the use of (i) Aho-Corasik for searching for multiple strings, and (ii) the lean extraction of words to feed into the classifiers. To the best of our knowledge, this is the first time that PII classification is achieved in real-time on a mobile device.

Setup. The tests were ran on a Nexus 6P with Android 7.1.1 and an 8-core QUALCOMM Snapdragon 810 processor with a clock speed of 2 GHz and battery capacity of 3450 mAh. We fed 10 HTTP packets of varying sizes (between 300-2000B) to each function under evaluation and timed how long it took using `System.nanoTime()`. We repeated each test case 100 times and calculated the average run-time and standard deviation. Each function was tested in isolation, running on the main thread, so as to minimize timing the overhead of possible thread switching.

Results. The results for the feature extraction approaches are as follows: (1) `ReCon`'s Java string parsing: 36 ms $\pm$ 17 ms; (2) Aho-Corasick search: 0.107 ms $\pm$ 0.149 ms. Clearly, `AntMonitor`

Library's efficient Aho-Corasick implementation brings orders of magnitude of benefit.

The results for classification techniques are: (1) ReCon's binary classification: 0.041 ms $\pm$ 0.029 ms; (2) Multi-Label classification: 0.751 ms $\pm$ 1.35 ms. As expected, the Multi-Label classification takes a little longer, but it is still reasonable and will not significantly impact user experience.

Training Time. Training Multi-Label classifiers generally takes twice as long as ReCon's binary classifiers. However, in both cases, a specialized classifier is trained within tens of *milliseconds* even when done on a standard Windows 10 laptop. General classifiers can take up to tens of minutes. A binary general classifier takes about one hour to train. Since the Multi-Label classifier needs to train a separate decision tree for each label, this training time scales linearly with the number of labels. In a naive implementation of BR, this would result in over 10 hours of training time. We exploited the fact that BR is easy to parallelize to reduce training time (we modified the Mulan library to train multiple trees at the same time) and we were able to produce a general classifier in about 90 minutes. When considering only *unknown* exposures, the number of labels is reduced, and both the binary and the Multi-Label general classifiers take *under 10 minutes* to train. However, since training is performed infrequently, and can be done at a remote server (the classifiers can be fetched later by user devices), we consider the training times a non-issue.

4.6 Summary

We presented AntShield - a system that performs, for the first time, on-device detection of *predefined* PII and classification of *unknown* PII, accurately and with low overhead. Rather than using binary classification and heuristics to determine PII types contained within packets, we explored the usage of Multi-Label classification and found that it outperforms prior art in terms of F-scores and variance. We have also shown that our approach can run in real-time on the device, detecting PII within milliseconds. Our code and dataset are available to the community at [10].

Chapter 5

NoMoAds: Detecting and Blocking Ads

5.1 Overview

Online advertising supports millions of free applications (apps) in the mobile ecosystem. Mobile app developers are able to generate revenue through ads that are served via third-party ad libraries such as AdMob and MoPub [1]. Unfortunately, the mobile advertising ecosystem is rife with different types of abuses. First, many mobile apps show intrusive ads that annoy users due to the limited mobile screen size [63]. Second, mobile ads consume significant energy and data resources [64]. Third, third-party mobile ad libraries have been reported to leak private information without explicit permission from app developers or users [65]. Finally, there have been reports of malware spreading through advertising in mobile apps [66]. Due to the aforementioned usability, performance, privacy, and security abuses, it is often desirable to detect and block ads on mobile devices.

Mobile ad-blocking apps such as Adblock Browser by Eyeo GmbH [67] and UC Browser by Alibaba [68] are used by millions of users. There are two key limitations of existing ad-blocking apps. First, most ad-blockers rely on manually curated filter lists (or blacklists) to block ads. For

example, EasyList [3] is an informally crowdsourced filter list that is used to block ads on desktop browsers. Unfortunately, these filter lists do not perform well in the app-based mobile ecosystem because they are intended for a very different desktop-based web browsing ecosystem. Second, most of the existing mobile ad-blocking apps are meant to replace mobile web browsers and can only block ads inside the browser app itself. Specifically, these browser apps cannot block ads across all apps because mobile operating systems use sandboxing to isolate apps and prevent them from reading or modifying each other's data.

In this chapter, we propose NoMoAds to effectively and efficiently detect and block ads across all apps on mobile devices while operating in user-space (without requiring root access). We make two contributions to address the aforementioned challenges. First, to achieve cross-app mobile ad-blocking, we inspect the network traffic leaving the mobile device. Our design choice of intercepting packets at the network layer provides a universal vantage point into traffic coming from all mobile apps. Our packet interception implementation is optimized to achieve real-time filtering of packets on the mobile device. Second, we train machine learning classifiers to detect ad-requesting packets based on automatically extracted features from packet headers and/or payload. Our machine learning approach has several advantages over manual filter list curation. It automates the creation and maintenance of filtering rules, and thus can gracefully adapt to evolving ad traffic characteristics. Moreover, it shortens the list of features and rules, making them more explanatory and expressive than the regular expressions that are used by popular blacklists to match against URLs.

Our prototype implementation of NoMoAds can run on Android versions 5.0 and above. We evaluate the effectiveness of NoMoAds on a dataset labeled using EasyList and manually created rules that target mobile ads. The results show that EasyList misses more than one-third of mobile ads in our dataset, which NoMoAds successfully detects. We evaluate different feature sets on our dataset and provide insights into their usefulness for mobile ad detection. In particular, network-layer features alone achieve 87.6% F-score, adding URL features achieves 93.7% F-score, adding

other header features achieves 96.3% F-score, and finally, adding personally identifiable information (PII) labels and application names achieves up to 97.8% F-score. Furthermore, when tested on applications not included in the training data, NoMoAds achieves more than 80% F-score for 70% of the tested apps. We also evaluate the efficiency of NoMoAds operating in real-time on the mobile device and find that NoMoAds can classify a packet within three milliseconds on average. To encourage reproducibility and future work, we make our code and dataset publicly available at [11].

5.2 Background

Deployment of ad-blockers has been steadily increasing for the last several years due to their usability, performance, privacy, and security benefits. According to PageFair [69], 615 million desktop and mobile devices globally use ad-blockers. While ad-blocking was initially aimed at desktop devices mainly as browser extensions such as AdBlock, Adblock Plus, and uBlock Origin, there has been a surge in mobile ad-blocking since 2015 [70]. Mobile browsing apps such as UC Browser and Adblock Browser are used by millions of iOS and Android users, particularly in the Asia-Pacific region due to partnerships with device manufacturers and telecommunication companies [70]. Moreover, Apple itself began offering ad-blocking features within their Safari browser since iOS9 [71]. As we discuss next, mobile ad-blocking is fundamentally more challenging as compared to desktop ad-blocking.

5.2.1 Challenges

Cross-App Ad-Blocking. It is challenging to block ads across all apps on a mobile device. Mobile operating systems, including Android and iOS, use sandboxing to isolate apps and prevent them from reading or modifying each other's data. Thus, ad-blocking apps like UC Browser or Adblock

Browser can only block ads inside their own browser unless the device is rooted. Specifically, Adblock has an Android app for blocking ads across all apps, but it can work only on rooted devices, or it has to be setup as a proxy to filter Wi-Fi traffic only [72]. Neither of these options are suitable for an average user who may not wish to root their device and may not know how to setup a proxy. A recent survey of ad-blocking apps on the Google Play Store found that 86% of the apps only block ads inside their browser app [73]. Recent work on leveraging VPNs for mobile traffic monitoring has considered interception in the middle of the network (*e.g.*`ReCon` [2]) as well as directly on the mobile device (*e.g.*`Haystack` [53]), primarily for the purpose of detecting privacy leaks and only secondarily for ad-blocking [64, 34]. In this Chapter, we build on top of the `AntMonitor Library` presented in Chapter 3 to efficiently implement a cross-app mobile ad-blocker.

Cross-app ad-blocking is not only technically challenging but is also considered a violation of the Terms of Service (ToS) of the official Apple and Android app stores [74]. However, there are still ways to install cross-app ad-blocking apps without rooting or jailbreaking a mobile device (*e.g.* through a third-party app store). Legally speaking, ad-blockers have withstood legal challenges in multiple European court cases [75]: acting on users' behalf with explicit opt-in consent, ad-blockers have the right to control what is downloaded. We are unaware of any successful challenges against ad-blockers under the Computer Fraud and Abuse Act (CFAA) in the U.S.

Avoiding Blacklists. Desktop ad-blockers rely on manually curated filter lists consisting of regular expressions such as the ones depicted in Tables 5.2 and 5.3. Unfortunately these lists are not tailored to the app-based mobile ecosystem, and hence we cannot simply reuse them to effectively block mobile ads. We either have to replicate the crowdsourcing effort for the mobile ecosystem or design approaches to automatically generate blacklist rules to block mobile ads.

Ad-blocking apps on the Google Play Store also rely on blacklists to block ads [73, 35]. More than half of these apps rely on publicly-maintained lists such as EasyList and some rely on customized

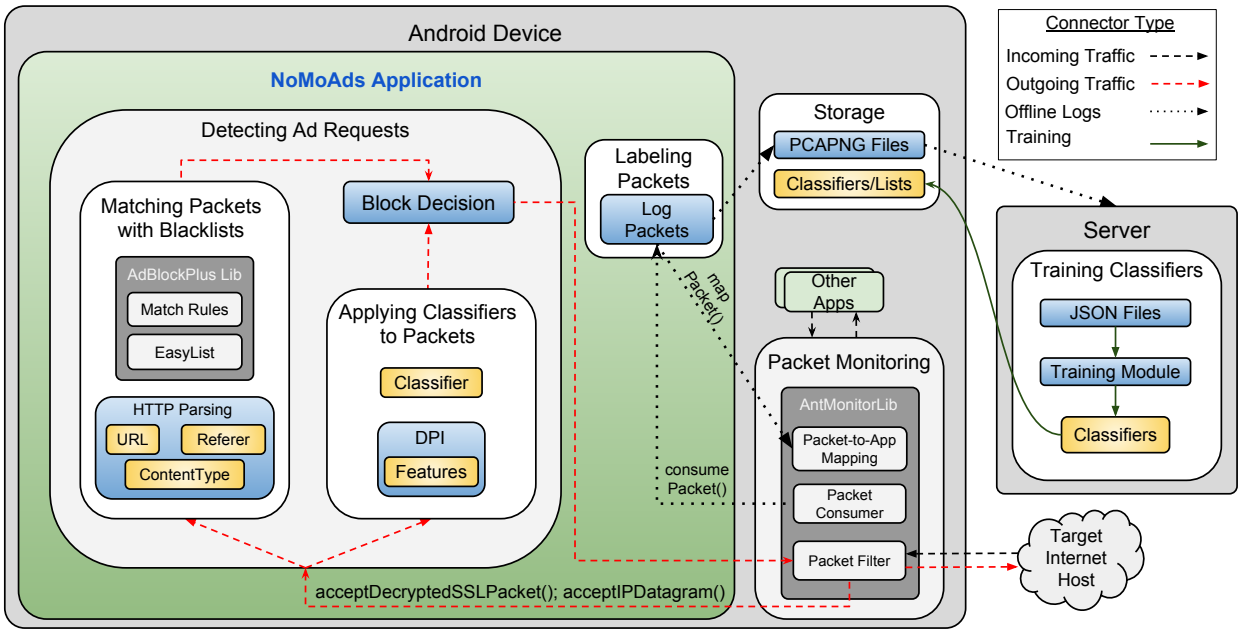


Figure 5.1: The NoMoAds system: it consists of the NoMoAds application on the device and a remote server used to (re)train classifiers. The app uses the `AntMonitor Library` to intercept, inspect, and save captured packets. All outgoing packets are analyzed for ads either by the AdblockPlus Library or by our classifiers. The former is used to label our ground truth dataset, as well, as a baseline for comparison, and the latter is the proposed approach of this Chapter.

filter lists. In addition, cross-app ad-blockers that are not allowed on the Google Play Store, such as DNS66 [31] and Disconnect [76], also rely on both public and customized blacklists. Unfortunately, these blacklists are manually curated, which is a laborious and error-prone process. They are also slow to update and do not keep up with the rapidly evolving mobile advertising ecosystem [77]. Furthermore, they contain obsolete filter rules that are redundant, which results in undesirable performance overheads on mobile devices.

5.3 The NoMoAds Approach

Fig. 5.1 provides an overview of our cross-app mobile ad-blocking system. It consists of user-space software NoMoAds and a server used for training classifiers. The NoMoAds app intercepts

every packet on the device and inspects it for ad requests, extracting features and passing them on to a classifier (Sec. 5.3.1). To obtain the ground truth, we match packets with blacklists (Sec. 5.3.2.1), log the labeled packets, and then upload them to the server for training (Sec. 5.3.3.1). While the detection of ad packets is done in real-time on the device itself, the selection of features and the training of classifiers is done offline at a server in longer time scales (Sec. 5.3.3.2).

5.3.1 Packet Monitoring

NoMoAds relies on the ability to intercept, analyze, and filter network traffic from all apps on a mobile device. To that end, NoMoAds leverages the APIs of the `AntMonitor Library` (Sec. 3.2.2), as described next.

Packet Interception. As shown in Fig. 5.1, we use the `acceptIPDatagram` and `acceptDecryptedSSLPacket` API calls provided by the `AntMonitor Library` to intercept unencrypted and successfully decrypted SSL/TLS packets, respectively. While the `AntMonitor Library` cannot decrypt SSL/TLS packets when certificate pinning is used, it can still analyze information from the TCP/IP packet headers.

Packet Analysis. Given a list of strings to search for, the `AntMonitor Library` can perform DPI within one millisecond (ms) per packet (see Chapter 3). When strings of interest are not known a priori, we can use `AntMonitor Library`'s visibility into the entire packet to parse and extract features from TCP/IP and HTTP/S headers and payload. For example, we can use IP address information in the IP header, port numbers and flag information in the TCP header, hostnames and query strings in the HTTP header, string signatures from the HTTP payload, and server name indication (SNI) from TLS extensions. In addition, the `AntMonitor Library` provides contextual information, such as which app is responsible for generating a given packet via the `mapPacket` API call.

Packet Filtering. For each packet, we can decide to block it (by returning `false` within one of the API calls) or to allow it (by returning `true`). By default, if our classifier returns a match, we block the packet and return an empty HTTP response back to the application that generated the ad request. It is critical to return feedback to the application, otherwise it triggers wasteful retransmissions that eat up the mobile device's scarce resources.

Leveraging the aforementioned packet interception, analysis, and filtering techniques, NoMoAds aims to detect and block packets that contain ad requests.

5.3.2 Detecting Ad Requests in Outgoing Packets

Ads are typically fetched from the network via HTTP/S requests. To detect them, we take the approach of inspecting every outgoing packet. Blocking requests for ads is consistent with the widely used practice of most ad-blockers. Note that ad-blockers also sometimes modify incoming ads (*e.g.* through CSS analysis) when it is impossible to cleanly block outgoing HTTP requests. The approach of outgoing HTTP request filtering is preferred because it treats the problem at its root. First, the ad request is never generated, which saves network bandwidth. Second, this approach prevents misleading the ad network into thinking that it served an ad, when it actually did not (this keeps attribution analytics and payments for ad placements honest and correct). Third, this approach circumvents the need to modify the rendered HTML content (*e.g.* CSS values).

The rest of this section compares two approaches for blocking ad requests: the traditional, blacklist-based approach (Sec. 5.3.2.1) and the proposed machine learning-based approach taken by NoMoAds (Sec. 5.3.2.2).

5.3.2.1 Blacklists

According to a recent survey, mobile ad-blocking apps on the Google Play Store rely on blacklists to block ads [73]. These blacklists (such as EasyList in AdblockPlus) capture the ad-blocking community's knowledge about characteristics of advertisements through informal crowdsourcing. However, blacklists suffer from the following limitations.

1. *Maintenance.* Blacklists are primarily created and maintained by humans domain-experts, often assisted by crowdsourcing. This is a tedious, time-consuming, and expensive process. Furthermore, as the characteristics of ad traffic change over time, some filer rules become obsolete and new filter rules need to be defined and added to the blacklist.

2. *Rules Compactness and Expressiveness.* Humans may not always come up with the most compact or explanatory filter rules. For example, they may come up with redundant rules, which could have been summarized by fewer rules. We faced this issue ourselves when coming up with our own set of rules tailored to mobile traffic (*e.g.* see rows 20 and 25 in Table 5.3). In addition, filter rules in today's blacklists are limited in their expressiveness: they are an "`OR`" or an "`AND`" of multiple rules. On the other hand, classifiers can come up with more complicated but intuitive rules, such as the decision tree depicted in Fig. 5.4.

3. *Size.* Blacklists can be quite lengthy. For instance, EasyList contains approximately 64K rules. This is a problem for implementations on the mobile device with limited CPU and memory resources.

4. *URL-focused Rules.* Most of today's blacklists were specifically created for browsers and web traffic, and they typically operate on the extracted URL and HTTP Referer header. As we show later, this is one of the reasons that these lists do not translate well when applied to mobile traffic. By exploiting `AntMonitor Library`'s visibility into the entire payload (beyond just URLs), we can leverage the information from all headers to more accurately detect ads in mobile traffic.

In this work, we used EasyList (the most popular publicly-maintained blacklist [73]) as (i) a baseline for comparison against our proposed learning approach – see Section 5.5.1, and for (ii) partially labeling packets as containing ads or not – see Section 5.4. In order to match packets against EasyList, we incorporated the open source AdblockPlus Library for Android [78] into NoMoAds, as shown in Fig. 5.1. The AdblockPlus Library takes as input the following parameters: URL, content type, and HTTP Referer. The content type is inferred from the requested file's extension type (*e.g.* `.js`, `.html`, `.jpg`) and is mapped into general categories (*e.g.* script, document, image). Relying on these parameters to detect ad requests restricts us to HTTP and to successfully decrypted HTTPS traffic. Hence, we first have to parse each TCP packet to see if it contains HTTP, and then extract the URL and HTTP Referer. Afterwards, we pass these parameters to the AdblockPlus Library, which does the matching with EasyList.

5.3.2.2 Classifiers

NoMoAds uses decision tree classifiers for detecting whether a packet contains an ad request. While feature selection and classifier training is conducted offline, the trained classifier is pushed to the NoMoAds application on the mobile device to match every outgoing packet in real-time. To extract features from a given packet and pass them to the classifier, one typically needs to invoke various Java string parsing methods and to match multiple regular expressions. Since these methods are extremely slow on a mobile device, we use the `AntMonitor Library`'s efficient DPI mechanism (approximately one millisecond per packet) to search each packet for features that appear in the decision tree. We pass any features found to the classifier, and based on the prediction result we can block (and send an empty response back) or allow the packet.

Classifiers vs. Blacklists. NoMoAds essentially uses a set of rules that correspond to decision tree features instead of blacklist rules. The decision tree classifier approach addresses the aforementioned limitations of blacklists.

1. *Mobile vs. Desktop.* Since EasyList is developed mostly for the desktop-based web browsing ecosystem, it is prone to miss many ad requests in mobile traffic. In contrast, NoMoAds uses decision tree classifiers that are trained specifically on mobile traffic. This leads to more effective classification in terms of the number of false positives and false negatives.

2. *Fewer and more Expressive Rules.* A classifier contains significantly fewer features than the number of rules in blacklists. While EasyList contains approximately 64K rules, our trained decision tree classifiers are expected to use orders of magnitude fewer rules. This ensures that the classifier approach scales well – fewer rules in the decision tree result in faster prediction times. Decision tree rules are also easier to interpret while providing more expressiveness than simple AND/OR.

3. *Automatically Generated Rules.* Since decision tree classifiers are automatically trained, it is straightforward to generate rules in response to changing advertising characteristics. These automatically generated rules can also help human experts create better blacklists.

5.3.3 Training Classifiers

This section explains our approach to training classifiers, which is done offline and at longer time scales. The trained classifier (*i.e.* decision tree model) is pushed to the mobile device and is applied to each outgoing packet in real-time (Sec. 5.3.2.2).

5.3.3.1 Labeling Packets (on the Mobile)

In order to train classifiers, we first need to collect ground truth, *i.e.* a dataset with packets and their labels (whether or not the packet contains an ad request). As shown in Fig. 5.1, we use the `AntMonitor Library`'s API to store packets in PCAPNG format, *i.e.* the packets in PCAP format plus useful information for each packet, such as the packet label. We make modifications to

the AntMonitor Library to allow us to block ad-related packets from going to the network, but still save and label them to be used as ground truth. We use tshark to convert PCAPNG to JSON, extracting any relevant HTTP/S fields, such as URI, host, and other HTTP/S headers. The JSON format offers more flexibility in terms of parsing and modifying stored information, and hence is a more amenable format for training classifiers.

We further extend the AntMonitor Library to annotate each packet with the following information: (i) its label provided by AdblockPlus, (ii) the name of the app responsible for the packet (available via AntMonitor Library's API calls), and (iii) whether or not the packet contains any personally identifiable information, as defined next.

We consider the following pieces of information as personally identifiable information (PII): Device ID, IMEI, Phone Number, Email, Location, Serial Number, ICC ID, MAC address, and Advertiser ID. Some of these identifiers (*e.g.* Advertiser ID) are used by major ad libraries to track users and serve personalized ads, and hence can be used as features in classification. PII values are available to the AntMonitor Library through various API calls provided by Android. Since these values are known, the library can easily search for them with DPI. We refer the reader to Chapter 4 for the full discussion of PII. Within the NoMoAds system, we use the AntMonitor Library's capability to find PII and label our packets accordingly.

5.3.3.2 Training Classifiers (at the Server)

We train decision tree classifiers to detect outgoing packet containing an ad request. We use the decision tree model for the following reasons. First, as discussed in Chapter 4, this model has performed well in terms of accuracy, training and prediction time. Second, decision trees provide insight into what features are useful (they end up closer to the root of the tree). Finally, decision trees make the real-time implementation on the device possible since we know which features to search for.

For training, we adopt the same approach as in Chapter 4 and ReCon [2], described in detail in Sec. 2.3.3.1. We adapt the idea for ads and learn which specific words within packets are useful features when it comes to predicting ad traffic. As an improvement on feature selection, we also discard words that are specific to our setup, such as version numbers and device/OS identifiers (*e.g.* "Nexus" and "shamu"), since we would like our classifier to be applicable to other users. We systematically extract features from different parts of the packet (*i.e.* TCP/IP headers, URL, and HTTP headers) to compare and analyze their relative importance (Sec. 5.5.1.1).

5.4 The NoMoAds Dataset

In order to train and test our classifiers for detecting ads, we collected and analyzed our own dataset consisting of packets generated by mobile apps and the corresponding labels that indicate which packets contain an ad request. Sec. 5.3.3.1 describes the format of our packet traces and the system used to collect them. In this section, we describe the selection process of mobile apps for generation of these packet traces.

App developers typically use third-party libraries to serve ads within their apps. We want to have sufficient apps in our dataset to cover a vast majority of third-party ad libraries. According to AppBrain [1], about 100 third-party ad libraries are used by a vast majority of Android apps to serve ads. Among these third-party ad libraries, only 17 are used by at least 1% of Android apps. The most popular third-party ad library, AdMob, alone is used by more than 55% of Android apps. Therefore, we can gain a comprehensive view of the mobile advertising ecosystem by selecting apps that cover the most popular third-party ad libraries.

We tested the most popular applications from the Google Play Store as ranked by AppBrain [1]. While we mainly focused on apps that contain third-party ad libraries, we also included a couple popular apps (Facebook and Pinterest) that fetch ads from first-party domains. More specifically,

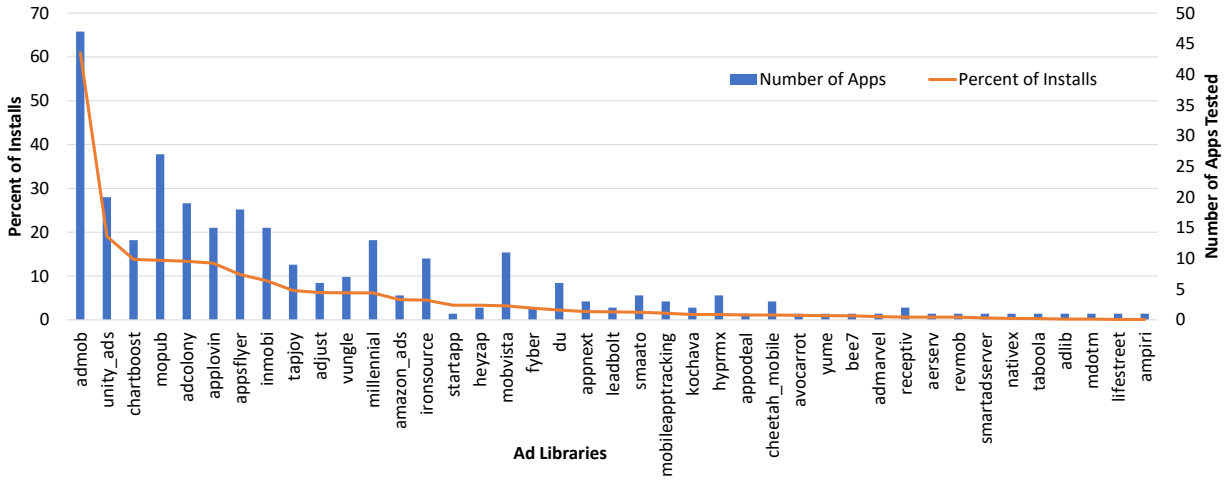


Figure 5.2: Third-party ad libraries that we tested and the number of apps that were used to test each library. The line plot in orange shows the percentage of installed apps from the Google Play Store that use each library according to AppBrain [1]. In order to obtain a representative dataset we made sure to test each ad library with a fraction of apps that is proportional to the fraction of this ad library's installs in the real world.

we selected 50 apps that display ads with the goal of capturing all third-party libraries that account for at least 2% of app installs on the Google Play Store (as reported by AppBrain [1]). Fig. 5.2 shows the 41 third-party ad libraries that are covered by at least one app in our selection of 50 apps. We note that the third-party ad libraries covered in our dataset account for a vast majority of app installs on the Google Play Store.

To label ad packets, we manually interacted with the aforementioned 50 apps from the Google Play Store using NoMoAds integrated with the AdblockPlus Library. We noticed that certain ads were still displayed which means that they were not detected by the filter rules in EasyList. We manually analyzed the outgoing packets using Wireshark [79] to identify the packets responsible for the displayed ads. For instance, some packets contained obvious strings such as `/network_ads_common` and `/ads`, and others were contacting advertising domains such as `applovin.com` and `api.appodealx.com`. To help us identify such strings, we utilized two more popular ad lists – AdAway Hosts [4] and hpHosts [5]. We picked AdAway Hosts because it is specific to mobile ad blocking; and hpHosts has been reported by [34] to find more mobile advertisers and trackers as compared to EasyList. However, we did not always find relevant matches with these lists because

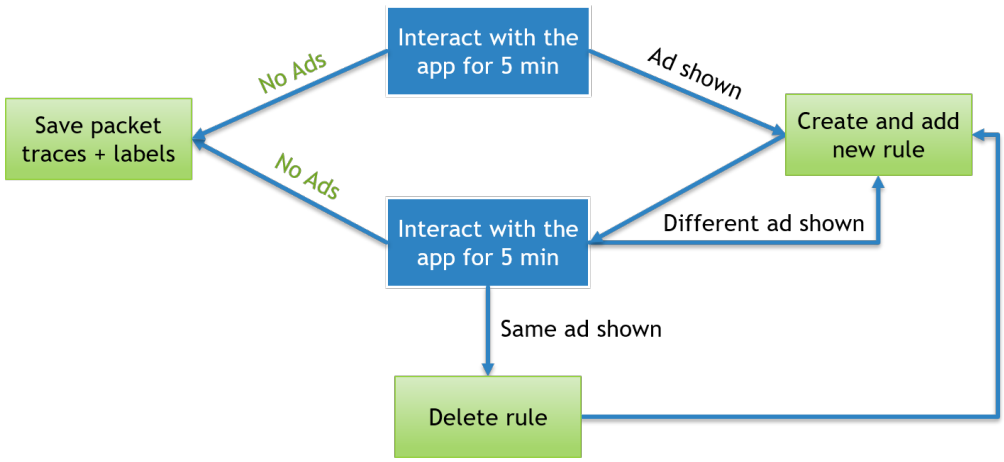


Figure 5.3: Manual interaction with apps to find blind spots in EasyList and create new, mobile-specific rules to cover these blind spots.

they tend to have many false positives and false negatives (see Table 5.4). Using manual inspection along with suggestions from AdAway Hosts and hpHosts, we were able to create Custom Rules, in the EasyList format, that are specifically targeted at mobile ads. In summary, we use the following strategy to develop a list of filter rules to detect all ads by each app, summarized in Fig. 5.3:

1. Run the app with NoMoAds using both EasyList and our Custom Rules. If there are no residual ads, then interact with the app for 5 minutes and save the packets generated during this time. If there are residual ads displayed, then proceed to the next step.

2. Each time an ad is displayed, stop and extract the capture, and inspect the last few packets to find the one responsible for the ad. Use AdAway Hosts and hpHosts for suggestions and develop new Custom Rules. Add the new rule to the list to be used by the AdblockPlus Library.

3. Run the app again to see if the freshly created rule was successful in blocking a given ad. If the new rule matched, but the same ad was still shown, that means the rule triggered a false positive. Remove the rule and repeat Step 2. If the new rule matched, and a different ad was shown, repeat Step 2. The repetition is important as applications often keep trying various ad networks available to them until they find one that will correctly produce an ad. We

	Count
Apps Tested	50
Ad Libraries Covered	41
Total Packets	15,351
Packets with Ads	4,866
HTTPS Packets with Ads	2,657
Ads Captured by EasyList	3,054
Ads Captured by Custom Rules	1,812

Table 5.1: Dataset Summary

stop repeating when there are no more ads being displayed for the duration of the 5 minute interaction with the app in question.

Table 5.1 summarizes key statistics of our dataset. The 50 tested apps in our dataset use 41 different third-party ad libraries. Our packet traces contain 15,351 outgoing HTTP(S) packets out of which 4,866 (over 30%) contain an ad request. Interestingly enough, about half of the ad requests are sent over HTTPS. This indicates good practices among ad libraries, but also demands the TLS-interception ability that is provided by the `AntMonitor Library`.

It is noteworthy that EasyList fails to detect more than one-third (37%) of ad requests in our dataset. We notice that EasyList seems to catch most of the ads generated by AdMob [80] and MoPub [81] – two of the most popular ad libraries, owned by Google and Twitter, respectively. Both of these companies also serve ads on desktop browsers, and hence it is expected that EasyList covers these particular ad exchanges. However, when applications use ad libraries that only have mobile components (*e.g.* UnityAds and AppsFlyer), EasyList misses many ads and we have to create Custom Rules for them. This observation highlights that EasyList is not well suited for today's app-based mobile advertising ecosystem. Table 5.2 shows some of the 91 EasyList rules that matched packets in our dataset. 91 is a tiny fraction of the approximately 64K filter rules in EasyList. Thus, we conclude that EasyList not only fails to capture one third of ad requests but also consists of mostly unused or redundant filter rules. Table 5.3 further shows the Custom Rules that we manually curated to detect ad requests that evaded EasyList. There were dozens of rules

	EasyList Rules	Number of Occurrences
1	/googleads.	951
2	://ads.$domain=...~ads.red…	686
3	://ads.$domain=...~ads.route.cc…	168
4	.com/adv_	135
5	||vungle.com^$third-party	124
6	||inmobi.com^$third-party	107
7	/pubads.	74
8	&ad_type=	64
9	||adcolony.com^$third-party	61
10	/videoads/*	60
11	.com/ad.$domain=~ad-tuning.de	47
12	||smaato.net^$third-party	36
13	||rubiconproject.com^$third-party	34
14	.com/ad/$~image,third-party,domain…	33
15	||adnxs.com^$third-party	28
16	||moatads.com^$third-party	28
17	||appnext.com^$third-party	24
18	||mobfox.com^$third-party	23
19	||andomedia.com^$third-party	23
20	/advertiser/*$domain=~affili.net|~bi…	19
21	/api/ad/*	19
22	||teads.tv^$third-party	17
23	||spotxchange.com^$third-party	17
24	/adunit/*$domain=~propelmedia.com	15
25	/securepubads.	14
26	/adserver.$~xmlhttprequest	13
27	||vdopia.com^$third-party	11
28	/curveball/ads/*	11
29	||ads.tremorhub.com^	10
30	&advid=$~image	10
	…	…
	Total	3054

Table 5.2: EasyList rules that matched at least 10 packets in our dataset. A total of just 91 rules were triggered by our dataset.

that we discarded as they triggered false positives or false negatives (in Step 3 above) and are thus omitted from the table. This finding illustrates the challenge of manually creating filter rules.

Our dataset is publicly available at [11].

	Custom Rules	Number of Occurrences
1	antsmasher_Advertiser ID	611
2	/bee7/*/advertiser_	414
3	||applovin.com^	203
4	/network_ads_common^	169
5	/adunion^	87
6	||ssdk.adkmob.com^	62
7	/mpapi/ad*adid^	49
8	/simpleM2M^	41
9	||placements.tapjoy.com^	36
10	||ads.flurry.com^	33
11	|https://t.appsflyer.com/api/*app_id^	26
12	||api.appodealx.com^	25
13	||cdn.flurry.com^	18
14	||api.tinyhoneybee.com/api/getAD*	6
15	||init.supersonicads.com^	5
16	/ads^	5
17	|https://publisher-config.unityads…	5
18	||ap.lijit.com^$third-party	3
19	||advs2sonline.goforandroid.com^	3
20	||doodlemobile.com/feature_server^	3
21	||live.chartboost.com/api^	2
22	|https://api.eqmob.com/?publisher_id^	2
23	||impact.applifier.com/mobile/camp…	1
24	|http://newfeatureview.perfectionholic…	1
25	||doodlemobile.com:8080/feature_ser…	1
26	||i.bpapi.beautyplus.com/operation/ad^	1
	…	0
	Total	1812

Table 5.3: The set of Custom Rules that we manually came up with to capture ad requests that escaped EasyList, and the number of packets that match each rule in our dataset. Only the rules that triggered true positives are shown.

5.5 Evaluation

In this section, we evaluate NoMoAds in terms of effectiveness of the classification (Section 5.5.1) as well as efficiency when running on the mobile device (Section 5.5.2). In terms of effectiveness, we show that NoMoAds achieves an F-score of up to 97.8% depending on the feature set. Furthermore, we show that NoMoAds performs effectively even when used to detect ads for previously unseen apps and third-party ad libraries. In terms of efficiency, we show that NoMoAds can op-

erate in real-time by adding approximately three milliseconds of additional processing time per packet.

5.5.1 Effectiveness

For evaluation of the classification methodology, we can split the packets in our dataset into training and testing sets, at different levels of granularity, namely packets (Section 5.5.1.1), apps (Section 5.5.1.2), or ad libraries (Section 5.5.1.3). Next, we show that our machine learning approach performs well for each of these three cases. Along the way, we provide useful insights into the classification performance as well as on practical deployment scenarios.

5.5.1.1 Testing on Previously Unseen Packets

First, we consider the entire NoMoAds dataset, described in Section 5.4, and we randomly split the packets into training and testing sets without taking into account any notion of apps or ad libraries that generated those packets.

We note that this splitting may result in overlap of apps in the training and test sets. Training on packets of apps that are expected to be used (and will generate more packets on which we then apply the classifiers) may be both desirable and feasible in some deployment scenarios. For example, if data are crowdsourced from mobile devices and training is done at a central server, the most popular apps are likely to be part of both the training and testing sets. Even in a distributed deployment that operates only on the device, users might want to do preliminary training of the classifiers on the apps they routinely use.

Setup. We train C4.5 decision tree classifiers on various combinations of features, extracted from each packet's headers and payload. The bottom rows of Table 5.4 summarize our results for each feature set. We report the F-score, accuracy, specificity, and recall based on five-fold cross-

	Approaches Under Comparison	F-score (%)	Accuracy (%)	Specificity (%)	Recall (%)	Number of Initial Features	Training Time (ms)	Tree Size	Per-packet Prediction Time (ms)
Ad-blocking lists	EasyList: URL + Content Type + HTTP Referer	77.1	88.2	100.0	62.8	63,977	N/A	N/A	0.54 ± 2.88
	hpHosts: Host	61.7	78.3	89.1	55.2	47,557	N/A	N/A	0.60 ± 1.74
	AdAwayHosts: Host	58.1	81.2	99.8	41.1	409	N/A	N/A	0.35 ± 0.10
NoMoAds with Different Sets of Features	Destination IP + Port	87.6	92.2	94.5	87.3	2	298	304	0.38 ± 0.47
	Domain	86.3	91.0	91.9	89.3	1	26	1	0.12 ± 0.43
	Path Component of URL	92.7	95.1	99.2	86.1	3,557	424,986	188	2.89 ± 1.28
	URL	93.7	96.2	99.7	88.7	4,133	483,224	196	3.28 ± 1.75
	URL+Headers	96.3	97.7	99.2	94.5	5,320	755,202	274	3.16 ± 1.76
	URL+Headers+PII	**96.9**	**98.1**	**99.4**	**95.3**	**5,326**	**770,015**	**277**	**2.97 ± 1.75**
	URL+Headers+Apps+PII	97.7	98.5	99.2	97.1	5,327	555,126	223	1.71 ± 1.83
	URL+Headers+Apps	97.8	98.6	99.1	97.5	5,321	635,400	247	1.81 ± 1.62

Table 5.4: Evaluation of decision trees trained on different sets of features, using our NoMoAds dataset (described in Section 5.4) and five-fold cross-validation. We report the F-score, accuracy specificity, recall, initial number of features, the training time (on the entire dataset at the server), the resulting tree size (the total number of nodes excluding the leaf nodes), and the average per-packet prediction time (on the mobile device). The top rows also show our baselines for comparison, namely three popular ad blocking lists: EasyList [3] (alone, without our Custom Rules), AdAway Hosts [4], and hpHosts [5].

validation. We also report the initial size of the feature set and the training time (how long it takes to train on our entire dataset on a standard Windows 10 laptop). Finally, we report the resulting tree size (number of nodes in the decision tree, excluding the leaf nodes), and the average per-packet prediction time on a mobile device. This helps us gain an understanding of which combination of features are essential for classifying packets as containing ad requests or not.

Network-based Features. We started by using destination IP and port number as our only features. With these features alone, we were able to train a classifier that achieved an F-score of 87.6%. However, IPs change based on a user's location since different servers may get contacted. A natural next step is to train on domain names instead. With this approach, our decision tree classifier achieved an F-score of 86.3%. As expected, training on domains performs similarly to training on IPs since these two features are closely related.

URL and HTTP headers. Domain-based ad blocking is often too coarse as some domains are

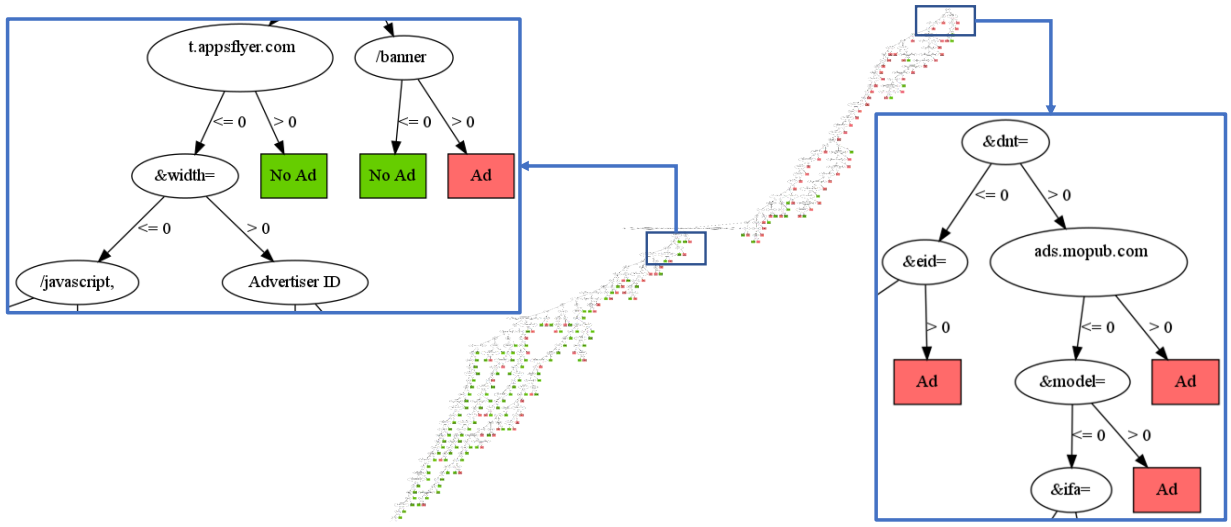


Figure 5.4: Partial view of the classifier tree when using URL, HTTP headers, and PII as features. Out of the 5,326 initial features, only 277 were selected for the decision tree depicted here. At the root of the tree is the feature with the most information gain – "&dnt=", which is a key that stands for "do not track" and takes in a value of 0 or 1. From there, the tree splits on "ads.mopub.com" – a well known advertiser, and when it is the destination host of the packet, the tree identifies the packet as an ad. Other interesting keys are "&eid" and "&ifa" both of which are associated with various IDs used to track the user. Finally, the keys "&model=" and "&width=" are often needed by advertisers to see the specs of a mobile device and fetch ads appropriate for a given screen size.

multi-purposed, which is why ad-blocking blacklists typically operate on URLs. Thus, we trained decision trees with the following combinations: using the path component of the URL only, using the full URL only, and using the full URL and all other HTTP headers. As shown in Table 5.4, breaking the contents of packets into words significantly increases the training time since the number of features grows dramatically from one or two to several thousands. However, having more features increases the F-score to more than 90%. We note that the F-score increases as we use more packet content.

PII as features. Since many ad libraries use various identifiers to track users and provide personalized ads, a natural question to ask is whether or not these identifiers can be useful for detecting ads. The `AntMonitor Library` already provides the capability to search for any PII contained within a packet, including Advertiser ID, Device ID, location, *etc.* Typically, these features cannot be used in lists since they change from user to user. But, since our system runs on-device, it has access to

these values and can provide labels (instead of the actual PII values) as features for classification. However, Table 5.4 shows that using PII as features has a very small effect on effectiveness: although it slightly decreases the amount of false positives (higher specificity), it does so at the cost of increasing the number of false negatives (lower recall). Fig. 5.4 shows a partial view of the final classifier tree when training on URLs, headers, and PII with zoom-ins on areas of interest. At the root of the tree is the feature with the most information gain – "&dnt=", which is a key that stands for "do not track" and takes in a value of 0 or 1. From there, the tree splits on "ads.mopub.com" – a well known advertiser, and when it is the destination host of the packet, the tree identifies the packet as an ad request. Other interesting keys are "&eid" and "&ifa" both of which are associated with various IDs used to track the user. The keys "&model=" and "&width=" are often needed by advertisers to see the specs of a mobile device and fetch ads appropriate for a given screen size. Finally, we note that PII (such as Advertiser ID shown in Fig. 5.4) does not appear until later in the tree, meaning it has little information gain. This is most likely due to the fact that advertisers are not the only ones collecting user information, and that there are services that exist for tracking purposes only.

App names. Next, we examined whether or not additional information available on the mobile device through the `AntMonitor Library` can further improve classification. We considered the application package name as a feature, *e.g.* whether a packet is generated by Facebook or other apps. This is a unique opportunity on the mobile device: a packet can be mapped to the application, which may not be the case if the packet is examined in the middle of the network. As shown in Table 5.4, adding the app as a feature slightly increased the F-score while decreasing the training time by 214 seconds and shrinking the resultant tree size. Training on URLs, headers, and app names alone (without PII) achieves the highest F-score. However, using app names is not ideal since this feature is not available when we classify packets that belong to applications that were not part of the training set. Yet, the better performance of the classifier and faster training time indicates the need to further explore contextual features. For instance, as part of future work, we

plan to use the set of ad libraries belonging to an app as a feature. Moreover, we expect the saving in training time to become more important when training on larger datasets. As we show in Chapter 6, training on a larger dataset takes over six hours, which may be acceptable when running on a server. However, more than that can quickly become unusable, as ideally, we would like to re-train our classifiers daily and push updates to our users, similarly to what EasyList does. Chapter 6 further explores how app names can be used to train per-app classifiers (as in Chapter 4) to reduce training times.

Blacklists as Baselines. The top rows of Table 5.4 also report the performance of three popular ad-blocking lists, which we use as baselines for comparison, namely: EasyList [3] (alone, without our Custom Rules), AdAway Hosts [4], and hpHosts [5]. EasyList is the best performing of the three, achieving an F-score of 77.1%, 88.2% accuracy, 100% specificity (no false positives), and 62.8% recall (many false negatives). Since EasyList filter rules operate on URL, content type and HTTP referer, they are most comparable to our classifiers trained on URL and HTTP Header features. AdAway Hosts and hpHosts perform worse than EasyList since they operate on the granularity of hosts and end up with many false positives and even more false negatives. hpHosts contains more rules than AdAway Hosts, and thus performs slightly better in terms of F-score. However, since hpHosts is more aggressive, it ends up with a lower specificity score.

5.5.1.2 Testing on Previously Unseen Apps

Setup. We split the NoMoAds dataset so that we train and test on different apps. From a classification point of view, this is the most challenging ("in the wild") scenario, where testing is done on previously unseen apps. This may occur as new apps get installed or updated, potentially using new ad libraries, and exhibiting behavior not captured in the training set. From a practical point of view, if one can do preliminary re-training of the classifier on the new apps, before using and pushing it to users, that would be recommended. However, we show that our classifiers perform

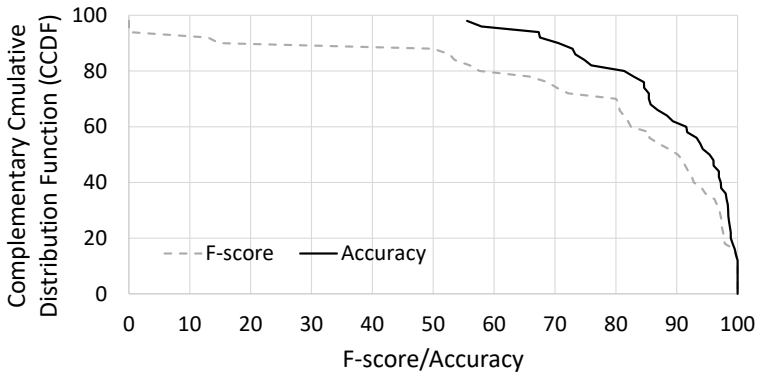


Figure 5.5: Complementary Cumulative Distribution Function (CCDF) of F-score and accuracy for the 50 apps in our dataset.

quite well even in this challenging scenario of testing on packets of previously unseen apps.

We use the decision tree with the URL, HTTP headers, and PII feature set because it performed quite well in Table 5.4 (see row highlighted in bold) and does not use the app name as a feature (which is not useful when testing on unseen apps). To test our system against apps that may not appear in our training set, we performed 10-fold cross-validation, this time separating the packets into training and testing sets based on the apps that generated those packets. Specifically, we divided our dataset into 10 sets, each consisting of five apps, randomly selected. We then trained a classifier on nine of those sets (*i.e.* 45 apps total) and tested on the remaining set of five apps. We repeated this procedure 10 times so that each set was tested exactly once. Therefore, each app was tested exactly once using a classifier trained on different apps (*i.e.* the 45 apps outside that app's test set).

Results. Fig. 5.5 summarizes the classification results (F-score and accuracy). We see that the classifier performs well on a large majority of the apps: over 70% of the apps have an F-score of 80% or higher, while half of the apps have an F-score above 90%. Accuracy is even better: over 80% have an accuracy of 80% or higher. This is expected since there are more negative samples than positive ones, making the true negative rate (and hence the accuracy) high.

Next, we investigated the reasons behind the good classification performance. One plausible hy-

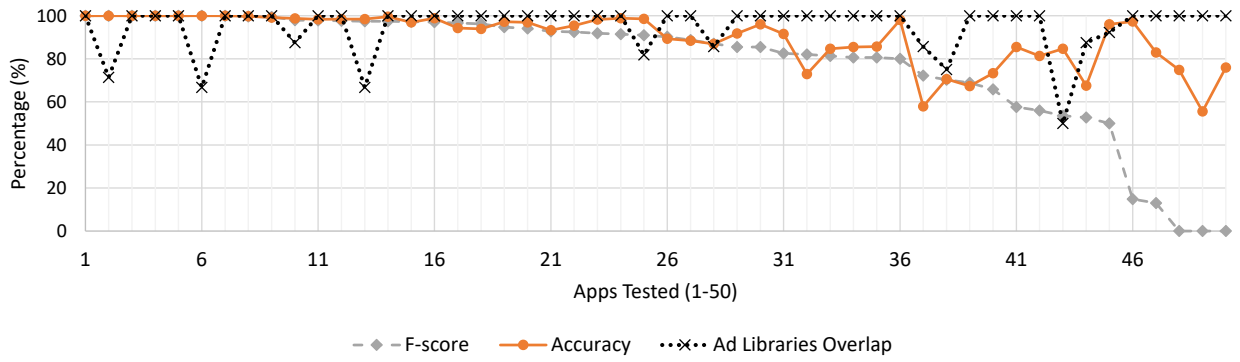


Figure 5.6: Accuracy and F-score of NoMoAds when training on 45 apps in our dataset and testing on the remaining five apps (see Sec. 5.5.1.2). We repeated the procedure 10 times so that all apps were in the test set exactly once. We list the F-score, the accuracy, and the Ad Libraries Overlap (the percentage of a given app's ad libraries that also appeared in the training set) for each individual app. The x-axis orders the 50 apps (*i.e.* the 50 most popular apps on Google Play) in our dataset in decreasing F-score.

pothesis is that the classifier performs well on previously unseen apps because apps use the same ad libraries. To assess this hypothesis, for each app, we computed not only its F-score/accuracy (on each individual app's data) but also the overlap in ad libraries. We define the overlap in ad libraries as the percentage of the app's ad libraries that appeared in the training set. An overlap of 100% means that all ad libraries used by the app in question were seen during training.

Fig. 5.6 shows the results for each app. The apps are ordered on the x-axis in terms of decreasing F-scores. We note that all apps have high overlap in ad libraries: all of them are above 50% and most of them have 100%. There are several apps (*e.g.* apps 2, 6, 10, 13, and 25) with near perfect F-scores even though they contain some ad libraries not seen during training (*i.e.* have a less than 100% overlap in ad libraries). However, there are a few apps with low F-scores despite the fact that they had all their ad libraries captured in the training set (overlap 100%). One possible explanation is that each app employs different functionalities of each ad library and some functions may have been absent from the training set.

False Negatives. We took a closer look into the five worst performing apps (*e.g.* apps 46-50, on the right of Fig. 5.6 that had a very low F-score). The main reason behind their poor performance was the low number of positive samples. For instance, App #48 is Spotify, which had no positive

samples, and hence no true positives, making the F-score equal to 0%. Apps #49 and #50 are YouTube and LINE: Free Calls & Messages, respectively. These two apps had eight and 19 positive samples, respectively – a small number in comparison to an average of about 97 positive samples per app. NoMoAds was unable to detect these positive samples correctly. However, in both cases NoMoAds achieved a 0% false positive rate, and hence the accuracy for both of these cases is relatively high.

Next, we examined the two apps with a low (13% and 14.8%) but non-zero F-score. The two apps are Facebook (App #47) and Pinterest (App #46), and they are the only ones in our dataset that are serving ads as a first party, while all the other apps use third-party ad libraries. In other words, the first-party ad serving behavior was not seen during training in both cases, which led to the poor performance of the classifiers.

Finally, for the four poorly performing apps with a non-zero amount of positive samples, we performed another experiment. Specifically, we trained on all 49 apps plus 50% of the packets belonging to each app in question and tested on the remaining 50% of packets belonging to that app. In some cases, the F-score improved significantly: the LINE: Free Calls & Messages app was able to achieve an F-score of 100%. YouTube stayed at 0% F-score due to a very low number of positive samples (eight total, with just four in the training set). Facebook and Pinterest improved only slightly: an F-score of 52.2% and 50%, respectively. This can be explained by the fact that both of these apps not only serve ads from a first-party, but also fetch very diverse content depending on what the user clicks. In contrast, most of the other apps in our dataset have a very specific set of functionalities and do not exhibit a wide range of network behavior. For example, games tend to contact their servers for periodic updates and third party servers for ads. Whereas Facebook and Pinterest can contact many different servers based on which pages the user visits. This finding suggests a deployment approach where the classifier is pre-trained on some packets of these popular apps before being deployed on the mobile device.

False Positives. We also examined apps with a high false positive rate. For instance, Shadow Fight 2 (App #40) only had 48% specificity, but looking at the packets that were incorrectly labeled as positive revealed that all of the false positives were actually related to ads and/or tracking. Specifically, the URLs within those packets contained the advertiser ID and strings such as the following: "ad_format=video," "tracker," and "rewarded." Similarly, other apps that had specificity below 80% (Angry Birds Seasons (App #17), Spotify (App #48), Plants vs. Zombies FREE (App #43), and BeautyPlus - Easy Photo Editor (App #28)) also had explainable false positives that contained the following strings within the URLs: "/impression/," "spotify.ads-payload," "/tracking/api/," and "pagead/conversion/." One possible explanation for these packets getting through our labeling process (Sec. 5.4) is that not all ad-related packets directly lead to an ad. For instance, some are simply specifying phone specs to fetch an ad of the correct size in the future, others track if and when an ad was actually served, and some track the user's actions to serve more personalized ads. This indicates that even our Custom Rules are incomplete, but our classifiers can suggest more rules for further improvement of ad-blocking.

5.5.1.3 Testing on Previously Unseen Ad Libraries

Another way to test the performance of our machine learning approach is to see if our classifiers can perform well on libraries that were not present in the training set. This would require to partition the packets in the dataset into testing and training parts, so as to separate different ad libraries, and then train on some and test on the remaining ones.

Unfortunately, this is not possible with network-based approach from user-space (VPN) alone: we can identify which app generated a packet, but we cannot reliably identify which ad library is responsible for a given packet. Using AppBrain's database, we can tell which ad libraries an app contains, but we do not know which ones are actually used. This is because 90% of the apps in our dataset use more than one ad library, and 80% use more than two. Facebook and Pinterest are examples of apps with no ad libraries, and we discussed them in detail in the previous section.

There are three apps with just one ad library, and they all contain AdMob [80] - the ad library that is present in all of the apps in our dataset (except Facebook and Pinterest). Therefore, we cannot partition our dataset based on ad libraries. The next chapter (Chapter 6) discusses how static analysis and dynamic instrumentation can be used to facilitate this mapping by tracing API calls to the network.

In this chapter, we analyzed the overlap in ad libraries to get some insight into the performance of our system in the presence of previously unseen ad libraries. Fig. 5.6 shows that NoMoAds does not need to see all the ad libraries in training in order to correctly classify packets belonging to apps with unseen ad libraries. Specifically, there are several apps (*e.g.* apps 2, 6, 10, 13, and 25) with near perfect F-scores even though they contain some ad libraries not seen during training (*i.e.* have a less than 100% overlap in ad libraries). This can be easily explained by the fact that multiple ad libraries may end up contacting the same ad networks even though their software implementation for mobile devices may differ. Moreover, as we stated earlier, some apps may contain multiple ad libraries, but use only a subset. We refer the reader to Section 5.5.1.2 for details on how the overlap of ad libraries relates to the classification performance.

5.5.2 Efficiency

5.5.2.1 Classification on the Mobile Device

In this section, we discuss experiments performed on a mobile device and we demonstrate that our decision tree classifiers can run in real-time – on the order of three milliseconds per packet. The experiments were performed on a Nexus 6 (Quad-Core 2.7 Ghz CPU, 3 GB RAM) running Android 6.0.1. To minimize noise from background processes, we kept only pre-installed apps and an instrumented version of NoMoAds.

To evaluate how much extra processing NoMoAds incurs, we fed 10 HTTP packets of vary-

ing sizes (between 300-2000B) to our classification function and timed how long it took using `System.nanoTime()`. As a baseline for comparison, we also timed how long parsing out HTTP features and using the AdblockPlus Library (with EasyList) takes to label a packet. We repeated each test case 100 times and calculated the average run-time and standard deviation. Each function was tested in isolation, running on the main thread, so as to minimize timing the overhead of possible thread switching. The results are as follows.

- The total time for NoMoAds to extract features and apply the decision tree classifier is: 2.96 ms $\pm$ 2.07 ms.
- The total time for HTTP parsing and applying the AdblockPlus Library is: 1.95 ms $\pm$ 0.75 ms.

Although the AdblockPlus Library outperforms NoMoAds by one millisecond (on average), it does so at the cost of a nearly 20% degradation in the F-score performance (Table 5.4).

In order to understand how much latency overhead prediction by itself adds, we tested the same 10 HTTP packets (100 times each) and timed the prediction time of each variant of our classifier (see the last column in Table 5.4). As we can see, the prediction time closely follows the tree size – the smaller the tree, the quicker we can make a prediction. Hence, it is important to know which features to train on in order to produce a small and efficient tree that can be used on mobile devices in real-time without significantly degrading user experience. Our tree of choice (URL, HTTP Headers, and PII), on average, predicts within three milliseconds. For comparison, we repeated the experiment with the AdblockPlus Library, this time isolating the matching of URL, Content Type, and HTTP Referer. We report the result in the last column of Table 5.4. The AdblockPlus Library is more efficient than our classifier in prediction time, indicating that most of the delay, when using the AdblockPlus Library approach, comes from HTTP parsing. Conversely, in the NoMoAds approach, most of the delay comes from the prediction itself, and not the search for features.

5.5.2.2 (Re)training Time

In Table 5.4, we reported the training time when using our entire dataset, as well as the size of the initial feature set extracted from all packets, and the final size of the tree (number of non-leaf nodes in the decision tree). We note that only a small subset of the features is selected by the decision tree. The selected features are up to an order of magnitude less in size than the initial feature set. This results in relatively small and intuitive classifiers, like the one depicted in Fig. 5.4. Furthermore, the selection of features significantly affects the training time and has a moderate effect on classification performance. In this Chapter, our training dataset was relatively small, and training our classifiers from scratch did not take more than 13 minutes (Table 5.4). This is acceptable since training is currently done offline at a remote server. In future work, we plan to further investigate training time as a function of the size of the training dataset and the selected features. Our goal is to be able to train and retrain our classifiers within a couple hours, in order to be able to push them from the server to mobile devices at least once a day, or a few times a day, as EasyList does.

5.6 Summary

NoMoAds is the first mobile ad-blocker to effectively and efficiently block ads served across all apps using a machine learning approach. Our work complements blacklist-based ad-blocking approaches, such as EasyList [3], DNS66 [31], and recent work on learning flow-based features [40]. To encourage reproducibility and future work, we make our code and dataset available at [11]. One limitation of NoMoAds is the limited size of the training set. In this Chapter, we manually labeled packets, which is not scalable if larger datasets are desired for training. The next Chapter discusses an automatic approach for labeling packets, which not only enables us to expand our dataset, but also provides the ability to map each packet to the library responsible for generating it.

Chapter 6

AutoLabel: Detecting Third-Parties

6.1 Overview

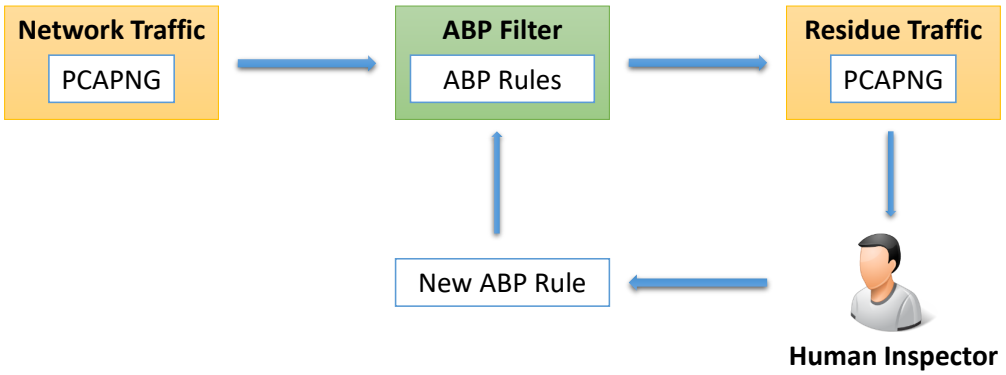
The NoMoAds system presented in the previous Chapter was an improvement over filter-based ad-blocking tools. However, NoMoAds had a major limitation – the system had no way of automatically labeling datasets. The same bottleneck applies to all other prior art on ad-blocking and anti-tracking. Specifically, to seed ground truth, prior art relies on either filter lists [39, 38, 34, 41], manually labeled data [40], or a combination of both [33].

In this Chapter, we aim to solve the problem of manual labeling of mobile network requests that are either requesting ads or are tracking the user (A&T requests). We start by noting that tracking and advertising on mobile devices is usually done by third-party libraries that app developers include in their apps to generate revenue. Throughout this Chapter, we will refer to a request as an A&T request, if it was generated by a library whose primary purpose is advertising or analytics (A&T libraries). Another key observation is that it's possible to determine if a network request came from the application itself or from a library by examining the Java stack trace leading to the network API call. More specifically, stack traces contain package names that identify different entities: app

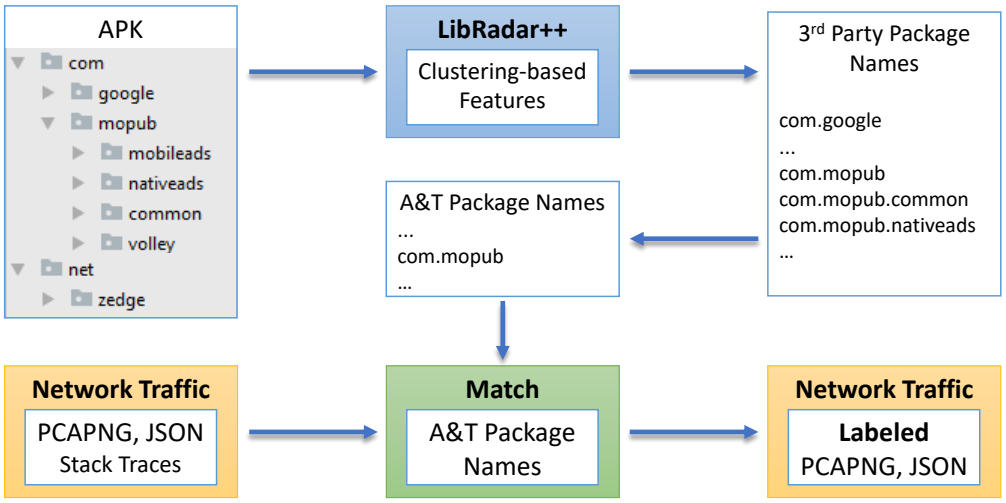
vs. library code. It is possible to manually curate a list of package names belonging to libraries by downloading and analyzing their JAR files, as was done in [22]. However, since we aim to minimize manual list curation efforts, we leverage advances in static analysis of apps in order to identify A&T libraries' package names [20, 82]. We combine the aforementioned ideas to make the following contributions:

- We design and develop AutoLabel – a system for automatically identifying and labeling A&T requests. Our approach does not rely on filter lists nor manual labeling of requests. Instead, it examines the Java stack trace leading to the network API call to determine whether the request was generated by an A&T library.
- We use AutoLabel to collect the first large-scale dataset of network requests, where each request is *automatically* labeled based on its origin: the app itself or an A&T library.
- Prior work has trained classifiers that use features extracted from packet headers and/or payload to predict ad requests [33], but it does not scale well on our large dataset. We build on and improve the state-of-the-art classification framework [33]: we use our labeled dataset to train a classifier on a per-app basis and to predict both advertising and tracking (A&T). Our classifiers can be trained quickly (on the orders of milliseconds) while achieving average F-scores of 94%.
- Finally, we evaluate both our labeling system and machine learning prediction against popular filter lists, namely EasyPrivacy [3] and Mother of All Ad-Blocking (MoaAB) [83]. Our labeling system discovers thousands of requests destined to hundreds of different hosts that evade filter lists and are potentially engaged in A&T activities. We also show that our classifiers generalize and find trackers that were missed by our labeling procedure.

Our classifiers can be applied in real-time on mobile devices to predict and block A&T requests when running on top of the `AntMonitor` system presented in Chapter 3, or any other efficient



(a) Without our approach: manual labeling of ground truth. ABP = AdblockPlus.



(b) With our approach: automatic ground truth labeling with LibRadar++.

Figure 6.1: Comparison of manual and automatic labeling approaches.

VPN-based interception system for mobile devices, such as [2, 29, 53]. To provide other researchers the ability to collect their own mobile tracking datasets, we will make AutoLabel open-source. We will also release our labeled dataset and will contribute back to the NoMoAds [33] project with our improvements on classifier training.

```
com.android.org.conscrypt.NativeCrypto.
        SSL_do_handshake
...
net.zedge.android.api.request.BaseApiRequest.
        run
net.zedge.android.config.ConfigLoaderImpl.
        loadConfigurationBlocking
net.zedge.android.config.ConfigLoaderImpl.
        loadWithBackoff
...
```

(a)

```
com.android.org.conscrypt.NativeCrypto.
        SSL_do_handshake
...
com.mopub.network.RequestQueueHttpStack.
        executeRequest
com.mopub.volley.toolbox.BasicNetwork.
        performRequest
com.mopub.volley.NetworkDispatcher.
        processRequest
...
```

(b)

Figure 6.2: Example stack traces captured from the *ZEDGE™Ringtones & Wallpapers* app: (a) The app itself is initiating an SSL handshake, as indicated by the presence of the package name `net.zedge.android` in the stack trace; (b) The *MoPub* ad library is starting an SSL handshake, as indicated by the `com.mopub` package name.

6.2 Labeling

In this Chapter, we are interested in automatically labeling which mobile network requests are either tracking the user or are requesting ads for the user. Prior art, such as NoMoAds (Chapter 5), was limited in its ability to label A&T requests: it required a human to first detect an ad and then to manually come up with a rule in AdblockPlus format [84] that would block future occurrences of such ads. We illustrate this manual and iterative process in Fig. 6.1(a). Not only is this approach prone to human error, it is also not scalable and is limited to ads only because tracking activities are invisible.

AutoLabel aims to solve the problem of manual labeling of mobile A&T requests. We start by noting that tracking and advertising on mobile devices is usually done by third-party libraries that app developers include in their apps to generate revenue. Throughout this Chapter, we will refer to a network request as an A&T request, if it was generated by a library whose primary purpose is advertising or analytics (A&T libraries). Another key observation is that it's possible to determine if a network request came from the application itself or from a library by examining the Java stack trace leading to the network API call. For example, consider the trimmed stack traces shown in Listings 6.2(a) and 6.2(b). Both were captured within the *ZEDGE™Ringtones & Wallpapers* app,

which has the package name `net.zedge.android`. Listing 6.2(a) shows the app starting an SSL handshake, as indicated by the presence of the package name `net.zedge.android` in the stack trace. On the other hand, Listing 6.2(b) shows the *MoPub* ad library starting an SSL handshake, as indicated by the `com.mopub` package name. Prior art, such as PmP [14], has also used stack trace analysis to infer which third-party libraries were accessing sensitive APIs. However, hooking into networking APIs is challenging (see Sec. 6.3.1.1), and thus PmP was unable to fetch such detailed traces for network access [14]. How we obtain the stack traces leading to each network request is described in Sec. 6.3.1.1.

In order to use stack traces for labeling A&T requests, we also need a list of package names belonging to libraries, and we need to know which ones are A&T libraries. To minimize manual list curation efforts, we use advances in static analysis of apps to help us identify A&T libraries' package names. Android apps are structured in a way where classes belonging to different entities (*e.g.* app vs. library) are separated into different folders (packages). One such structure is shown in the "APK" box in Fig. 6.1(b). As with the stack traces shown in Fig. 6.2, the APK pictured in Fig. 6.1(b) also belongs to the *ZEDGE*™ app. Note how the APK is split between packages belonging to *Google*, *MoPub*, and *ZEDGE*™ itself. We can use this splitting to extract package names belonging to third-party libraries. In fact, that is exactly what LibRadar [20] does: they build signatures for each packaged folder and then use clustering to identify third-party libraries. Using this technique they have built an initial database of 29k libraries. Based on the extracted library signatures, LibRadar can identify libraries in new apps and can provide the corresponding packages names even when package name obfuscation is used. Recently, an updated version of LibRadar was released – LibRadar++ [82]. This version of the tool is built over a larger set of apps (six million) and libraries (5,102).

Thus, AutoLabel uses LibRadar++ [82] to analyze apps and automatically produce a list of library package names contained within (Fig. 6.1(b)). Note that LibRadar++ provides two package names as output: the first is the package name used in the APK and the second is the original package

name of the library. Most of the time the two names are the same. If an app uses package-name obfuscation, then the names are different, but LibRadar++ is still able to identify the library via its database of library signatures. Although there are some cases in which this identification fails (see Sec. 6.5.2), the LibRadar++ approach is still more resilient than matching against a static list of library package names. Furthermore, if an updated version of LibRadar++ becomes available, it can easily be plugged into AutoLabel. Based on the original package name that LibRadar++ provides it is trivial to identify popular A&T libraries: one can simply search the Internet or use an existing list of library names and their purposes, such as AppBrain [1]. To identify A&T libraries, we use the list prepared by LibRadar [85], which maps library package names to their primary purpose.

Fig. 6.1(b) summarizes the AutoLabel method: we match the collected stack traces against a list of package names belonging to A&T libraries produced by LibRadar++. We note that unlike the approach depicted in Fig. 6.1(a), our method has minimal human involvement. The only point where manual effort might be needed is in mapping package names to their primary purpose. However, we note that unlike a list of rules to match against URLs, a mapping of library names to their purpose will always be available and will remain up-to-date so that app developers can select which libraries to use. In order to perform this labeling, we need a system that can collect network requests and map them to the stack traces that led to each networking API call. The next section describes how AutoLabel achieves this mapping.

6.3 Data Collection System

In this section, we describe the AutoLabel data collection system that can be used to collect outgoing network traffic and the Java stack traces that have led to each request. At a later stage, the stack traces can be used to identify which requests were sent by A&T libraries (Sec. 6.2). As shown in Fig. 6.3, our data collection system consists of two main components, described next. First, we

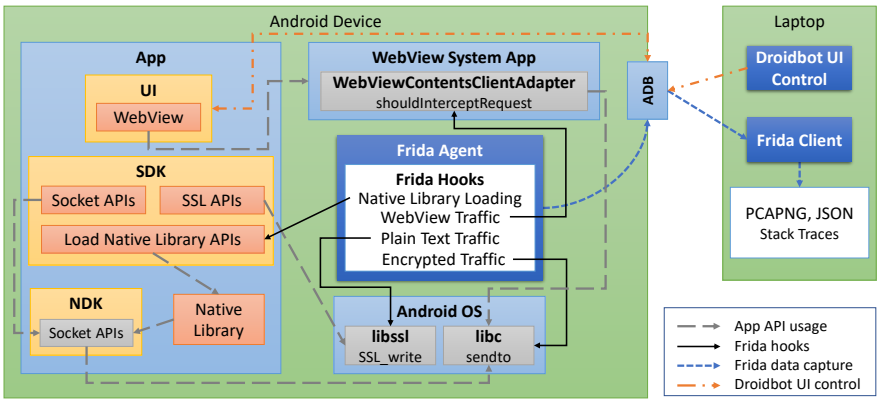


Figure 6.3: Our data collection system (Sec. 6.3): using Frida to hook into Android networking APIs, capture network traces, and the Java stack traces leading to the networking API call.

use the Frida [16] dynamic instrumentation toolkit to hook into various Android API calls of interest (Sec. 6.3.1). Second, we use the Droidbot [86] UI automation tool to automatically exercise Android apps (Sec. 6.3.2).

6.3.1 Hooking Android Networking APIs

To hook into networking APIs and collected the stack traces that led to them, we use the Frida [16] dynamic instrumentation toolkit. We choose Frida over Xposed [15] for two reasons. First, Xposed can only be used on Android devices, whereas Frida can also hook into iOS. Therefore, with some modifications, our approach has the potential to be applied on iOS devices. Second, Frida is professionally maintained and is sponsored and used by NowSecure [87] – a mobile security company. As shown in Fig. 6.3, Frida consists of two components: a Frida agent that runs on the mobile device and a Frida client that runs on a connected computer. Sec. 6.3.1.1 describes our Frida agent: how we hook into APIs that result in outgoing network requests and save the Java stack traces that have led to each hooked function call. Sec. 6.3.1.2 describes how our Frida client saves the captured network requests in either PCAPNG or JSON format, along with the Java stack traces.

6.3.1.1 Frida Agent

Frida allows developers to write JavaScript code that gets injected at run-time into an application process. The injected code is referred to as the agent. The agent can monitor and modify the behavior of the application by hooking into API calls of interest. Once inside a hooked function, we can fetch the Java stack trace that led to the function being hooked. Since we are interested in networking API calls only, we begin by providing some background on the Android OS and then explain our hooking methodology. Our methods can be used by future researchers to gain deeper insight into network activities within Android apps, not just in terms of advertising and tracking, but across all types of third-party libraries.

Background. As shown in Fig. 6.3, Android apps typically access the network by using Java API calls available through the Android Software Development Kit (SDK). These APIs can be high level, such as `java.net.URLConnection.openConnection`, or low level, such as `java.net.Socket`. Regardless of which API is used, every network operation eventually ends up in a `Posix` socket operation, which then gets translated into a standard C library (`libc`) socket API call. Previous studies, such as TaintDroid [12], have hooked into `Posix` sockets to gain visibility into outgoing network traffic. However, apps can also use the Android Native Development Kit (NDK) to write additional C modules for their apps. These native modules can access the `libc` socket APIs and bypass any hooks installed in Java space (see Fig. 6.3). To send encrypted traffic, apps typically use Java SSL APIs, such as `javax.net.ssl.SSLSocket`. These API calls get translated into native OpenSSL operations. Although OpenSSL is not available through the NDK, apps sometimes include their own SSL libraries [88].

To ensure that we catch all outgoing communication performed by an app along with the detailed stack traces leading to each network request, we place the following Frida hooks, as depicted in Fig. 6.3:

Plain Text Traffic. Prior work in both static (*e.g.* [22]) and dynamic (*e.g.* [12, 13]) analysis of Android apps suffered from not having visibility into native code. To overcome this limitation, we hook into the `libc` `sendto` and `write` functions to collect plain text data sent via both the SDK and the NDK API calls.

Encrypted Traffic. In order to read plain text traffic before it becomes ciphertext, we hook into Android's OpenSSL `SSL_write` function.

WebView Traffic. WebView is a special GUI component in Android that provides developers with ready-made browser-like functionalities. Although our previous two hooks can successfully capture WebView traffic, we also want the ability to accurately label if the traffic is coming from the app or from a third-party library by examining stack traces. Unfortunately, in the case of WebView, all network requests are handled by the Android System WebView app (see Fig. 6.3). This means that regardless of if the WebView element was created by the app or by a library, the network requests it generates will appear to be coming from the System WebView app. To handle this special case, we hook into the `WebViewClient` constructor since applications are required to create a `WebViewClient` object in order to control a WebView. From the `WebViewClient` constructor we get a stack trace leading to the app or a library and we save each `WebViewClient` and stack trace pair for later query. We also hook the `shouldInterceptRequest` function of the WebView app's `WebViewContentsClientAdapter` class. This function gets called every time any WebView in the system attempts to load a resource. Since each `WebViewContents-ClientAdapter` instance is tied to a specific `WebViewClient` object, we can refer to our saved data structure containing `WebViewClient` and stack trace pairs to get the stack trace responsible for each WebView request.

Native Library Loading. Since apps can include their own versions of OpenSSL, we hook into the `java.lang.System` `loadLibrary` and `load` function calls to catch occurrences of native libraries being loaded. We check each loaded library for the inclusion of the OpenSSL `SSL_write`

method, and if such a method exists, we add a Frida hook to it. Although it is possible for apps to include SSL libraries other than OpenSSL, a 2017 study has shown that only 14% of apps (out of the studied 7,258) use third-party SSL libraries [88]. Furthermore, apps that do use other SSL libraries are usually browser-type apps which are not the focus of this Chapter (see Sec. 6.4.1.1).

6.3.1.2 Frida Client

The Frida client runs outside of the inspected application. When Frida is used to inspect mobile devices, the Frida client runs on a PC that is connected to the mobile device via USB. In the case of Android, the Frida client and agent can communicate with each other through the Android Debug Bridge (ADB), as shown in Fig. 6.3. When a hook has been triggered, our Frida agent sends the Java stack trace leading to the hooked function, along with any relevant data, to the client. The client can then save the collected data on the PC for any future analysis. Depending on the hook triggered, the client saves the data in either PCAPNG or JSON format, as described next.

Non-WebView Traffic. Whenever a `sendto`, `write`, or `SSL_write` is triggered, the Frida agent has access to the full packet bytes that were about to be sent since the packet buffer is passed as a pointer argument to each function. In the case of `sendto` and `write`, the socket file descriptor is also passed. With the help of various `libc` functions and the file descriptor, the agent can learn auxiliary information about the intercepted connection: the IP address of the remote server, the destination port, and whether the traffic is TCP or UDP. To learn the same information about an SSL connection, the agent can utilize a `libssl` function to fetch the file descriptor from the `SSL` object which is passed as a pointer argument to `SSL_write`. The agent then sends this auxiliary information along with the captured packet bytes to the Frida client for further processing. Based on the provided information, the client can reconstruct parts of the network and transport layer headers and save the packet in PCAPNG format. We chose the PCAPNG format as it allows the usage of common tools such as *tshark* for correctly parsing packet bytes. In addition, the

PCAPNG format allows adding a comment to each captured packet. We utilize the packet comment to store the Java stack trace leading to hooked function call.

WebView Traffic. WebView traffic is always sent over HTTP/S, and the `shouldIntercept-Request` function's argument `ShouldInterceptRequestParams` can be used to extract all HTTP fields of interest. Since `shouldInterceptRequest` operates on the application layer, these fields are available in plain text, even in the case of HTTPS. We extract the following fields and save them in JSON format along with the stack trace of the responsible `WebViewClinet` (see Sec. 6.3.1.1): the full URL, the HTTP headers, and the HTTP method.

6.3.2 Droidbot UI Control

As shown in Fig. 6.3, we use Droidbot [86] to automatically exercise apps at scale. Droidbot is a lightweight tool that requires no modifications to the Android OS and no application instrumentation. Droidbot consists of two components, as shown in Fig. 6.3: an Android app and a Python script that runs on a connected PC. The Droidbot Android app utilizes the Android Accessibility API [89] to find UI elements of an app in testing, in real-time. The app sends this information through ADB to the Droidbot Python script. Upon receipt of the UI data, the script can decide which UI element to exercise and send the command through ADB. Since UI elements can be thought of as a graph, Droidbot offers two algorithms for UI exploration: Depth First Search (DFS) and Breadth First Search (BFS). During our experiments we found the DFS variant to cause apps to crash, hence we decided to use the BFS algorithm for exercising apps. We note that most previous studies that collect mobile packet traces exercise apps either manually [33], or both manually and with the UI/Application Exerciser Monkey [61] (*e.g.* [2]). However, manual testing does not scale well. On the other hand, Monkey, when used as a standalone tool, can only send random events to the device – it has no knowledge of the GUI. This can lead to limited coverage of the app, and can even lead to other apps being exercised instead of the intended one. For example, Monkey

can end up clicking on ads or links which open up browser apps. Droidbot can detect when it has left the intended app and can send a command to go back to the application in testing. Jin et al. [90] have also used Droidbot to exercise apps and collect network traces. We believe that this is the correct direction for future research on mobile network traffic.

6.4 Machine Learning Application

One application of AutoLabel is providing a ground truth dataset to train machine learning classifiers. In this section, we explore this application, specifically for training classifiers to predict A&T packets. We begin by collecting a new, diverse dataset using AutoLabel (Sec. 6.4.1) and then we use that dataset as our ground truth for training machine learning classifiers (Sec. 6.4.2).

6.4.1 Data Collection

In this section, we describe our collected dataset. First, in Sec. 6.4.1.1, we discuss how we selected and downloaded apps to test, and the testing environment we used when exercising the selected apps. Next, in Sec. 6.4.1.2, we summarize our collected dataset.

6.4.1.1 Selecting Apps

To diversify our dataset, we used the Google Play Unofficial Python API v0.4.3 [91] to download the top 10 free apps from each of the 35 Google Play categories on January 29, 2019. Previous studies, such as [33] and [2] used top lists (*e.g.* AppAnnie) to select apps to test without taking categories into consideration. However, in our experience we found that such lists are heavy on certain categories, such as games and entertainment, and can sometimes miss less popular categories, such as art & design and family. This can cause problems because some categories of apps

	Count
Apps Tested	307
Total Packets	37,438
A&T Packets	13,512
Apps with Packets	298
Apps with 10+ Positive Samples	143
A&T Libraries	37
A&T Libraries with Packets	26
WebView Packets	17,057
SSL_Write Packets	17,028
libc sendto Packets	3,353

Table 6.1: Dataset summary. Packets are HTTP/S packets.

are more likely to use specific third-party libraries. For instance, game apps often use the Unity Ads library [92] to display in-game ads. Since we want to gain a comprehensive view of the mobile advertising and tracking ecosystem, we chose to diversify our app categories. In addition, we manually verified that none of the selected apps are browser apps, such as Firefox. Since browser apps function like any desktop browser, they can open up the entire web ecosystem. In this Chapter we focus on learning mobile A&T behavior, and leave the in-browser anti-tracking protection to standalone tools, such as the Brave browser [93].

Since some apps appear in multiple categories, we ended up with a total of 339 distinct apps. We ran each app using the system described in Sec. 6.3 on a Nexus 6 device running Android 7.1. We disabled the Chrome WebView and used the Android System WebView to ensure compatibility with our Frida hooks (Sec. 6.3.1.1). Since the Frida hooks may be affected by updates of Google Play Services and of the Android System WebView, for reproducibility purposes we indicate that we used versions 9.8.79 and 55.0.2883.91, respectively. Each app was exercised with Droidbot for 5 minutes, sending inputs every 2 seconds. Certain apps could not be installed or had run-time errors. For instance, a couple apps detected the rooted state of the device and would not start. Some apps required a Google Play Services update, which we could not update as it would affect our Frida hooks. In total, we had to exclude 32 apps, leaving us with a total of 307 apps to test.

6.4.1.2 Data Summary

Table 6.1 summarizes our dataset. Out of 307 apps that we tested, 298 sent at least one HTTP/S packet. In total, we collected 37,438 HTTP/S packets, 13,512 of which were A&T packets. This means that over 36% of outgoing HTTP/S packets belong to A&T libraries. With our dataset of 307 apps we were able to test 37 different A&T libraries, 26 of which have sent out at least one network request. Surprisingly, most (17,057) of the packets that we captured were sent by `WebView` objects. This means that had we not placed the appropriate hooks within `WebViewClient` constructors (Sec. 6.3.1.1), we would have been unable to obtain stack traces for over 45% of captured packets. Finally, although we have seen apps load their own versions of OpenSSL (Sec. 6.3.1.1), we were unable to trigger the functionalities required to make the apps use their OpenSSL modules.

To facilitate parsing and training of machine learning classifiers, we convert all PCAPNG files to JSON, keeping only the relevant HTTP/S fields, namely the full URL, the HTTP headers, and the HTTP method. This also provides us consistency with the data collected from WebViews (Sec. 6.3.1.2), which is already in the JSON format. Each JSON data point also contains a label indicating whether or not it contains an A&T packet. These JSON files can then be used to train machine learning classifiers, as shown in Fig. 6.6 and described in the next section.

6.4.2 Machine Learning Approach

First, we discuss our training setup in Sec. 6.4.2.1. Second, we train a general classifier on our entire dataset of 37,438 packets (Sec. 6.4.2.2). We find that the hours-long training times of the general classifier makes it unsuitable for real deployment. Third, we train and evaluate per-app classifiers (Sec. 6.4.2.3).

Feature Set	F-score (%)	Accuracy (%)	Specificity (%)	Recall (%)	Training Time	Tree Size
Domain	85.33	91.18	93.99	84.71	182 ms	1
Host	90.28	92.70	92.01	93.93	96 ms	1
Path Component of URL	84.05	91.11	97.09	77.35	6.07 hr	500

Table 6.2: General Classifier Performance. Tree size here is the number of non-leaf nodes.

6.4.2.1 Setup

For training, we adopt the same approach as in Chapter 5 and ReCon [2], described in detail in Sec. 2.3.3.1. To evaluate which features have the most importance, we selectively extract words from various parts of packets as discussed next.

Host. First, we evaluate how well can our classifiers perform when operating only with second-level domains (SLDs) or host names. Evaluating on SLDs can help identify which SLDs are only involved in A&T activities. This can make filter lists, such as EasyList, shorter by eliminating the need to list all possible hosts within an SLD. Using host names only is representative of a situation where TLS traffic cannot be decrypted, and we are only left with DNS requests or with the server name identification (SNI) headers from TLS handshakes.

URL. Second, we evaluate how using more of the packet helps with classification. Specifically, how well can a classifier perform when using only the path component of the URL (including query parameters) and when using the full URL (host name and the path component).

URL and HTTP Headers. Finally, we evaluate how adding HTTP headers helps in classification. We note that some filter lists, such as EasyList, also use the Referer header and Content-Type headers. Thus, for a fair comparison, we evaluate these two headers separately, and then we evaluate all the HTTP headers.

6.4.2.2 General Classifier

Table 6.2 summarizes the results of 10-fold cross-validation when training a general classifier across the entire dataset. Using the SLD as the only feature yields an F-score of 85%. Unsurprisingly, using host names as a feature increases the F-score to 90%. This means that even when traffic cannot be decrypted, we can still block 90% of A&T request by blocking DNS requests to the corresponding hosts or by matching the SNI header from TLS handshakes. However, it is possible for hosts to collect tracking data and to also serve content necessary for the functionality of the app. For example, we found that the host `lh3.googleusercontent.com` is often contacted by the *AdMob* library. However, `lh3.googleusercontent.com` is also often used to fetch various Google content, such as images of apps displayed on the Google Play Store. In such a scenario, more features are needed.

To that end, we train a general classifier using the path component of the URL, including query parameters. As shown in Table 6.2, using these features actually decreases the performance of the classifier. Furthermore, the resultant tree size is 500 (the maximum size we allow for our DTs), which is much larger than the 188 non-leaf nodes reported in [33]. We believe the difference in these findings is caused by our larger and more diverse dataset. In addition, we find that training a general classifier on the path component of the URL takes over six hours. This means that to perform 5-fold cross-validation for such a classifier, more than 24 hours are required. Even if training time is not considered an issue as it can be done offline, prediction time is of essence when operating on live network traffic. As was shown in [33], prediction time is closely correlated with the tree size. Specifically, a tree consisting of close to 300 non-leaf nodes can cause a three millisecond delay on every outgoing packet [33]. Our tree of 500 nodes can cause even larger delays that can negatively impact user experience. Because of these shortcomings, we do not train a general classifier on the remaining feature sets from Sec. 6.4.2.1, and instead propose a more scalable approach, as discussed in the next section.

Feature Set	Avg. F-score (%)	Avg. Accuracy (%)	Avg. Specificity (%)	Avg. Recall (%)	Avg. Training Time (ms)	Avg. Tree Size
Domain	93.30 ± 10.16	94.64 ± 6.50	82.96 ± 27.64	94.64 ± 11.59	0.06 ± 0.23	0.93 ± 0.24
Host	93.87 ± 6.91	94.93 ± 5.59	85.14 ± 24.32	93.46 ± 9.93	0.13 ± 0.40	0.97 ± 0.16
Path Component of URL	90.74 ± 9.81	93.45 ± 5.94	83.08 ± 27.65	89.02 ± 14.21	25.24 ± 46.33	8.29 ± 7.34
URL (Host & Path)	93.81 ± 7.44	95.59 ± 5.10	88.04 ± 21.94	92.76 ± 10.87	26.99 ± 52.24	7.81 ± 6.81
URL + Referer + Content Type	94.42 ± 6.39	95.89 ± 4.50	89.61 ± 20.93	93.49 ± 8.04	19.65 ± 36.46	4.98 ± 4.30
URL and HTTP Headers	94.96 ± 6.65	96.54 ± 4.28	90.56 ± 19.97	94.97 ± 7.84	22.14 ± 37.37	3.24 ± 2.65

Table 6.3: Performance of per-app classifiers. Tree size here is the number of non-leaf nodes.

6.4.2.3 Per-App Classifiers

Rather than training one classifier over the entire dataset, we explore the idea of training a classifier per-application. The advantages of such classifiers were discussed in Sec. 4.2.3.3. One problem with both per-app classifiers is that sometimes there is not enough training data. For instance, as shown in Table 6.1, our dataset contains only 143 apps that have at least 10 A&T packets – positive samples. ReCon [2] proposed the general classifier as a means to deal with cases for which no specialized classifier could be trained. However, this method does not solve the problem of long training times of general classifiers. Fortunately, our per-app approach allows us a more flexible solution.

In our dataset, the low number of samples is caused by the inability of Droidbot to adequately exercise certain apps. For example, one app in our dataset, *Extreme City GT Car Stunts*, is a game app written in Unity 3D [94]. Since, as was confirmed in [90], Droidbot struggles with extracting UI elements when Unity is involved, the tool was unable to perform many meaningful actions inside *Extreme City GT Car Stunts*. This led to the collection of just seven negative samples and three positive ones – not enough to train a reasonable classifier. To confirm our hypothesis about Droidbot, we manually exercised the app for five minutes. As expected, the manual interaction generated more packets: eleven negative samples and 226 positive ones. This procedure can be repeated for any app with a low number of samples. In contrast, in the case of per-domain classifiers, it is unclear how to increases samples for a given SLD. In the future, a crowdsourcing platform, such as CHIMP [95] can be used to generate inputs for apps with a low number of samples. We note that certain apps will always have zero positive samples. For instance, in our dataset we had four

apps that contained no A&T libraries and thus they could not generate any A&T packets. Such apps can have their traffic always be allowed by the `AntMonitor` Library.

For the purposes of this Chapter, we skip training classifiers for apps that have less than 10 positive samples and report results on the remaining 143 apps. Specifically, we evaluate each feature set from Sec. 6.4.2.1 with 10-fold cross-validation for each of the 143 apps. In Table 6.3, we report the average and standard deviation for the following metrics: F-score, accuracy, specificity, recall, training time, and tree size (non-leaf nodes). We find that, on average, the per-app classifiers outperform the general one – Table 6.3 vs. Table 6.2. We find that using more features (the full URL and all HTTP headers) yields the best results – an average F-score of close to 95%. Using the full URL with just HTTP `Referer` and `Content Type` headers performs very closely to using the full feature set. This justifies the choice list curators make when designing rules for their lists. Interestingly, using hosts as a single feature also yields a high average F-score of close to 94%. Training on the path component of the URL performs the worst – an average F-score of 90%. This shows that, as expected, the host name is one of the most important features for identifying trackers, and that is why many common filter lists (*e.g.* MoaAB [83]) operate on the host level alone. In addition, we note that training per-app classifiers takes *milliseconds*. This means that if we need to provide a classifier for an app that was not part of our dataset or adapt a classifier for an updated app, we can test the given for five minutes (either with Droidbot or manually) and build a classifier for it, all in under 10 minutes. Finally, we note that the per-app classifiers have an average tree size below 10 – orders of magnitude less than the size of the general classifier described in the previous section. This means that they can be applied in real-time on top of a VPN service (Fig. 6.6) and predict packets within milliseconds, as was shown in Chapter 5.

For the rest of this section, we will focus on the best performing classifier – the one trained on the full URL with all the HTTP headers. Fig. 6.4 shows the Complementary Cumulative Distribution Function (CCDF) of F-scores of our per-app classifiers. We note that 23% of our per-app classifiers achieve F-scores of 100%. 80% of our classifiers reach F-scores 90% or higher, and all of the clas-

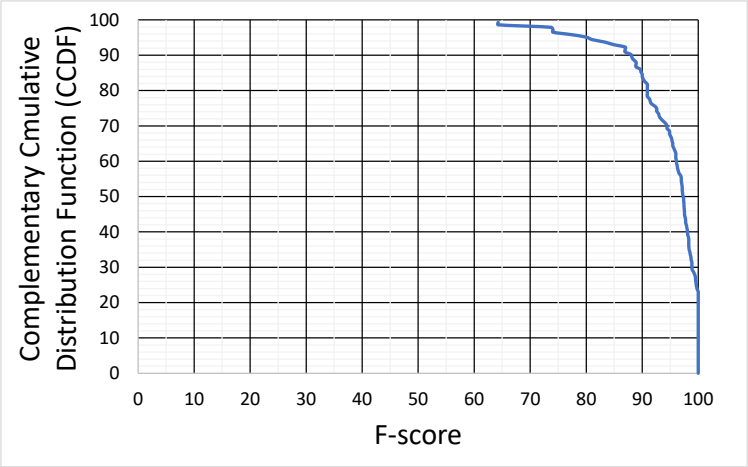


Figure 6.4: Complementary Cumulative Distribution Function (CCDF) of F-scores of the 143 per-app classifiers: 80% of the classifiers reach F-scores of 90% or above.

sifiers achieve an F-score of at least 64%. Next, we examine the trees that our classifiers produce. Fig. 6.5 shows one such tree. We note that the tree first splits on the query key `app_id`. Such keys are often followed by package names of apps from which they are sent. When an app displays an ad, the ad network must know which app has shown an ad so that the corresponding developer can get paid. However, this also leads to user profiling: over time ad networks learn what type of apps a particular user uses. The last feature in the tree, `x-crashlytics-api-client-version` is even more interesting. *Crashlytics* is considered to be a development library (not an A&T library), but it also has the potential to track the user since it is installed across many apps. In fact, some popular ad-blocking filter lists have began including hosts belonging to *Crashlytics* [83]. Fig. 6.5 shows that our classifiers generalize: the depicted classifier has learned to identify *Crashlytics* as a tracking service even though it was not labeled in our ground truth. Therefore, although Table 6.3 shows a high false positive rate (low specificity with high variance), it does not mean that these false positives are actual false positives that will cause app breakage. We explore this in more detail in Sec. 6.5.

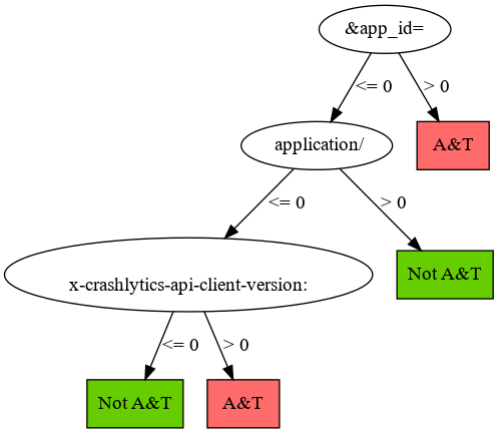


Figure 6.5: An example decision tree of a per-app classifier trained on the full URL and all HTTP headers.

6.4.2.4 Deployment

While the AutoLabel system described in Sec. 6.3 is useful as a research tool, it is not suitable for large-scale adoption by users as it requires root. However, the classifiers that we described in this section can be applied on top of a user-space VPN service, such as `AntMonitor` (Chapter 3), as we did in `AntShield` (Chapter 4) and NoMoAds (Chapter 5). We illustrate this deployment scenario in Fig. 6.6: at the server, we use AutoLabel to test apps and label their traffic, then we train classifiers using the labeled dataset, and finally, we push the classifiers to the mobile device. Since our proposed approach is to train per-app classifiers, the (re)training can be done on a per-app basis, as needed. For example, if an application gets updated, we can re-run AutoLabel and re-train our classifiers within minutes since the labeling is automatic and training classifiers takes milliseconds (see Table 6.3).

6.5 Evaluation Results

To further analyze the quality of our A&T labels, by both AutoLabel and our classifiers, we look into the disagreements between popular anti-tracking filter lists and our labels. We also seek to

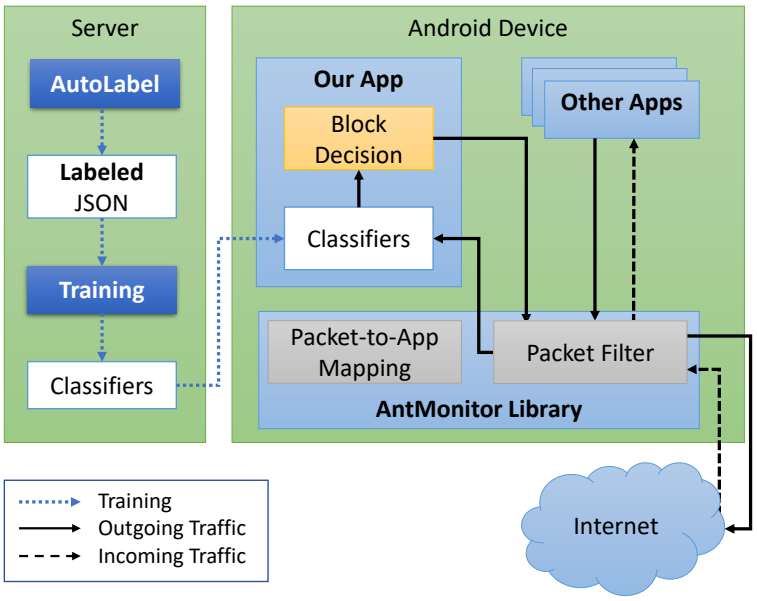


Figure 6.6: End-user system: classifiers are trained offline and are used to predict and block A&T requests intercepted by a VPN service, such as `AntMonitor` (Chapter 3).

understand if our classifiers can generalize and learn to recognize A&T requests that were not part of the training set. For a fair comparison this section of the Chapter focuses on the 24,786 requests that we were able to label with our per-app classifiers, *i.e.* requests belonging to apps with less than 10 positive samples are absent from this analysis. We start by discussing our filter lists-based labeling strategy (Sec. 6.5.1). Next, we evaluate the quality of our labeling methodology by comparing our labels with those of filter lists (Sec. 6.5.2). Finally, we compare the labels provided by filter lists with our prediction results and provide insight onto our false positives and false negatives (Sec. 6.5.3).

6.5.1 Baseline of Comparison

As our baseline of comparison, we use popular ad-blocking and anti-tracking filter lists that are publicly available, and we match them against the requests in our dataset. Specifically, we compare against the filter list used by AdblockPlus – EasyPrivacy, and against the Mother of All Ad-Blocking (MoaAB) [83] list. Although the latter is less popular and is mostly aimed at ads, it was

reported in [35] to be the most effective list for mobile devices: it blocked the most requests to third parties and also successfully blocked many fingerprinting scripts. To match against the filter list rules, we use an open source Python parser – *adblockparser* [96]. The tool can work with any list as long as it is in AdblockPlus format specified in [84]. EasyPrivacy already comes in this format. To convert MoaAB into AdblockPlus format, we follow the guidelines outlined in [84]. Since MoaAB operates by modifying a rooted Android's `hosts` file, the conversion is straightforward: we simply need to write an AdblockPlus rule that blocks all the host names specified in MoaAB. Rules that match against host names are written by appending two pipe symbols in front of the host in question.

Once our lists are ready, we need to parse our collected requests and feed the required information to *adblockparser*. The tool takes in the full URL and several options that are support by AdblockPlus. Key options are: whether the request is a request to a third-party (*e.g.* a site fetching content from a different domain than its own), whether the request is an XML HTTP request (contains the HTTP Header `X-Requested-With: XMLHttpRequest`), and what type of content is being request (*e.g.* an image or an HTML document). We note that AdblockPlus has even more options, however they are not available when operating on a mobile device. For example, the option "websocket" is used to identify requests initiated by `WebSocket` object. Such information is only available when operating within a browser with the ability to hook into various APIs. On a mobile device where we operate on a per-packet basis we can only use the three options described earlier. In fact, the AdblockPlus library for Android [78] uses the exact same three options. To determine the content type of the request, we follow the same logic as written in the AdblockPlus library for Android [78]: we match the requested object from the path component of the URL against file endings. For instance, to determine if the requested file is an image, we match against the following file endings: `.gif`, `.png`, `.jpg`, `.jpeg`, `.bmp`, `.ico`. Determining whether the request is an XML HTTP request is straightforward: we look for the presence of the `X-Requested-With: XMLHttpRequest` HTTP header. Finally, to determine is the request is to a third-party, we compare the origin (defined by the scheme, host, and port) of the URL being requested to the origin of

	Labels by each Method				
	Auto	Filter Lists			
Row	Label	EasyPrivacy	MoaAB	Count (%)	Notes
1	0	0	0	8,077 (32.59%)	negative samples
2	1	0	1	8,279 (33.40%)	agreements with filter lists
3	1	1	1	1,607 (6.48%)	
4	1	1	0	174 (0.70%)	
5	1	0	0	2,960 (11.94%)	disagreements: new A&T samples found
6	0	0	1	3,089 (12.46%)	disagreements: AutoLabel false negatives
7	0	1	1	362 (1.46%)	
8	0	1	0	238 (0.96%)	
Total Requests				24,786 (100%)	

Table 6.4: Comparing AutoLabel to popular anti-tracking filter lists. "0" = negative label; "1" = a positive label (A&T request). Example: row five means that 2,960 requests were detected by AutoLabel, but were not detected by state-of-the-art filter lists (EasyPrivacy and MoaAB).

the URL specified in the HTTP `Referer` header. This methodology allows us to extract the three options needed to feed into *adblockparser* along with the full URL request.

Tables 6.4 and 6.5 summarize the results of matching against EasyPrivacy and MoaAB and comparing the outcome against AutoLabel and our machine learning prediction, respectively. A label of "0" indicates a negative label, and a label of "1" indicates a positive label – an A&T request.

6.5.2 Evaluating AutoLabel

First, we examine the quality of the labels provided by AutoLabel by analyzing the agreements and disagreements between our labels and the filter lists' labels. Out of 24,786 requests, 8,077 (32.6%) were labeled as negative (0 in Table 6.4) by all three approaches (first row).

Agreements with Filter Lists. We begin by analyzing rows two through four which represent agreements between AutoLabel and the filter lists. We find that for over 33% of requests, the filter lists agree with our labels. Since EasyList is less aggressive than MoaAB, there is greater overlap between AutoLabel and MoaAB.

Disagreements with Filter Lists: New A&T Samples Found. Next, we examine the cases where AutoLabel disagreed with the other lists and found extra positive samples. Row five indicates that 2,960 (11.9%) positive samples were identified by our approach, but were undetected by either of the filter lists. We examine the host names contacted by these 2,960 requests and find 150 distinct hosts. Most (559) of these requests were destined to `lh3.googleusercontent.com`. Since MoaAB operates at the host level (vs. the full URL level), it cannot block hosts belonging to the `googleusercontent.com` SLD as it often servers essential content. However, in the case of our 559 requests, they were all sent by either the `com.google.android.gms.ads` or the `com.google.android.gms.internal.ads` package. This highlights the value of going beyond host names and using other features from intercepted requests (Sec. 6.4.2.1). Similarly, 302 of the samples were destined to `fonts.googleapis.com`. Although these fonts are often requested by first-parties, in the case of these 302 samples, they were generated by the *AdMob* library. For closer examination, we randomly selected an application out of the 55 that requested the fonts via *AdMob*. We found that these fonts were requested for display apps. Specifically, they contained a `doubleclick.net` URL in the HTTP `Referer` header. Thus, these are indeed true positives as the fonts were not being requested for a main document. It is possible that MoaAB and other lists do not need to block these as they would never be generated if the ad from `doubleclick.net` was never fetched. This finding illustrates that future work may need to reconstruct the `Referer` chain and exclude requests that would never be generated to avoid training on them. In the case of fonts, such filtering can be critical – we don't want the classifiers to get confused on which `fonts.googleapis.com` to block. Some of the other samples out of our 2,960 positive ones detect hosts that should be included into MoaAB. For example: 278 requests were destined to the `startappservice.com` SLD and *Startapp* is a popular mobile advertising library; 67 requests contacted the `tapjoy.com` SLD and *Tapjoy* is yet another advertising SDK; 22 requests contacted `img.applovin.com` and *AppLovin* is another mobile advertiser.

Disagreements with Filter Lists: False Negatives. Finally, we examine rows six through eight, which represent AutoLabel false negatives. Row six shows that MoaAB found an additional 3,089 (12.5%) positive samples that were undetected by other methods. Most of these requests (589) were going to `graph.facebook.com`, which is a known tracking domain. AutoLabel failed to mark these requests as positive because they were sent by the social library – *Facebook*. Such libraries, as well as various development libraries (*e.g. Unity 3D*), may sometimes need Internet access for legitimate reasons. Distinguishing between legitimate and A&T requests of other (non-A&T) libraries is the current limitation of our work. Another 160 requests (out of the 3,089) were going to `googleads.g.doubleclick.net`. We examined some of the stack traces that led to these request but were not marked to contain an A&T package name. We found that these requests were sent by variations of the obfuscated `com.google.android.gms.internal.-zzabm.zza` package. The parent `com.google.android.gms` package name belongs to *Google Play Services* [97] and is responsible for fetching *AdMob* ads. However, we cannot simply block all requests sent by *Google Play Services* as this library of libraries provides additional services that may be required for proper application functionality. For example, `com.google.-android.gms.maps` provides *Google Maps* APIs for other apps to use. We note that most requests to the `doubleclick.net` SLD are sent by `com.google.android.gms.ads` and `com.google.android.gms.internal.ads` packages. It is difficult to say if the obfuscated package name is indeed the `.ads` package that was able to avoid detection by LibRadar++. In either case, this suggests that future work may benefit from combining static and dynamic analysis to discover more package names in apps. To summarize, the 3,089 samples that were uniquely identified by MoaAB as positive, were responsible for contacting 194 different hosts. Of these hosts, 125 were never identified by AutoLabel. In contrast to MoaAB, EasyPrivacy was able to identify only 238 positive samples that were undetected by AutoLabel (row eight). 362 samples were labeled as positive by both EasyPrivacy and MoaAB (row seven), indicating that there is some overlap between the two lists which is not captured by our methodologies.

	Labels by each Method					
	Our Approaches		Filter Lists			
Row	Prediction	AutoLabel	EasyPrivacy	MoaAB	Count (%)	Notes
1	1	0	0	1	949 (3.83%)	agreements with filter lists
2	1	0	1	0	37 (0.15%)	
3	1	0	1	1	28 (0.11%)	
4	1	0	0	0	799 (3.22%)	disagreements: new A&T samples found
5	0	1	0	1	514 (2.07%)	disagreements: prediction false negatives
6	0	1	1	1	127 (0.51%)	
7	0	1	1	0	21 (0.08%)	
Total Requests					24,786 (100%)	

Table 6.5: Comparing our machine learning predictions to popular anti-tracking filter lists. "0" = negative label; "1" = a positive label (A&T request). Example: row four means that 799 requests were labeled as positive by our classifiers, but were labeled as negative by state-of-the-art filter lists (EasyPrivacy and MoaAB).

Summary. We find that the AutoLabel approach is accurate when labeling requests generated by A&T libraries. The reason our method misses the 125 hosts identified by MoaAB is partly caused by the fact that we do not consider first-party tracking and tracking by social (*e.g. Facebook*) and development (*e.g. Unity 3D*) third-party libraries. Another reason for missing these hosts are package names that avoid detection by LibRadar++. On the other hand, our approach can also provide suggestions for filter lists – it detected 150 hosts some of which should be included ad-blocking or anti-tracking filter lists.

6.5.3 Prediction Evaluation

In this section, we evaluate the performance of our classifiers by examining the agreements and disagreements between our predictions, AutoLabel, and filter lists. To that end, in Table 6.5, we extend Table 6.4 to include a column for prediction.

Agreements with Filter Lists. If we examine Table 6.3 alone we will find an alarmingly high false positive rate: specificity below 91% and with high variance between different per-app classifiers. In total, out of 11,766 samples labeled as negative by AutoLabel, 1,813 (over 15%) are predicted as positive. However, Table 6.5 shows that 949 of those "false positives" are also labeled

as true positives by MoaAB (row one). Similarly, 37 of the samples are labeled as positive by EasyPrivacy, and 28 are labeled as positive by both MoaAB and EasyPrivacy. Therefore, a total of 1,014 samples (over 55%) out of 1,813 "false positive" ones are actually true positives that were missed by AutoLabel for reasons discussed in the previous section. This finding illustrates that our classifiers can pick up patterns that extend past the used training data.

Disagreements with Filter Lists: New A&T Samples Found. Next, we examine the remaining 799 false positives that were not detected by either filter list – row four in Table 6.5. We found that 95 of these samples were destined to `api.jaumo.com` – a dating service. Since the application sending this request was a dating app, these 95 sample are indeed false positives that could lead to app breakage. Examining the tree built for the dating app, we found that it contained only one decision node which marked all requests containing the word "Accept-Language" as positive. This indicates that we can further improve our feature selection process by eliminating common HTTP headers. Other samples (of the 799) include 115 requests that were contacting the `unity3d.com` SLD. It is likely that these requests were generated by the *Unity 3D* development library. Unfortunately, we found that Frida was unable to provide the Java stack trace leading to these requests. Addressing this limitation is outside the scope of this thesis and we hope that future releases of Frida will be able to provide the needed stack traces. If the 115 requests were indeed sent by *Unity 3D*, it is possible they were true tracking requests as *Unity* also has a *Unity Ads* library. In total, we found that the 799 false positive samples contacted 116 different hosts. Some of these identified hosts should be considered for inclusion in filter lists. For example, we found 61 requests going to the `tapas.net` SLD. Some of the hosts under this SLD are identified as advertisers by the VirusTotal [98] URL categorization service. Other examples include 31 requests going to the `ytimg.com` (an advertising SLD owned by *YouTube*), 28 requests going to `app-measurement.com` (another tracking service), and 19 requests going to `settings.crashlytics.com` (*Crashlytics* is considered a development library for crash-reporting library, but it can also engage in tracking activities).

Disagreements with Filter Lists: False Negatives. Rows five through seven show that our machine learning approach has 662 false negatives. These false negatives belonged to 56 distinct apps – therefore only 56 per-app classifiers (out of 143) were suffering from false negatives. We examined the apps in question and found that most of the false negatives were clustered in a few apps. The app with the most (115) false negatives was *Drum Pad Machine - Make Beats*. We examined the classifier tree generated for this app and found that its size was above average – 11 non-leaf nodes. As with some of the examples we discussed earlier, the tree contained some common HTTP headers which could have been the cause of confusion. The same problem plagued app responsible for next most (74) false negatives. With better feature selection these false negatives should disappear. We leave such feature engineering to future work.

Summary. We find that most of our false positives are actually false negatives in the AutoLabel labeling mechanism. This finding is similar to the reports provided by [41]: machine learning algorithms trained with popular filter lists result in "false positives" that highlight false negatives in the ground truth data. Our findings illustrate that our classifiers can generalize past the ground truth that was fed to them and can find new A&T hosts. In terms of false negatives, our classifiers can be further improved with better feature selection.

6.6 Summary

In this Chapter, we presented a system that can automatically label outgoing mobile network packets with the A&T (advertising or analytics/tracking) library that was responsible for generating them. Using our labeling system we collect a large dataset of mobile advertising and tracking. We use our dataset to train machine learning classifiers that can detect A&T requests on mobile devices in real-time using a VPN interception service, such as `AntMonitor` (Chapter 3). We have shown that our classifiers can achieve high F-scores of 94% on average, and they can be trained in the orders of milliseconds. Finally, using popular filter lists (EasyList and MoaAB), we have shown that

our classifiers generalize past the ground truth that they have been trained on: they find tracking hosts that were not labeled by our ground truth but are present in the filter lists. To enable further research, we will release our labeled dataset and will open-source our data collection and labeling systems. We will also contribute back to the NoMoAds project [33] with our improvements on classifier training.

Chapter 7

Conclusion

7.1 Summary

The usage of mobile devices has been rapidly increasing over the last few years. People now spend more time on their smartphones as opposed to their desktop computers [99]. Furthermore, when users interact with their mobile device, they spend over 80% of their time on apps as opposed to browsers [99]. Unfortunately, the mobile apps ecosystem is rife with abuses. Mobile phones contain a multitude of PII and applications often transmit this sensitive data to third-party servers for tracking and advertising purposes. Although certain PII are protected by permissions, the current permissions model does not separate the app and the third-party libraries contained within it. Thus, whatever permissions the app is granted, the same permission is also granted to any library the app includes. This provides third-parties with an opportunity to track user activities across different apps.

In this thesis, we proposed several tools to provide mobile device users with transparency and control over their data. We started by developing and efficient packet interception system, called `AntMonitor`, that runs on the mobile device without requiring root privileges (Chapter 3). Specif-

ically, we leverage Android VPN APIs and we perform layer-3 to layer-4 translation to avoid routing traffic through a VPN server. Making such a system run efficiently on a mobile device poses various challenges which we solve with various optimization techniques. To allow other researchers to take advantage of our optimizations, we made `AntMonitor` as a ready-to-use Android library and have made it open-source at [10]. In subsequent Chapters, we also presented various applications built on top of `AntMonitor`.

In Chapter 4, we presented a system for detecting blocking PII exposures. We started by noting that some PII are available to all apps (including `AntMonitor`) via Android APIs. For such PII, we proposed using DPI to detect it in outgoing network traffic. For PII that is unknown to our system, we proposed a machine learning approach. Specifically, we collected a dataset of over 47 thousand packets from 400 most popular free apps. Using `AntMonitor`'s DPI capability, we automatically labeled each packet with the type of PII that they contained. Since we used test accounts, we were able to have `AntMonitor` search for PII that are normally unavailable through Android APIs, such as usernames and passwords. Using the labeled data, we trained multi-label machine learning classifiers for predicting the type of PII a given packet contains. Our classifiers are able to achieve F-scores of over 90% and can predict PII in real-time on the device. To encourage reproducibility, we have made our dataset and code available at [10].

In Chapter 5, we presented a system, called NoMoAds, for blocking requests for ads. We started by integrating `AntMonitor` with the AdblockPlus Library to match outgoing URL requests against the list of rules in EasyList. Next, we manually interacted with 50 most popular free apps that serve ads. If, during our interaction, an ad was shown, we created new rules to add to EasyList and block packets responsible for the residue ads. Using our custom rules and EasyList we labeled over 15 thousand packets and found that close five thousand packets contained requests for ads. We used our labeled dataset to train a classifier for predicting which packets contain requests for ads. To the best of our knowledge, NoMoAds is the first mobile ad-blocker to effectively and efficiently block ads served across all apps using a machine learning approach. Our dataset and code are available

at [11].

In Chapter 6, we presented AutoLabel – a complete system for detecting packets that are generated by third-party libraries. We are particularly interested in identifying packets generated by A&T libraries, which is the bottleneck in blocking ads and trackers today. Using a combination of dynamic instrumentation and static analysis, AutoLabel addresses this bottleneck. AutoLabel can be used to automatically label datasets, which can in turn be used to train machine learning classifiers to detect A&T packets. In other words, there is no longer a need to maintain lengthy filter lists. To the best of our knowledge, our system is the first to provide this automatic ground truth labeling. We used our system to collect a dataset of over 37 thousand packets from 307 popular free apps. Next, we used our dataset to train machine learning classifiers for predicting A&T packets. We showed that both our ground-truth labeling mechanism and our classifiers are in agreement with popular filter lists and can also find new A&T services. To enable future research on mobile tracking, we will release our collected dataset and we will open-source AutoLabel.

7.2 Future Directions

Over the last few years, concern over online privacy has led to several new laws. For example, in Europe, the General Data Protection Regulation (GDPR) [100] has come into effect in 2018. The law restricts how companies can collect data and requires full transparency over the data collection. Similarly, in 2020, the California Consumer Privacy Act (CCPA) [101] will become effective, enhancing privacy rights of California residents. While these laws will hopefully decrease the amount of personal data collection and tracking, tools such as `AntMonitor` will still be needed to check and enforce compliance.

In addition, as data collection becomes more regulated, advertisers and publishers will need to come up with new ways to support online content and mobile apps. One idea is to use privacy-

preserving techniques when collecting data for personalized ads purposes. For example, as of 2019, Google is already using federated learning to train machine learning classifiers for predictive typing without collecting personal data [102]. The same idea can be extended to personalized ads. Another approach is to re-define the advertising industry. For instance, the Brave browser [93] is exploring the idea of blockchain-based advertising [103] where publishers are rewarded based on user's attention while browsing. Users' attention is monitored within the browser only, and their private data never leaves the device. As the advertising industry adapts, the need for ad-blockers and anti-tracking tools may decrease. However, in order for the industry to change, the research community needs to maintain the pressure on advertisers by continuously developing innovative solutions to ad-blocking and anti-tracking, building on top of and extending tools such as NoMoAds and AutoLabel.

Finally, in the years to come, Android APIs may change and may restrict functionalities that `AntMonitor` and its derivatives rely on. For example, apps that target Android 7.0, by default, do not trust user-added certificates [104]. This impairs `AntMonitor`'s ability to decrypt and analyze TLS traffic. Although it is possible to perform limited analysis on various TLS features (*e.g.* the SNI header, supported algorithms, *etc.*), TLS 1.3 encrypts all handshake messages after the ServerHello [105], and, as of 2018, Cloudfare has started supporting encrypted SNI (ESNI) [106]. While these measures provide users with extra security, they limit the type of analysis that can be done with tools such as `AntMonitor`. Nevertheless, users should be allowed control over their data and the community needs to develop a solution that provides users with both control and security. For instance, on browsers, users can install extensions that provide an extra layer of control (*e.g.* ad-blocking, cookie-blocking, *etc.*). A similar solution can be developed for mobile devices. In fact, Facebook has already taken a step in this direction by allowing users to change their apps' (Facebook, Messenger, Instagram) settings to allow user installed Certificate Authorities [107].

Bibliography

[1] "AppBrain," https://www.appbrain.com/stats/libraries/ad, (accessed Nov. 2017).

[2] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, "ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic," in *Proc. 13th Annu. Int. Conf. on Mobile Systems, Applications, and Services*, vol. 16, New York, NY, USA, 2016.

[3] "EasyList," https://easylist.to/, (accessed Feb. 2019).

[4] "AdAway hosts," https://adaway.org/hosts.txt, (accessed Jan. 2017).

[5] "hpHosts," https://hosts-file.net/ad_servers.txt, (accessed Nov. 2017).

[6] "Digital in 2019," https://wearesocial.com/uk/digital-2019, (accessed Mar. 2019).

[7] "VNI Mobile Forecast Highlights Tool," https://www.cisco.com/c/m/en_us/solutions/service-provider/forecast-highlights-mobile.html, (accessed Mar. 2019).

[8] "Android VpnService," http://goo.gl/kV7ZZL, (accessed Feb. 2019).

[9] "iOS NetworkExtension," https://developer.apple.com/documentation/networkextension, (accessed Feb. 2019).

[10] "AntMonitor Open Source," https://athinagroup.eng.uci.edu/projects/antmonitor/, (accessed Mar. 2019).

[11] "NoMoAds Data and Code," http://athinagroup.eng.uci.edu/projects/nomoads/, (accessed Mar. 2019).

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Real-Time Privacy Monitoring on Smartphones," *ACM Transactions Computer Systems*, vol. 32, no. 2, p. 5, 2014.

[13] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, ""These Arent the Droids Youre Looking For": Retrofitting Android to Protect Data from Imperious Applications," in *Proc. 18th ACM Conf. on Computer and Communications Security*, Chicago, IL, USA, Oct. 2011, pp. 639–652.

[14] S. Chitkara, N. Gothoskar, S. Harish, J. I. Hong, and Y. Agarwal, "Does this App Really Need My Location?: Context-Aware Privacy Management for Smartphones," in *Proc. ACM Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, 2017, p. 42.

[15] "Xposed Module Repository," https://repo.xposed.info/module/de.robv.android.xposed.installer, (accessed Feb. 2019).

[16] "Frida," https://www.frida.re/, (accessed Feb. 2019).

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[18] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information Flow Analysis of Android Applications in DroidSafe," in *Proc. Network and Distributed System Security Symp.*, vol. 15, San Diego, CA, USA, Feb. 2015, p. 110.

[19] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale," in *Proc. Int. Conf. Trust and Trustworthy Computing*, Vienna, Austria, Jun. 2012, pp. 291–307.

[20] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps," in *Proc. 38th Int. Conf. Software Engineering Companion*, Austin, TX, USA, May 2016, pp. 653–656.

[21] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe Exposure Analysis of Mobile In-App Advertisements," in *Proc. 5th ACM Conf. Security and Privacy in Wireless and Mobile Networks*, Tucson, AZ, USA, Apr. 2012, pp. 101–112.

[22] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps," in *Proc. 13th Annu. Int. Conf. Mobile Systems, Applications, and Services*, Florence, Italy, May 2015, pp. 89–103.

[23] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications," in *Proc. 5th ACM Conf. Data and Application Security and Privacy*, San Antonio, TX, USA, Mar. 2015, pp. 37–48.

[24] M. Ikram, N. Vallina-Rodriguez, S. Seneviratne, M. A. Kaafar, and V. Paxson, "An Analysis of the Privacy and Security Risks of Android VPN Permission-enabled Apps," in *Proc. 2016 Internet Measurement Conf.*, Santa Monica, CA, USA, Nov. 2016, pp. 349–364.

[25] 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," Northeastern University, Tech. Rep., Dec. 2013.

[26] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou, "AntMonitor: A System for Monitoring from Mobile Devices," in *Proc. SIGCOMM Workshop Crowdsourcing and Crowdsharing of Big Internet Data*, London, UK, Aug. 2015.

[27] "tPacketCapture," www.taosoftware.co.jp/en/android/packetcapture, (accessed Mar. 2019).

[28] 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:1510.01419v1*, Oct. 2015.

[29] Y. Song and U. Hengartner, "PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices," in *Proc. 5th Annual ACM CCS Workshop Security and Privacy in Smartphones and Mobile Devices*, Denver, CO, USA, Feb. 2015, pp. 15–26.

[30] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[31] "DNS-based Host Blocker for Android," https://github.com/julian-klode/dns66, (accessed Nov. 2017).

[32] "AdGuard for Android," https://adguard.com/en/adguard-android/overview.html, (accessed May 2019).

[33] A. Shuba, A. Markopoulou, and Z. Shafiq, "NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking," vol. 2018, no. 4, Barcelona, Spain, Jul. 2018.

[34] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, and C. K. P. Gill, "Apps, Trackers, Privacy, and Regulators," in *Proc. Network and Distributed System Security Symp.*, vol. 2018, San Diego, CA, USA, Feb. 2018.

[35] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl, "Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools," in *Proc. 2017 IEEE European Security and Privacy*, Paris, France, Apr. 2017, pp. 319–333.

[36] "Privacy Badger," https://www.eff.org/privacybadger, (accessed May 2019).

[37] "Safari," https://www.apple.com/safari/, (accessed May 2019).

[38] S. Bhagavatula, C. Dunn, C. Kanich, M. Gupta, and B. Ziebart, "Leveraging Machine Learning to Improve Unwanted Resource Filtering," in *Proc. 2014 Workshop Artificial Intelligent and Security Workshop*, Scottsdale, AZ, USA, Nov. 2014, pp. 95–102.

[39] J. Bau, J. Mayer, H. Paskov, and J. C. Mitchell, "A Promising Direction for Web Tracking Countermeasures," in *Proc. of W2SP*, San Francisco, CA, USA, May 2013.

[40] D. Gugelmann, M. Happe, B. Ager, and V. Lenders, "An Automated Approach for Complementing Ad Blockers Blacklists," vol. 2015, no. 2, Philadelphia, PA, USA, Jun. 2015, pp. 282–298.

[41] U. Iqbal, Z. Shafiq, P. Snyder, S. Zhu, Z. Qian, and B. Livshits, "AdGraph: A Machine Learning Approach to Automatic and Effective Adblocking," *arXiv preprint arXiv:1805.09155*, 2018.

[42] A. Shuba, A. Le, E. Alimpertis, M. Gjoka, and A. Markopoulou, "AntMonitor: System and Applications," *arXiv preprint arXiv:1611.04268*, 2016.

[43] A. Shuba, "AntMonitor: A System for Mobile Network Monitoring," Master's thesis, Dept. Elect. Eng. and Comp. Sci., UC Irvine, Irvine, CA, USA, 2016.

[44] "The pcap Next Generation Capture File Format," https://github.com/pcapng/pcapng/, (accessed Feb. 2019).

[45] "Android Versions," https://developer.android.com/about/dashboards, (accessed Feb. 2019).

[46] "strongSwan VPN Client," https://play.google.com/store/apps/details?id=org.strongswan.android, (accessed Feb. 2019).

[47] "Secure Android Proxy," https://code.google.com/archive/p/sandrop/, (accessed Feb. 2019).

[48] "Google, Android ToyVPN Example," https://android.googlesource.com/platform/development/+/master/samples/ToyVpn/src/com/example/android/toyvpn/ToyVpnService.java, (accessed Mar. 2019).

[49] A. V. Aho and M. J. Corasick, "Efficient String Matching: an Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[50] K. Kanani, "MultiFast," http://multifast.sourceforge.net/, (accessed May 2019).

[51] "UTF-8 and UTF-16 Strings," http://developer.android.com/training/articles/perf-jni.html#UTF_8_and_UTF_16_strings, (accessed Mar. 2019).

[52] "Android BatteryManager API," http://developer.android.com/reference/android/os/BatteryManager.html, (accessed Mar. 2019).

[53] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson, "Haystack: A multi-purpose mobile vantage point in user space," *arXiv preprint arXiv:1510.01419v3*, Oct. 2016.

[54] "AntShield Dataset," https://athinagroup.eng.uci.edu/projects/antmonitor/antshield-dataset/, (accessed Mar. 2019).

[55] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas, "Mulan: A Java Library for Multi-Label Learning," *JMLR*, vol. 12, pp. 2411–2414, 2011.

[56] G. Tsoumakas and I. Katakis, "Multi-Label Classification: An Overview," *IJDWM*, vol. 3, no. 3, 2006.

[57] "Top-Level Domain Zone File Information," https://www.verisign.com/channel-resources/domain-registry-products/zone-file/index.xhtml, (accessed Feb. 2019).

[58] "Spotlight on Consumer App Usage," https://www.appannie.com/insights/market-data/global-consumer-app-usage-data/, (accessed Feb. 2019).

[59] "tldextract," https://pypi.org/project/tldextract/, (accessed Mar. 2019).

[60] "App annie," https://www.appannie.com, (accessed Mar. 2019).

[61] "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html, (accessed Mar. 2019).

[62] S. Godbole and S. Sarawagi, "Discriminative Methods for Multi-Labeled Classification," in *PAKDD*, 2004, pp. 22–30.

[63] D. G. Goldstein, R. P. McAfee, and S. Suri, "The Cost of Annoying Ads," in *Proc. 2013 World Wide Web Conf.*, Rio de Janeiro, Brazil, May 2013, pp. 459–470.

[64] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft, "Breaking for Commercials: Characterizing Mobile Advertising," in *Proc. 2012 Internet Measurement Conf.*, Boston, MA, USA, Nov. 2012, pp. 343–356.

[65] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee, "The Price of Free: Privacy Leakage in Personalized Mobile In-App Ads," in *Proc. Network and Distributed System Security Symp.*, San Diego, CA, USA, Feb. 2016.

[66] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, "The Dark Alleys of Madison Avenue: Understanding Malicious Advertisements," in *Proc. 2014 Internet Measurement Conf.*, Vancouver, BC, Canada, Nov. 2014, pp. 373–380.

[67] "Adblock Browser," https://adblockbrowser.org/, (accessed May 2017).

[68] "UC Browser," https://play.google.com/store/apps/details?id=com.UCMobile.intl, (accessed Nov. 2017).

[69] "PageFair. The state of the blocked web – 2017 Global Adblock Report," https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf, 2017.

[70] J. Hercher, "Mobile Ad Blocking Takes Off In Asia, Sparked By User Data Costs," https://adexchanger.com/mobile/mobile-ad-blocking-takes-off-asia-sparked-user-data-costs/, (accessed May 2019).

[71] A. Hern, "A proxy war: Apple ad-blocking software scares publishers but rival Google is target," https://www.theguardian.com/technology/2016/jan/01/publishers-apple-ad-blockers-target-google/, 2016, (accessed May 2019).

[72] "Adblock Plus for Android," https://adblockplus.org/en/android-about, (accessed May 2017).

[73] M. Ikram and M. A. Kaafar, "A First Look at Mobile Ad-Blocking Apps," in *Proc. 2017 IEEE 16th Int. Symp. Network Computing and Applications*, Cambridge, MA, USA, 2017, pp. 1–8.

[74] W. Palant, "Adblock Plus for Android Removed from Google Play Store," https://adblockplus.org/blog/adblock-plus-for-android-removed-from-google-play-store, 2013, (accessed May 2017).

[75] B. Williams, "Adblock Plus and (a little) more," https://adblockplus.org/blog/five-and-oh-look-another-lawsuit-upholds-users-rights-online, 2016, (accessed May 2019).

[76] "Disconnect," https://disconnect.me/, (accessed Mar. 2018).

[77] U. Iqbal, Z. Shafiq, and Z. Qian, "The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists," in *Proc. 2017 Internet Measurement Conf.*, London, UK, Nov. 2017.

[78] "Adblock Plus Library for Android," https://github.com/adblockplus/libadblockplus-android, (accessed Nov. 2017).

[79] "Wireshark," https://www.wireshark.org/, (accessed May 2019).

[80] "AdMob," https://www.google.com/admob/, (accessed Mar. 2018).

[81] "MoPub," https://www.mopub.com, (accessed Mar. 2018).

[82] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu, "Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets," in *Proc. 2018 Internet Measurement Conf.*, Boston, MA, USA, Oct. 2018, pp. 293–307.

[83] "Moaab: Mother of all ad-blocking," https://forum.xda-developers.com/showthread.php?t=1916098, (accessed Feb. 2019).

[84] "Writing Adblock Plus Filters," https://adblockplus.org/filters, (accessed Feb. 2019).

[85] "LiteRadar," https://github.com/pkumza/LiteRadar, (accessed Feb. 2019).

[86] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: a Lightweight UI-guided Test Input Generator for Android," in *Proc. 2017 IEEE/ACM 39th Int. Conf. Software Engineering Companion*, Buenos Aires, Argentina, 2017, pp. 23–26.

[87] "NowSecure," https://www.nowsecure.com/, (accessed Feb. 2019).

[88] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill, "Studying TLS Usage in Android Apps," in *Proc. 13th ACM Int. Conf. Emerging Networking Experiments and Technologies*, Seoul/Incheon, South Korea, Dec. 2017, pp. 350–362.

[89] "Android Accessibility API," https://developer.android.com/guide/topics/ui/accessibility/, (accessed Feb. 2019).

[90] H. Jin, M. Liu, K. Dodhia, Y. Li, G. Srivastava, M. Fredrikson, Y. Agarwal, and J. I. Hong, "Why Are They Collecting My Data?: Inferring the Purposes of Network Traffic in Mobile Apps," vol. 2, no. 4, 2018, p. 173.

[91] "Google Play Unofficial Python API," https://github.com/NoMore201/googleplay-api, (accessed Feb. 2019).

[92] "Unity Ads," https://unity.com/solutions/unity-ads/, (accessed Feb. 2019).

[93] "Brave browser," https://brave.com/, (accessed Feb. 2019).

[94] "Unity 3D," https://www.unity3d.com/, (accessed Feb. 2019).

[95] M. Almeida, M. Bilal, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Varvello, and J. Blackburn, "CHIMP: Crowdsourcing Human Inputs for Mobile Phones," in *Proc. 2018 World Wide Web Conf.*, Lyon, France, Apr. 2018, pp. 45–54.

[96] "adblockparser," https://github.com/scrapinghub/adblockparser, (accessed Feb. 2019).

[97] "Google APIs for Android," https://developers.google.com/android/reference/packages, (accessed Feb. 2019).

[98] "VirusTotal," https://www.virustotal.com/, (accessed Feb. 2019).

[99] Comscore, "Global Digital Future in Focus," Tech. Rep., 2018.

[100] "General Data Protection Regulation," https://eugdpr.org/, (accessed May 2019).

[101] "California Consumer Privacy Act," https://www.caprivacy.org/, (accessed May 2019).

[102] E. Miraglia, "Privacy that Works for Everyone," https://www.blog.google/technology/safety-security/privacy-everyone-io/, 2019, (accessed May 2019).

[103] "Basic Attention Token," https://basicattentiontoken.org/, (accessed May 2019).

[104] "Android 7.0 for Developers," https://developer.android.com/about/versions/nougat/android-7.0, (accessed May 2019).

[105] "The Transport Layer Security (TLS) Protocol Version 1.3," https://tools.ietf.org/html/rfc8446, (accessed May 2019).

[106] "Encrypt that SNI: Firefox edition," https://blog.cloudflare.com/encrypt-that-sni-firefox-edition/, (accessed May 2019).

[107] "Facebook: Researcher Settings," https://www.facebook.com/whitehat/researcher-settings/help, (accessed May 2019).