ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic

Jingjing Ren[1], Ashwin Rao[3], Martina Lindorfer[4], Arnaud Legout[2], David Choffnes[1]
[1]*Northeastern Univ.*,[2]*Inria*, [3]*Univ. of Helsinki*, [4]*Vienna Univ. of Technology*

Abstract

It is well known that apps running on mobile devices extensively track and leak users' personally identifiable information (PII); however, these users have little visibility into PII leaked through the network traffic generated by their devices, and have poor control over how, when and where that traffic is sent and handled by third parties. In this paper, we present the design, implementation, and evaluation of *ReCon*: a cross-platform system that reveals PII leaks and gives users control over them without requiring any special privileges or custom OSes. *ReCon* leverages machine learning to reveal potential PII leaks by inspecting network traffic, and provides a visualization tool to empower users with the ability to control these leaks via blocking or substitution of PII. We evaluate *ReCon*'s effectiveness with measurements from controlled experiments using leaks from the 100 most popular iOS, Android, and Windows Phone apps, and via an IRB-approved user study with 31 participants. We show that *ReCon* is accurate, efficient, and identifies a wider range of PII than previous approaches.

1 Introduction

There has been a dramatic shift toward using mobile devices such as smartphones and tablets as the primary interface to access Internet services. Unlike their fixed-line counterparts, these devices also offer ubiquitous mobile connectivity and are equipped with a wide array of sensors (*e.g.* GPS, camera, and microphone).

The combination of rich sensors and ubiquitous connectivity make these devices perfect candidates for privacy invasion. Apps extensively track users and leak their personally identifiable information (PII) [13, 18, 22, 28, 47], and users are generally unaware and unable to stop them [16, 24]. Cases of PII leaks dramatically increased from 13.45% to 49.78% from 2010 and 2014, and the vast majority of these leaks occur over IP networks (less than 1% of apps leak data over SMS) [36].

Previous attempts to address PII leaks face challenges of a *lack of visibility* into network traffic generated by mobile devices and the *inability to control* the traffic. Passively gathered datasets from large mobile ISPs [47, 49] provide visibility but give users no control over network flows. Likewise, custom Android extensions that are often integrated in dynamic analysis tools provide control over network flows but measurement visibility is limited to the devices running these custom OSes or apps [19], often requiring warranty-voiding "jailbreaking". Static analysis tools can identify PII leaks based on the content of the code implementing an app, but suffer from imprecision and cannot defend against dynamic code loading at run time.

We argue that improving mobile privacy requires (1) trusted third-party systems that enable auditing and control over PII leaks, and (2) a way for such auditors to identify PII leaks. Our key observation is that a PII leak must (by definition) occur over the network, so interposing on network traffic is a naturally platform-independent way to detect and mitigate PII leaks. Based on this insight, we propose a simpler, more effective strategy than previous approaches: interposing on network traffic to improve visibility and control for PII leaks.

Using this approach, we focus on the problem of identifying and mitigating PII leaks at the network level. We describe the design and implementation of a system to address this problem called *ReCon*, which detects PII leaks from network flows alone, presents this information to users, and allows users fine-grained control over which information is sent to third parties. We use machine learning and crowdsourcing-based reinforcement to build classifiers that reliably detect PII in network flows, even when we do not know a priori what information is leaked and in what format. To address flows using SSL or obfuscation, we describe techniques that allow our system to detect PII leaks in encrypted flows with user opt in, and adapt to obfuscation.[1]

By operating on network traffic alone, *ReCon* can be deployed in mobile networks [4], in home networks, in the cloud, or on mobile devices. *ReCon* is currently deployed using VPN tunnels to software middleboxes running on popular cloud platforms, because this allows us to immediately deploy to arbitrary mobile device OSes and ISPs.

Our key contributions are as follows:

- A study using controlled experiments to demonstrate how PII leaks from iOS, Android, and Windows Phone devices, motivating the need for (and potential effectiveness of) systems that identify PII leaks from network flows. We find extensive leaks of device identifiers ($>$ 50% of the top 100 apps), user

[1] We support SSL decryption for controlled experiments and private *ReCon* instances, but disable them in user studies for privacy reasons.

identifiers (> 14% of top 100 Android/iOS apps), locations (14-26% of top 100 Android/iOS apps) and even passwords (3 apps) in *plaintext traffic*.

- An approach to detect and extract PII leaks from arbitrary network flows, using machine learning informed by extensive ground truth for more than 72,000 flows generated by mobile apps.
- A system that enables users to view PII leaks from network flows, provide feedback about relevant leaks, and optionally modify leaks.
- An evaluation of our system, showing it is efficient (classification can be done in less than one ms), and that it accurately identifies leaks (with 98.1% accuracy for the vast majority of flows in our dataset). We show that a simple C4.5 Decision Tree (DT) classifier is able to identify PII leaks with accuracy comparable to several ensemble methods atop DTs (AdaBoost, Bagging, and Blending) that take significantly more processing time (by a factor of 7.24).
- A comparison with three alternative techniques for detecting PII leaks using information flow analysis. We show that overall *ReCon* finds more PII leaks than all three approaches. Further, *ReCon* can leverage information flow analysis techniques to improve its coverage, as we demonstrate in §5.3.
- A characterization of our approach on traffic generated by user devices as part of an IRB-approved user study. We demonstrate that our approach successfully identifies PII leaks (with users providing 4,250 labels for PII leaks) and characterize how these users' PII is leaked "in the wild." For example, we find sensitive information such as usernames and passwords (10 apps) being leaked in plaintext flows, in addition to personal attributes such as gender and locations.

In the next section, we motivate our work using the results of controlled experiments identifying extensive information leakage in popular apps. We then describe the design (§3) and implementation (§4) of *ReCon*. We evaluate the effectiveness of our approach using controlled experiments and data from users in §5. We discuss related work in §6 and conclude in §7.

2 Motivation and Challenges

In this section, we use controlled experiments to measure PII leakage with ground-truth information. We find a surprisingly large volume of PII leaks from popular apps from four app stores, particularly in plaintext (unencrypted) flows. Based on these results, we identify several core challenges for detecting PII leaks when we do not have ground-truth information, *i.e.* for network traffic generated by arbitrary users' devices. In the next section, we describe how to automatically infer PII leaks in network flows when the contents of PII is not known in advance.

2.1 What is PII?

Personally identifiable information (PII) is a generic term referring to "information which can be used to distinguish or trace an individual's identity" [30]. These can include geographic locations, unique identifiers, phone numbers and other similar data.

Central to this work is identifying PII leaked by apps over the network. In this paper, we define PII to be either (1) **Device Identifiers** specific to a device or OS installation (ICCID, IMEI, IMSI, MAC address, Android ID, Android Advertiser ID, iOS IFA ID, Windows Phone Device ID), (2) **User Identifiers**, which identify the user (name, gender, date of birth, email address, mailing address, relationship status), (3) **Contact Information** (telephone numbers, address book information), (4) **Location** (GPS latitude/longitude, zip code), or (5) **Credentials** (username, password). This list of PII is informed by information leaks observed in this study. While this list is not exhaustive, we believe it covers most of the PII that concerns users. We will update the list of tracked PII as we learn of additional types of PII leaks.

2.2 Controlled Experiments

Our goal with controlled experiments is to obtain ground-truth information about network flows generated by apps and devices. We use this data to identify PII in network flows and to evaluate *ReCon* (§5).

To capture privacy leaks regardless of the OS or network that the device uses, we employ *Meddle* [40]. *Meddle* provides visibility into network traffic through redirection, *i.e.* sending all device traffic to a proxy server using native support for virtual private network (VPN) tunnels. Once traffic arrives at the proxy server, we use software middleboxes to intercept and modify the traffic. We additionally use SSLsplit [7] to decrypt and inspect SSL flows only during our controlled experiments where *no human subject traffic is intercepted*.

Device setup. We conducted our controlled experiments using Android devices (running Android 5.1.1), an iPhone (running iOS 8.4.1) and a Windows Phone (running Windows 8.10.14226.359). We start each set of experiments with a factory reset of the device followed by connecting the device to *Meddle*.

Manual tests. We manually test the 100 most popular free apps for Android, iOS, and Windows Phone from the *Google Play* store, the iOS *App Store*, and the *Windows Phone Store* on August 9, 2015 as reported by App Annie [2]. For each app, we install it, interact with it for up to 5 minutes, and uninstall it. We give apps permission to access to all requested resources (*e.g.* contacts or location). This allows us to characterize real user interactions with popular apps in a controlled environment. We enter unique and distinguishable user credentials when interacting with apps to easily extract the corresponding PII from network flows (if they are not obfuscated).

Automated tests. We include fully-automated experiments on the 850 of the top 1,000 Android apps from the free, third-party Android market *AppsApk.com* [3] that were successfully downloaded and installed on an Android device.[2] We perform this test because Android users can install *third-party apps* without rooting their device. Our goal is to understand how these apps differ from those in the standard *Google Play* store, as they are not subject to *Google Play*'s restrictions and vetting process. We automate experiments using *adb* to install each app, connect the device to the *Meddle* platform, start the app, perform approximately 10,000 actions using *Monkey* [8], and finally uninstall the app and reboot the device to end any lingering connections. We limit the automated tests to Android devices because iOS and Windows do not provide equivalent scripting functionality.

Analysis. We use *tcpdump* and *bro* to analyze network traffic, then search for the conspicuous PII that we loaded onto devices and used as input to text fields. We classify some of the destinations of PII leaks as *trackers* using a publicly available database of tracker domains [1], and recent research on mobile ads [17, 27, 35].

2.3 PII Leaked from Popular Apps

We use the traffic traces from our controlled experiments to identify how apps leak PII over HTTP and HTTPS. For our analysis we focus on the PII in §2.1. Some of this information may be required for normal app operation; however, sensitive information such as credentials should never travel across the network in plaintext.

Table 1 presents PII leaked by iOS, Android and Windows apps in plaintext. Device identifiers, which can be used to track user's behavior, are the PII leaked most frequently by popular apps. Table 1 shows that other PII—user identifiers, contacts, location, and *credentials such as username and password*—are also leaked in plaintext. Importantly, our manual tests identify important PII not found by automated tests (*e.g.* Monkey) such as user identifiers and credentials. Thus, previous studies based on automation significantly underestimate leakage and are insufficient to good coverage of PII leaks.

Cross-platform app behavior. We observed that the information leaked by an app varied across OSes. Of the top 100 apps for Android, 16 apps are available on all the three OSes. Of these 16 apps, 11 apps leaked PII in plaintext on at least one OS: 2 apps leaked PII on all the three OSes, 5 apps leaked PII in exactly one OS, and the remaining 4 apps leaked PII in two of the three OSes. A key take-away is that *PII analysis based only on one OS does not generalize to all OSes, suggesting the need for a cross-platform solution.*

Leaks over SSL. During our experiments, we observed that PII is also sent over encrypted channels. In many cases, this is normal app behavior (*e.g.* sending credentials when logging in to a site, or sending GPS data to a navigation app). However, when such information leaks to third parties, there is a potential PII leak. We focus on the PII leaked to tracker domains [1], and find that 6 iOS apps, 2 Android apps and 1 Windows app send PII to trackers over SSL. The vast majority of this information is device identifiers, with three cases of username leaks. While SSL traffic contains a minority of PII leaks, there is clearly still a need to address leaks from encrypted flows.

Our observations are a conservative estimate of PII leakage because we did not attempt to detect obfuscated PII leaks (*e.g.* via salted hashing), and several apps used certificate pinning (10 iOS, 15 Android, and 7 Windows apps) or did not work with VPNs enabled (4 iOS apps and 1 Android app).[3] Our results in §5.3 indicate that obfuscation is rare today, and our results above show that significant PII leaks are indeed visible in plaintext.

2.4 Summary and Challenges

While the study above trivially revealed significant PII leaks from popular mobile apps, several key challenges remain for detecting PII leaks more broadly.

Detection without knowing PII. A key challenge is how to detect PII when we do not know the contents of PII in advance. One strawman is to simply block all advertising and tracking sites. However, this is a blunt and indiscriminate approach that can disrupt business models supporting free apps. In fact, the developers of the top paid iOS app Peace (which blocks all ads) recently withdrew their app from the App Store due to this [32].

Another strawman solution is to automatically run every app in every app store to determine when PII is leaked. This allows us to formulate a regular expression to identify PII leaks from every app regardless of the user: we simply replace the PII with a wildcard.

There are several reasons why this is insufficient to identify PII leaks for arbitrary user flows. First, it is impractically expensive to run this automation for all apps in every app store, and there are few tools for doing this outside of Android. Second, it is difficult (if not impossible) to use automation to explore every possible code path that would result in PII leaks, meaning this approach would miss significant PII. Third, this approach is incredibly brittle – if a tracker changes the contents of flows leaking PII at all, the regular expression would fail.

[2] There is an overlap of 14 apps between the AppsApk and Google Play apps we tested, but AppsApk hosts significantly older versions.

[3] For more details, see Appendix A.1

OS	Store	Testing Technique	# of Apps	# Apps leaking a given PII Device Identifier	User Identifier	Contact Info.	Location	Credentials
iOS	App Store	Manual	100	47 (47.0%)	14 (14.0%)	2 (2.0%)	26 (26.0%)	8 (8.0%)
Android	Google Play	Manual	100	52 (52.0%)	15 (15.0%)	1 (1.0%)	14 (14.0%)	7 (7.0%)
Windows	WP Store	Manual	100	55 (55.0%)	3 (3.0%)	0 (0.0%)	8 (8.0%)	1 (1.0%)
Android	AppsApk	Auto.(Monkey)	850	155 (18.2%)	6 (0.7%)	8 (0.9%)	40 (4.7%)	0 (0.0%)
Android	Google Play	Auto.(Monkey)	100	52 (52%)	0 (0.0%)	0 (0.0%)	6 (6%)	0 (0.0%)

Table 1: **Summary of PII leaked in plaintext (HTTP) by Android and iOS apps.** *Popular iOS apps leak location information more often than other OSes (26 iOS apps leak location info.) while Android and Windows apps are slightly more likely to leak device identifiers. User identifiers and credentials are leaked across all platforms.*

These issues suggest an alternative approach to identifying PII in network flows: use machine learning to build a model of PII leaks that accurately identifies them for arbitrary users. This would allow us to use a small set of training flows, combined with user feedback about suspected PII leaks, to inform the identification of a PII leaks for a large number of apps.

Encryption. It is well known that flows in the mobile environment increasingly use encryption (often via SSL). Sandvine reports that in 2014 in North American mobile traffic, approximately 12.5% of upstream bytes use SSL, up from 9.78% the previous year [43]. By comparison, 11.8% of bytes came from HTTP in 2014, down from 14.66% the previous year. A key challenge is how to detect PII leaks in such encrypted flows. *ReCon* identifies PII leaks in plaintext network traffic, so it would require access to the original plaintext content to work. While getting such access is a challenge orthogonal to this work, we argue that this is feasible for a wide range of traffic if users run an SSL proxy on a trusted computer (*e.g.* the user's own computer) or use recent techniques for mediated access to encrypted traffic [39, 44].

Obfuscation of PII. The parties leaking PII may use obfuscation to hide their information leaks. In our experiments, we found little evidence of this (§ 5.3). In the future, we anticipate combining our approach with static and dynamic analysis techniques to identify how information is being obfuscated, and adjust our system to identify the obfuscated PII. For example, using information flow analysis, we can reverse-engineer how obfuscation is done (*e.g.* for salted hashing, learn the salt and hash function), then use this information when analyzing network traces to identify leaked PII. In the ensuing cat-and-mouse game, we envision automating this process of reverse engineering obfuscation.

3 ReCon Goals and Design

The previous section highlights that current OSes are not providing sufficient visibility into PII leaks, provide few options to control it, and consequently significant amounts of potentially sensitive information is exfiltrated from user devices. To address this problem, we built *ReCon*, a tool that detects PII leaks, visualizes how users' information is shared with various sites, and allows users to change the information shared with them (including modifying PII or even blocking connections entirely).

The high-level goal of this research is to explore the extent to which we can address privacy issues in mobile systems at the network level. More specifically, the subgoals of *ReCon* are as follows:

- Accurately identify PII in network flows, *without* requiring knowledge of users' PII *a priori*.
- Improve awareness of PII leaks by presenting this information to users.
- Automatically improve the classification of sensitive PII based on user feedback.
- Enable users to change these flows by modifying or removing PII.

To achieve the first three goals, we determine what PII is leaked in network flows using network trace analysis, machine learning, and user feedback. We achieve the last goal by providing users with an interface to block or modify the PII shared over the network. This paper focuses on how to address the research challenges in detecting and revealing PII leaks; as part of ongoing work outside the scope of this paper, we are investigating other UIs for modifying PII leaks, how to use crowdsourcing to help design PII-modifying rules, and how we can use *ReCon* to provide other types of privacy (*e.g.* k-anonymity).

Figure 1 presents the architecture of the *ReCon* system. In the "offline" phase, we use labeled network traces to determine which features of network flows to use for learning when PII is being leaked, then train a classifier using this data, finally producing a model for predicting whether PII is leaked. When new network flows enter *ReCon* (the "online" phase), we use the model to determine whether a flow is leaking PII and present the suspected PII leak to the user via the *ReCon* Web UI (Fig. 2). We collect labels from users (*i.e.* whether our suspected PII is correct) via the UI and integrate the results into our classifier to improve future predictions (left). In addition, *ReCon* supports a map view, where we display the location information that each do-

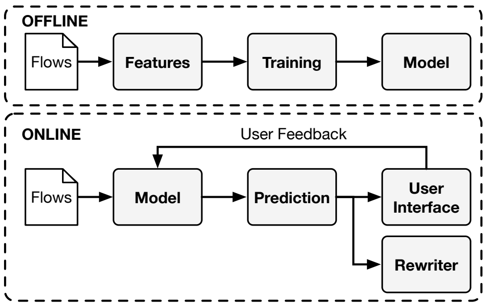
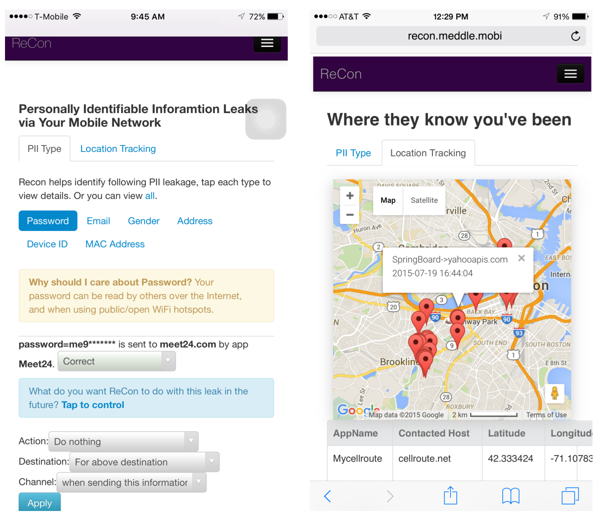


Figure 1: *ReCon* **architecture**. *We select features and train a model using labeled network flows, then use this model to predict whether new network flows are leaking PII. Based on user feedback, we retrain our classifier.*



(a) PII leaks and actions (b) Map view of location leaks

Figure 2: **Screen capture of the** *ReCon* **user interface**. *Users can view how their PII is leaked, validate the suspected PII leaks, and create custom filters to block or modify leaks.*

main is learning about the user (right). By using a Web interface, *ReCon* users can gain visibility and control into their PII leaks without installing an app. A demo of *ReCon* is available at `http://goo.gl/rAFzxc`.

To support control of PII, *ReCon* allows users to instruct the system to replace the PII with other text (or nothing) for future flows (see the drop-down boxes in Fig. 2(a)). We allow users to specify blocking or replacement of PII based on PII category (shown in the figure), domain, or app. This protects users' PII for future network activity, but does not entirely prevent PII from leaking in the first place. To address this, we support *interactive* PII labeling and filtering, using push notifications or other channels to notify the user of leaks immediately when they are detected (as done in a related study [11]).[4]

3.1 Non-Goals

ReCon is not intended as a blanket replacement for existing approaches to improve privacy in the mobile environment. For example, information flow analysis [19] may identify PII leaks not revealed by *ReCon*. In fact, *ReCon* can leverage information flow analysis techniques to improve its coverage, as we demonstrate in §5.3. Importantly, *ReCon* allows us to identify and block unobfuscated PII in network flows from arbitrary devices without requiring OS modifications or taint tracking.

3.2 Deployment Model and User Adoption

Because *ReCon* needs access only to network traffic to identify and modify PII leaks, it admits a variety of deployment models, *e.g.*, in the cloud, in home devices, inside an ISP, or on mobile devices. We are currently hosting this service on *Meddle* in a cloud-based deployment because it provides immediate cross-platform support with low overheads [40]. We are also in discussions with Telefonica to deploy *ReCon* on their Awazza [4] APN proxy, which has attracted thousands of users.

3.3 Protecting User Privacy

An important concern with a *ReCon* user study is privacy. Using an IRB-approved protocol [6], we encrypt and anonymize all captured flows before storing them (more details in Appendix A.2). The secret key is stored on a separate secure server and users can delete their data at any time. We will make the *ReCon* software publicly available. For those who want to run their own *ReCon* instance (*e.g.* if they do not want to participate in our study), the *ReCon* system requires only that a user has root on a modern Linux OS. *ReCon* can be deployed in a single-machine instance on a home computer, as Raspberry Pi plugged into a home router, a dedicated server in an enterprise, on the device itself, or VM in the cloud. One can also selectively route traffic to different *ReCon* instances, *e.g.*, to a cloud instance for HTTP traffic and a trusted home instance to decrypt HTTPS connections to identify PII leaked over SSL.[5]

4 Recon Implementation

We now discuss several key aspects of our system implementation. We evaluate our design decisions in the following section. The *ReCon* pipeline begins with parsing network flows, then passing each flow to a machine learning classifier for labeling as a PII leak or not.

4.1 Machine Learning Techniques

We use the *weka* data mining tool [23] to train classifiers that predict PII leaks. We train our classifier by extract-

[4]Push notifications require a companion app, and we currently support only Android (we plan to release iOS and Windows versions soon).

[5]See Appendix A.4 for more details about privacy-protecting deployment models.

ing relevant features and providing labels for flows that leak PII as described below.

Feature extraction. The problem of identifying whether a flow contains a PII leak is similar to the document classification problem,[6] so we use the "bag-of-words" model [26]. In this model, all flows are separated into words (using tokens) to form a set of all words in the dataset. Then for each flow, we produce a vector of binary values where each word that appears in a flow is set to 1, and each word that does not is set to 0.

A key challenge for feature extraction in network flows is that there is no standard token (*e.g.* whitespace or punctuation) to use for splitting flows into words. For example, a colon (`:`) could appear as part of a MAC address (*e.g.* `02:00:00:00:00`), a time-of-day (*e.g.* `11:59`), or can even be a delimiter in JSON data (*e.g.* `username:user007`). Further frustrating attempts to select features, one domain uses "=>" as a delimiter (in `username =>user007`). In these cases, there is no single technique that covers all flows. Instead, we use a number of different delimiters "`, ; / ( ) { } [ ]`" to handle the common case, and treat ambiguous delimiters by inspecting the surrounding content to determine the encoding type based on context (*e.g.* looking at content-encoding hints in the HTTP header or whether the content appears in a GET parameter).

Feature selection. A simple bag-of-words model produces too many features to be useful for training accurate classifiers that make predictions within milliseconds (to intercept PII leaks in real time). To reduce the feature set, we make the assumption that low-frequency words are unlikely to be associated with PII leaks, because when PII does leak, it rarely leaks just once. On the other hand, session keys and other ephemeral identifiers tend to appear in exactly one flow. Based on this intuition, we apply a simple threshold-based filter that removes a feature if its word frequency is too small. We select a reasonable threshold value empirically, by balancing accuracy and classification time for labeled data (discussed in §5.2.3).

While the above filter removes ephemeral identifiers from our feature set, we must also address the problem of words that commonly appear. Several important examples include information typically found in HTTP flows, such as "content-length:", "en-us", and "expires". We thus add stop-word-based filtering on HTTP flows, where the stop words are determined by term frequency—inverse document frequency (tf-idf). We include only features that have fairly low tf-idf values.

Per-domain and general classifiers. We find that PII leaks to the same destination domain use the same (or similar) data encodings to transfer data over the network. Based on this observation, we build per-domain models (one classifier for each destination domain) instead of one single general classifier. We identify the domain associated with each flow based on the *Host:* parameter in the HTTP header. If this header is not available, we can also identify the domain associated with each IP address by finding the corresponding DNS lookup in packet traces. This improves prediction accuracy because the classifier typically needs to learn a small set of association rules. Further, per-domain classifiers improve performance in terms of lower-latency predictions, important for detecting and intercepting PII leaks in-band.

The above approach works well if there is a sufficiently large sample of labeled data to train to the per-domain classifier. For domains that do not see sufficient traffic, we build a (cross-domain) general classifier. The general classifier tends to have few labeled PII leaks, making it susceptible to bias (*e.g.* 5% of flows in our general classifier are PII leaks). To address this, we use undersampling on negative samples, using 1/10 sampling to randomly choose a subset of available samples.

4.2 Automatically Extracting PII

A machine learning classifier indicates whether a flow contains PII, but does not indicate *which content in the flow is a PII leak.* The latter information is critical if we want to present users with information about their leaks and allow them to validate the predictions.

A key challenge for extracting PII is that the key/value pairs used for leaking PII vary across domains and devices; *e.g.* the key "device_id" or "q" might each indicate an IMEI value for different domains, but "q" *is not always associated with a PII leak.* While we found no solution that perfectly addresses this ambiguity, we developed effective heuristics for identifying "suspicious" keys that are likely associated with PII values.

We use two steps to automatically extract PII leaks from a network flows classified as a leak. The first step is based on the relative probability that a suspicious key is associated with a PII leak, calculated as follows:

$$P_{type,key} = \frac{K_{PII}}{K_{all}}$$

where *type* is the PII type (*e.g.* IMEI, e-mail address), *key* is the suspicious key for that *type* of PII, K_{PII} is the number of times the key appeared in PII leaks, and K_{all} is the number times the key appeared in all flows. The system looks for suspicious keys that have $P_{type,key}$ greater than a threshold. We set this value to an empirically determined value, 0.2, based on finding the best trade-off between false positives (FPs) and true positives (TPs) for our dataset. For users wanting more or less sensitivity, we will make this a configurable threshold in *ReCon* (*e.g.* if a user wants to increase the likelihood of increasing TPs at the potential cost of increased FPs).

[6] Using network flows as a documents, and structured data as words.

In the second step, we use a decision tree classifier structure, and make the observation that the root of each decision tree is likely a key corresponding to a PII value. We thus add these roots to the suspicious key set and assign them a large P value.

5 Evaluation

In this section, we evaluate the effectiveness of *ReCon* in terms of accuracy and performance. First, we describe our methodology, then we describe the results from controlled experiments, and we conclude by presenting the results of a user study, focusing on the impact of user feedback and characterizing observed PII leaks.

To summarize the key findings: 1) we demonstrate that a decision-tree classifier is both accurate (98.1% overall) and efficient (trains in seconds, predicts in sub-milliseconds); 2) *ReCon* not only identifies more PII than static and dynamic information-flow analysis techniques, but also can learn from the results of these approaches to improve its coverage of PII leaks; 3) our user study shows that our approach accurately identifies substantial amounts of PII leaks, and that users changed their app usage in response to using our tool. Note that this paper focuses on reliably identifying leaks and enabling control, but does not evaluate the control functionality.

5.1 Dataset and Methodology

To evaluate *ReCon* accuracy, we need app-generated traffic and a set of labels indicating which of the corresponding flows leak PII. For this analysis, we reuse the data from controlled experiments presented in §2.2; Table 2 summarizes this dataset using the number of flows generated by the apps, and fraction that leak PII. We identify that more than 6,500 flows leak PII, and a significant fraction of those flows leak PII to known trackers.

We use this labeled dataset to train classifiers and evaluate their effectiveness using the following metrics. We define a positive flow to be one that leaks PII; likewise a negative flow is one that *does not* leak PII. A false positive occurs when a flow does not leak PII but the classifier predicts a PII leak; a false negative occurs when a flow leaks PII but the classifier predicts that it does not. We measure the false positive rate (FPR) and false negative rate (FNR); we also include the following metrics:

- **Correctly classified rate (*CCR*)**: the sum of true positive (TP) and true negative (TN) samples divided by the total number of samples. $CCR = (TN + TP)/(TN + TP + FN + FP)$.
  A good classifier has a CCR value close to 1.
- **Area under the curve (*AUC*)**: where the curve refers to receiver operating characteristic (ROC). In this approach, the x-axis is the false positive rate and y-axis is the true positive rate (ranging in value from 0 to 1). If the ROC curve is $x = y$ ($AUC = 0.5$), then the classification is no better than randomly guessing. *A good classifier has a AUC value near 1.*

To evaluate the efficiency of the classifier, we investigate the runtime (in milliseconds) for predicting a PII leak and extracting the suspected PII. We want this value to be significantly lower than typical Internet latencies.

We use the *weka* data mining tool to investigate the above metrics for several candidate machine learning approaches to identify a technique that is both efficient and accurate. Specifically, we test Naive Bayes, C4.5 Decision Tree (DT) and several ensemble methods atop DTs (AdaBoost, Bagging, and Blending).

5.2 Lab Experiments

In this section, we evaluate the impact of different implementation decisions and demonstrate the overall effectiveness of our adopted approach.

5.2.1 Machine Learning Approaches

A key question we must address is which classifier to use. We believe that a DT-based classifier is a reasonable choice, because most PII leaks occur in structured data (*i.e.* key/value pairs), and a decision tree can naturally represent chained dependencies between these keys and the likelihood of leaking PII.

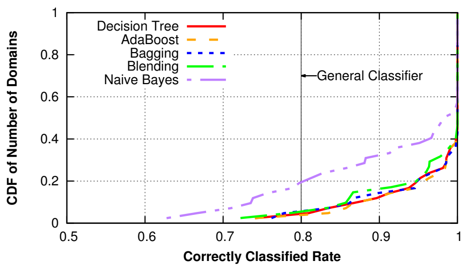
To evaluate this claim, we tested a variety of classifiers according to the accuracy metrics from the previous section, and present the results in Fig. 3. We plot the accuracy using a CDF over the domains that we use to build per-domain classifiers as described in §4.1. The top two graphs (overall accuracy via CCR and AUC), show that Naive Bayes has the worst performance, and nearly all the DT-based ensemble methods have high CCR and AUC values. (Note that the x-axis does not start at 0.)

Among the ensemble methods, Blending with DTs and k-nearest-neighbor (kNN) yields the highest accuracy; however, the resulting accuracy is not significantly better than a simple DT. Importantly, a simple DT takes significantly less time to train than ensemble methods. For ensemble methods, the training time largely depends on the number of iterations for training. When we set this value to 10, we find that ensemble methods take 7.24 times longer to train than a simple DT on the same dataset. Given the significant extra cost with minimal gain in accuracy, we currently use simple DTs.

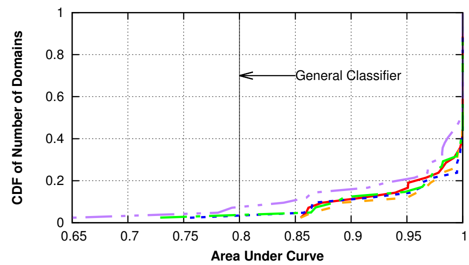
The bottom figures show that most DT-based classifiers have zero FPs (71.4%) and FNs (76.2%) for the majority of domains. Further, the overall accuracy across all per-domain classifiers is >98.1%. The domains with poor accuracy are the trackers `rlcdn.com` and `turn.com`, due to the fact their positive and negative flows are very similar. For example, the key `partner_uid` is associated both with an Android ID value and another unknown identifier.

	Manual tests			Automated tests (Monkey)	
OS (Store)	iOS (App)	Android (Play)	Windows (WP)	Android (Play)	Android (AppsApk)
Apps tested	*100*	*100*	*100*	*100*	*850*
HTTP flows	14683	14355	12487	7186	17499
Leaking PII	845	1800	969	1174	1776
Flows to trackers	1254	1854	1253	1377	5893
Leaking PII to trackers	508	567	4	414	649

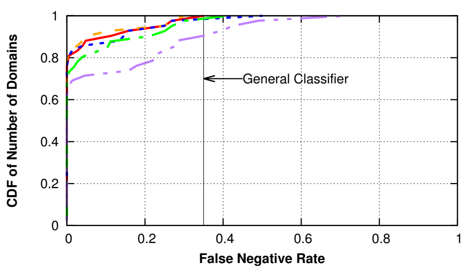
Table 2: **Summary of HTTP flows from controlled experiments.** *Manual tests generated similar numbers of flows across platforms, but Android leaked proportionately more PII. Collectively, our dataset contains more than 6500 flows with PII leaks.*



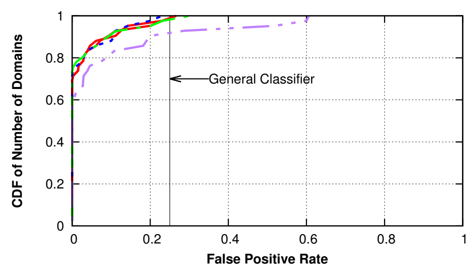
(a) CCR (x-axis does not start at 0)



(b) AUC (x-axis does not start at 0)



(c) FNR



(d) FPR

Figure 3: **CDF of per-domain per-device classifier accuracy, for alternative classification approaches.** *For the 42 per-domain classifiers, DT-based classifiers outperform Naive Bayes, and they exhibit good accuracy (high CCR and AUC, low FPR and FNR). The vertical line depicts accuracy when using one classifier across all domains, which leads to significantly worse performance.*

To provide intuition as to why DTs work well, and why PII leak detection presents a nontrivial machine-learning problem, we include several examples of DTs trained using our data. Of course, some cases of PII leaks are simple: Fig. 4(a) shows that Android Advertiser ID is always leaked to the tracker `applovin.com` when the text `idfa` is present in network traffic. Other cases are not trivial, as seen in Fig. 4(b). Here, we find that `auid` is not always associated with an IMEI value, and the DT captures the fact that the IMEI will *not* be present for a `getImage.php5` request if the `urid` is present. Finally, Fig. 4(c) gives an example of a non-trival DT for a different type of PII—e-mail address. Here, the term `email` appears in both positive and negative flows, so this feature cannot be used alone. However, our classifier learns that the leak happens in a `/user/` request when the terms `session` and `deviceId` are not present in the flow.[7] Overall, 62% of DTs are the simple case (Fig. 4(a)), but more than a third have a depth greater than two and thus indicate a significant fraction of cases where association rules are nontrivial.

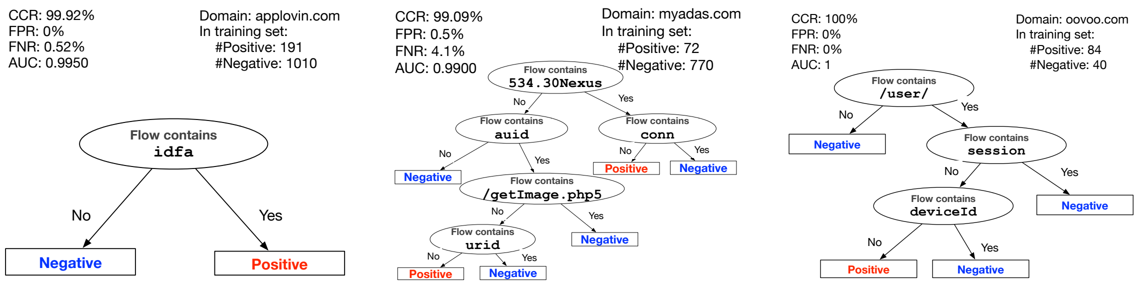
[7]Note that in this domain `deviceId` is actually used for an app-specific identifier, not a device identifier.

5.2.2 Per-Domain Classifiers

We now evaluate the impact of using per-domain classifiers instead of one classier for all flows. We build per-domain classifiers for all domains with greater than 100 samples (*i.e.* labeled flows), at least one of which leaks PII. For the remaining flows, there is insufficient training data to inform a classifier, so we create a general classifier based on the assumption that a significant fraction of the flows use a common structure for leaking PII.[8]

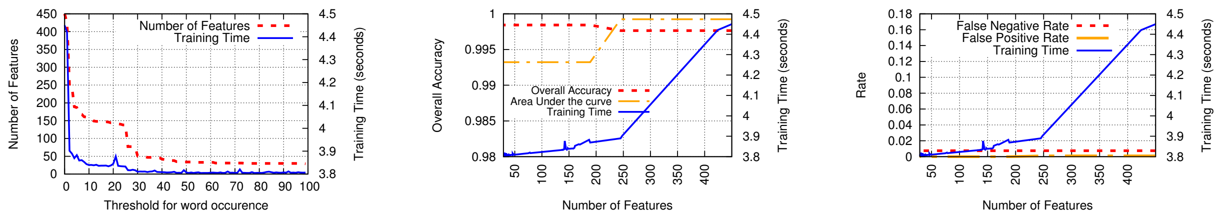
We evaluate the impact of per-domain classifiers on overall accuracy in Figure 3. The vertical lines in the subgraphs represent values for the general classifier, *which is trained using all flows from all domains*. We can see that the general classifier has lower accuracy (80.0% CCR) than >95% of the per-domain classifiers. Further, training such general classifiers is expensive in terms of runtime: it takes *minutes* to train per-domain classifiers for thousands of flows, but it takes *hours* to train general classifiers for the same flows.

[8]Note that once *ReCon* acquires sufficient labeled data (*e.g.* from users or controlled experiments) for a destination domain, we create a per-domain classifier.



(a) Simple DT for device identifier (b) Non-trivial DT for device identifier (c) Non-trivial DT for e-mail address

Figure 4: **Example decision trees (DTs) for** ***ReCon*****'s per-domain classifiers.** *The classifier beings at the root (top) node, and traverses the tree based on whether the term at each node is present. The leaves (boxes) indicate whether there is a PII leak (positive) or not (negative) for each path. The top right of each figure shows the number of positive/negative samples used to train each DT.*



(a) #features changes as threshold changes (b) accuracy and training time over #features (y-axes do not start at 0) (c) false negative and false positive rate and training time over #features

Figure 5: **Feature selection** *for the tracker domain mopub.com. Using ≈200 features leads to high accuracy and low training times; however, adding more features increases training time with no benefit to accuracy.*

5.2.3 Feature Selection

The accuracy of the classifiers described above largely depends on correctly identifying the subset of features for training. Further, the training time for classifiers increases significantly as the number of features increases, meaning that an efficient classifier requires culling of unimportant features. A key challenge in *ReCon* is determining how to select such features given the large potential set derived from the bag-of-words approach.

We use Figure 5 to illustrate this problem and how we address it. Here, we focus on statistics for the tracker domain `mopub.com` (266 flows out of 1276 leak PII); other domains exhibited similar properties.

First, we focus on the threshold to use for including features in our training set. As described in Section 4.1, we filter out features from words that appear infrequently. Fig. 5(a) shows the impact of this decision on training time, where the x-axis is the minimum number of appearances for a word to be included as a feature, and the y-axis is the time required to train a classifier on the resulting features. The figure shows that including all words (threshold = 1) significantly increases training time, but there is a minimal impact on training time if the threshold is greater than or equal to 20. The corresponding number of features decreases from 450 to 29 as the threshold for word occurrence increases from 1 to 99.

Picking the right number of features is also important for classifier accuracy, as too many features may lead to overfitting and too few features may lead to an incomplete model. We evaluate this using Fig. 5(b), where the x-axis is the number of features, the left y-axis is accuracy (the y-axis does not start at zero), and the right y-axis is training time. Even small numbers of features lead to high accuracy for this domain, but increasing the number of features significantly beyond 250 does not improve performance at all (but does increase training time). We see a similar effect on the FP rate in Fig. 5(c).

While the training time may not seem particularly high in this context, we note that this cost must be incurred for each domain and each time we want to update the classifier with user-labeled flows. With potentially thousands of flows and labels in large-scale deployments, such training times can significantly affect the scalability and responsiveness of *ReCon*.

With this in mind, we propose the following strategies for picking threshold values. First, we can use the above analysis to find the best threshold, then periodically update this threshold based on new labeled data. Second, we can pick a fixed threshold based on the average threshold across all domains (word frequency = 21). We evaluated the impact of these two approaches, and found they were nearly identical for our dataset. This suggests that a fixed value is sufficient for our dataset, but we propose periodically updating this threshold by performing the above analysis daily or weekly as a low-priority background process.

5.2.4 PII Extraction Strategies

As discussed in Section 4.2, we use two heuristics to identify key/value pairs that are likely to leak PII. We use our dataset to evaluate this approach, and find that the FP and FN rates are 2.2% and 3.5%, respectively. By comparison, a naive approach that treats each key/value pair equally yields FP and FN rates of 5.1% and 18.8%, respectively. Our approach is significantly better than this naive approach, and our FP and FN rates are sufficiently low to correctly extract PII the vast majority of the time.

5.3 Comparison with Information Flow Analysis

Our labeled dataset in the above analysis may miss PII leaks that are obfuscated or otherwise hidden from our analysis. We now evaluate our approach by comparing with one that is resilient to such issues: information flow analysis (IFA). We experiment with three IFA techniques: (1) static IFA with FlowDroid [10], (2) dynamic IFA with TaintDroid [19] (via Andrubis [36]), and (3) AppAudit [48], which uses a combination of both static and approximated dynamic analysis. Each of these tools has limitations: some are very resource intensive and some pose restrictions on the type of apps they can successfully analyze.

Static IFA. FlowDroid detects PII leaks as data flowing between sensitive sources and sinks, which are configured via a list of Android API calls. However, the analysis is quite resource intensive: for 7.83% of apps, our available memory of 8GB was insufficient for analysis; for 25.60% of apps the analysis exceeded our analysis timeout of 30 minutes. Further, the detected leaks are reported as paths between the API calls of interest and are thus difficult to interpret for non-expert users.

Dynamic IFA. Andrubis is an app analysis sandbox that uses TaintDroid to identify PII leaks from Android apps during dynamic analysis. Andrubis installs each app in an emulated Android environment and monitors its behavior for 240 seconds. Besides calling all of the app's registered components and simulating common events, such as incoming SMS and location changes, it uses Monkey [8] to generate approximately 8,000 pseudo-random streams of user events. In addition to detailed analysis report including all detected data leaks, it also provides the recorded network packet traces. However, this analysis fails for 38.57% of apps because they exceed the file size and/or API level limit of Andrubis.

Hybrid IFA. AppAudit flags functions that potentially leak PII through static analysis and then performs a simulated dynamic analysis to filter out candidates functions to confirm PII leaks. AppAudit reports leaks to the network, file system and through SMS from various sources such as the location, contacts and device specific identifiers. The analysis failed for 17.14% of apps, but runtimes are very fast compared to the previous approaches with an average of 14.16s per app. However, as AppAudit only approximates the execution of suspicious functions, it does not record any network packet traces.

		Type of PII being leaked				
	# leaks detected	**Device Id.**	**User Id.**	**Con-tacts**	**Loca-tion**	**Cred-entials**
Andrubis	plaintext	173	N-A	10	8	N-A
	obfuscated	124	N-A	16	0	N-A
	incorrect	140	N-A	24	6	N-A
	Total	457	N-A	50	14	N-A
ReCon	TP	146	17	7	35	0
	FN	27	0	0	0	0

Table 4: **Comparison with Andrubis (which internally uses TaintDroid), for Android apps only.** *TaintDroid has a higher false positive rate than ReCon, but catches more device identifiers. After retraining ReCon with these results, ReCon correctly identifies all PII leaks. Further, ReCon identifies PII leaks that TaintDroid does not.*

Methodology and results. For our comparison we reuse the 850 apps from *AppsApk.com* and the top 100 apps from *Google Play* from §2.2, and focus on the 280 that produced network traffic in our experiments. Since static and hybrid IFA approaches do not provide network flows, we base our comparison on the number of apps that leak a certain type of PII. We flag an app as a leaking a certain type of PII, if any of the tested tools detected a PII leak in that category, which is the case for 208 apps in our dataset. Table 3 shows the number and percentage of apps that were correctly flagged by FlowDroid, Andrubis, AppAudit and *ReCon*. FlowDroid mainly identified location leaks and the phone number, while AppAudit mainly identified IMEI leaks. Andrubis performed well in detecting device identifiers (ICCID, IMEI, IMSI) and the phone number. *Importantly, ReCon identifies more PII leaks overall (except for contact information), and in more categories than previous approaches.*

The above results are encouraging for *ReCon*, and we further investigated mismatches between *ReCon* and TaintDroid results, since the latter provides network traces that we can process via *ReCon*. Note, as the authors of TaintDroid themselves acknowledge [19], it may generate false positives (particularly for arrays and IMSI values), due to propagating taint labels per variable and IPC message. We thus manually inspected flows flagged as leaking PII, and discarded cases where the identified PII did not appear in plaintext network flows (*i.e.* false positives). Table 4 shows the results of our analysis, grouped by PII type.

We use the plaintext leaks identified by Andrubis as ground truth, and evaluate our system by sending the Andrubis network traffic through *ReCon*. The *ReCon* false positive rate was quite low (0.11%), but the false negative rate was relatively high (14.9%). The vast majority of false negative flows were Device ID leaks (124/457 are

Approach	#apps leaking PII (#reports)	Device Identifier	User Identifier	Contact Info Information	Location	Credentials
FlowDroid (Static IFA)	44 (187)	28 (14.58%)	×	8 (53.33%)	25 (48.08%)	×
Andrubis (Dynamic IFA)	72 (172)	68 (35.42%)	×	9 (60.00%)	3 (5.77%)	×
AppAudit (Hybrid IFA)	46 (232)	42 (21.88%)	×	3 (20.00%)	1 (1.92%)	×
ReCon	155 (280)	145 (75.52%)	6 (100%)	4 (26.66%)	29 (55.77%)	0 (-)
Union of all approaches	208 (280)	192	6	15	52	0

Table 3: **Comparison of *ReCon* with information flow analysis tools.** *This comparison is based on 280 Android apps (apps from the Google Play and AppsApk dataset for which we observed network flows). We present the number of Android apps detected as leaking PII, as well as the percentage of leaking apps detected by each tool out of all leaking apps detected by any of the tested tools in each category (× means the tool does not track that type of information).*

obfuscated and 140/457 are false positive reports from Andrubis). *Importantly, when we retrain ReCon's classifier with the Andrubis data, we find that all of the false negatives disappear.* Thus, *ReCon* is *adaptive* in that its accuracy should only improve as we provide it more and diverse sets of labeled data. In the next section we describe results suggesting that we can also use crowdsourcing to provide labeled data.

We also note that *ReCon* identified several instances of PII leaks that are not tracked by IFA. These include the Android ID, MAC address, user credentials (username and password), gender, birthdays, ZIP codes, and e-mail addresses.

5.4 User Study

We now describe the results of our IRB-approved user study, where participants used *ReCon* for at least one week and up to 101 days, interacted with our system via the UI, and completed a follow-up survey. Our study was biased in that most participants (75%) are students in computer science and located in the Boston area. While we cannot claim representativeness, we can use the user feedback quantitatively, to understand the impact of labeling on our classifiers. We also use the study qualitatively, to understand what information was leaked from participant devices but not those in our controlled experiments, and to understand users' opinions about privacy.

The study includes 31 users in total, with 24 iOS devices and 13 Android devices (some users have more than one device). We initialized the *ReCon* classifiers based on the results of the controlled experiments, then retrained the classifiers based on user feedback.

Runtime. While the previous section focused on runtime in terms of training time, an important goal for *ReCon* is to predict and extract PII in-band with network flows so that we can block/modify the PII as requested by users. As a result, the network delay experienced by *ReCon* traffic depends on the efficiency of the classifier.

We evaluated *ReCon* performance in terms of PII prediction and extraction times. The combined cost of these steps is less than 0.25 ms per flow on average (std. dev. 0.88), and never exceeds 6.47 ms per flow. We believe this is sufficiently small compared to end-to-end delays 10s or 100s of milliseconds in mobile systems.

User feedback. Study participants were asked to view their PII leaks via the *ReCon* UI, and label them as correct or incorrect. As of Sep 22, 2015, our study covers 565,128 flows, of which 7,560 were predicted to contain PII. Of those, there are 4,077 TP flows, 173 FP flows and 3310 unlabeled flows. Table 5 shows the results across all users. *The users in the study found few cases when ReCon incorrectly labeled PII leaks.* The vast majority of unlabeled data is device identifiers, suggesting that users might not understand or care about this type of leak.

For those flows that were incorrectly labeled, we retrained the classifier with these user labels. After this step, we found 11 false positive flows only, but missed 16 true positive flows (due to using a general classifier).

User survey. To qualitatively answer whether *ReCon* is effective, we conducted a survey where we asked participants "Have you changed your ways of using your smartphone and its applications based on the information provided by our system?" A majority (20/26) of responses from our users indicated that they found the system useful and changed their habits related to privacy when using mobile devices (full results in Appendix A.3). This is in line with results from Balebako et al. [11], who found that users "do care about applications that share privacy-sensitive information with third parties, and would want more information about data sharing."

PII leak characterization. We now investigate the PII leaked in the user study. As Table 5 shows, the most commonly leaked PII is device identifiers, likely used by advertising and analytic services. The next most common leak is location, which typically occurs for apps that customize their behavior based on user location. We also find user identifiers commonly being leaked (*e.g.* name and gender), suggesting a deeper level of tracking than anonymous device identifiers. Depressingly, even in our small user study we found 165 cases of credentials being leaked in plaintext (94 verified by users). These results highlight the negative impact of closed mobile systems—even basic security is often violated by sending passwords in plaintext (10 apps in our study).

We further investigate the leaks according to OS (Table 5). We find that the average iOS user in our study experienced more data leaks than the average Android

	Leak Type	total	Feedback on leaks: correct	wrong	no label
iOS	Device ID.	2853	12	165	2676
	User ID.	321	103	1	217
	Contact Info.	6	3	1	2
	Location	3643	3620	5	18
	Credential	31	22	0	9
Android	Device ID.	317	2	0	315
	User ID.	31	30	0	1
	Contact Info.	8	8	0	0
	Location	216	205	0	11
	Credential	134	72	1	61

Table 5: **Summary of leaks predicted by OS.** *We observe a higher number of leaks for iOS because the number of iOS devices (24) is more than the number of Android devices (13).*

user, and particularly experienced higher relative rates of device identifier, location, and credential leaks.

We investigated the above leaks to identify several apps responsible for "suspicious" leaks. For example, the ABC Player app is inferring and transmitting the user's gender. The Brainscape app leaks user credentials, including password, in plaintext. Last, All Recipes—a cookbook app—is tracking user locations even when there is no obvious reason for it to do so.

6 Related Work

Our work builds upon and complements a series of related work on privacy and tracking. Early work focused on tracking via Web browsers [5, 42]. Mobile devices make significant PII available to apps, and early studies showed PII such as location, usernames, passwords and phone numbers were leaked by popular apps [46]. Several efforts systematically identify PII leaks from mobile devices, and develop defenses against them:

Dynamic analysis. One approach, dynamic taint tracking, modifies the device OS to track access to PII at runtime [19] using dynamic information flow analysis, which taints PII as it is copied, mutated and exfiltrated by apps. This ensures that all access to PII being tracked by the OS is flagged; however, it can result in large false positive rates (due to coarse-granularity tainting), false negatives (*e.g.* because the OS does not store leaked PII such as a user's password), and incur significant runtime overheads that discourage widespread use. Running taint tracking today requires rooting the device, which is typically conducted only by advanced users, and can void the owner's warranty. Other approaches that instrument apps with taint tracking code still either require modifications to platform libraries [12], and thus rooting, or re-signing the app under analysis [41], essentially breaking Android's app update and resource sharing mechanisms. In addition, taint tracking does not address the problem of which PII leaks should be blocked (and how), a problem that is difficult to address in practice [27].

Static analysis. Another approach is to perform static analysis (*e.g.* using data flow analysis or symbolic execution) to determine *a priori* whether an app will leak privacy information [9, 10, 15, 18, 20, 25, 29, 31, 38, 48, 50–52]. This approach can avoid run-time overhead by performing analysis before code is executed, but state-of-the-art tools suffer from imprecision [14] and symbolic execution can be too time-intensive to be practical. Further, deploying this solution generally requires an app store to support the analysis, make decisions about which kinds of leaks are problematic, and work with developers to address them. Static analysis is also limited by obfuscation, and tends not to handle reflection and dynamically loaded code [53]. A recent study [36] finds dynamically loaded code is increasingly common, comprising almost 30% of goodware app code loaded at runtime.

Network flow analysis. *ReCon* analyzes network flows to identify PII leaks. Previous studies using network traces gathered inside a mobile network [21, 47], in an ISP [37], and in a lab setting [33] identified significant tracking, despite not having access to software instrumentation. In this work, we build on these observations to both identify how users' privacy is violated and control these privacy leaks *regardless of the device OS or network being used.*

PrivacyGuard [45] and AntMonitor [34] use the Android VPNService API to intercept traffic on Android devices and perform traffic analysis. A key limitation of these approaches is they rely on hard-coded identifiers for PII, or require knowledge of (and direct access to) a user's PII to work. Further, these approaches work only for the Android OS. In contrast, *ReCon* is cross-platform, does not require a priori knowledge of PII, and is adaptive to changes in how PII leaks.

7 Conclusion

In this paper we presented *ReCon*, a system that improves visibility and control over privacy leaks in traffic from mobile devices. We argued that since PII leaks occur over the network, detecting these leaks at the network layer admits an immediately deployable and cross-platform solution to the problem. Our approach based on machine learning has good accuracy and low overhead, and adapts to feedback from users and other sources of ground-truth information.

We believe that this approach opens a new avenue for research on privacy systems, and provides opportunities to improve privacy for average users. We are investigating how to use *ReCon* to build a system to provide properties such as k-anonymity, or allow users to explicitly control how much of their PII is shared with third parties—potentially doing so in exchange for micropayments or access to app features (see Appendix A.5).

References

[1] Ad blocking with ad server hostnames and ip addresses. http://pgl.yoyo.org/adservers/.

[2] App Annie App Store Stats. http://www.appannie.com/.

[3] AppsApk.com. http://www.appsapk.com/.

[4] AwaZza. http://www.awazza.com/web/.

[5] Lightbeam for Firefox. http://www.mozilla.org/en-US/lightbeam/.

[6] Meddle IRB consent form. https://docs.google.com/forms/d/1Y-xNg7cJxRnlTjH_56KUcKB_6naTfRLqQlcZmHtn5IY/viewform.

[7] SSLsplit - transparent and scalable SSL/TLS interception. http://www.roe.ch/SSLsplit.

[8] UI/Application Exerciser Monkey. https://developer.android.com/tools/help/monkey.html.

[9] Y. Agarwal and M. Hall. ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing. In *Proc. of MobiSys*, 2013.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of PLDI*, 2014.

[11] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen. "Little Brothers Watching You:" Raising Awareness of Data Leaks on Smartphones. In *Proc. of SOUPS*, 2013.

[12] J. Bell and G. Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proc. of OOPSLA*, 2014.

[13] T. Book and D. S. Wallach. A Case of Collusion: A Study of the Interface Between Ad Libraries and Their Apps. In *Proc. of ACM SPSM*, 2013.

[14] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. of NDSS*, 2015.

[15] X. Chen and S. Zhu. DroidJust: Automated Functionality-aware Privacy Leakage Analysis for Android Applications. In *Proc. of WiSec*, 2015.

[16] S. Consolvo, J. Jung, B. Greenstein, P. Powledge, G. Maganis, and D. Avrahami. The Wi-Fi Privacy Ticker: Improving Awareness & Control of Personal Information Exposure on Wi-Fi. In *Proc. of UbiComp*, 2010.

[17] J. Crussell, R. Stevens, and H. Chen. MAdFraud: Investigating Ad Fraud in Android Applications. In *Proc. of MobiSys*, pages 123–134. ACM, 2014.

[18] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proc. of NDSS*, 2011.

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

[20] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proc. of TRUST*, 2012.

[21] P. Gill, V. Erramilli, A. Chaintreau, B. Krishnamurthy, D. Papagiannaki, and P. Rodriguez. Follow the Money: Understanding Economics of Online Aggregation and Advertising. In *Proc. of IMC*, 2013.

[22] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proc. of WiSec*, 2012.

[23] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[24] S. Han, J. Jung, and D. Wetherall. A Study of Third-Party Tracking by Mobile Apps in the Wild. Technical Report UW-CSE-12-03-01, University of Washington, 2012.

[25] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *Proc. of MobiSys*, 2014.

[26] Z. Harris. Distributional structure. *Word*, 10(23):146–162, 1954.

[27] 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 ACM CCS*, 2011.

[28] M. Huber, M. Mulazzani, S. Schrittwieser, and E. Weippl. Appinspect: Large-scale Evaluation of Social Networking Apps. In *Proc. of ACM COSN*, 2013.

[29] J. Jeon, K. K. Micinski, and J. S. Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. Technical Report CS-TR-5022, University of Maryland, College Park, 2012.

[30] C. Johnson, III. US Office of Management and Budget Memorandum M-07-16. `http://www.whitehouse.gov/sites/default/files/omb/memoranda/fy2007/m07-16.pdf`, May 2007.

[31] J. Kim, Y. Yoon, K. Yi, and J. Shin. SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *Proc. of MoST*, 2012.

[32] H. King. No. 1 paid app on iTunes taken down by developer. `http://money.cnn.com/2015/09/18/technology/peace-ad-blocking-app-pulled/index.html`, September 2015.

[33] B. Krishnamurthy and C. Wills. Privacy Diffusion on the Web: A Longitudinal Perspective. In *Proc. of ACM WWW*, 2009.

[34] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou. AntMonitor: A system for monitoring from mobile devices. In *Proc. of Wrokshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*, 2015.

[35] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market. In *Proc. of ACM HotMobile*, 2012.

[36] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proc. of BADGERS*, 2014.

[37] Y. Liu, H. H. Song, I. Bermudez, A. Mislove, M. Baldi, and A. Tongaonkar. Identifying personal information in internet traffic. In *Proceedings of the 3rd ACM Conference on Online Social Networks (COSN'15)*, Palo Alto, CA, November 2015.

[38] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proc. of ACM CCS*, 2012.

[39] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. In *Proc. of ACM SIGCOMM*, 2015.

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

[41] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, Y. Chen, W. Zhu, and W. Chen. Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android (to appear). In *Proc. of SecureComm*, 2015.

[42] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. *Proc. of USENIX NSDI*, 2012.

[43] Sandvine. Global Internet Phenomena Report. `https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/1h-2014-global-internet-phenomena-report.pdf`, 1H 2014.

[44] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep packet inspection over encrypted traffic. In *Proc. of ACM SIGCOMM*, 2015.

[45] Y. Song and U. Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices (to appear). In *Proc. of ACM SPSM*, 2015.

[46] The Wall Street Journal. What They Know - Mobile. `http://blogs.wsj.com/wtk-mobile/`, December 2010.

[47] N. Vallina-Rodriguez, J. Shah, A. Finamore, H. Haddadi, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft. Breaking for Commercials: Characterizing Mobile Advertising. In *Proc. of IMC*, 2012.

[48] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective Real-time Android Application Auditing. In *IEEE Symposium on Security and Privacy*, 2015.

[49] N. Xia, H. H. Song, Y. Liao, M. Iliofotou, A. Nucci, Z.-L. Zhang, and A. Kuzmanovic. Mosaic: Quantifying Privacy Leakage in Mobile Networks. In *Proc. of ACM SIGCOMM*, 2013.

[50] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proc. of USENIX Security*, 2012.

[51] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proc. of ACM CCS*, 2013.

[52] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proc. of ACM CCS*, 2013.

[53] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proc. of ACM CODASPY*, 2015.

A Appendix

A.1 Observations from Manual Tests

We observed that some iOS apps implemented certificate pinning: Several apps use pinning at least for login and registration (Facebook/Facebook Messenger, WhatsApp, Google, Gmail, Dropbox), others throughout the app's functionality (iTunes U, Vine, Twitter, Periscope). Some games seem to use certificate pinning on startup when they are downloading additional game data (Angry Birds 2, Game of War - Fire Age, Jurassic World). Furthermore, 4 apps prohibited usage over VPN (Snapchat, Snap Upload For Snapchat, OfferUp, Google Translate). Candy Crash Saga produced an error message about not being able to connect to Facebook, yet still had access to the account's friends. Tumblr also seems to perform some kind of certificate checking: It produced an error message about not being able to complete the registration, but the account was still created. Finally, the registration and login for both Netflix and Ibotta did not work with and even without intercepting the traffic.

A.2 IRB details

We are using *ReCon* for an IRB-approved study (#13-08-04) that reports data from capturing all of a subject's Internet traffic, which raises significant privacy concerns. The study protocol entails informed consent from subjects who are interviewed, where the risks and benefits of our study are explained. The incentive to use *ReCon* is Amazon.com gift certificates. To protect the data collected, we use public key cryptography to encrypt the captured data before it is stored on disk. Further, subjects can delete their data and disable monitoring at any time. Per the terms of our IRB, we cannot make this data public due to privacy concerns.

In our second deployment model, we have IRB approval (#13-11-17) for a follow-up study where we record only the first few bytes of the HTTP payload, reducing the risk of recording sensitive information. We conduct informed consent using an online form, allowing us to enroll users worldwide. The incentive to use our system is increased privacy; in return, we collect limited information that allows us to validate the effectiveness of *ReCon* and improve its accuracy with user feedback.

Response	Count
I spent more time reviewing claims made by applications regarding access to my data, like contacts, location and so on.	6
I stopped using certain applications because Meddle shows they leak too much personally identifiable information.	11
I learned to keep location service off unless needed.	4
I used Meddle to block information that I do not want leaked.	2
No change.	3

Table 6: **User survey results for the question of whether information revealed by *ReCon* changed participant habits.** Most users took action to address privacy as a result of information provided by *ReCon*. Some users chose multiple options.

A.3 Full survey results

Table 6 presents the full set of questions and responses from our user study about the effectiveness of *ReCon*.

A.4 Privacy and Incentives

Beyond the context of the user study, we will provide incentives and deployment models that balance privacy and utility for *ReCon*. First, we will make our software source code publicly available to build trust from users, Second, we will provide easy-to-use hardware and software that allows users to run the *ReCon* system on their own devices inside their own network. This substantially reduces the privacy risk because user traffic never traverses an untrusted machine, and it opens up exciting research opportunities, such as bumping SSL connections to identify and block PII in HTTPS flows.

An interesting challenge is how to incorporate a crowdsourced classifier in this deployment model. We believe that we can retrain each user's classifier locally based on feedback, then exchange the models themselves with other users. Because the models should not contain any PII (rather, they store the features associated with PII), the privacy risk should be minimal. However, it is an open question whether we can ensure that PII does not leak via side channels.

A.5 Alternative Architectures for PII Sharing

In the current implementation, *ReCon* relies on being able to identify PII in plaintext flows. Naturally, if users begin to block or change their PII using *ReCon*, trackers and advertisers may resort to obfuscation and encryption to avoid detection. In response, we can simply retrain *ReCon* to identify obfuscated PII leaks, using available static and dynamic analysis tools that are resilient to these evasion techniques. Of course, this could lead to an endless cat-and-mouse game of PII detection evasion. We hope to avoid this using *ReCon* to promote explicit PII sharing, where users and third parties engage in an incentive-driven, mutually beneficial service. In the case that third parties choose not to participate in such a scheme, we can provide strong incentives by *blocking all traffic to those sites* unless they cooperate.