AntShield: On-Device Detection of Personal Information Exposure

Evita Bakopoulou
University of California, Irvine

Milad Asgari Mehrabadi
University of California, Irvine

AntShield: On-Device Detection of Personal Information Exposure

Anastasia Shuba[1], Evita Bakopoulou[1], Milad Asgari Mehrabadi[1], Hieu Le[1], David Choffnes[2], and Athina Markopoulou[1]

[1]University of California, Irvine

[2]Northeastern University

ABSTRACT

Mobile devices have access to personal, potentially sensitive data, and there is a growing number of applications that transmit this personally identifiable information (PII) over the network. In this paper, we present the AntShield system that performs *on-device* packet-level monitoring and detects the transmission of such sensitive information accurately and in real-time. A key insight is to distinguish PII that is *predefined* and is easily available on the device from PII that is *unknown* a priori but can be automatically detected by classifiers. Our system not only combines, for the first time, the advantages of *on-device* monitoring with the power of *learning* unknown PII, but also outperforms either of the two approaches alone. We demonstrate the real-time performance of our prototype as well as the classification performance using a dataset that we collect and analyze from scratch (including new findings in terms of leaks and patterns). AntShield is a first step towards enabling distributed learning of private information exposure.

1. INTRODUCTION

Mobile devices have access to a wealth of personal, potentially sensitive information and there is a growing number of applications that access, process and transmit some of this information over the network. Sometimes this is justified (required for the intended operation of the applications, *e.g.* location is needed by GoogleMaps) and controllable (*e.g.* by the user through permissions), but for the most part, users are not in control of their data today. Applications and third party libraries routinely transmit user data to remote servers, including adservers and trackers, and users are typically unaware of how their personal data is shared and for what purpose.

Prior work on improving data transparency and identifying potential privacy leaks includes static and dynamic analysis and network-centric approaches. In this paper, we take the latter approach: personal information leaks happen, by definition, over network traffic, therefore a natural and comprehensive vantage point to identify and control leaks is at the network layer. Traffic can be monitored in the middle of the network (as in Meddle [1] and Recon [2]) and/or on the device itself (as in AntMonitor [3][4] and Lumen (a.k.a. Haystack) [5]). A key challenge for network-based monitoring is how to analyze traffic both efficiently and securely. The current state-of-the-art consists of the following complementary approaches. On one hand, AntMonitor [3] and Lumen [5] detect leaks on the device, but require a blacklist of strings (potential PII leaks) known a priori to search for. Therefore, they are unable to detect leakage of information that changes dynamically or is not part of the list. On the other hand, Recon [2] recently addressed this limitation, by training classifiers in a fully centralized way. However, the implementation relied on a trusted, remote proxy to route and analyze traffic, which potentially impacts scalability and security.

We adopt the on-device network monitoring paradigm, which presents both opportunities and challenges. On the upside, it obviates the need for a trusted infrastructure and gives full control to the user, which we believe is the right approach in privacy. Devices also have access to important contextual information, such as certain personal information available on the phone, and which apps are responsible for transmitting packets. On the downside, mobile devices have limited resources to conduct traffic analysis, including deep packet inspection (DPI), and training and applying machine learning classifiers for inferring leaks of PII. It is currently an open question as to how to train machine learning classifiers to retain high accuracy in a truly distributed manner.

In this paper, we take the first step towards enabling distributed learning of personal information leaks from network traffic. We present AntShield - a system that performs efficient on-device analysis, provides accurate and comprehensive data privacy protection, and gives users transparency and control over their personal information in real-time. A key insight is the distinction between PII that is *predefined* by the user or is readily available on the device, from PII that is a priori *unknown* and should be inferred by classifiers. We propose a hybrid String Matching-classification approach: (i) we build on the AntMonitor Library [3] for intercepting packets on the device and looking for *predefined* strings in real-time and (ii) we build classifiers for the remaining *unknown* PII.

The contributions of this paper are the following:

- the AntShield *System.* We present the first system to detect PII exposure (using a *hybrid* DPI and classification approach), 100% on the device (from user space and without routing traffic through a remote VPN server), and in real-time (in ~1 ms). This is enabled by our system design and multiple optimizations.

- *Classification Methodology.* Our *multi-label* classification methodology (Binary Relevance with Decision Trees) achieves significantly higher accuracy (8-25% improvement) and lower variance (a factor of 2-5) compared to state-of-the-art. We also design and advocate for *per-app*, instead of *per-domain*, classifiers: they achieve similar classification accuracy, but allow faster and more scalable operation while covering more traffic.

- *Dataset and Analysis.* In order to demonstrate the effectiveness of our approach, we collect a new (larger and richer than previously available) dataset of privacy leaks on mobile devices, which we will make available to the community. As a side contribution, we analyzed the dataset, identified previously unseen leaks (including leaks over plain TCP and UDP, leaks while the app is in the background, and malicious scanning for rooted devices) and behavioral patterns (*e.g.* communities of domains and mobile apps involved in exposing private information).

The structure of the rest of the paper is as follows. Section 2 briefly reviews related work. Section 3 describes our system's rationale, design and implementation. Section 4 evaluates AntShield's classification accuracy and run-time performance; it also presents our collected dataset and findings therein. Section 5 concludes the paper.

2. RELATED WORK

Different communities are working on improving data transparency and exposing or preventing potential privacy leaks. *Permissions* are useful but not sufficient: (i) users typically accept to install apps by default; (ii) permissions do not protect against inter-app communication and poorly documented system calls; and (iii) they do not capture run-time behavior. Using a *custom OS* or a rooted phone one can get access to fine-grained information on the device, including network traffic. Phonelab [6] and others [7, 8] use packet capturing APIs such as tcpdump or iptables-log. These are powerful but inherently limited to small scale-deployment as the overwhelming majority of users do not have rooted phones, and wireless providers and phone manufacturers strongly discourage rooting. The same limitation applies to approaches that use a custom OS to dynamically intercept leaks (*e.g.* TaintDroid [9]) or permission requests to certain resources (*e.g.* AppFence [10]). Static analysis tools such as AndroidLeaks [11] and PiOS [12] are limited by having to constantly download and analyze all available apps, which is not scalable. Moreover, static analysis suffers from inherent imprecisions, is not representative of what can happen at run-time, and cannot deal with native or dynamically loaded code.

Within the network measurements community, a number of prior works [1, 2, 3, 4, 5] have looked for personal information leaks in network traffic. This includes monitoring in the middle of the network (as in Meddle [1] and Recon [2]) or on the device itself (as in AntMonitor ([3, 4]) and Lumen (a.k.a. Haystack) [5]). AntMonitor and Lumen detect leaks on the device, but require a blacklist of strings (potential PII leaks) known a priori to search for; therefore, they are unable to detect leakage of information that changes dynamically or is not part of the list.

To remedy this limitation, Recon recently applied machine learning techniques to predict whether or not a given packet contains PII [2]. They broke packets into words based on delimiters (*e.g.* '?', '=', ':') and then used these words as features in classification. Various methods were used to ensure that the PII themselves and strings that occur too often or too infrequently are not part of the feature list, see [2] for details. To decide whether or not a packet contains PII (a binary classification problem), Recon used the Java Weka library's [13] C4.5 Decision Tree (DT), and then heuristics for extracting the type of leak. To improve classification accuracy Recon built specialized classifiers for each destination domain that received enough data to train such a classifier. For the rest of the domains, a *general* classifier was built. For the heuristic step, Recon maintained a list of probabilities that a particular key-word corresponds to a PII value. For each PII type, the probability was calculated by taking the number of times the key was present in a packet with the given PII, and dividing it by the number of times the key appeared in all packets. During PII extraction, Recon looked for keys with probability higher than an empirically computed threshold. Their code and dataset are available at [14].

Recon is the closest to our work, thus we use it as our baseline for comparison throughout the paper. The key difference lies in the centralized vs distributed approach. Recon collected its datasets in the middle of the network, trained and applied the classifiers in a centralized way. AntShield operates on the device, which poses unique system challenges and learning opportunities, and paves the way for truly distributed learning of privacy leakage.

3. SYSTEM DESIGN & IMPLEMENTATION

3.1 Goals and Design Rationale

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

- Device Identifiers: IMEI, AndroidID, phone number, serial number, ICCID, MAC Address.

- User identifiers: credentials (per app, usually transmitted over HTTPS), advertiserID, email.
- User demographic: first/last name, gender, zipcode, city, *etc.* - unavailable through Android APIs.
- Location: (latitude and longitude coordinates, available through Android APIs.
- User-defined: the user can also define any custom string that should be monitored (*e.g.* see GUI in Fig. 2(a)), such as digits of her credit card.

A key insight of our design is the distinction on whether PII of interest is known to the device or not. We refer to PII that consists of strings known a priori on the device (*e.g.* via Android APIs, or defined by the user) as *predefined*. We refer to PII that is not known to our AntShield system (*e.g.* hidden from apps or changing dynamically) as *unknown*. By default, we assume that any PII available via Android API calls are *predefined* (*e.g.* IMEI, AndroidID, phone number, serial number, ICCID, MAC Address, advertiserID, email, and location), and the rest are *unknown* (*e.g.* username login, password, first/last name, gender, zipcode, and city).

Our system employs different techniques to detect the transmission of *predefined* PII (String Matching) and *unknown* PII (classification). We refer to the transmission of a packet from the device to the network, containing at least one PII, as a *privacy exposure (or leak)*. This transmission may be: (i) intended to collect information about the user; (ii) benign, *e.g.* necessary for the operation of the app, acceptable to the user, or (iii) of the honest-but-curious nature. Distinguishing between privacy *exposure* and an actual privacy *leak* is out of the scope of this paper, and we refer to the two terms interchangeably, meaning "exposure." Our goal is to detect privacy exposures *on the device* with low overhead, accurately and in real-time. This is a first step towards enabling distributed learning of PII exposures.

Design Objectives and Choices. First, we want a solution that can be used by the non-sophisticated end-user: a mobile app that the user can simply install (as an app, without rooting the phone), enable in the background, and occasionally interact with. Second, we want a solution that operates purely on the device and does not redirect traffic through a middle server. This has several advantages: it does not need to expand the trust base (data does not need to leave the user's device), and it is well positioned to have access to rich information available on the device (app names, and *predefined* PII). To meet both of these goals, we use the AntMonitor Library v0.1.5 [3]. To the best of our knowledge, AntMonitor is the most efficient implementation (in terms of throughput, battery and other resources) for on-device packet interception and inspection of both unencrypted and encrypted traffic, today; see [3] for details. AntMonitor relies on a VPN service on the device (but not on a remote VPN server), which is the only way to intercept traffic today without rooting the phone. If more efficient libraries for packet interception become available in the future, AntShield's modular design allows to replace this component.

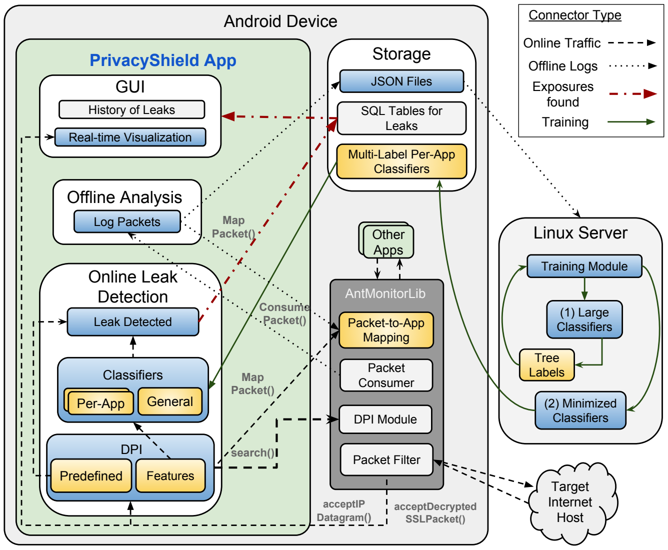


Figure 1: AntShield Architecture. It consists of a mobile app on the device and a remote server. Real-time classification consists of the following steps: each packet is intercepted by AntMonitor Library, mapped to an app, and analyzed for multiple *predefined* and *unknown* leaks; detection occurs before the packet is forwarded towards its remote destination (and an action may be taken to block the leak). Offline operations include loading and (re)training the classifiers, and (if the user agrees) uploading logs to the server.

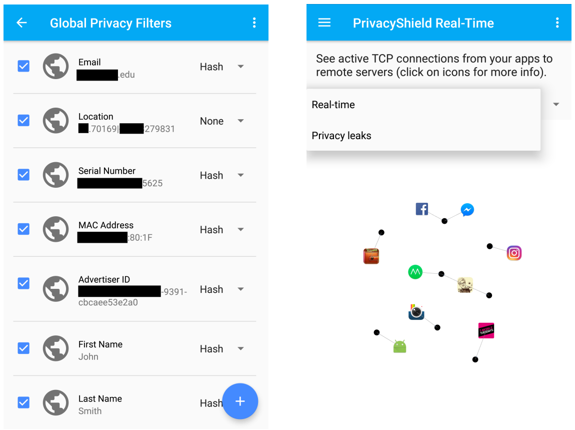
Third, we want to accurately detect a comprehensive range of PII in real-time, both *predefined* (through String Matching), and the remaining *unknown* ones through machine learning models. Towards the first goal, we utilize the DPI API provided by AntMonitor [3]. Towards the second goal, we build specialized classifiers defined in Sec. 3.3.

3.2 PrivacyShield Architecture

The overview of the AntShield architecture is depicted in Fig. 1. It consists of a mobile app and an (optional) server. A brief overview of each component is described next.

Online Leak Detection. This is the core functionality of our PII exposure detection. As shown in Fig. 1, AntShield leverages calls to AntMonitor Library (acceptIPDatagram and acceptDecryptedSSLPacket) to intercept packets (in clear text or decrypted SSL, respectively). Each outgoing intercepted packet is analyzed with DPI for *predefined* leaks and for features. The features are then passed to classifiers to detect *unknown* leaks (described in detail in Sec. 3.3). Either way, if a PII exposure is detected, the user is notified and the exposure is logged. If the user chooses to, the leaky packet can be blocked, or it can be allowed to continue towards its remote destination.

Offline Analysis. This module can be used when heavier processing is required. For example, to generate logs


Global Privacy Filters

☑ Email [redacted].edu Hash

☑ Location [redacted].70169|[redacted]279831 None

☑ Serial Number [redacted]5625 Hash

☑ MAC Address [redacted]:80:1F Hash

☑ Advertiser ID [redacted]-9391-cbcaee53e2a0 Hash

☑ First Name John Hash

☑ Last Name Smith Hash

PrivacyShield Real-Time

See active TCP connections from your apps to remote servers (click on icons for more info).

(a) *predefined* PII, possible actions, and custom filters (name)

(b) Real-Time Visualization: which apps transmit PII to which remote servers

Figure 2: Select Screenshots of the AntShield Android App.

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

Storage (on-Device and/or Server). The AntShield app comes pre-loaded with classifiers trained on our existing dataset (Sec. 4.1), so that users have no need to contact any server and can use the system as-is. Only if the user chooses to do so, logs (packet traces, JSON or other meta data) can be maintained on the device and/or occasionally be uploaded to a server. The use of the server is optional – by default, logs do not need be collected or leave the device. The user may choose to share her data with the server to get the benefits of crowdsourcing, and retrained classifiers. As an example scenario, we used the logging capability of the Offline Analysis module to generate the dataset used in our evaluation. Specifically, each captured packet was labeled with the type of PII that it was leaking and the app name that generated the packet. The PII itself was replaced, and the packet was converted to a JSON format for easier processing at the machine learning training stage (see Sec. 4.1 for details). In general, this feature is useful for other researchers who may wish to generate their own datasets manually or from user studies.

GUI. This component has two purposes. It allows the user to specify various preferences, most importantly, the *predefined* PII to be monitored. By default, these include PII available on the device through Android APIs, as shown in Fig. 2(a). Users that trust AntShield can also opt-in and predefine additional PII, such as name and gender or any string (*e.g.* digits of a credit card). Second, AntShield's GUI notifies the user about PII exposed. From here, users can decide to allow the leak to happen, replace the exposed PII with a random string of the same length (so as not to alter the payload size), or block the packet completely. Whatever action the user selects, it is remembered for future occurrences of the same PII/app combination. Users can view a history of leaks at any time and can also see where each app sends data as a graph of connections updated in real-time (Fig. 2(b)). The edges of the graph can be filtered or annotated by the leaked PII, and more information about the remote servers receiving the leak can be displayed.

3.3 Leak Detection Methodology

At the heart of AntShield lies the online inspection of network packets to detect if they contain PII.

First, we use a *hybrid* String Matching-classification methodology. As described in Sec. 3.1, a key insight is that PII can be split into two categories: *predefined* and *unknown*, depending on whether they are known a priori or not. This is an inherent advantage of operating on the device: AntShield has access to all the *predefined* strings and can use DPI to search for them; we refer to this method as String Matching. This not only gives us 100% accuracy on finding *predefined* leaks, if they are not obfuscated, but also reduces the set of PII that classifiers must learn, thereby improving the accuracy of finding *unknown* leaks and reducing variance (see Sec. 4.3).

Second, we treat PII detection as a Multi-Label problem, since a packet may contain zero, one, or multiple PII. Our classifiers decide, in one step, if any PII are contained in a packet, and if so - what type. More specifically, we use Mulan [15] to perform multi-label classification using the Binary Relevance (BR) transformation method [16]. The idea is to train a separate binary classifier for each label. Since the C4.5 DTs worked well for classifying leak vs non-leak, we use them as the independent classifiers in BR.

Third, we build classifiers *per-app*, instead of *per destination domain*. This is possible thanks to AntShield running on the device: it can accurately map a packet to the app that generated it. From a classification point of view, per-app classifiers perform similarly to per-domain classifiers, as shown in Sec. 4.3 and explained in Sec. 4.2. However, per-app classifiers have important system advantages. First, they allow for easy setup and scalability: only the few classifiers for the installed apps on the particular device must be loaded into memory. This is much smaller than hundreds of domains contacted by those apps and the third-party libraries contained within them. Second, they apply to all TCP and UDP traffic, not just to HTTP(S) traffic. Third, per-app classifiers obviate the need for DNS lookups, which are costly and inaccurate, but are necessary when using per-domain classifiers. Recon parsed HTTP(S) packets to extract the host name (which is also costly in terms of CPU) and decided which per-domain classifier to apply. One possible solution is to do reverse-DNS lookup to map (all TCP and UDP, not only HTTP(S)) packets to their intended hosts. However, many companies opt-in to use third-party web service providers (such as Amazon AWS), and for them, reverse-DNS returns host names that are not very useful (*e.g.* ec2-54-164-159-29.compute-1.amazonaws.com). As a work-around, it may be possible to implement a reverse-DNS cache on the device by keeping track of all the DNS requests. Unfortunately, we have seen many cases where the same IP maps to multiple host names (again, due to third-party web service usage). Finally, even if we could somehow achieve perfect mapping of IPs to host names, there is still the problem of domain name extraction. One solution is to maintain a public suffix list, but that would take up too much memory on the mobile device. Another solution is to keep removing prefixes from the host name and do DNS queries until a Start of Authority record is reached; but this would cause too much network delay on each packet before it can even be assigned a classifier.

3.4 Real-time Implementation on the Mobile Device

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

Detecting PII in an Outgoing Packet. Our hybrid approach relies on String Matching to search for the *predefined* leaks and on classification methodology to detect *unknown* ones. The former benefits from the good performance of AntMonitor Library's efficient DPI module. The latter needs to parse packets to extract words that are used as features of the classifiers. With off-the-shelf Recon, to extract words from a packet, several invocations of Java string parsing methods would be required, which are extremely slow on a mobile device. We were able to extract features from the traffic while completely avoiding parsing by exploiting the following observation: most decision trees are one-level deep and only a third of the trees have a depth greater than two. Therefore, we only need to extract the words that appear in the decision tree nodes and we can use DPI to search for them. Since the Aho-Corasick algorithm used in the AntMonitor Library can search for many strings in one pass of the packet, having these extra words to search for does not affect performance.

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

Minimizing Classifiers to Load in RAM. With limited RAM, care must be taken when loading machine learning models from disk to memory. To minimize the impact on RAM, we: (i) load per-app models only for those apps that are installed on the device; and (ii) perform a two-step training method to reduce the general classifier feature set (see Fig. 1). Specifically, the general classifier has a feature set size of over 12k, and during prediction needs the allocation of a double array with size 12k+. While this is a small size for a server, on the mobile device it causes major issues – if one loads the full general classifier, most web pages and applications do not load. This is because each time a packet has to be predicted by the general classifier (when there is no corresponding per-app classifier), the memory allocation becomes so large that a *blocking garbage collection call* has to be executed by the Android OS after every prediction. This blocks the main networking thread, causes connections to time-out, and prevents pages from loading. We were able to reduce the feature set by exploiting the existing classifier tree: we re-trained the general classifier using only the words that appear in the tree nodes as features. This resulted in a feature set size of only 509, *i.e.* a 24x reduction for the general classifier, which in itself allowed AntShield to run in real time. Further improvements were achieved by reducing the feature set of per-app classifiers. Overall, AntShield's memory usage was around 100 MB, which is acceptable: many popular apps, *e.g. Facebook*, use as much as 200 MB RAM.

Real-Time Packet-to-App Mapping. In order to call the per-app classifiers, we first need to map a packet to the application that generated it. The AntMonitor Library provides packet-to-app mapping but only off-line (*e.g.* after a packet has been read off a queue on a different thread that does not block the main networking thread) [3], which is not fast enough to run on-line. Specifically, when using AntMonitor Library's original mapping implementation, we were only able to reach a throughput of 1 Mbps when testing with *Speedtest*. Upon further code and CPU usage analysis, we found that the inefficiency stemmed from two issues: (i) the AntMonitor Library was doing some Java string parsing to extract the app UID, source IP/port, and destination IP/port that were separated by a comma when returned from the native C module; (ii) the AntMonitor Library was storing the mappings in a HashMap keyed by a String (made of concatenating source/ destination IP/port numbers), which caused many String comparisons whenever an item needed to be fetched from the HashMap. To avoid these costly operations, we changed AntMonitor Library's native C module to return the app UID and the source port number only, as separate elements in an array. (it is best to avoid using complex data structures in native C). This way the Java part of the code could separate out the UID (and fetch the corresponding app name) and the source port of each open connection without doing any parsing. The source port number is then

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

Table 1: Summary of Datasets. Recon is the previous state-of-the-art, collected in the middle of the network [2]. AntShield's Manual and Automated datasets were collected on the device.

used as the key to the HashMap that fetches the corresponding app names. These improvements allowed us to do real-time packet-to-app mapping while achieving network speeds close to regular device operation speeds.

Real-time. The evaluation in Sec. 4.4 shows that our optimizations make the crucial difference for being able to detect PII in real-time on the device: 1ms for extracting words (as opposed to 30ms+ if parsing out all words) and 1ms for classification.

4. EVALUATION

4.1 PrivacyShield Datasets

In order to evaluate the effectiveness of our methodology in detecting private information exposure, we collected and analyzed two AntShield Datasets. We logged all packets generated by different apps on a test device (Nexus 6) and converted each packet into a JSON object that reported any PII exposures (see 3.1 for a list) and broke the packet into any relevant fields (destination IP address/port, HTTP method, if applicable, and etc). We collected two different datasets, depending on how we interacted with apps, described next.

Manual Testing. First, in order to assess PII leaks during typical user behavior, we tested 100 most popular and free Android apps, based on rankings in *AppAnnie* [17]. We tested in batches: we installed 5 apps on the test device and then used AntShield to intercept and log packets while interacting with each app for 5min. After all apps in the batch were tested, we switched off the screen and waited 5min to catch any packets sent in the background. Next, we uninstalled each app and finally, turned off AntShield.

Automatic Testing. We also used the *UI/Application Exerciser Monkey* [18] to automatically interact with apps. This does not capture typical user behavior but enables extensive and stress testing of more apps. We installed 4 batches of 100 applications each, and had *Monkey* perform 1,000 random actions in each tested app while AntShield logged the generated traffic. At the end of each batch, we switched off the screen of the test device and waited for 10min to catch additional exposures sent in the background.

Summary. Since the two (Automatic and Manual) AntShield Datasets capture different behaviors, we describe and analyze them separately. However, for the purposes of training and testing classifiers, we merged them into one, referred to as the AntShield Dataset. The AntShield datasets are summarized in Table 1, next to the prior state-of-the-art PII datasets collected by Recon [2].

Using AntShield to capture packets on the device has several advantages compared to previous datasets collected in the middle of the network: (1) we were able to accurately map each packet to the app that generated it; (2) we kept track of foreground vs. background apps, to see what kind of data apps send while in the background; (3) we gained insight into TLS, UDP, and regular TCP traffic, in addition to HTTP; (4) scrubbing PII and labeling packets with the type of PII they leak was fully automated: AntShield already provides *predefined* strings, and we entered the *unknown* strings (*e.g.* fake test account credentials) as custom filters (as in Fig. 2(a)). The resulting dataset contains more and richer information about exposures than before. Some advantages are inherent to running on the device (*i.e.* the ability to capture contextual information, including the app names). Other differences are due to changes in app versions and leak behavior over time. Therefore, in addition to being used to evaluate our methodology (Section 4.3), our datasets have value on their own and we will make them available to the community.

4.2 Exposures Found in the Datasets

Our datasets provide us with insights into the current state of privacy leaks in the Android ecosystem. Some of the captured patterns were previously unknown, and are revealed for the first time here. For example, we were able to detect leaks happening in the background, leaks in plain TCP and UDP (not belonging to HTTP(S) flows), two orders of magnitude more *unknown* leaks than before (which is crucial for training classifiers), several hundreds of packets with not one but multiple leaks (which motivated our Multi-Label approach), and malicious scanning for rooted devices.

Background Leaks. AntShield is in a unique position to capture leaks that happen in the background vs. foreground, and other contextual information that is only available on the device. Table 1 shows that there is a substantial number of background leaks (*e.g.* half of all leaks in the automatic dataset) that should be brought to users' attention and be incorporated into learning algorithms. We observed an order of magnitude more background leaks in the top apps in the automatic vs the manual datasets. One possible explanation is that the random clicks in the automated test lead to click-

App Name	Leak Type	# Leaks
com.roblox.client 2.280.107211	Username, Location	1234
com.ss.android.article.master3.2.7	City, Adid, Location, AndroidId, IMEI	766
com.cleanmaster.security3.2.6	Adid, AndroidId	511
com.cyberpony.stickman.warriors.epic1.3	Adid, City, Location, Zipcode	434
com.paypal.android.p2pmobile6.9.0	City, FirstName, LastName, SerialNumber, Zipcode, Adid, AndroidId, Password, Email	257
com.weather.Weather7.7.1	Adid, Location	172
com.pof.android3.45.2.1417399	Adid, Username, AndroidId	136
com.bitmango.go.wordcookies1.1.9	Adid, AndroidId	135
com.kiloo.subwaysurf1.68.0	Adid, IMEI, AndroidId	101
com.qisiemoji.inputmethod5.5.8.1570	Adid, IMEI, AndroidId	99
com.bitmango.go.blockhexapuzzle1.3.7	Adid, AndroidId	97
com.bitmango.rolltheballunrollme1.6.5	Adid, AndroidId	95
com.jb.zcamera2.48	Adid, AndroidId, IMEI, Email, IMSI	87
. . .	. . .	. . .
All	All	3819

Domain Name	Leak Type	# Leaks
isnssdk.com	Adid, IMEI. AndroidId	739
roblox.com	Location	679
facebook.com	Adid	651
rbxcdn.com	Location	561
mopub.com	Adid	340
bitmango.com	Adid	262
paypal.com	AndroidId	257
appsflyer.com	Adid, AndroidId	239
goforandroid.com	Adid, IMEI, IMSI, AndroidId	171
applovin.com	Adid	157
pof.com	Adid, AndroidId	121
adjust.com	Adid	96
adkmob.com	Adid, AndroidId	88
pandora.com	Adid, AndroidId, Zipcode	78
wish.com	Adid	78
lyft.com	Location	68
. . .	. . .	. . .
All	All	3819

Table 2: Manual dataset: Summary of applications and domain names with most leaks and their leak types

App Name	Leak Type	# Leaks
cmbinc12.mb32b5.98	City, Adid, Location, AndroidId, Zipcode	1326
com.kitkatandroid.keyboard3.9.9	Adid, , Location, AndroidId, SerialNumber	1046
com.episodeinteractive.android.catalog5.61.1+g	Adid, Gender, SerialNumber, AndroidId	438
com.myyearbook.m11.8.0.681	Adid, City, Location, Zipcode	263
System0.1.5	Username, City, Zipcode, Adid, AndroidId, Location, IMEI, IMSI	255
com.clearchannel.iheartradio.controller7.2.2	Adid, Zipcode	220
com.cmcm.live3.4.9	Adid, AndroidId, Location, IMEI, SerialNumber, IMSI	213
com.apalon.myclockfree2.29	Adid, City	206
com.freecraft.pocket.edition2.0	Adid, City, Location, Zipcode	174
com.madebyappolis.spinrilla2.2.4	Adid, City, Location, AndroidId, Zipcode	146
. . .	. . .	. . .
All	All	4760

Domain Name	Leak Type	# Leaks
mopub.com	Adid	2878
pocketgems.com	AndroidId	416
applovin.com	Adid	409
appsflyer.com	Adid	312
ksmobile.net	SerialNumber, Location, AndroidId	216
ihrhls.com	Adid	215
goforandroid.com	AndroidId	210
tapjoyads.com	IMEI, AndroidId	169
amplitude.com	Adid	152
lkqd.net	Adid, AndroidId	150
tapjoy.com	Adid, AndroidId	148
pinterest.com	AndroidId	112
bitmango.com	Adid	109
3g.cn	AndroidId	107
instagram.com	Username	105
crashlytics.com	Adid, AndroidId	98
. . .	. . .	. . .
All	All	4760

Table 3: Auto dataset: Summary of applications and domain names with most leaks and their leak types

App Name	Leak Types	Port
System	IMEI, IMSI, AndroidId	8080
com.jb.gosms	AndroidId	10086
com.jiubang.go.music	AndroidId	10086
air.com.hypah.io.slither	Username	10086
com.jb.emoji.gokeyboard	AndroidId	10086
com.gau.go.launcherex	AndroidId	10086
com.steam.photoeditor	AndroidId	10086
com.jb.zcamera	AndroidId	10086
com.flashlight.brightestflashlightpro	AndroidId	10086

Domain Name	Leak Types	Port
206.191.155.105	Username	454
206.191.154.41	Username	454
23.236.120.208	AndroidId	10086
3g.cn	IMEI, IMSI, AndroidId	8080
23.236.120.220	AndroidId	10086

Table 4: TCP packets (non HTTP/S) sending PII over ports other than 80, 443, 53

ing on ads, which generate traffic even after the app moves to the background.

Auto vs. Manual. Tables 2 and 3 show the top apps/domains that collect the most PII in our manual and auto datasets, respectively. We find that the top apps and domains differ for each dataset, indicating that it is important to test apps manually so as to fairly represent what happens to real users. For instance, we see that the auto dataset is more ad-oriented, while the top domains for the manual dataset include non-ad networks such as `facebook`, `paypal`, and `pandora`. This is most likely due to the fact that our *Monkey* tests ended up clicking on ads during the random events, whereas real users tend to avoid ads.

Non-HTTP Leaks. Prior state-of-the-art datasets [2] reported only HTTP(S) leaks. Table 1 reports, for the first time, leaks in non-HTTP(S), including plain TCP or UDP packets. Our dataset contains 29 UDP leaks, all of which were exposing Advertiser Id and Location. As shown in Table 4, we also found some apps (mostly games and photo-editing apps) that leaked the device ID over non-standard (80, 443, 53) TCP ports, such as 8080 or 10086 (a port known to be used by trojans, Syphillis and other threats [19]). The destination IPs could not be resolved by DNS, indicating that the application may have hard-coded those IPs. We were also able to detect ver 3000 TCP packets with PII expo-

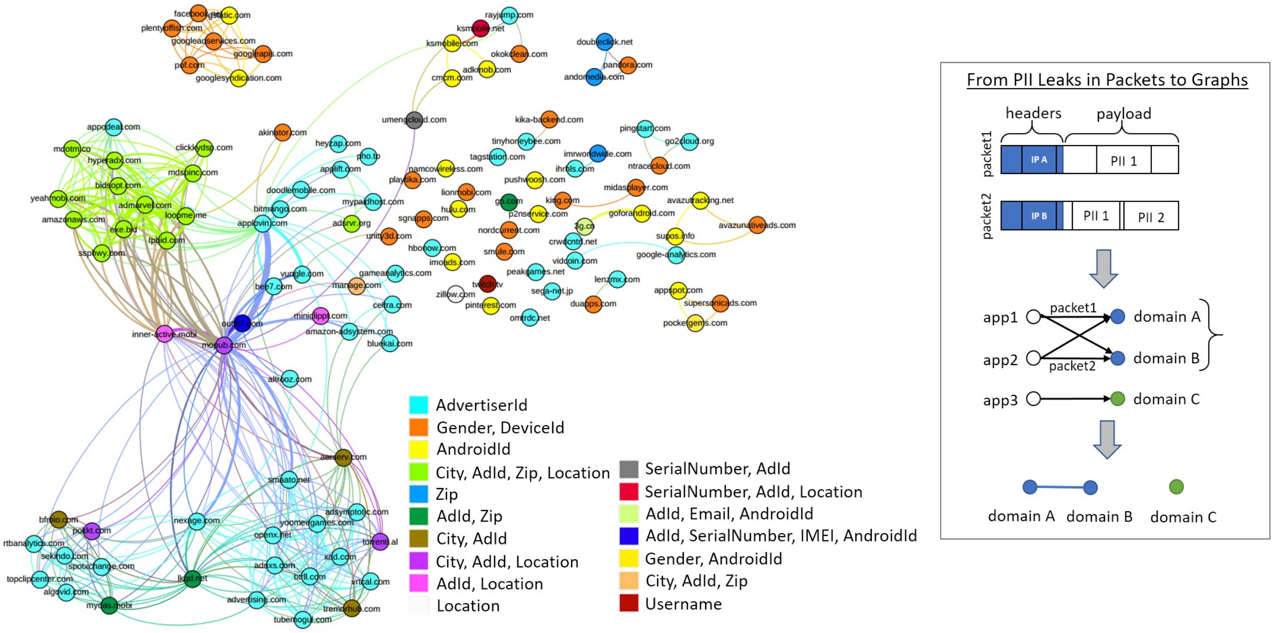


Figure 3: **Understanding the behavior of leaking through graph analysis of the** `AntShield` **dataset.** The graph consists of nodes corresponding to destination domains and edges representing the similarity of two domains. Two domains are similar if there are common apps that send packets with PII exposures to both domains; the more common apps leak to these domains, the more similar they are, the larger the width of an edge between them. The color of a domain node indicates the types of PII it receives. One can observe from the graph structure that domains form communities that capture interesting patterns: (1) The large communities on the left and bottom consist mostly of ad networks; ad exchanges are nodes in between ad communities; (2) Facebook/Google domains are a different community on their own, on the top left; (3) small apps contact only their own domain, leading to isolate domain nodes; (4) domains in the same community receive the same type of leaks (as indicated by the color of nodes).

sure, most of which are TCP segments, belonging to a larger HTTP packet. It is important to be able to classify these packets as well, since we will be receiving them through the VPN during real time inspection in `AntShield`.

HTTPS Leaks. Since traffic is increasingly over HTTPS than HTTP, we need to inspect and train on HTTPS traffic as well. However, due to their sensitive nature, previous HTTPS datasets [2] were not made publicly available and we had to collect our own. Table 5 summarizes the leaks we discovered in HTTPS traffic. The top app `com.ss.android.-article.master` is a news app, thus it makes sense for it to query the user's city, perhaps to fetch localized news. However, it is unclear why the app needs the user's IMEI (when it already has the AdId) and the specific longitude and latitude coordinates of the user. Another example is `com.cmcm.live` - it leaks 5 different device identifiers for no apparent reason. Hence, although well-behaving apps should use HTTPS, they should also be inspected for potential privacy leaks as not all information that they gather is necessary for their functionality. We also found that the majority of top domains receiving PII over HTTPS were ad-related.

Checking for Rooted Devices. We noticed a suspicious flag called "jailbroken" or "device.jailbroken" leaked by several apps (*e.g.* com.bitstrips.imoji, com.yelp.android, `com.zeptolab.ctr.ads`, etc). This flag was found in the URI content or in the body of a POST method in the packets, and it was set to 1 if the device was rooted, or to 0 otherwise. In Table 6, we show the applications that contain this field in our dataset and the domain to which the "jailbroken" flag is being sent. We also show other types of leaks that the particular domain collects. From the table, we see that the flag is usually accompanied with a device identifier. Several apps send this flag to the same domain (`upsight-api.com`, an ad network), which indicates that an ad library is probably leaking this information, rather than the app itself.

Behavioral Analysis of PII Leaks. An interesting direction for analyzing the `AntShield` dataset is via behavioral analysis. For instance, we can ask: (i) what can the communication between mobile apps and destination domains reveal about tracking and advertising? (ii) what type of information leaks to what domains and how to define similarity of apps or domains with respect to leaks? Fig. 3 showcases one graph that visualizes similar destination domains with respect to leaks they received, as captured in the `AntShield` dataset. We define two domains to be similar if they

App Name	Leak Type	# Leaks
com.ss.android.article.master3.2.7	City, Adid, Location, AndroidId, IMEI	752
com.cleanmaster.security3.2.6	Adid, AndroidId	174
com.paypal.android.p2pmobile6.9.0	City, FirstName, LastName, Zipcode, Adid, SerialNumber, AndroidId, Password, Email	131
com.offerup2.3.12	Adid, Username, FirstName, Location, Zipcode, AndroidId	114
com.cmcm.live3.4.9	Adid, AndroidId, Location, IMEI, SerialNumber, IMSI	114
me.lyft.android4.20.3.1439781	City, FirstName, LastName, SerialNumber, Zipcode, PhoneNumber, Location, AndroidId	112
com.pinterest5.6.2	Adid, AndroidId	111
com.weather.Weather7.7.1	Adid, Location	110
com.qisiemoji.inputmethod5.5.8.15709	Adid, IMEI, AndroidId	83
. . .	. . .	. . .
All	All	3039

Domain Name	Leak Type	# Leaks
mopub.com	Adid	2380
isnssdk	AndroidId, IMEI	805
roblox.com	Location	679
applovin.com	Adid	566
rbxcdn.com	Location	561
appsflyer.com	Adid	549
facebook.com	Adid	391
bitmango.com	Adid	371
goforandroid.com	AndroidId	262
ihrhls.com	Adid	219
pocketgems.com	AndroidId	211
ksmobile.net	SerialNumber, Location, AndroidId	159
tapjoy.com	Adid, AndroidId	151
tapjoyads.com	IMEI, AndroidId	147
wish.com	Adid	139
paypal.com	AndroidId	131
pof.com	pof.com	122
. . .	. . .	. . .
All	All	3039

Table 5: Summary of applications and domain names with HTTPS leaks in our dataset (manual and auto)

App Name	Domain	Leak Types
com.bitstrips.imoji 10.2.32, 10.3.76	pushwoosh.com	AndroidId
com.nianticlabs-.pokemongo 0.57.4	upsight-api.com	Location, AndroidId
com.psafe.msuite 3.11.6 , 3.11.8	upsight-api.com	AndroidId
com.yelp.android 9.5.1	bugsnag.com	AndroidId
com.zeptolab.ctr.ads 2.8.0	onesignal.com	AndroidId
com.namcobandai-games.pacmantournaments 6.3.0	namcowireless.com	AndroidId
com.huuuge.casino.slots 2.3.185	upsight-api.com	AndroidId
com.cmplay.dancingline 1.1.1	pushwoosh.com	AndroidId

Table 6: Applications with "jailbroken" field

are contacted by the same set of applications (see the box on the right inside Fig. 3). For example, domains A and B are similar because they are contacted by two apps (app1, app2). We depict the similarity of domains A and B as an edge on the graph of domains, at the bottom of the box. This data can be readily extracted from our trace, together with the type of information that was transmitted from apps to domains.

The graph depicted on the left side of Fig. 3 shows a projection of the underlying bipartite graph (middle step in the box) on domains (last step in the box); the graph is plotted and analyzed using Gephi [20]. Nodes in this graph represent domains; the edges indicate similar nodes as per above definition; the width of the edge indicates the number of common applications; and the domain color corresponds to the type of leaked PII. The clusters of domains in the graph are the output of a community detection algorithm, which is a heuristic that tries to optimize modularity.[1]

The graph in Figure 3 reveals interesting patterns about PII leakage in the AntShield dataset. First, advertising is the result of coordinated behavior. For example, it is easy to identify ad exchanges: mopub.com is in the center of all communication; and inner-active.mobi and nexage.com are also clearly shown as hubs. All three large communities on the bottom and left of the graph correspond to ad networks. Second, on the top left, there is a community of domains that belong mostly to Google and Facebook, and two domains (pof.com and plentyoffish.com) of a dating service. The latter could be because the dating app also sends statistics (*e.g.* for advertising purposes) to Google and Facebook, in addition to its own servers, as suggested by the type of PII being sent (gender and device ID, represented by the yellow color). Third, not all domains belong to a community: some are well-behaved and are contacted only by their own app. For instance the white-colored domain zillow.com towards the bottom center of the graph is an isolate node and only receives information about the user's location, which makes sense since it provides a real-estate service. Another example is the blue-colored domain hbonow.com: it is only contacted by its own app and only receives the advertising ID to serve ads. Another observation from Figure 3 is that most domains in the same community receive the same type of PII (as indicated by the domain color). This can be explained by the common ad libraries shared among different apps that fetch the same PII.

In general, similarity of apps and domains based on their network activity can be exploited to infer abusive behavior (*e.g.* advertising, tracking, or malware) in mobile traffic, and this is one promising direction for future work.

4.3 Classification Evaluation

Classification Schemes under Comparison. In this section, we use our datasets to compare the classification accuracy of the proposed AntShield approach (see Section 3.3) to the previous state-of-the-art Recon approach (Section 2).

[1] The main idea is that for specific node i, it tries to assign different communities of its neighbors like node j's community as i's community and compute the gain of modularity for whole network. The community which maximize the modularity will be the proper one. If the gain of modularity be negative or zero, i keeps its community. This process is an iterative process which is done for all nodes. This algorithm is implemented in Gephi software [20], and works with weighted graphs also.

Since our proposed method combines several ideas, we also report results from the evaluation of individual ideas, to help assess which idea brings the most benefit:

1. Complete Recon approach as per Section 2: classify all (*predefined* and *unknown*) exposures, using binary classifiers first to detect a leak, then heuristics to determine the type of leak.

2. Recon classifying *unknown* exposures only.

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

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

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

6. Complete AntShield as per Section 3.3: String Matching for *predefined* and Multi-Label classification for *unknown* leaks; Multi-Label trained on *unknown* only, testing done on all exposures.

Per-app vs. Per-domain classifiers. In Section 3.3, we discussed the system advantages of using per-app instead of per-domain classifiers. In this section, we show that their classification performance is similar (which is also justified by the insights at the end of Section 4.2). For each method, we compare how well the per-domain, per-app, and general classifiers perform. We train specialized classifiers for those domains and apps that contain at least one positive sample (packet with an exposure), and one negative sample (packet with no exposure). In that sense, we find that per-app classifiers are able to cover more data than the per-domain classifiers. In particular, we obtain the following numbers for packets covered by a classifier:

- All PII, per-app classifiers: 211 (93.3% of traffic, 99.5% of packets with PII)

- All PII, per-domain classifiers: 182 (63.6% of traffic, 95.0% of packets with PII)

- *unknown* PII, per-app classifiers: 47 (54.4% of traffic, 99.5% of packets with *unknown* PII)

- *unknown* PII, per-domain classifiers: 49 (24.5% of traffic, 87.4% of packets with *unknown* PII)

This is expected since apps generally exhibit more diverse behavior by connecting to various domains, some of which collect PII and some of which do not. Thus, we are more likely to find apps that have sent at least one packet containing PII and one packet without PII, as opposed to domains that receive packets with and without PII.

Evaluation Approaches and Metrics. After classifying a packet, either a leak is detected with a particular PII type, or No Leak is detected. Depending on how one summarizes these numbers over all packets classified, we may have different assessments. In particular, whether or not we consider packets that do not contain PII, affects the numbers, since this is the majority of the packets. We considered three evaluation schemes, summarized in Fig. 4:

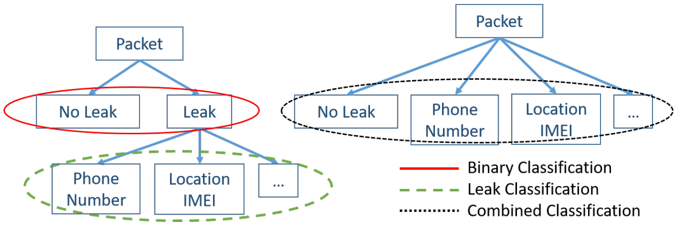


Figure 4: Evaluation approaches: (1) Binary Classification: we assess how well we identify whether or not a packet contains a PII (Sec. 4.3.1); (2) Leak Classification: we assess how well we infer the PII type from packets that already contain a PII, ignoring packets without PII (Sec. 4.3.2); (3) Combined Classification - assess how well we identify the PII type *and* the No Leak label, considering all packets (Sec. 4.3.3).

1. *Binary Classification*: this approach evaluates how well the applicable algorithms classify a packet as containing an exposure or not (Sec. 4.3.1).

2. *Leak Classification*: this approach evaluates how well each algorithm distinguishes PII types in packets that contain an exposure (Sec. 4.3.2), *i.e.* packets without a PII are not taken into account.

3. *Combined Classification*: this approach evaluates how well each algorithms distinguishes among PII types and "no leak" (Sec. 4.3.3), *i.e.* packets without a PII are taken into account.

For each approach, we perform 5-fold cross-validation on the given model (unless otherwise specified), and calculate the average and the standard deviation across the trained specialized classifiers. (Since Recon's second (non-binary) step and String Matching are both heuristic, we did not perform cross-validation on these methods when evaluating leak and combined classification, but simply ran the algorithms on the entire applicable dataset (columns 1-3 and 6) in the Tables.)

Because a packet can leak more than one PII type, for the latter two approaches, we use evaluation metrics specific to multi-labeling problems [21]. We report : (i) *accuracy*: the number of correct labels, divided by the number of predicted and true labels, (ii) *precision*: the number of correct labels, divided by the number of predicted labels, (iii) *recall*: the number of correct labels, divided by the number of true labels.

4.3.1 Binary Classification

We report the binary classification results in Table 7 for the two machine learning algorithms under consideration: Recon's DT, and our Multi-Label BR. We report the standard metrics for binary classification: F-measure, specificity,

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

Table 7: Binary Classification Results (Sec. 4.3.1)

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

Table 8: Leak Classification Results (Sec. 4.3.2).

		Method					
		(1) Recon on All PII	(2) Recon on *unknown*	(3) String Matching & Recon on *unknown*	(4) Multi-Label on All PII	(5) Multi-Label on *unknown*	(6) Complete AntShield
Per-Domain Avg	accuracy	**89.1% ± 22.1**	91.8% ± 17.7	98.5% ± 7.81	99.2% ± 2.02	99.3% ± 2.54	**99.5% ± 3.99**
	precision	**90.0% ± 21.5**	91.8% ± 17.7	98.7% ± 7.55	99.2% ± 2.10	99.3% ± 2.82	**99.5% ± 3.99**
	recall	**89.2% ± 22.0**	91.8% ± 17.7	98.5% ± 7.80	99.2% ± 1.83	99.5% ± 1.87	**99.8% ± 1.60**
Per-App Avg	accuracy	**91.3% ± 15.2**	95.0% ± 12.6	99.5% ± 3.09	98.7% ± 2.31	98.9% ± 2.83	**99.1% ± 7.35**
	precision	**92.7% ± 14.1**	95.0% ± 12.6	99.5% ± 2.96	98.7% ± 2.24	99.0% ± 2.89	**99.1% ± 7.35**
	recall	**91.4% ± 15.2**	95.0% ± 12.6	99.5% ± 3.06	98.7% ± 2.26	99.1% ± 2.11	**99.4% ± 6.91**
General	accuracy	**89.8%**	99.1%	99.3%	78.5%	76.5%	**99.8%**
	precision	**91.3%**	99.1%	99.4%	80.6%	79.1%	**99.8%**
	recall	**90.4%**	99.1%	99.4%	77.0%	74.4%	**99.9%**

Table 9: Combined Classification Results (Sec. 4.3.3).

and recall. The first column is consistent with Recon's own reports in [2] - the model achieves high accuracy and low false positives/negatives. The second column shows that there is little benefit in focusing on *unknown* exposures only. This makes sense, since in this binary step, we only want to see whether or not a packet contains a leak, and not to extract what type of leak it is (see Fig. 4). The third and fourth columns also show little benefit from our Multi-Label approach since within the BR, we still use a similar decision tree to classify exposure vs non-exposure. We also note that the standard deviation is higher when focusing on *unknown* exposures only (columns 2 and 4). This is expected since there is now less data to work with and some domains send *unknown* PII only once in a while. Furthermore, in the case of binary classification, the general classifiers perform close to the specialized ones. However, we are interested in improving the accuracy on the type of PII classification, and as we show in the next two subsections, our approaches and the specialized classifiers bring benefit there.

4.3.2 Leak Classification

The main results are summarized in Table 8. First, standard deviation is high because certain domains are easy to learn and get near 100%, while a small set of domains are difficult (some even have 0% accuracy). Recon's heuristic scores low when attempting to extract the PII type (column 1); see Sec. 2 and [2] for a description of the heuristic. Second, when we reduce the set of PII types to look for (column 2), the heuristic performs slightly worse, probably due to not having enough samples of *unknown* exposures. Third, as expected, String Matching can find *predefined* exposures with 100% accuracy, thus the overall accuracy improves by ~20% (column 3 vs column 1), and standard deviation decreases. Fourth, the Multi-Label approach shows significant improvement when compared to Recon's heuristic (column 4 vs column 1, and column 5 vs column 2); this is expected, since we do not need to estimate probabilities or calculate out thresholds. Fifth, the complete AntShield achieves near perfect performance, and decreases the standard deviation (column 6 vs columns 1-3). Finally, in all cases: (i) the specialized classifiers outperform the general ones, and (ii) the per-app classifiers achieve higher accuracy and lower standard deviation in our final method (column 6).

4.3.3 Combined Classification

The results for combined classification are shown in Table 9 and the difference between the performance of different classification methods is less pronounced than before. This is because the majority of packets do not contain a leak, the binary classifiers work well (see Sec. 4.3.1) and classify the "no exposure" packets correctly, making the results look deceivingly good. This is why we also report the Leak Classification performance (Sec. 4.3.2), as it provides deeper insight into the classifiers' performance. We note that in this case, the Multi-Label general classifiers appear to do worse than the corresponding ones in Recon, because the results reported in columns 4 and 5 are based on cross-validation, so the general classifiers do not see all the training data and do worse on some folds.

4.4 Real-Time Performance on the Device

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

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

Results. The results for the feature extraction approaches are as follows: (1) Recon's Java string parsing: 36 ms ± 17 ms; (2) Aho-Corasick search: 0.107 ms ± 0.149 ms. Clearly, AntMonitor Library's efficient Aho-Corasick implementation brings orders of magnitude of benefit.

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

Training Time. Training Multi-Label classifiers generally takes twice as long as Recon's binary classifiers. However, in both cases, a specialized classifier is trained within tens of *milliseconds* even when done on a standard Windows 10 laptop. General classifiers can take up to tens of minutes. When considering only *unknown* leaks, the number of labels is reduced, and both the binary and the Multi-Label general classifiers take *under 10 min* to train. However, since training is performed infrequently, and can be done at a remote server (the classifiers can be fetched later by user devices), we consider the training times a non-issue.

5. CONCLUSION & FUTURE DIRECTIONS

We presented AntShield - a system that performs, for the first time, on-device detection of *predefined* PII and classification of *unknown* PII, accurately (with higher accuracy and lower variance than state-of-the-art) and with low overhead (in real-time on the device). In the process, we collected and analyzed a new dataset, which reveals interesting PII leaks and patterns, some of which were previously unknown. Preliminary graph analysis revealed interesting patterns of apps and domains colluding to leak private information; behavioral analysis of PII leaks is one promising direction for future work. We will make the AntShield work available to the research community, including the AntShield plugin for the AntMonitor Library on the device, the training module, and the AntShield dataset.

This work focused on a single device and it is the first necessary step towards enabling distributed learning of PII leakage, where multiple devices running AntShield collaborate with each other and/or a central entity to share training data and/or classifiers. This is an important direction for future work. If the users do not completely trust the central entity, distributed machine learning frameworks for enabling collaborative learning, while preserving user privacy, are currently an active research area. We will consider Federated Learning [22, 23] (which enables mobile phones to collaboratively learn a shared prediction model while keeping all the training data on device, and is supported by Google); the Teacher-Student model [24] (ensemble-based machine learning on private datasets); and hybrid approaches such as Blender [25] (a hybrid differential privacy model where users have different privacy requirements).

6. REFERENCES

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

[2] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *In ACM MobiSys)*, volume 16, 2016.

[3] A. Shuba, A. Le, E. Alimpertis, M. Gjoka, and A. Markopoulou. Antmonitor: System and applications. *arXiv:1611.04268*, 2016.

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

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

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

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

[8] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-Layer Profiling of Android Applications. In *ACM MobiCom*, Aug. 2012.

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

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

[11] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *TRUST*, 2012.

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

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[14] Recon code and dataset. `https://github.com/Eyasics/recon`.

[15] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas. Mulan: A java library for multi-label learning. *JMLR*, 12, 2011.

[16] G. Tsoumakas and I. Katakis. Multi-label classification: An overview. *IJDWM*, 3(3), 2006.

[17] App annie. `https://www.appannie.com`.

[18] Ui/application exerciser monkey. `https://developer.android.com/studio/test/monkey.html`.

[19] Speed guide: Ports database. `http://www.speedguide.net/ports.php`.

[20] M. Bastian, S. Heymann, M. Jacomy, et al. Gephi: an open source software for exploring and manipulating networks. *Icwsm*, 8, 2009.

[21] S. Godbole and S. Sarawagi. Discriminative methods for multi-labeled classification. In *PAKDD*. Springer, 2004.

[22] Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data. https://research.googleblog.com/2017/04/federated-learning-collaborative.html, April.

[23] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical Secure Aggregation for Privacy Preserving Machine Learning. *Cryptology ePrint Archive: Report 2017/281*, 2017.

[24] Nicolas Papernot, Martín Abadi, Úlfar Erlingsson, Ian Goodfellow, and Kunal Talwar. Semi-supervised knowledge transfer for deep learning from private training data. *stat*, 1050:3, 2017.

[25] Brendan Avent and Aleksandra Korolova. Blender: Enabling local search with a hybrid differential privacy model. In *In the Proc. of 26th Usenix Security Symposium*, 2017.