sciendo

Proceedings on Privacy Enhancing Technologies ; 2020 (2):45–66

Anastasia Shuba* and Athina Markopoulou

NoMoATS: Towards Automatic Detection of Mobile Tracking

Abstract: Today's mobile apps employ third-party advertising and tracking (A&T) libraries, which may pose a threat to privacy. State-of-the-art detects and blocks outgoing A&T HTTP/S requests by using manually curated filter lists (*e.g.* EasyList), and recently, using machine learning approaches. The major bottleneck of both filter lists and classifiers is that they rely on experts and the community to inspect traffic and manually create filter list rules that can then be used to block traffic or label ground truth datasets. We propose NoMoATS – a system that removes this bottleneck by reducing the daunting task of manually creating filter rules, to the much easier and scalable task of labeling A&T libraries. Our system leverages stack trace analysis to automatically label which network requests are generated by A&T libraries. Using NoMoATS, we collect and label a new mobile traffic dataset. We use this dataset to train decision tree classifiers, which can be applied in real-time on the mobile device and achieve an average F-score of 93%. We show that both our automatic labeling and our classifiers discover thousands of requests destined to hundreds of different hosts, previously undetected by popular filter lists. To the best of our knowledge, our system is the first to (1) automatically label which mobile network requests are engaged in A&T, while requiring to only manually label libraries to their purpose and (2) apply on-device machine learning classifiers that operate at the granularity of URLs, can inspect connections across all apps, and detect not only ads, but also tracking.

Keywords: mobile; privacy; tracking; advertising; filter lists; machine learning

DOI 10.2478/popets-2020-0017
Received 2019-08-31; revised 2019-12-15; accepted 2019-12-16.

***Corresponding Author: Anastasia Shuba:** Broadcom Inc. (the author was a student at the University of California, Irvine at the time the work was conducted), E-mail: ashuba@uci.edu
Athina Markopoulou: University of California, Irvine, E-mail: athina@uci.edu

1 Introduction

The mobile ecosystem is rife with third-party tracking. App developers often integrate with third-party libraries, which can be broken into roughly three categories: advertisement and analytics libraries, social libraries (*e.g. Facebook*), and development libraries [1]. These libraries inherit the same permissions as their parent app, and can thus collect rich personal and contextual information [2, 3].

To protect themselves, privacy-conscious users rely on tools such as DNS66 [4] and AdGuard [5]. These apps require no rooting and instead rely on VPN APIs to intercept outgoing traffic and match it against a list of rules, such as EasyPrivacy [6]. Such lists are manually curated, by experts and the community, and are thus difficult to maintain in the quickly changing mobile ecosystem. More recently, multiple works [7–9] have proposed to train machine learning models, which are more compact and generalize. However, in order to obtain ground truth (*i.e.* labeled datasets) to train the machine learning models, current state-of-the-art still relies on filter lists [7, 8] or a combination of filter lists and manual labeling [9]. Therefore, obtaining accurate ground truth is a crucial part and a major bottleneck of both filter-lists and machine learning approaches.

In this paper, we aim to reduce the scope of manual labeling required to identify 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 whose primary purpose is advertising or analytics (A&T libraries). Throughout this paper, we will refer to a an HTTP request (or a decrypted HTTPS request) as an A&T request (or packet), if it was generated by an A&T library. Another key observation is that it is possible to determine if a network request came from the application itself or from a library by examining the stack trace leading to the network API call. More specifically, stack traces contain package names that identify different entities: app vs. library code. Thus, to label which network requests are A&T, we just need a list of libraries that are known to be A&T.

(cc) BY-NC-ND

We combine these ideas to make the following contributions. First, we design and implement NoMoATS – a system for automatically labeling A&T requests. Our approach does not rely on filter lists nor manual labeling of requests. Instead, it examines stack traces leading to the network API call to determine if the request was generated by an A&T library. Second, using NoMoATS we collect the first large-scale dataset of mobile network requests, where each request is labeled based on its origin: the app itself or an A&T library. Third, we evaluate the effectiveness of NoMoATS-produced labels for training machine learning classifiers that predict HTTP/S A&T requests on mobile devices. Our classifiers can be trained quickly (on the orders of milliseconds) while achieving average F-scores of 93%. Furthermore, our classifiers can be applied in real-time on mobile devices to predict and block A&T requests when running on top of a VPN-based interception system, such as [7, 10–12]. Using popular filter lists, namely EasyList, EasyPrivacy [6] and Mother of All Ad-Blocking (MoaAB) [13], we evaluate both our labeling approach and our classifiers. We find that NoMoATS 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.

We envision that this work will be useful to the community by minimizing human involvement in labeling A&T requests: from creating (tens of thousands of) rules to only identifying libraries that are A&T (on the order of hundreds). In addition to this scalability advantage, NoMoATS provides a more stable labeling: the network behavior of trackers and apps change in much shorter time scales than a library's purpose, eliminating the need for frequent manual updates. Our work can assist both expert list curators in creating and updating filter rules (especially as A&T change their behavior) and researchers to label their own datasets (for apps and libraries of their own interest). To that end, we make NoMoATS open-source and release our dataset[1][14].

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the NoMoATS system. Section 4 describes how we trained machine learning classifiers for predicting A&T requests using a dataset collected and labeled with NoMoATS. Section 5 evaluates the NoMoATS approach and the classifiers from Sec. 4 against popular filter lists. Section 6 discusses the limitations of our system and Section 7 concludes the paper. Appendices A-C provide additional details on the NoMoATS approach.

2 Related Work

Various studies have shown the prevalence of tracking on the web [15–18] and on mobile devices [19, 20]. These studies have shown that there are two types of tracking: stateful and stateless. In stateful tracking, an explicit identifier is collected, which can be a cookie or a type of personally identifiable information (PII), such as a person's email address. In contrast, stateless tracking aims to fingerprint users by collecting seemingly benign information, such as the screen resolution of a device or how a device processes audio. To prevent both types of tracking, multiple tools have been developed in industry and in academia. These tools can be categorized into three approaches: (i) static analysis, (ii) rooted devices, and (iii) traffic analysis. In this section, we exclude the discussion of static analysis tools, such as [2, 21–24], and focus on the latter two approaches as they are most closely related to NoMoATS.

Using a rooted device or building a modified OS allows power users and researches to exert greater control over a device. For instance, TaintDroid [25] 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 [26] extended the system to also consider SSL traffic. However, both approaches could not deal with PII sent over native code. Similarly, ProtectMyPrivacy (PmP) [3] hooked into PII-accessing APIs and recorded stack traces leading to each API call. From the stack traces, they were able to tell whether PII was being accessed by the app or by a third-party library. However, PmP was not able to hook into networking APIs to trace which part of the app was sending out information. Unfortunately, rooting a device or installing a custom OS is 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 (Sec. 4.2.4). To the best of our knowledge, our system is the first to be able to map network requests (including ones sent from native code) to the libraries that generated them.

Traffic analysis is the most promising approach as it is the most user friendly. In fact, all of the anti-A&T tools that are popular today (*i.e.* have millions

1 Please note that the data collection process was automated. No human subjects were involved, thus no IRB needed.

Tool	Blocks					Inspects	Decision Mechanism	Labeling Procedure
	3rd Parties		Identifiers					
	Ads	Trackers	Stateful	Stateless	PII			
AntMonitor [12]	✗	✗	✗	✗	✓	Full HTTP payload	FL	Automated
ReCon [7]	✗	✗	✗	✗	✓	Full HTTP payload	ML	Automated
AdGuard [5]	✓	✓	✓	✓	✓	URL, HTTP Headers	FL	Manually create rules
Lumen [8]	✓	✓	✓	✓	✓	Hosts	ML	Use existing FL
NoMoAds [9]	✓	✗	✓	✓	✓	URL, HTTP Headers	ML	Use existing FL and manually create rules
NoMoATS	✓	✓	✓	✓	✓	URL, HTTP Headers	ML	Label libraries as being A&T related or not

Table 1. Comparing NoMoATS to existing VPN-based anti-A&T tools. Some tools focus on blocking PII only, while others adopt a more aggressive approach and block all connections destined to third parties, thereby reducing both stateless and stateful tracking. Some tools leverage machine learning (ML) for decision making, while others rely on filter lists (FL). Of the approaches that target A&T beyond PII, only NoMoATS offers minimal manual labeling – one only needs to label libraries with their purpose.

of downloads) are traffic analysis-based. For example, AdGuard [5] has reached over five million downloads on the Google Play Store alone [27]. We note that multiple anti-A&T tools exist that are tailored to the web ecosystem and are offered as standalone browsers (*e.g.* Safari [28]) or browser extensions (*e.g.* Adblock Plus [29]). While these tools and their research extensions (*e.g.* [30–34]) also fall into the traffic analysis category, we exclude them from our discussion as our focus is on mobile anti-A&T tools, which are fundamentally different in the technology they use and are tailored towards a different ecosystem. Specifically, operating as a browser or a browser extension provides a level of control that is similar to using a rooted mobile device: all browser traffic is intercepted and read in plain text, all rendered web content (*e.g.* HTML and JavaScript) can be read and modified, and various JavaScript APIs can be intercepted. On a mobile device, the only way to intercept traffic from all apps without rooting the device, is to use a VPN, which has certain limitations. First, inspecting HTML and JavaScript is not possible without reconstructing pages from HTTP responses, which is difficult to achieve in real-time on a mobile device with limited battery. Second, the increasing adoption of TLS limits the analysis to HTTP traffic and HTTPS traffic that can be decrypted (*e.g.* when certificate pinning is not used). Since VPN-based traffic analysis is also the approach taken by NoMoATS, we focus the rest of this section on discussing existing tools in this space.

Table 1 compares prior mobile anti-A&T solutions to NoMoATS by evaluating them with respect to several criteria listed next. (i) What does the tool *block* in terms of third-parties and identifiers? (ii) What traffic does the tool *inspect*? (iii) What *decision mechanism* does the tool use? (iv) What *labeling procedure* is required? The remainder of this section explores each of our criteria serially using example tools from Table 1.

The first design choice in building mobile anti-A&T tools is choosing what to block. For instance, tools such as AntMonitor [12] and ReCon [7] only block connections that contain PII. While this approach has the advantage of preventing tracking by first parties, it can also lead to apps breaking since first parties often have legitimate reasons for collecting PII (*e.g. Google Maps* requires location data). Furthermore, blocking PII alone does not account for other identifiers, such as cookies and fingerprints. A more aggressive approach is to block all third-party connections that are A&T related, which is the approach taken by the majority of the tools in Table 1, including NoMoATS. This approach will miss tracking done by first parties, but will be able to block all types of identifiers, including stateless ones, that are sent to A&T hosts. In fact, a recent study [19] showed that such approaches are able to significantly reduce the number of fingerprinting scripts loaded into web pages.

The second design choice is what traffic to inspect. Inspecting the entire HTTP payload, as done by AntMonitor and ReCon, ensures that PII sent as part of HTTP data are also caught. On the other extreme, is the choice to block connections based on the destination host alone, as in Lumen [8]. However, this approach cannot effectively block multi-purposed hosts that serve essential content and track users. As a middle ground, most tools opt in to inspect the requested URL and HTTP headers. This approach can strategize against multi-purposed hosts and is more lightweight than inspecting the entire payload.

Another critical design choice for anti-A&T tools is the underlying decision mechanism that identifies which connections should be blocked. In the simplest case of AntMonitor [12], that mechanism is a filter list containing known PII values: when a PII is detected in an outgoing packet, the connection is dropped. This is also the approach taken by AdGuard [5], albeit with much more

complex filter lists that match keywords in URLs and HTTP headers. One of the filter lists used by AdGuard, is EasyList [6], which, as of August 2019, contains close to 74K manually curated rules. These rules are difficult to maintain in the quickly changing mobile ecosystem: over 90% of the rules in EasyList are not triggered in common use-cases [35] and it can take months to update the list in response to changes [36]. An alternative to filter lists is machine learning, proposed by multiple works [8, 9]: instead of matching HTTP requests against a set of rules, a classifier decides what to block. Unfortunately, machine learning solves only half of the problem. As we discuss next, obtaining ground truth can prove challenging.

In the case of training classifiers to identify HTTP requests that contain PII (*e.g.* ReCon [7]), labeling data is straightforward: one simply needs to setup a testing phone for which all PII values are known and then search the collected traces for occurrences of these PII. On the other hand, labeling network traffic as A&T related is difficult. Prior art [8, 9] relied on existing filter lists (*e.g.* EasyList) to label their data. This approach circulates back to the labeling procedure adopted by filter lists, which is still manual. Specifically, to create new rules, filter list curators manually inspect network traffic to determine which requests should be blocked and then create generic blocking rules for them. In our own prior work, NoMoAds [9], we also took this manual labeling approach to supplement EasyList with mobile-specific ad-blocking rules. Due to the scalability issues of this approach, the NoMoAds dataset was limited to just 50 apps. Scalability is a problem for both approaches: filter list-based and machine learning-based (when it comes to labeling ground truth). The mobile ecosystem is a rapidly changing environment, and thus it is important to have a scalable approach for blocking A&T. NoMoATS aims to aid with scalability of blocking A&T, as discussed next.

In our approach, network traffic is labeled automatically. The human only has to label which libraries are A&T, and such information is easily found by visiting a given library's web page. Furthermore, there are only hundreds of libraries [1] to label, which is orders of magnitude less than the tens of thousands of rules that exist in filter lists [6] or the millions of websites and HTTP requests that constantly change. Finally, the purposes of libraries (A&T or not) change in much longer time scales than the network behavior of apps, which is important in the arms-race. Future work has the potential to automate the library labeling process as well by searching for keywords on the library's page. Continuously expanding automation in the task of blocking A&T traffic will allow the privacy community to keep up with the rapidly changing mobile advertising and tracking ecosystem.

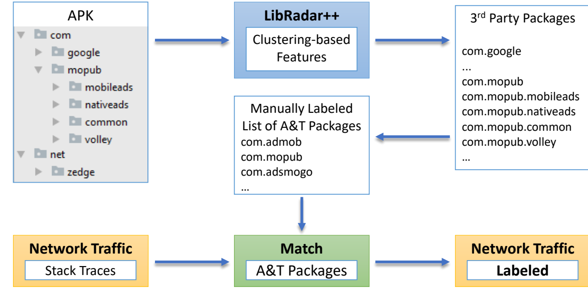

Fig. 1. Our labeling approach with LibRadar++ (Sec 3.1). How we obtain stack traces pertaining to each network request is described in Sec. 3.2.

3 The NoMoATS System

In order to move the labeling process from HTTP/S requests to libraries, we need two components. First, we need to be able to analyze stack traces to determine if a request was generated by an A&T library (Sec. 3.1). Second, we need to be able to map network requests to stack traces that led to them (Sec. 3.2).

3.1 Stack Trace Analysis

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, belonging to the *ZEDGE™* app, is shown in the "APK" box in Fig. 1. 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. Prior art, LibRadar [37], did just that: they built signatures for each packaged folder and then used clustering to identify third-party 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++ [38], which was built over a larger set of apps (six million).

As shown in Fig. 1, NoMoATS uses LibRadar++ [38] to analyze apps and automatically produce a list of library package names contained within (Fig. 1). Note

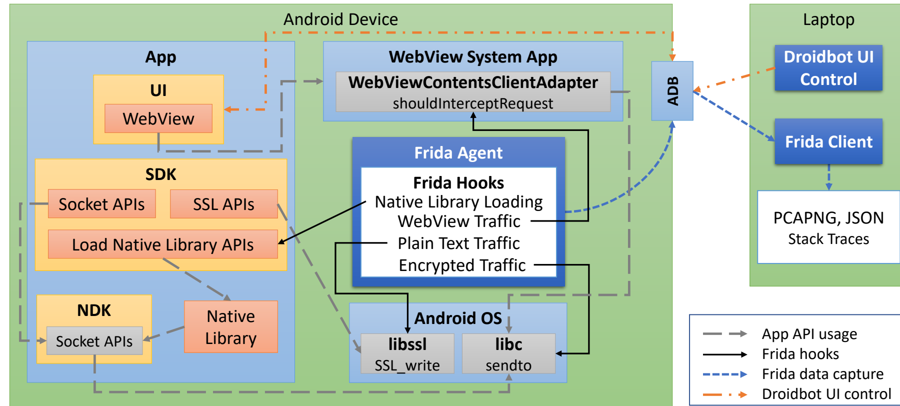

Fig. 2. Our data collection system (Sec. 3 and Appendix A): using Frida to hook into Android networking APIs, capture network traces, and the Java stack traces leading to the networking API call.

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. 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 NoMoATS. 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 [39]. We note that this is the only point where manual effort is needed: in mapping package names to their primary purpose.

Once we have a list of A&T package names, we can search stack traces for their presence – as shown in Fig. 1. If an A&T package name appears in a stack trace leading to a networking API call, we can mark the corresponding network activity as A&T. The next section describes how NoMoATS collects network requests and the responsible stack traces.

3.2 Hooking Android Networking APIs

Fig. 2 provides an overview of the part of the NoMoATS system that is used to collect network requests and the stack traces leading to them. To intercept network requests, we use the Frida [40] dynamic instrumentation toolkit. Frida allows us to write JavaScript code that gets injected at run-time into an application process. The injected code is referred to as the agent, which 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. 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. We begin by providing some background on the Android OS and then explain our hooking method.

Background. As shown in Fig. 2, 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 [25], 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 directly and bypass any hooks installed in Java space (see Fig. 2). 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, which are only available through the SDK. We note that apps can also use their own versions of OpenSSL and other SSL libraries – Appendix A.2 discusses this scenario further. 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. 2:

Plain Text Traffic. Prior work in both static (*e.g.* [2]) and dynamic (*e.g.* [25, 26]) analysis of Android apps suffered from not having visibility into native code. To overcome this limitation, we hook into the `libc sendto` function 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 that provides developers with ready-made browser-like functionalities. This component is often used in conjunction with regular GUI components to display web elements, such as banner ads. 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. 2). 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 apps need 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 `WebViewContentsClientAdapter` 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 request.

Saving Traces. To save the captured network traces, we utilize the Frida client that runs outside of the inspected application (see Fig. 2). When a hook has been triggered, our Frida agent sends the captured network request and the responsible Java stack trace to the client via the Android Debug Bridge (ADB). 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 in Appendix A.3.

4 Machine Learning Application

The motivating application for developing NoMoATS was training classifiers to detect ads and trackers. With NoMoATS, we were able to collect a large dataset with minimal human involvement (Sec. 4.1) and then use that dataset to train machine learning classifiers to predict not just ad but also tracking requests (Sec. 4.2).

4.1 Data Collection

In this section, we describe how we used NoMoATS to collect a training dataset. First, in Sec. 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. 4.1.2, we summarize our collected dataset.

4.1.1 Selecting and Exercising Apps

To diversify our dataset, we used the Google Play Unofficial Python API v0.4.3 [41] to download the top 10 free apps from each of the 35 Google Play Store categories on January 29, 2019. Previous studies, such as [9] and [7] 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 are more likely to use specific third-party libraries. For instance, game apps often use the Unity Ads library [42] 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 paper we focus on learning mobile A&T behavior, and leave the in-browser anti-tracking protection to standalone tools, such as Safari [28] (see Sec. 2 for further discussion about browser tools).

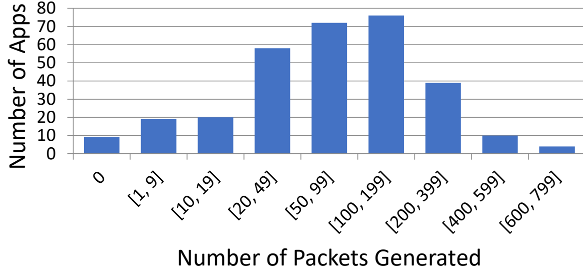

Fig. 3. Distribution of packets in our dataset: y-axis shows the number of apps that generated the corresponding number of packets on the x-axis. Packets are HTTP/S requests.

	Count
Apps Tested	307
Total Packets	37,438
A&T Packets	13,512
Apps with Packets	298
Apps with 10+ Positive and Negative Samples	128
A&T Libraries	37
A&T Libraries with Packets	26

(a)

	Number of		
	Packets	Apps	Devs
WebView	17,057	167	132
SSL_Write	17,028	295	239
libc sendto	3,353	129	108

(b)

Table 2. Statistics about: (a) our dataset; (b) the number of packets, apps, and developers (devs) that triggered each Frida hook (Sec. 3.2). Packets are HTTP/S requests.

Since some apps appear in multiple categories, we ended up with a total of 339 distinct apps. We ran each app using NoMoATS (Sec. 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. 3.2). 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.

To automatically exercise apps at scale, we used Droidbot [43], as shown in Fig. 2. Each app was exercised with Droidbot for five minutes, sending inputs every two 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. Future work can update Frida hooks to work with new versions of Google Play Services and to hook into common methods for root detection and provide false responses to the apps (Sec. 6). In total, we had to exclude 32 apps, leaving us with a total of 307 apps.

4.1.2 Data Summary

First, to understand how well Droidbot explores apps, we have plotted the distribution of packets (HTTP/S requests) in Fig. 3. We note that most apps generated at least 50 packets, with over a dozen apps generating 400 or more packets. We examined the nine apps that did not generate any traffic and found that three of them required an account. Two of them were meant for moving data to another device, and thus required additional hardware for proper testing. Another two apps required a SIM card or a valid phone number, which our test device did not have. Yet another pair of apps could not start due to incompatibilities with Frida. Similarly, the 19 apps that generated less than 10 packets also either required logins or were gaming apps that Droidbot had trouble exploring. We discuss these limitations further in Sec. 6. Next, in Table 2a we provide additional summaries of our dataset. In total, we collected 37,438 packets, 13,512 (36%) of which were A&T packets. 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.

In Table 2b, we explore the value of our Frida hooks based on the number of packets, apps, and developers that triggered each hook. 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 the `WebViewClient` constructor (Sec. 3.2), we would have been unable to obtain stack traces for over 45% of captured packets that range across 167 unique apps and 132 unique developers. Unfortunately, we also had to discard 83 WebView packets because Frida was unable to acquire stack traces for them. We discuss this limitation in Sec. 6 and omit these packets from our dataset of 37,438 packets. Out of 298 apps that sent at least one packet, 295 used OpenSSL and triggered our `SSL_Write` hook. Our `libc sendto` hook saw the least usage: only 129 apps (108 unique developers) triggered this hook, resulting in just 3,353 packets. This indicates that more apps and developers are moving towards better security practices by using encryption.

The NoMoATS Dataset. We use this dataset to train the classifiers discussed in the next section, and this is the strength of the machine learning approach presented in this paper. Our main contribution is in our dataset: it has labels for trackers (not just ads is in our prior work [9]) and it is larger in scale. We have

released our dataset and open-sourced NoMoATS [14] to allow other researchers to collect ever larger datasets by letting NoMoATS run longer and on more apps.

4.2 Machine Learning Approach

In this section, we present our machine learning approach. First, we discuss our training setup in Sec. 4.2.1. Second, we train a classifier on our entire dataset of 37,438 packets (Sec. 4.2.2). For the rest of the paper, we will refer to this classifier as *general classifier* – a term coined by ReCon [7]. Third, we explore the possibility of training *per-app classifiers* – classifiers built by training on a specific app's dataset only (Sec. 4.2.3). Finally, in Sec. 4.2.4, we discuss a potential deployment scenario and evaluate the time it takes for our classifiers to make a prediction on a mobile device.

4.2.1 Setup

We build on our prior NoMoAds open-source code [9] to train C4.5 Decision Tree (DT) classifiers since it was shown in [7, 9] that they work well when classifying mobile requests. Moreover, [9] has shown that DT classifiers can run in real-time on mobile devices. We follow the standard approach for feature extraction from HTTP requests, as was done in [7, 9, 44, 45]. Specifically, we use a bag-of-words model where we break each packet into words based on standard URL delimiters (*e.g.* "/", "=", "?"). We discard standard HTTP headers [46], words that appear too infrequently, and any words that consist of only one character. The remaining words are used as our features. To evaluate which features have the most importance, we selectively extract words from various parts of requests as discussed next.

Host. First, we evaluate how well our classifiers can perform when operating only with second-level domains (SLDs) or hostnames. 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 hostnames 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) field 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 (hostname and the path component).

Feature Set	F-score (%)	Accuracy (%)	Specificity (%)	Recall (%)	Training Time	Tree Size	Prediction Time (ms)
Domain	87.72	90.82	90.80	90.84	284ms	1	0.12 ± 0.08
Host	90.25	92.68	92.03	93.84	331ms	1	0.11 ± 0.06
Path Component of URL	86.21	90.49	95.11	82.31	5.59hrs	535	0.85 ± 1.13
URL (Host & Path)	90.63	93.35	95.76	89.08	6.27hrs	556	0.87 ± 1.10
URL + Referer + Content Type	94.77	96.18	96.31	95.95	5.09hrs	488	0.83 ± 0.92
URL and HTTP Headers	95.56	96.74	96.50	97.18	5.50hrs	371	0.92 ± 1.15

Table 3. General Classifier Performance (Sec. 4.2.2) in terms of F-score, accuracy, specificity, recall, training time (at the server), tree size (number of non-leaf nodes), and average per-packet prediction time on the mobile device (Sec. 4.2.4).

URL and HTTP Headers. Finally, we evaluate how adding HTTP headers helps with classification. First, since some filter lists (*e.g.* EasyList) use the Referer and Content-Type headers, we evaluate the added benefit of these two headers alone. Next, we evaluate how using all the HTTP headers helps with classification.

4.2.2 General Classifier

Table 3 summarizes the results of five-fold cross-validation when training a general classifier across the entire dataset. We report the following metrics: F-score, accuracy, specificity, recall, training time, and tree size (non-leaf nodes). We find that using the SLD as the only feature yields an F-score of 87%. Unsurprisingly, using hostnames 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 requests by blocking DNS requests to the corresponding hosts or by matching the SNI field 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 3, using these features actually decreases the performance of the classifier. Furthermore, the resultant tree size is 535, which is much larger than the 188 non-leaf nodes reported in [9]. 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 five hours. Next, we train a general classifier using the full URL and find that the F-score increases, but at the cost of an increased tree size and longer training time. Interestingly, when adding the HTTP Referer and Content Type header values as features, the F-score increases even more while the training time and tree size both decrease. This signifies the importance of these two features and validates the EasyList community's choice to use them in their filter lists. Finally, using all the headers (last row in Table 6) achieves the highest F-score of over 95% and the smallest tree size (371 non-leaf nodes). This is because some A&T libraries use custom HTTP headers, which can be used to easily identify a request as A&T. For example, *Chartboost* [47] (a mobile advertising library), uses the following HTTP header key-value pair: "X-Chartboost-Client: Chartboost-Android-SDK..." We found this HTTP header value early in the decision tree, indicating its high information gain.

While the general classifiers perform well in terms of F-scores, the slow training times may become problematic. With our current dataset of 298 apps daily updates are still possible: performing five-fold cross validation takes about 15 hours when using all features. However, as of September 2019, the Google Play Store contains over 2.7 million apps [48]. Thus, to achieve better coverage, we would have to train on thousands more apps, which would further increase our feature space and training time. One way to avoid this would be to follow the approach we proposed in [9] and select apps that contain specific A&T libraries, until all A&T libraries were explored. Another approach, discussed in the next section, is to train per-app classifiers, using a particular app's data alone.

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. Since VPN libraries, such as AntMonitor [12], can provide packet-to-app mapping, we can apply a separate classifier to each app. Thus, we train a separate classifier for each application in our dataset, using that app's data only. ReCon [7] proposed a similar approach where they trained per-SLD classifiers for predicting PII exposures. Although it is possible for us to also train separate classifiers for each SLD in our dataset, we argue that per-app classifiers are more appropriate for realistic deployment.

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 [49]. In contrast, according to a 2017 report by AppAnnie [50], 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. Thus, with per-app classifiers, a new user in 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. 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.

One problem with both per-app and per-SLD classifiers is that sometimes there is not enough training data. For instance, as shown in Table 2, our dataset contains only 128 apps that have at least 10 positive and 10 negative samples. ReCon [7] proposed the general classifier as a means to deal with those SLDs for which no specialized classifier could be trained. However, this method does not eliminate the need to train a large general classifier. Fortunately, our per-app approach allows us a more flexible solution, described next.

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 [51]. Since, as was confirmed in [45], 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-SLD classifiers, it is unclear how to increases samples for a given

Feature Set	Average ± Standard Deviation						
	F-score (%)	Accuracy (%)	Specificity (%)	Recall (%)	Training Time (ms)	Tree Size	Prediction Time (ms)
Domain	92.87 ± 10.4	94.51 ± 6.6	86.77 ± 22.6	93.96 ± 11.9	1.09 ± 4.82	0.95 ± 0.21	0.11 ± 0.07
Host	93.53 ± 7.02	94.91 ± 5.40	88.01 ± 20.4	92.90 ± 10.0	1.12 ± 4.75	0.99 ± 0.09	0.12 ± 0.13
Path Component of URL	89.99 ± 9.77	92.98 ± 6.24	86.39 ± 21.8	88.60 ± 13.7	66.8 ± 119	8.88 ± 7.42	0.17 ± 0.20
URL (Host & Path)	92.81 ± 8.65	95.15 ± 5.53	91.98 ± 13.4	92.00 ± 10.9	69.9 ± 133	8.54 ± 6.83	0.16 ± 0.18
URL + Referer + Content Type	94.14 ± 6.90	96.02 ± 4.54	94.21 ± 10.3	92.90 ± 8.96	45.6 ± 79.5	5.57 ± 4.50	0.18 ± 0.20
URL and HTTP Headers	93.23 ± 8.97	95.79 ± 6.24	91.71 ± 18.4	92.94 ± 10.6	48.6 ± 90.4	3.71 ± 2.67	0.16 ± 0.18

Table 4. Performance of per-app classifiers (Sec. 4.2.3) and their average per-packet prediction time on the mobile device (Sec. 4.2.4). Tree size here is the number of non-leaf nodes.

SLD. In the future, a crowdsourcing platform, such as CHIMP [52] can be used to generate inputs for apps with a low number of samples (Sec. 6). 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 VPN.

Next, we discuss how we train per-app classifiers using the NoMoATS dataset. For the purposes of this section, we skip training classifiers for apps that have less than 10 samples of each class and report results on the remaining 128 apps. Specifically, we evaluate each feature set from Sec. 4.2.1 with five-fold cross-validation on each of the 128 per-app classifiers. In Table 4, 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 perform similarly to the general one – Tables 4 and 3. We find that in the case of per-app classifiers, using the full URL along with the HTTP Referer and Content Type features yields the best results – an average F-score of over 94%. Interestingly, in the case of per-app classifiers, using the full feature set slightly reduces the F-score, but also decreases the average tree size. Furthermore, using hosts as a single feature also yields a high average F-score of over 93%. Training on the path component of the URL performs the worst – an average F-score of ~90%. This shows that, as expected, the hostname is one of the most important features for identifying trackers, and that is why many common filter lists (*e.g.* MoaAB [13]) 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 it for five minutes (either with Droidbot or manually) and build a specialized classifier, all in under 10 minutes. In contrast, in the case of a general classifier, an additional or an updated app would require re-training over the entire dataset. 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. As we show in Sec. 4.2.4, the tree size has an impact on real-time prediction on the device, and thus it is beneficial to have smaller trees. Moreover, smaller trees are easier for list curators to examine and adjust.

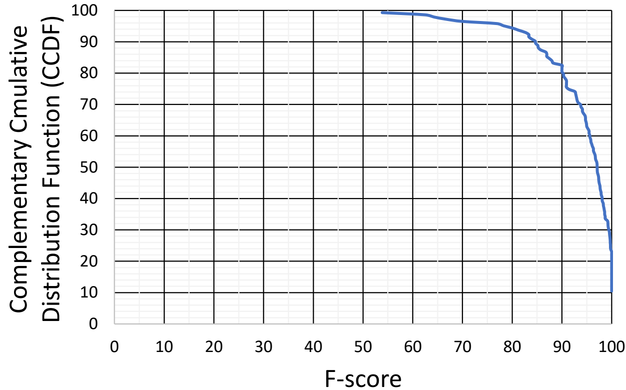

Fig. 4. Complementary Cumulative Distribution Function (CCDF) of F-scores of the 128 per-app classifiers trained on the full feature set (URL with all the HTTP headers). Over 80% of the classifiers reach F-scores of 90% or above, and over half of the classifiers outperform the general one with F-scores of over 95.6%.

For the rest of this section, we will focus on the classifiers trained using the full URL with all the HTTP headers (last row in Table 4). Fig. 4 shows the Complementary Cumulative Distribution Function (CCDF) of F-scores of these per-app classifiers. We note that over 14% of our per-app classifiers achieve F-scores of 100%. 80% of our classifiers reach F-scores 90% or higher, and 56% outperform the general one with F-scores of over 95.6%. Next, we examine the trees that our classifiers produce. Fig. 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 net-

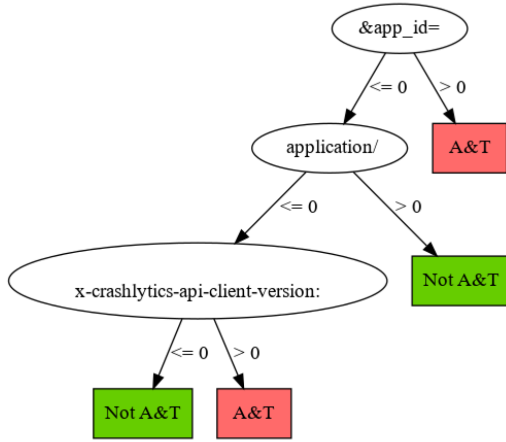

Fig. 5. An example decision tree of a per-app classifier trained on the full URL and all HTTP headers.

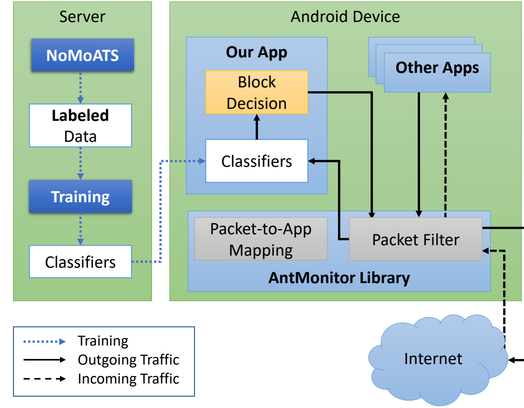

Fig. 6. Potential deployment: classifiers are trained offline and are used to predict and block A&T requests intercepted by a VPN service, such as AntMonitor [12].

work 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* [13]. Fig. 5 shows that our classifiers generalize: the depicted classifier has learned to identify *Crashlytics* as a tracking service even though it was not labeled by NoMoATS. Therefore, although Table 4 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 (see Sec. 5).

4.2.4 Deployment

The NoMoATS system described in Sec. 3 requires a rooted device. As such, it is meant to be used by researchers and filter list curators to assist in collecting datasets and in creating new rules, respectively. On the other hand, the classifiers presented in this section, can be applied on top of a user-space VPN service, such as the open-source AntMonitor [12]. This deployment scenario is illustrated Fig. 6: we use NoMoATS to test apps, label their traffic, and train classifiers at the server. 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 NoMoATS and re-train our classifiers within minutes since the labeling is automatic and training classifiers takes milliseconds (see Table 4).

Once the classifiers are ready, they can be downloaded on the mobile device, where they need to act on live traffic without significantly impacting user experience. To that end, we evaluate both our general and per-app classifiers in terms of their efficiency when applied on a mobile device. In both cases, we use the classifiers trained on the full feature set. For a more even comparison, we took the per-app classifier that had the largest tree size of 12 non-leaf nodes. We then took five negative and five positive samples from the app that resulted in this tree to act as test packets. The packets ranged in size from 63 bytes to 2836. Using a Nexus 6 (Quad-Core 2.7 Ghz CPU, 3 GB RAM) running Android 7.1, we fed these 10 packets to the general classifier and to the per-app one, and recorded the time to classify using `System.nanoTime()`. Each test case was repeated 100 times and we recorded the averages and the standard deviation. The results are reported in Tables 3 and 4 in the last column. We find that both the general classifier and the per-app one can classify a packet within one millisecond, making both models suitable for real-time deployment. As expected, the smaller tree size of the per-app classifier allows for even faster classification – within a quarter of a millisecond.

5 Evaluation Results

In this section, we analyze the quality of NoMoATS labels (Sec. 3) and our classifiers' predictions (Sec. 4.2). Specifically, we look into agreements and disagreements

between popular anti-tracking filter lists and our labels. We also seek to understand if our classifiers can generalize and learn to recognize A&T requests that were not part of the training set. We start by introducing our baseline for comparison – popular filter lists, and how we labeled our dataset with them (Sec. 5.1). Next, we evaluate the quality of NoMoATS labels by comparing them with those of filter lists (Sec. 5.2). Finally, we compare the labels provided by filter lists with both our general classifier (Sec. 5.3) and our per-app classifiers (Sec. 5.4), and we provide insight on our false positives and false negatives.

Row	Auto Label	Filter Lists	Count (%)	Notes
1	0	0	16,054 (42.9%)	negative samples
2	1	1	10,549 (28.2%)	agreements with filter lists
3	1	0	2,963 (7.91%)	disagreements: new A&T samples found
4	0	1	7,872 (21.0%)	disagreements: AutoLabel false negatives
Total Requests			37,438 (100%)	

Table 5. Sec. 5.2: comparing NoMoATS to popular filter lists (EasyList, EasyPrivacy and MoaAB). "0" = negative label; "1" = a positive label (A&T request). Example: row three means that 2,963 requests were detected by NoMoATS, but were not detected by any of the three filter lists.

5.1 Baseline for Comparison

As our baseline for comparison, we use popular adblocking and anti-tracking filter lists that are publicly available, and we match them against the requests in our dataset. Specifically, we compare against EasyList, EasyPrivacy [6], and MoaAB [13]. EasyPrivacy is a small list of less than 17k rules aimed at blocking trackers that are not already blocked by the ad-blocking EasyList that contains ~74k rules. Thus, we use these two lists in conjunction. Although MoaAB is less popular and is mostly aimed at ads, it was reported in [19] to be the most effective list for mobile devices: it blocked the most requests to third parties and fingerprinting scripts. We fetched all three lists on August 7, 2019 and used an open-source Python parser, *adblockparser* [53], to match our data against the rules. The parser can work with any list as long as it is in Adblock Plus format [54]. Both EasyList and EasyPrivacy already come in this format. To convert MoaAB into Adblock Plus format, we follow the guideline for writing filter rules [54]. Since MoaAB operates by modifying a rooted Android's `hosts` file, the conversion is straightforward: for each hostname specified in MoaAB we write an Adblock Plus rule that blocks that hostname. Once the lists are ready, we need to parse our collected requests and feed the required information to *adblockparser*. The details of this procedure are described in Appendix B.

Tables 5, 6, and 7 summarize the results of matching against the three filter lists and comparing the outcome against NoMoATS, our general classifier, and our per-app classifiers, respectively. A label of "0" indicates a negative label, and a label of "1" indicates a positive label – an A&T request. To simplify the tables, we merged all three lists into one column, where "1" indicates that at least one of the filter lists labeled a given sample as positive, and a "0" indicates that all three lists labeled a given sample as negative.

5.2 Evaluating NoMoATS

First, we examine the quality of the labels provided by NoMoATS by analyzing the agreements and disagreements between our labels and the filter lists' labels. Out of 37,438 requests, 16,054 (42.9%) were labeled as negative by both approaches (first row).

Agreements with Filter Lists. We begin by analyzing row two, which represents agreements between NoMoATS and the filter lists. We find that for over 28% of requests, at least one filter list agrees with our labels.

Disagreements with Filter Lists: New A&T Samples Found. Next, we examine the cases where NoMoATS disagreed with the filter lists and found extra positive samples. Row three indicates that 2,963 (7.91%) positive samples were identified by our approach, but were undetected by any of the filter lists. We examine the hostnames contacted by these 2,963 requests and find 162 distinct hosts. A plurality (563) of these requests were destined to `lh3.googleusercontent.com`. Out of these 563 requests, 499 were generated by a prior A&T request, as indicated by the presence of A&T SLDs (*e.g.* `doubleclick.net`) in the HTTP Referer header. It is possible that filter lists do not need to block these as they would never be generated if the ad from `doubleclick.net` was never fetched. However, the remaining 64 requests (out of the 563), did not contain a Referer header and were sending various width and height measurements. Such measurements could be used to fetch ads or to potentially fingerprint the user. Since the `googleusercontent.com` SLD often servers essential content, a more complex rule should potentially be added to EasyList to block A&T requests destined

to this multi-purposed SLD. Some of the other samples out of our 2,963 positive ones detect hosts that should be included filter lists. For example, 286, 71, and 23 requests were destined to SLDs belonging to the *Startapp*, *Tapjoy*, and *AppLovin* mobile advertisers, respectively.

Disagreements with Filter Lists: False Negatives. Finally, we examine row four, which represents NoMoATS false negatives – 7,872 (21%) requests. A plurality of these requests (1,272) were going to `graph.facebook.com`, which is a known tracking domain. NoMoATS 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 (Sec. 6). Another 625 requests (out of the 7,872) were going to `googleads.g.doubleclick.net`. We examined some of the stack traces that led to these requests 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 [55], which provides not just ad libraries, but development libraries as well (*e.g. Google Maps*). Thus, we cannot simply block all requests sent by Google Play Services. 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. Thus, it is possible that the obfuscated package name is the `.ads` package that was able to avoid detection by LibRadar++. This suggests that future work may benefit from combining static and dynamic analysis to discover more package names in apps (Sec. 6). We note that out of the 7,872 requests, 5,246 were labeled as positive by MoaAB only, indicating that MoaAB could contain false positives. Indeed, all 233 requests going to `fls-na.amazon.com` that were blocked by MoaAB alone, were sent by an *Amazon* app. Even more problematic, MoaAB blocked 49 requests destined for `www.okcupid.com`, sent by the *OkCupid* dating app. Thus, some of the 7,872 NoMoATS false negatives could actually be false positives in filter lists.

Summary. We find that the NoMoATS approach is accurate when labeling requests generated by A&T libraries. The reason our method misses certain A&T requests 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 some A&T hosts are package names that avoid detection by LibRadar++. We leave improving LibRadar++ to future work. Overall, our approach can provide suggestions for filter lists: it detected 162 hosts that are candidates for inclusion in filter lists and highlighted potential false positives in MoaAB.

Row	General Prediction	Auto Label	Filter Lists	Count (%)	Notes
1	1	0	1	577 (1.54%)	agreements with filter lists
2	1	0	0	261 (0.70%)	disagreements: new A&T samples found
3	0	1	1	235 (0.63%)	disagreements: prediction false negatives
Total Requests				37,438 (100%)	

Table 6. Sec. 5.3: comparing general classifier predictions to NoMoATS and popular filter lists (EasyList, EasyPrivacy and MoaAB). “0” = negative label; “1” = a positive label (A&T request). Example: row four means that 261 requests were labeled as positive by the classifier, but were labeled as negative by all three filter lists and NoMoATS.

5.3 Evaluating the General Classifier

In this section, we evaluate the performance of the general classifier (Sec. 4.2.2) trained on the full feature set (URL and all the HTTP headers). We examine the agreements and disagreements between the classifier’s predictions, NoMoATS, and filter lists. To that end, in Table 6, we extend Table 5 to include a column for prediction. Note that we omit the discussion of agreements between NoMoATS and the classifier’s predictions as these samples were discussed in the previous section.

Agreements with Filter Lists. Table 3 indicates that the general classifier has a specificity of 96.5%, which signifies a low false positive rate. In total, out of 23,926 samples labeled as negative by NoMoATS, 838 (~3.5%) are predicted as positive. However, Table 6 shows that 577 of such “false positives” are also labeled as positive by at least one filter list. In other words, 577 samples (over 68%) out of 838 “false positive” ones are actually true positives that were missed by NoMoATS for reasons discussed in the previous section. This finding illustrates that our classifiers can pick up patterns that extend past the training data.

Disagreements with Filter Lists: New A&T Samples Found. Next, we examine the remaining 261 false positives that were not detected by any of the filter lists – row two in Table 6. Out of the 261, the majority (138)

were going to the unity3d.com SLD and were either sharing various app events performed by Droidbot or were requesting ads from the unityads.unity3d.com sub-domain. Another 61 requests were destined to other A&T SLDs, such as tapjoy.com, crashlytics.com, and urbanairship.com. These samples demonstrate the general classifier's ability to generalize past the training data and find new A&T hosts that should be included in filter lists. Out of the remaining 77 samples, only 14 were obvious false positives that could cause app breakage (*e.g. Hulu* app sending a request to hulu.com). As for the remaining 63 samples, it is unclear whether or not they are A&T. For instance, 10 requests were destined to googlevideo.com, but it is difficult to tell if they were fetching video ads or content.

Disagreements with Filter Lists: False Negatives. Row three shows that the general classifier resulted in 235 false negatives. We further examined these requests and found that they were affecting 29 distinct SLDs, including some popular A&T SLDs, such as doubleclick.net. Investigating further, we found that for most of such domains, there was at least one request that was marked as negative by NoMoATS (see Sec. 5.2 for more information about NoMoATS false negatives). These false negatives in the labeled data then confused the general classifier and led to false negative predictions. Only four out of the 29 SLDs were never marked as negative by NoMoATS, and in these cases, each domain experienced at most 2 false negative predictions.

Summary. We find that most of the general classifier's false positives are actually false negatives in the NoMoATS labeling mechanism. This illustrates that our classifier can generalize past the ground truth and find new A&T hosts. Similar results were reported by [33]: machine learning algorithms trained with popular filter lists result in "false positives" that highlight false negatives in the ground truth data. In terms of false negatives of the general classifier, most of them also stem from the false negatives of NoMoATS caused by limitations of our system (see Sec. 6).

5.4 Evaluating Per-App Classifiers

In this section, we compare the performance of our per-app classifiers (Sec. 4.2.3) with NoMoATS, the general classifier, and the filter lists. All the classifiers in this section are trained using the full feature set (URL and all the HTTP headers). For a fair comparison, this section of the paper focuses on the 24,100 requests that

	Prediction					
Row	Per-App	General	Auto Label	Filter Lists	Count	Notes
1	1	0	0	1	54 (0.22%)	agreements with filter lists
2	1	1	0	1	41 (0.17%)	
3	1	0	0	0	207 (0.86%)	disagreements: new A&T samples found
4	1	1	0	0	65 (0.27%)	
5	0	0	1	1	43 (0.18%)	disagreements: prediction false
6	0	1	1	1	193 (0.80%)	
Total Requests					24,100 (100%)	

Table 7. Comparing per-app classifiers' predictions to general classifier's predictions and to filter lists. "0" = negative label; "1" = a positive label (A&T request). Example: row three means that 207 requests were labeled as positive by per-app classifiers, but were labeled as negative by all other approaches.

we were able to label with our per-app classifiers, *i.e.* requests belonging to apps with less than 10 samples of each type are absent from this analysis. To that end, in Table 7, we extend Table 6 to include a column for per-app classifiers. Note that we omit the discussion of agreements between NoMoATS and the per-app classifiers' predictions as these samples were discussed in Sec. 5.2.

Agreements with Filter Lists. Table 4 shows that the per-app classifiers suffer from a higher false positive rate than the general classifier (Table 3): specificity is below 92%, and there is high variance among the per-app classifiers. In total, out of 11,693 samples labeled as negative by NoMoATS, 367 (over 3%) are predicted as positive. However, Table 7 shows that 95 of those "false positives" are also labeled as positive by at least one filter list (rows one and two). Moreover, 54 out of the 95 samples were correctly labeled as positive by the per-app classifiers, but not by the general one (row one). This illustrates that the per-app classifiers are able to learn patterns that not only extend past the training data but are also different from the patterns picked up by the general classifier.

Disagreements with Filter Lists: New A&T Samples Found. Next, we examine the 207 false positives that were not detected by any filter list nor by the general classifier – row three in Table 7. A plurality (67) of these requests were destined to the unity3d.com SLD, similar to what we observed with the general classifier in Sec. 5.4. Examining these 67 requests further, we found that *all* of them were sending various information about the device, including a custom identifier that remained the same across seven different apps. Similarly, 40 requests destined to settings.crashlytics.com were also sending a unique identifier – potentially performing cross-app tracking. This highlights that blocking stan-

dard PII, such as advertiser ID, is not enough to prevent tracking since third-party libraries can create their own identifiers and use them across apps (see Appendix C for further discussion about PII). 30 more requests were going to app-measurement.com (an A&T domain), and the remaining 70 requests were spread among 38 different SLDs, with five or less requests going to each. We randomly selected five of these SLDs for manual inspection. One request was going to googlevideo.com and it is difficult to tell if it was requesting a video ad or content. Curiously, two requests were going to a hardcoded IP - 203.107.1.1. Looking up the IP reveals that it belongs to the Alibaba Advertising Co. This illustrates another shortcoming of filter lists - they can be circumvented by hard-coding IP addresses. Two requests by the *Zillow* app were sent to splkmobile.com (an analytics company). The remaining two requests were false positives that could lead to app breakage: one was going to moovitapp.com and it was sent by the *Moovit* app, and another was going to cyberlink.com, sent by an app developed by *CyberLink.com*.

Disagreements with Filter Lists: False Negatives. Finally, we examine the 193 false negatives that were marked as negative by the per-app classifiers but not by the general one – row six in Table 7. These requests were sent by 76 unique apps to 22 unique SLDs. We randomly selected five of these apps to examine their trees and the reasons behind the false negatives. In all the apps we examined, there were one to three false negatives, and each sample was generated by a different cross-validation split. In three of the five apps, the false negatives were samples that appeared infrequently in the dataset, and thus were not part of the training set in some cross-validation splits. For example, in one app all three false negatives were the only ones in the app's dataset to be destined to the android.bugly.qq.com host. In the remaining two cases, the decision trees contained words such as "Mobile" and "Android," which are common words that appear in the User-Agent HTTP header field. This indicates that we can further improve our feature selection to help the classifiers pick better features.

Summary. We find that our per-app classifiers are able to uncover A&T samples that are missed by NoMoATS, the general classifier, and by filter lists. As with the general classifier, the per-app ones also suffer from actual false positives that could lead to app breakage. We also find that our false negative rate can be further improved by better feature selection and by data balancing to increase appearances of infrequent samples. While the per-app classifiers perform similarly to the general one, the per-app classifiers have the advantage of easy inspection and correction of faulty decision trees that are only a few levels deep.

6 Limitations and Future Work

In this section, we discuss the limitations of this work and outline future directions to remedy them. First, NoMoATS relies on having a labeled list of A&T libraries, which still requires human involvement, albeit decreased from millions of packers and tens of thousands of rules to just hundreds of libraries. In the future, this process can be further automated by crawling webpages of libraries and searching for keywords.

Second, some mobile libraries are multi-purposed (*e.g.* they provide development tools, but may also engage in tracking). In certain cases, our proposed classifiers have already learned to identify tracking behavior in requests sent by non-A&T libraries. We can use these samples to feed back into our labels. Another idea is to statically analyze which development and social libraries explicitly request the Internet permission. Certain libraries have legitimate reasons for accessing the Internet, *e.g.* the *Facebook* library providing a login option. In contrast, other libraries first check if their host app has Internet access and then abuse that permission. Such libraries can be safely blocked by our classifiers. A third option is to supplement NoMoATS with filter lists. Relatedly, throughout the paper, we have considered advertising and tracking together, which is reasonable. Indeed, it is difficult to distinguish the two functionalities as many advertising libraries are also engaged in tracking. Furthermore, tracking on mobile devices is most often done for the purpose of showing personalized ads. For instance, it was shown in [8] that most of the data collection on mobile devices is done by Google and Facebook, whose primary source of revenue is advertising [56, 57]. Future work can explore the possibility of looking into libraries that only do tracking but do not show ads (*e.g.* analytics or development libraries) and then using the collected data to learn what patterns can be used to distinguish advertising from tracking.

Third, our current method does not check if blocking A&T requests causes applications to break. In the future, we plan to explore if Droidbot can be used to help us detect broken apps. Since Droidbot can explore UI elements in a predictable manner, any broken functionality should be detectable across different test

runs: one with blocking disabled and one with blocking enabled. Another option is to follow the filter list approach: release our tool and rely on user reports for detecting app breakage.

Fourth, for real-time A&T blocking (not labeling and training), our design choice of using VPN to intercept traffic applies to HTTP traffic, and HTTPS where it can be decrypted. However, decryption is not possible when certificate pinning is used. In such cases, we can train hostname-based classifiers, which still achieve F-scores of over 90% and can be applied on DNS requests or on the TLS SNI field. Another approach is to use static analysis and modify a given app to trust user-installed certificates, or to use a rooted device – see Sec. 2 for alternative design choices and their trade-offs.

Fifth, the tools we use (Frida, Droidbot, LibRadar++) have certain limitations of their own. For example, in the case Frida, apps can detect that they are being executed in a different, rooted environment and can choose to act differently or to not work at all (as we saw in some of our experiments). Fortunately, we can use Frida hooks to hook into common methods for root detection and provide false responses to apps. In addition, we saw that in some cases Frida failed to acquire stack traces or had incompatibilities with certain apps. We hope that future releases of Frida will address these issues. As for Droidbot, it often fails to fully explore applications requiring a login. However, most modern apps support third-party login functionality (*e.g.* via *Facebook* or *Google*). Future releases of Droidbot can include the option to seek out these buttons and press on them. Another approach is to use a system such as CHIMP [52] to crowdsource inputs for more complex apps that Droidbot cannot handle, such as games written in Unity. Finally, LibRadar++ sometimes fails to detect obfuscated A&T package names. One way to address this limitation is to have a feedback loop from our classifiers to LibRadar++ to help it learn new obfuscated A&T package names.

Other, minor, possible extensions include the following. Our machine learning approach can be further improved with better feature engineering and data balancing. We can also avoid confusing our classifiers by reconstructing the HTTP Referer chain and excluding HTTP requests that would not have been generated if their referrer was blocked.

7 Conclusion

In this paper, we presented NoMoATS – a system that can automatically label outgoing mobile network requests with the A&T (advertising or tracking) library that was responsible for generating them. NoMoATS addresses the major bottleneck of both filter list-based and machine learning-based approaches to ad-blocking and anti-tracking. Specifically, it removes the need for humans to label network traces, and instead requires labeling of libraries only. This allows for improved scalability of labeling: there are only hundreds of A&T libraries [1] as opposed to thousands of filter list rules [6] that aim to correctly filter millions of HTTP requests. Furthermore, the purposes of libraries (A&T or not) change in much longer time scales than the network behavior of apps and trackers, eliminating the need for frequent manual updates. Network traces labeled with NoMoATS can be used to train classifiers or to provide suggestions to filter list curators. In our work, we used NoMoATS to collect a new dataset, the first of its kind: outgoing packets labeled with not just ad requests (*e.g.* as in [9]) but also tracking (which is more difficult to visually inspect), and at the fine granularity of HTTP/S requests (not just tracking domains or hostnames, *e.g.* as in [8]). We then used 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 the open-source AntMonitor [12]. We showed that our classifiers can achieve high F-scores of 93% on average, and can be trained in the order of milliseconds. Finally, using popular filter lists (EasyList, EasyPrivacy, and MoaAB), we showed that our classifiers generalize past the training data: they find samples that are also labeled as positive by filter lists but are missed by NoMoATS. To enable further research, we release our dataset and open-source the NoMoATS tool [14].

Acknowledgements

This work was supportedby the NSF Awards 1815666 and 1649372 and by a UCI Seed Funding Award. A special thanks goes to Hieu Le for his help with PII analysis. We would also like to thank the anonymous PETS reviewers for helping us improve this paper.

References

[1] AppBrain. Android library statistics. https://www.appbrain.com/stats/libraries/. (accessed Feb. 2019).

[2] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Proc. 13th Annu. Int. Conf. Mobile Systems, Applications, and Services*, pages 89–103, Florence, Italy, May 2015.

[3] 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. In *Proc. ACM Interactive, Mobile, Wearable and Ubiquitous Technologies*, volume 1, page 42, 2017.

[4] Julian Andres Klode. DNS-based Host Blocker for Android. https://github.com/julian-klode/dns66. (accessed Nov. 2019).

[5] AdGuard. AdGuard for Android. https://adguard.com/en/adguard-android/overview.html. (accessed Nov. 2019).

[6] EasyList. EasyList. https://easylist.to. (accessed Nov. 2019).

[7] 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*, volume 16, New York, NY, USA, 2016.

[8] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, and Christian Kreibich Phillipa Gill. Apps, Trackers, Privacy, and Regulators. In *Proc. Network and Distributed System Security Symp.*, volume 2018, San Diego, CA, USA, Feb. 2018.

[9] Anastasia Shuba, Athina Markopoulou, and Zubair Shafiq. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. volume 2018, Barcelona, Spain, Jul. 2018.

[10] 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.

[11] 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*, pages 15–26, Denver, CO, USA, Feb. 2015.

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

[13] BSDgeek_Jake. MoaAB: Mother of All AD-BLOCKING. https://forum.xda-developers.com/showthread.php?t=1916098. (accessed Nov. 2019).

[14] UCI Networking Group. NoMoAds Open Source. http://athinagroup.eng.uci.edu/projects/nomoads. (accessed Dec. 2019).

[15] Peter Eckersley. How Unique Is Your Web Browser? In *Proc. Privacy Enhancing Technologies*, pages 1–18, Berlin, Germany, Jul. 2010.

[16] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting. In *2013 IEEE Symp. Security and Privacy*, pages 541–555, San Francisco, CA, USA, May 2013.

[17] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the Web for Fingerprinters. In *Proc. 2013 ACM SIGSAC Conf. Computer and Communications Security*, pages 1129–1140, Berlin, Germany, Nov. 2013.

[18] Steven Englehardt and Arvind Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *Proc. 2013 ACM SIGSAC Conf. Computer and Communications Security*, pages 1388–1401, Berlin, Germany, Nov. 2016.

[19] 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 *Proc. 2017 IEEE European Security and Privacy*, pages 319–333, Paris, France, Apr. 2017.

[20] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proc. 2013 ACM SIGSAC Conf. Computer and Communications Security*, pages 1515–1532, Berlin, Germany, Nov. 2018.

[21] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices*, 49(6):259–269, 2014.

[22] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proc. Network and Distributed System Security Symp.*, volume 15, page 110, San Diego, CA, USA, Feb. 2015.

[23] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale. In *Proc. Int. Conf. Trust and Trustworthy Computing*, pages 291–307, Vienna, Austria, Jun. 2012.

[24] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proc. 5th ACM Conf. Security and Privacy in Wireless and Mobile Networks*, pages 101–112, Tucson, AZ, USA, Apr. 2012.

[25] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An Information-Flow Tracking System for Real-Time Privacy Monitoring on Smartphones. *ACM Transactions Computer Systems*, 32(2):5, 2014.

[26] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In *Proc. 18th ACM Conf. on Computer and Communications Security*, pages 639–652, Chicago, IL, USA, Oct. 2011.

[27] AdGuard: Content Blocker for Samsung and Yandex. https://play.google.com/store/apps/details?id=com.adguard.android.contentblocker. (accessed Nov. 2019).

[28] Apple. Safari. https://www.apple.com/safari. (accessed Nov. 2019).

[29] Adblock Plus. Adblock Plus. https://adblockplus.org. (accessed Nov. 2019).

[30] Jason Bau, Jonathan Mayer, Hristo Paskov, and John C Mitchell. A Promising Direction for Web Tracking Countermeasures. In *Proc. of W2SP*, San Francisco, CA, USA, May 2013.

[31] Sruti Bhagavatula, Christopher Dunn, Chris Kanich, Minaxi Gupta, and Brian Ziebart. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *Proc. 2014 Workshop Artificial Intelligent and Security Workshop*, pages 95–102, Scottsdale, AZ, USA, Nov. 2014.

[32] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. An Automated Approach for Complementing Ad Blockers' Blacklists. volume 2015, pages 282–298, Philadelphia, PA, USA, Jun. 2015.

[33] Umar Iqbal, Zubair Shafiq, Peter Snyder, Shitong Zhu, Zhiyun Qian, and Benjamin Livshits. AdGraph: A Machine Learning Approach to Automatic and Effective Adblocking. *arXiv preprint arXiv:1805.09155*, 2018.

[34] Zain ul Abi Din, Panagiotis Tigas, Samuel T King, and Benjamin Livshits. Percival: Making In-Browser Perceptual Ad Blocking Practical With Deep Learning. *arXiv preprint arXiv:1905.07444*, 2019.

[35] Antoine Vastel, Peter Snyder, and Benjamin Livshits. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. *arXiv preprint arXiv:1810.09160*, 2018.

[36] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *Proc. 2017 Internet Measurement Conf.*, London, UK, Nov. 2017.

[37] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proc. 38th Int. Conf. Software Engineering Companion*, pages 653–656, Austin, TX, USA, May 2016.

[38] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets. In *Proc. 2018 Internet Measurement Conf.*, pages 293–307, Boston, MA, USA, Oct. 2018.

[39] Ma Zi'ang. LiteRadar. https://github.com/pkumza/LiteRadar. (accessed Nov. 2019).

[40] Ole André V. Ravnås. Frida. https://www.frida.re. (accessed Nov. 2019).

[41] Domenico Iezzi. Google Play Unofficial Python API. https://github.com/NoMore201/googleplay-api. (accessed Nov. 2019).

[42] Unity. Unity Ads. https://unity.com/solutions/unity-ads. (accessed Nov. 2019).

[43] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: a Lightweight UI-guided Test Input Generator for Android. In *Proc. 2017 IEEE/ACM 39th Int. Conf. Software Engineering Companion*, pages 23–26, Buenos Aires, Argentina, 2017.

[44] 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.

[45] Haojian Jin, Minyi Liu, Kevan Dodhia, Yuanchun Li, Gaurav Srivastava, Matthew Fredrikson, Yuvraj Agarwal, and Jason I Hong. Why Are They Collecting My Data?: Inferring the Purposes of Network Traffic in Mobile Apps. volume 2, page 173, 2018.

[46] iana. Permanent Message Header Field Names. http://www.iana.org/assignments/message-headers/message-headers.xhtml#perm-headers. (accessed Sep. 2019).

[47] Chartboost. Chartboost. https://www.chartboost.com. (accessed Nov. 2019).

[48] AppBrain. Android and Google Play statistics. https://www.appbrain.com/stats/. (accessed Sep. 2019).

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

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

[51] Unity. Unity 3D. https://www.unity3d.com. (accessed Nov. 2019).

[52] Mario Almeida, Muhammad Bilal, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Matteo Varvello, and Jeremy Blackburn. CHIMP: Crowdsourcing Human Inputs for Mobile Phones. In *Proc. 2018 World Wide Web Conf.*, pages 45–54, Lyon, France, Apr. 2018.

[53] Scrapinghub. adblockparser. https://github.com/scrapinghub/adblockparser. (accessed Nov. 2019).

[54] eyeo. Writing Adblock Plus Filters. https://adblockplus.org/filters. (accessed Nov. 2019).

[55] Google. Google APIs for Android. https://developers.google.com/android/reference/packages. (accessed Nov. 2019).

[56] Google. Alphabet Announces Fourth Quarter and Fiscal Year2018 Results. https://abc.xyz/investor/static/pdf/2018Q4_alphabet_earnings_release.pdf, 2019. (accessed Nov. 2019).

[57] Facebook. Facebook Reports Third Quarter 2019 Results. https://investor.fb.com/investor-news/press-release-details/2019/Facebook-Reports-Third-Quarter-2019-Results, 2019. (accessed Nov. 2019).

[58] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. Studying TLS Usage in Android Apps. In *Proc. 13th ACM Int. Conf. Emerging Networking Experiments and Technologies*, pages 350–362, Seoul/Incheon, South Korea, Dec. 2017.

[59] Google. Android Accessibility API. https://developer.android.com/guide/topics/ui/accessibility. (accessed Nov. 2019).

[60] Google. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey. (accessed Nov. 2019).

[61] Adblock Plus. Adblock Plus Library for Android. https://github.com/adblockplus/libadblockplus-android. (accessed Nov. 2019).

[62] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M Pujol. Tracking the Trackers. In *Proc. 2013 World Wide Web Conf.*, pages 121–132, Montreal, Canada, April 2016.
[63] Enric Pujol, Oliver Hohlfeld, and Anja Feldmann. Annoyed users: Ads and Ad-Block Usage in the Wild. In *Proc. 2017 Internet Measurement Conf.*, pages 93–106, Tokyo, Japan, Oct. 2015.

A System Details

In this appendix, we provide details on the NoMoATS system (Sec. 3).

A.1 Stack Traces Intuition

In this section, we provide the intuition behind how we find which requests were made by A&T libraries (Sec. 3.1). Figures 7a and 7b present trimmed stack traces that were captured within the *ZEDGE™Ringtones & Wallpapers* app, which has the package name `net.zedge.android`. Fig. 7a 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, Fig. 7b shows the *MoPub* ad library starting an SSL handshake, as indicated by the `com.mopub` package name.

A.2 SSL Libraries

In Sec. 3.2, we described how we hooked the OpenSSL `SSL_write` function to captured TLS/SSL traffic before it becomes ciphertext. This hook works when apps use the version of OpenSSL provided by the Android OS. To address scenarios where apps include their own versions of OpenSSL, we use the following procedure. We hook into the `java.lang.System` `loadLibrary` and `load` function calls to catch occurrences of native libraries being loaded. From there, 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. In our experiments, we have seen apps load their own versions of OpenSSL, but Droidbot did not trigger the functionalities required to make the apps use their OpenSSL modules. We note that it is also possible for apps to include SSL libraries other than OpenSSL, but a 2017 study has shown that only 14% of apps (out of the studied 7,258) use third-party SSL libraries [58]. Furthermore, apps that do use other SSL libraries are usually browser-type apps which are not the focus of this paper (see Sec. 4.1.1).

A.3 Network Requests Formats

In this section, we provide details on how we save network requests captured by Frida in Sec. 3.2. In the case of non-WebView traffic, we save the requests in PCAPNG format with the following procedure. 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 the function call responsible for generating the packet.

To save WebView traffic, we utilize the JSON format as follows. WebView traffic is always sent over HTTP/S, and the `shouldInterceptRequest` 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. 3.2): the full URL, the HTTP headers, and the HTTP method.

Finally, to facilitate parsing and training of machine learning classifiers, we convert all PCAPNG files to JSON using *tshark*. For consistency with the data collected from WebViews, we keep only the relevant HTTP fields, namely the full URL, the HTTP headers, and the HTTP method. Based on our stack trace analysis from Sec. 3.1, we also add a label to each JSON data point, indicating whether or not it contains an A&T request. These JSON files can then be used to train machine learning classifiers, as described in Sec. 4.2.

```
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)

Fig. 7. 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` and the absence of A&T package names in the stack trace; (b) The *MoPub* ad library is starting an SSL handshake, as indicated by the `com.mopub` package name.

A.4 Droidbot

To automatically exercise apps at scale (Fig. 2), we used Droidbot – a lightweight tool that requires no modifications to the Android OS and no application instrumentation. Droidbot consists of two components: an Android app and a Python script that runs on a connected PC. The Droidbot Android app utilizes the Android Accessibility API [59] 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 automatic and repeatable 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 [9], or both manually and with the UI/Application Exerciser Monkey [60] (*e.g.* [7]). 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 UI. 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. [45] 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.

B Matching Against Filter Lists

In this appendix, we describe how we parsed our collected requests and fed the required information to *adblockparser* (Sec. 5.1). The tool takes in the full URL and several options that are used by Adblock Plus. 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 requested (*e.g.* an image or an HTML document). We note that Adblock Plus 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 Adblock Plus library for Android [61] uses the exact same three options. To determine the content type of the request, we follow the same logic as written in the Adblock Plus library for Android [61]: 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 if the request is to a third-party, we compare the origin (defined by the scheme, host, and port) of the URL be-

ing requested to the origin of the URL specified in the HTTP Referer header. This method allows us to extract the three options needed to feed into *adblockparser* along with the full URL request.

C Measuring Tracking

Throughout the Evaluation Section (Sec. 5), we used the number of A&T requests as a measure for the volume of tracking. Although reasonable, and consistent with prior work [9, 19, 62, 63], this is not necessarily the only measure for tracking. Other approaches, such as ReCon [7] and AntMonitor [12], have measured the number of PII-containing requests. For comparison against such approaches, we considered seven popular PII and searched for them in the NoMoATS dataset. Specifically, we searched for the Android Device ID, IMEI, email, location coordinates, device serial number, advertiser ID, and MAC address. We also searched for the MD5 and SHA1 hashes of these values. We found that only 2,684 requests in our dataset contain PII, as opposed to the 13,512 requests that NoMoATS identified as A&T. Furthermore, out of these 2,684 PII-containing requests, less than 400 were labeled as A&T by NoMoATS and EasyList with EasyPrivacy. This finding is consistent with what was discovered in [8] as a recent trend – only 14.4% of A&T services collect explicit identifiers, suggesting that trackers have moved from collecting PII to other techniques, such as fingerprinting, in order to evade detection. Thus, measuring tracking on the use of PII alone also has limitations.