Anastasia Shuba*, Athina Markopoulou, and Zubair Shafiq

NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking

Abstract: Although advertising is a popular strategy for mobile app monetization, it is often desirable to block ads in order to improve usability, performance, privacy, and security. In this paper, we propose NoMoAds to block ads served by any app on a mobile device. NoMoAds leverages the network interface as a universal vantage point: it can intercept, inspect, and block outgoing packets from all apps on a mobile device. NoMoAds extracts features from packet headers and/or payload to train machine learning classifiers for detecting ad requests. To evaluate NoMoAds, we collect and label a new dataset using both EasyList and manually created rules. We show that NoMoAds is effective: it achieves an F-score of up to 97.8% and performs well when deployed in the wild. Furthermore, NoMoAds is able to detect mobile ads that are missed by EasyList (more than one-third of ads in our dataset). We also show that NoMoAds is efficient: it performs ad classification on a per-packet basis in real-time. 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.

Keywords: ad-blocker; machine learning; mobile; privacy

DOI 10.1515/popets-2018-0035
Received 2018-02-28; revised 2018-06-15; accepted 2018-06-16.

1 Introduction

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 [2]. Second, mobile ads consume significant energy and data resources [3]. Third, third-party mobile ad libraries have been reported to leak private information without explicit permission from app developers or users [4]. Finally, there have been reports of malware spreading through advertising in mobile apps [5]. 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 [6] and UC Browser by Alibaba [7] 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 [8] 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 paper, we propose NoMoAds to effectively and efficiently 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 Ea-

***Corresponding Author: Anastasia Shuba:** University of California, Irvine, E-mail: ashuba@uci.edu
Athina Markopoulou: University of California, Irvine, E-mail: athina@uci.edu
Zubair Shafiq: University of Iowa, E-mail: zubair-shafiq@uiowa.edu

syList 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 `http://athinagroup.eng.uci.edu/projects/nomoads/`.

The rest of this paper is organized as follows. Section 2 discusses the background and prior work related to mobile ad-blocking. Section 3 describes NoMoAds' design and implementation. Section 4 describes our data collection and ground truth labeling procedure. Section 5 evaluates NoMoAds in terms of effectiveness and efficiency and compares it to state-of-the-art filtering approaches. Section 6 concludes the paper and outlines directions for future work.

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 [9], 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 [10]. 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 [10]. Moreover, Apple itself began offering ad-blocking features within their Safari browser since iOS9 [11]. As we discuss next, mobile ad-blocking is fundamentally more challenging as compared to desktop ad-blocking.

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 [12]. 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 [13]. Recent work on leveraging VPNs for mobile traffic monitoring has considered interception in the middle of the network (*e.g.,* ReCon [14]) as well as directly on the mobile device (*e.g.,* AntMonitor [15], Lumen [16]), primarily for the purpose of detecting privacy leaks and only secondarily for ad-blocking [3, 17].

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 [18]. 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 [19]: 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.

Efficient Traffic Interception. While a mobile app can intercept network traffic from all other apps in user-space by setting up a VPN, it is challenging to efficiently analyze packet headers and/or payload to block ads. Ad-blocking typically operates by filtering URLs pointing to advertising domains. Given limited battery and processing capabilities of mobile devices, it is particularly challenging to open up network traffic headers to inspect URLs of every packet from all apps. As compared to remote traffic interception (through a VPN server), local (on-device) mobile traffic interception provides local context but needs to be done efficiently due to the limited CPU, memory, and battery resources on the mobile device. We build on AntMonitor [15], a system for analyzing network traffic at the mobile device, to efficiently implement a cross-app mobile ad-blocker.

Avoiding Blacklists. Desktop ad-blockers rely on manually curated filter lists consisting of regular expressions such as the ones depicted in Tables 2 and 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 [13, 20]. More than half of these apps rely on publicly-maintained lists such as EasyList and some rely on customized filter lists. In addition, cross-app ad-blockers that are not allowed on the Google Play Store, such as DNS66 [21] and Disconnect [22], 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 [23]. Furthermore, they contain obsolete filter rules that are redundant, which results in undesirable performance overheads on mobile devices.

2.2 Related Work

In this section, we survey the most closely related literature to this paper. Bhagavatula et al. [24] 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. [25] also used machine learning to identify tracking domains within the web ecosystem. Later, Gugelmann et al. [26] 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. Rodriguez et al. [17] leveraged graph analysis to discover 219 mobile ad and tracking services that were unreported by EasyList. They identified third-party domains by noting the domains contacted by more than one app, and then inspected each third party domain's landing page for certain keywords that would mark it as an ad or tracking service. In a similar fashion, PrivacyBadger [27] learns which domains are potential trackers by analyzing the number of web pages a certain domain appears on. Going a step further, to avoid broken pages for cases where domains are multi-purposed (*i.e.,* both functional and tracking), PrivacyBadger only blocks cookies belonging to such domains.

Compared to prior work, our approach trains per-packet classifiers (thus maintaining less state than per-flow) to detect ad-requesting packets in mobile traffic. By efficiently analyzing full packets, as we discuss later, our approach can make use of more information than just flow-level or URL-based features. To the best of our knowledge, prior research is lacking an effective approach to automatically detect ads directly on the mobile device.

Aside from inspecting network traffic, there have been other approaches for blocking ads on Android devices. For instance, PEDAL [28] decompiles applications and trains classifiers to distinguish the bytecode of apps from that of ad-libraries. However, static analysis and code re-writing can lead to unusable apps (*e.g.,* due to broken third party functionality), and cannot deal with native code. Modifications to the Android Operating System (OS) have also been proposed to mitigate privacy exposure to ad libraries (*e.g.,* AdDroid [29]). However, OS modification is not suitable for mass adoption as most users are not comfortable with the complex procedure of changing their mobile OS. In the future, we plan to build on the OS modification approach to automatically label ads in packet traces, which can then be used as ground truth to train our machine learning classifiers.

3 The NoMoAds Approach

Fig. 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. 3.1). To obtain the ground truth, we match packets with blacklists (Sec. 3.2.1), log the labeled packets, and then upload them to the server for training (Sec. 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. 3.3.2).

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 [15], as described next.

Packet Interception. We make the design choice to intercept packets at the network layer, because it provides a universal vantage point to traffic from all mobile apps and allows us to build a cross-app mobile ad-blocker. We leverage our prior work on the AntMonitor Library [15], which is a lightweight VPN-based tool for monitoring packets coming in and out of the device. The AntMonitor Library has the following desirable properties: it operates in real-time at user-space, without the need for root access, proxy or special configuration. As shown in Fig. 1, we use the `acceptIPDatagram`

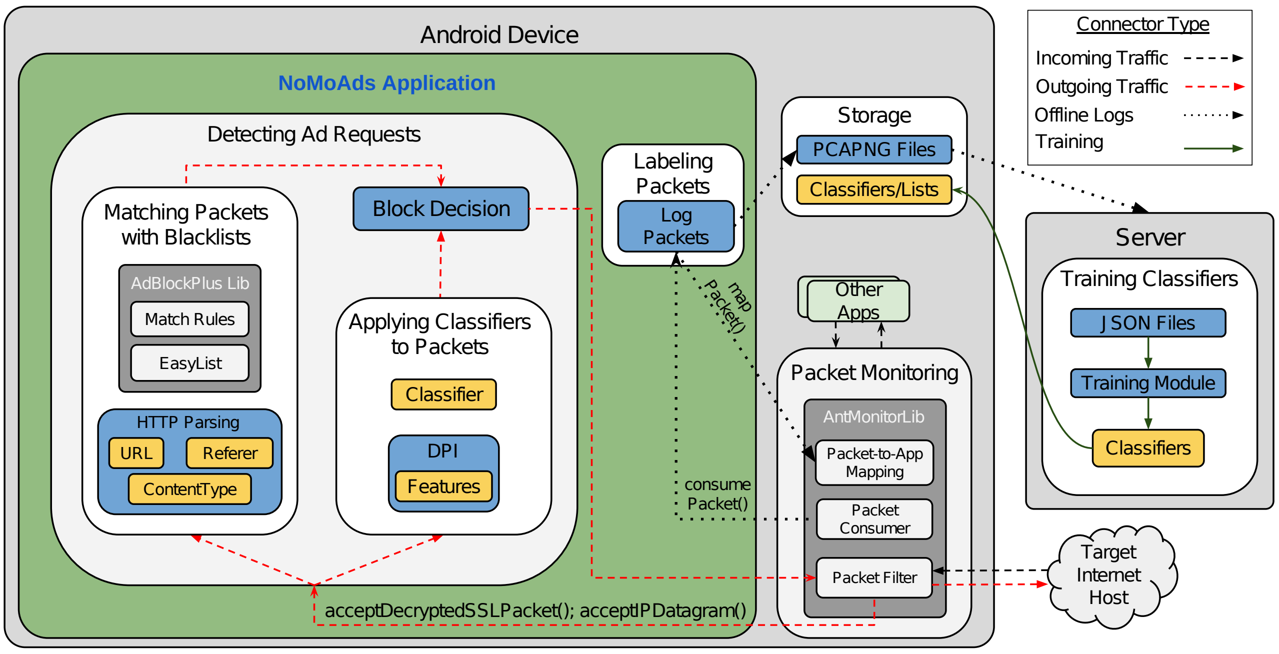

Fig. 1. The NoMoAds system: it consists of the NoMoAds application on the device and a remote server used to (re)train classifiers. The app uses the AntMonitor Library to intercept, inspect, and save captured packets. All outgoing packets are analyzed for ads either by the AdblockPlus Library or by our 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 paper.

and `acceptDecryptedTLSPacket` 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. Note that certificate pinning is currently not widely deployed: Oltrogge et al. [30] reported that only 45 out of 639,283 mobile apps employ certificate pinning.

Packet Analysis. Given a list of strings to search for, the AntMonitor Library can perform Deep Packet Inspection (DPI) within one millisecond (ms) per packet (see [15]). 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.

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. 3.2.1) and the proposed machine learning-based approach taken by NoMoAds (Sec. 3.2.2).

3.2.1 Blacklists

According to a recent survey [13], mobile ad-blocking apps on the Google Play Store rely on blacklists to block ads [13]. 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 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. 3.
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 headers and payload to more accurately detect ads in mobile traffic.

In this work, we used EasyList (the most popular publicly-maintained blacklist [13]) as (i) a baseline for comparison against our proposed learning approach – see Section 5.1, and for (ii) partially labeling packets as containing ads or not – see Section 4. In order to match packets against EasyList, we incorporated the open source AdblockPlus Library for Android [31] into NoMoAds, as shown in Fig. 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.

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.

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. 3.2.2).

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. 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. The full discussion of PII is out of the scope of this paper, and we refer the reader to [15] and [14] for details. Within the NoMoAds system, we use the AntMonitor Library's capability to find PII and label our packets accordingly.

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, in our past experience this model has performed well in terms of accuracy, training and prediction time [32, 33]. 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.

During training, we adopt a bag-of-words model to extract features from a given packet. This approach has been used in the past, *e.g.,* by ReCon [14], as a general way to detect PII leakage. We adapt this idea for ads and learn which specific words are useful features when it comes to predicting ad traffic.

In particular, we break the packet into words based on delimiters (*e.g.,* "?", "=", ":") and then use these words as features in classification. As a preliminary phase of feature selection, we discard words that appear too infrequently, since ad requests typically follow the same structure in each packet sent. 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, HTTP headers, and payload) to compare and analyze their relative importance (Sec. 5.1.1).

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. 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.

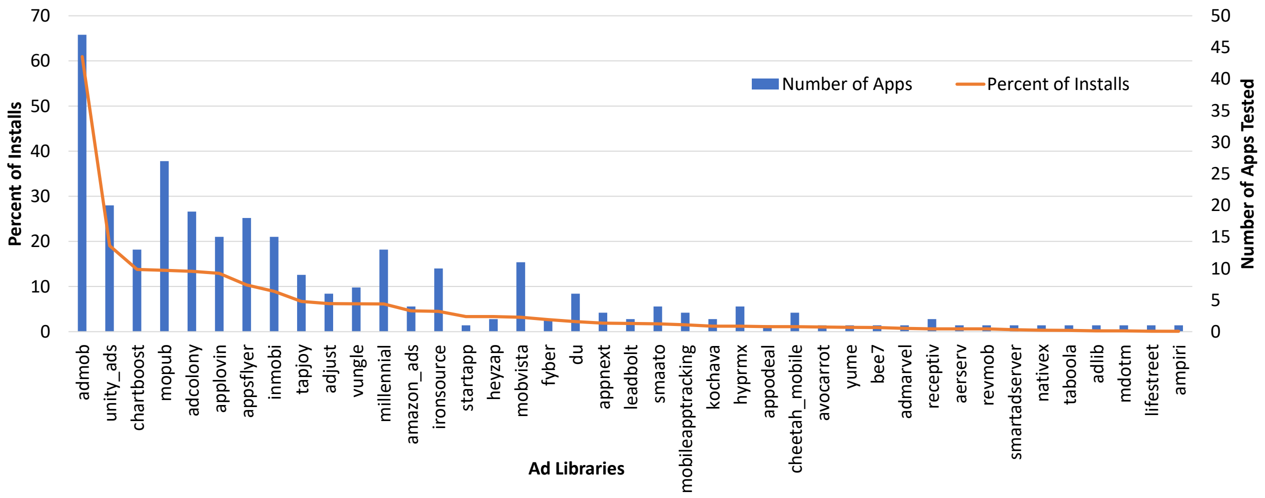

Fig. 2. Third-party ad libraries that we tested and the number of apps that were used to test each library. The line plot in orange shows the percentage of installed apps from the Google Play Store that use each library according to AppBrain [1]. In order to obtain a representative dataset we made sure 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.

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, 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. 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 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 [34] 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 [35] and hpHosts [36]. We picked AdAway Hosts because it is specific to mobile ad blocking; and hpHosts has been reported by [17] to find more mobile advertisers and trackers as compared to EasyList. However, we did not always find relevant matches with these lists because they tend to have many false positives and false negatives (see Table 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:

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

	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 1. Dataset Summary

often keep trying various ad networks available to them until they find one that will correctly produce an ad. We stop repeating when there are no more ads being displayed for the duration of the 5 minute interaction with the app in question.

Table 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 [37] and MoPub [38] – 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 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 3 further shows the Custom Rules that we manually curated to detect ad requests that evaded EasyList. There were dozens of rules 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 `http://athinagroup.eng.uci.edu/projects/nomoads/`.

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

5 Evaluation

In this section, we evaluate NoMoAds in terms of effectiveness of the classification (Section 5.1) as well as efficiency when running on the mobile device (Section 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 operate in real-time by adding approximately three milliseconds of additional processing time per packet.

	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 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.

5.1 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 5.1.1), apps (Section 5.1.2), or ad libraries (Section 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.

5.1.1 Testing on Previously Unseen Packets

First, we consider the entire NoMoAds dataset, described in Section 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 4 summarize our results for each feature set. We report the F-score, accuracy, specificity, and recall based on a 5-fold cross-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, extracted from headers and/or payload, 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 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 4, breaking the contents of packets into words significantly increases the training time since the number of features grows dramatically from 1-2 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

	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 4. Evaluation of decision trees trained on different sets of features, using our NoMoAds dataset (described in Section 4) and 5-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 [8] (alone, without our Custom Rules), AdAway Hosts [35], and hpHosts [36].

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 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. 3 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. 3) do not appear until later in the tree, meaning they have 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. If in the future we expand NoMoAds to blocking ads *and* trackers, we expect PII to play a bigger role.

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 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 a packet that belongs 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. In our past experience, for every ~5K packets in the dataset, the number of features increases by 50%. For instance, for a dataset with ~45K packets, the number of features would be ~13K. Training on ~13K features could take several hours, which is 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. Therefore, as our training datasets grow, we will need to be more selective about

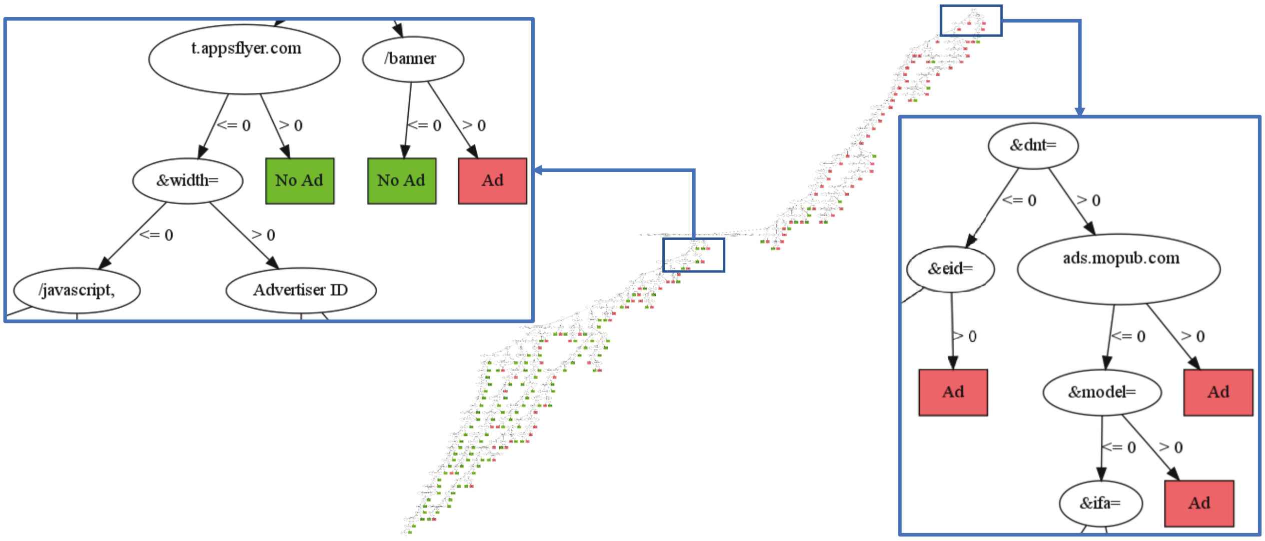

Fig. 3. Partial view of the classifier tree when using URL, HTTP headers, and PII as features. Out of the 5,326 initial features, only 277 were selected for the decision tree depicted here. At the root of the tree is the feature with the most information gain – "&dnt=", which is a key that stands for "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.

which data and which features we feed to our classifiers; this will be part of future work.

Blacklists as Baselines. The top rows of Table 4 also report the performance of three popular ad-blocking lists, which we use as baselines for comparison, namely: EasyList [8] (alone, without our Custom Rules), AdAway Hosts [35], and hpHosts [36]. 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.

5.1.2 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 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 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 5 apps, randomly selected. We then trained a classifier on 9 of those sets (*i.e.,* 45 apps total) and tested on the remaining set of 5 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. 4 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

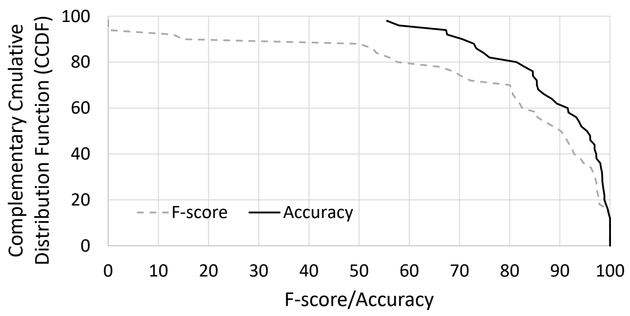

Fig. 4. Complementary Cumulative Distribution Function (CCDF) of F-score and accuracy for the 50 apps in our dataset.

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 hypothesis 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 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 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. 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

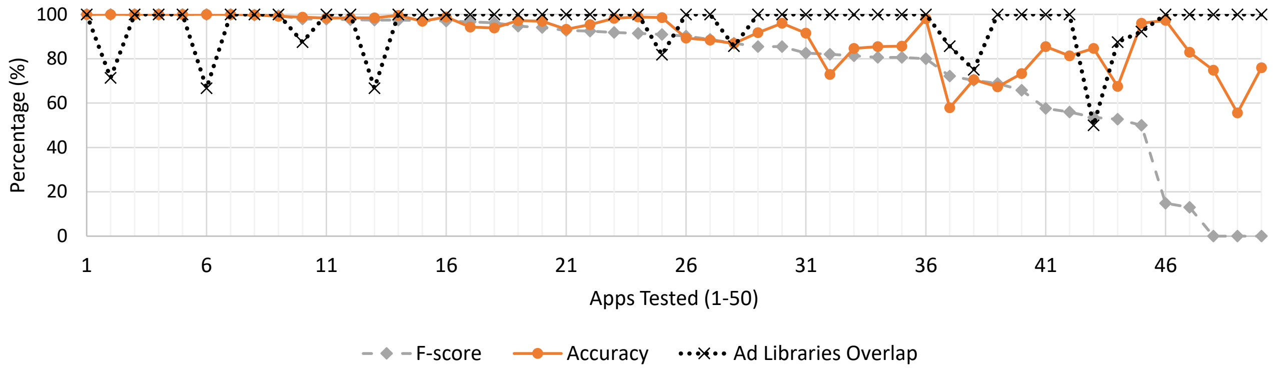

Fig. 5. Accuracy and F-score of NoMoAds when training on 45 apps in our dataset and testing on the remaining 5 apps (see Sec. 5.1.2). We repeated the procedure 10 times so that all apps were in the test set exactly once. We list the F-score, the accuracy, and the Ad Libraries Overlap (the percentage 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.

are incomplete, but our classifiers can suggest more rules for further improvement of ad-blocking.

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 our current network-based approach from user-space (VPN): 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 [37] - 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. As part of future work, we plan to build a system that can facilitate this mapping by tracing API calls to the network, either with static analysis (as was done in PEDAL [28]) or at the OS-level (as was done in ProtectMyPrivacy [39]). This effort is outside the scope of this paper.

Despite this limitation, 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 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 5.1.2 for details on how the overlap of ad libraries relates to the classification performance.

5.2 Efficiency

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 varying 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 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 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 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.

5.2.2 (Re)training Time

In Table 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. 3. Furthermore, the selection of features significantly affects the training time and has a moderate effect on classification performance. In this paper, our training dataset was relatively small, and training our classifiers from scratch did not take more than 13 minutes (Table 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.

6 Conclusion and Future Directions

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 work complements blacklist-based ad-blocking approaches, such as EasyList (which uses only the URL and HTTP Referer), DNS66 [21] (which operates on the coarse granularity of domains), and recent work on learning flow-based features [26]. To encourage reproducibility and future work, we make our code and dataset publicly available at `http://athinagroup.eng.uci.edu/projects/nomoads/`.

We conclude by discussing the limitations of NoMoAds and outline future research directions to address them.

First, the size of the training set used in this paper is limited. We currently manually label packets, which is not scalable if larger datasets are desired for training. Hence, in the future, we will explore options for automatic labeling of packets by separating ad library code from application code, either with static analysis and re-compilation (as done in [28]) or with OS-level modifications (as done in [39]). This will enable us to not only expand our dataset, but also to map each packet to the ad library responsible for generating (*e.g.,* by tracing API calls). An alternative way to increase the size of the dataset is through crowdsourcing, along the lines of Lumen [17].

Second, ad libraries may be able to circumvent our system by employing certificate pinning. However, certificate pinning is currently not widespread. Oltrogge et al. [30] reported only 45 out of 639,283 mobile apps employing certificate pinning. The authors explained that there are certain implementation hurdles that come with certificate pinning and it is generally not recommended that third-parties (such as ad libraries) use pinning since certificates must be kept up-to-date and it is difficult for app and ad library developers to coordinate certificate updates. Furthermore, network-level features can still be used to classify certificate-pinned packets: as shown in Table 4, destination IP and port, or destination domain lead to F-scores of 86% and above on our current dataset. In addition, with the automatic labeling approach proposed above, we will be able to label encrypted packets and explore using TCP/IP and various TLS-specific fields (*e.g.,* cipher suites, TLS extension headers) as features for classifying pinned packets.

Third, ad libraries may attempt to obfuscate other features in order to circumvent NoMoAds. While some apps and libraries do already obfuscate PII, the practice is uncommon as was shown in [40]. Even when PII are obfuscated, NoMoAds can use keys that correspond to PII values (*e.g.,* “uid=”, “idfa=”) as was done in [14] to detect PII that are a priori unknown to a system. Such features, as well as URL paths, are difficult to change, making NoMoAds resilient to feature obfuscation. Apps may also attempt to use anti ad-blockers to detect presence of ad-blockers [41, 42]. While we did not observe such behavior, if and when that happens, we can block anti ad-blocking scripts that are downloaded from third-parties [23] or use more sophisticated dynamic analysis techniques to circumvent anti ad-blocking logic that is part of the app [43].

Acknowledgements

This work is supported in part by the National Science Foundation under grant numbers 1715152 and 1649372, by the Data Transparency Lab (DTL), and CPCC at UCI. A. Shuba has been partially supported by an ARCS Fellowship. We would like to thank the anonymous reviewers whose thorough comments helped improve this paper.

References

[1] AppBrain. `https://www.appbrain.com/stats/libraries/ad`.

[2] Daniel G Goldstein, R Preston McAfee, and Siddharth Suri. The Cost of Annoying Ads. In *Proceedings of the 22nd international conference on World Wide Web*, pages 459–470. ACM, 2013.

[3] Narseo Vallina-Rodriguez, Jay Shah, Alessandro Finamore, Yan Grunenberger, Konstantina Papagiannaki, Hamed Haddadi, and Jon Crowcroft. Breaking for Commercials: Characterizing Mobile Advertising. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 343–356. ACM, 2012.

[4] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. The Price of Free: Privacy Leakage in Personalized Mobile In-App Ads. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[5] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. The Dark Alleys of Madison Avenue: Understanding Malicious Advertisements. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 373–380. ACM, 2014.

[6] Adblock Browser. `https://adblockbrowser.org/`.

[7] UC Browser. `https://play.google.com/store/apps/details?id=com.UCMobile.intl`.

[8] EasyList. `https://easylist.to/`.

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

[10] James 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/`, 2017.

[11] Alex 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.

[12] Adblock Plus for Android. `https://adblockplus.org/en/android-about`.

[13] Muhammad Ikram and Mohamed Ali Kaafar. A First Look at Mobile Ad-Blocking Apps. *IEEE Network Computing and Application (NCA)*, 2017.

[14] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 361–374. ACM, 2016.

[15] Anastasia Shuba, Anh Le, Emmanouil Alimpertis, Minas Gjoka, and Athina Markopoulou. AntMonitor: System and Applications. *arXiv preprint arXiv:1611.04268*, 2016.

[16] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. Haystack: A Multi-Purpose Mobile Vantage Point in User Space. *arXiv:1510.01419v3*, Oct. 2016.

[17] Narseo Vallina-Rodriguez, Srikanth Sundaresan, Abbas Razaghpanah, Rishab Nithyanand, Mark Allman, Christian Kreibich, and Phillipa Gill. Tracking the Trackers: Towards Understanding the Mobile Sdvertising and Tracking Ecosystem. *arXiv preprint arXiv:1609.07190*, 2016.

[18] Adblock Plus for Android Removed from Google Play Store. `https://adblockplus.org/blog/adblock-plus-for-android-removed-from-google-play-store`.

[19] Ben Williams. Adblock Plus and (a little) more. `https://adblockplus.org/blog/five-and-oh-look-another-lawsuit-upholds-users-rights-online`, 2016.

[20] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 319–333. IEEE, 2017.

[21] DNS-based Host Blocker for Android. `https://github.com/julian-klode/dns66`.

[22] Disconnect. `https://disconnect.me/`.

[23] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *ACM Internet Measurement Conference (IMC)*, 2017.

[24] Sruti Bhagavatula, Christopher Dunn, Chris Kanich, Minaxi Gupta, and Brian Ziebart. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 95–102. ACM, 2014.

[25] Jason Bau, Jonathan Mayer, Hristo Paskov, and John C Mitchell. A Promising Direction for Web Tracking Countermeasures. *Proceedings of W2SP*, 2013.

[26] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. An Automated Approach for Complementing Ad Blockers' Blacklists. *Proceedings on Privacy Enhancing Technologies*, 2015(2):282–298, 2015.

[27] Privacy Badger. `https://www.eff.org/privacybadger`, 2018.

[28] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient Privilege De-escalation for Ad Libraries in Mobile Apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 89–103. ACM, 2015.

[29] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. Acm, 2012.

[30] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. To Pin or Not to Pin-Helping App Developers Bullet Proof Their TLS Connections. In *USENIX Security Symposium*, pages 239–254, 2015.

[31] Adblock Plus Library for Android. `https://github.com/adblockplus/libadblockplus-android`.

[32] Anastasia Shuba, Evita Bakopoulou, and Athina Markopoulou. Privacy Leak Classification on Mobile Devices. In *Signal Processing Advances in Wireless Communications (SPAWC), 2017 IEEE 18th International Workshop on*. IEEE, 2018. To Appear.

[33] Anastasia Shuba, Evita Bakopoulou, Milad Asgari Mehrabadi, Hieu Le, David Choffnes, and Athina Markopoulou. AntShield: On-Device Detection of Personal Information Exposure. *arXiv preprint arXiv:1803.01261*, 2018.

[34] Wireshark. `https://www.wireshark.org/`.

[35] AdAway hosts. `https://adaway.org/hosts.txt`.

[36] hpHosts. `https://hosts-file.net/ad_servers.txt`.

[37] AdMob. `https://www.google.com/admob/`.

[38] MoPub. `https://www.mopub.com`.

[39] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I Hong, and Yuvraj Agarwal. Does this App Really Need My Location?: Context-Aware Privacy Management for Smartphones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):42, 2017.

[40] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[41] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E. Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven J. Murdoch. Ad-Blocking and Counter Blocking: A Slice of the Arms Race. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2016.

[42] Muhammad Haris Mughees, Zhiyun Qian, and Zubair Shafiq. Detecting Anti Ad-blockers in the Wild. In *Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[43] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, , and Heng Yin. Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2018.