ReCon: Revealing and Controlling Privacy 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 Sophia Antipolis, [3]Univ. of Helsinki, [4]Vienna Univ. of Technology

Abstract

Mobile systems have become increasingly popular for providing ubiquitous Internet access; however, recent studies demonstrate that software running on these systems extensively tracks and leaks users' personally identifiable information (PII). We argue that these privacy leaks persist in large part because mobile 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 describe *ReCon*, a cross-platform system that reveals PII leaks and gives users control over them without requiring any special privileges or custom OSes. Specifically, our key observation is that PII leaks must occur over the *network*, so we implement our system in the network using a software middlebox built atop the Meddle platform. While this simplifies access to users' network flows, the key challenges for detecting PII from the network perspective are 1) how to efficiently and accurately detect users' PII without knowing a priori what their PII is and 2) whether to block, obfuscate, or ignore the PII leak. To address this, we use a machine learning approach to detect traffic that contains PII, display these behaviors via a visualization tool and let the user decide how the system should act on transmitted PII. We discuss the design and implementation of the system and evaluate its methodology with measurements from controlled experiments and flows from 16 users (19 devices) as part of an IRB-approved user study.

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 via WiFi and cellular data access, 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. Previous work shows that apps extensively track users and leak their personally identifiable information (PII) [13, 16, 20, 26, 40], and users are generally unaware and unable to stop them [15, 22].

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 [40, 42] provide visibility but give researchers no control over network flows. Likewise, custom Android extensions provide control over network flows but measurement visibility is limited to the devices running these custom OSes or apps [17], often requiring warranty-voiding "jailbreaking". Static and dynamic analysis tools can identify and block privacy leaks based on the content of the code implementing an app, but cannot defend against dynamic code loading at run time nor explore every possible code path.

We argue that improving privacy in this environment 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 privacy leak must (by definition) occur over the network, so interposing on network traffic is a natural way to detect and mitigate PII leaks. Based on this insight, we propose a simpler, more effective strategy than previous approaches: *using indirection [37, 38] to improve visibility and control for PII leaks in mobile network traffic.* We use natively supported OS features to redirect all of a device's Internet traffic to a third-party server to identify and control privacy leaks in network traffic. This allows us to explore the potential of detecting privacy leaks from network flows *without* needing privileged access to a mobile ISP; rather, we use software middleboxes running atop trusted servers (*e.g.* in users' home networks, or on a trusted cloud platform). This solution does not require rooting devices or deploying hardware, so it is immediately deployable globally.

This paper focuses on the problem of understanding and mitigating privacy 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 private information in network flows, even when we do not know a priori what information is leaked and in what format. Because mobile traffic increasingly uses SSL, we describe techniques that allow our system to detect private information leaks in encrypted flows with user opt in as well.

Our key contributions are as follows:

- A study using controlled experiments to demonstrate that significant PII leaks from mobile devices in plaintext, motivating the need for (and potential effectiveness of) systems that identify PII leaks from network flows. We find extensive use of unique identifiers ($> 20\%$ of the top 100 Android apps) and even six of the top 100 iOS apps leaking passwords over plaintext.
- An approach to detect and extract PII leaks from arbitrary network flows, using machine learning informed by extensive ground truth for more than 33,900 flows generated by mobile apps.
- A system that enables users to access information about privacy leaks from network flows, provide feedback about relevant leaks, and optionally change information sent to third parties.
- 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 98.2% 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 the accuracy comparable to the several ensemble methods atop DTs (AdaBoost, Bagging, and Blending) that take significantly more time (by a factor of 6.43) to achieve the same level of accuracy.
- 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 581 labels for PII leaks) and characterize how these users' PII is leaked "in the wild." For example, we find sensitive information such as usernames, passwords, gender, and locations being leaked in plaintext flows.

In the next section, we discuss related work and describe the *Meddle* platform on which *ReCon* is built. Section 3 presents the results of controlled experiments identifying extensive information leakage in popular apps. We then describe the design (Section 4) and implementation (Section 5) of *ReCon*. We evaluate the effectiveness of our approach using controlled experiments and data from users in Section 6. In Section 7 we discuss limitations and future work, and conclude in Section 8.

2. BACKGROUND

In this section, we describe previous work in the area of privacy leaks and provide background information on the *Meddle* system on which our *ReCon* system is built.

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" [28]. 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. For the purpose of this work, we define PII to be either 1) **Device Identifiers** specific to a device or OS installation (ICCID, IMEI, IMSI, Android ID), 2) **User Identifiers** which identify the user (name, gender, birthday), 3) **Contact Information** (telephone numbers, address book information), 4) **Location** (GPS latitude/longitude), or 5) **Credentials** (username, password). This list of PII is informed by information leaks observed in this study, but is not exhaustive. As we identify other types of PII leaks we will incorporate them into our analysis.

2.2 Related Work

Recent studies show that most of our time spent online is being tracked by third parties, and the apps we use are leaking personally identifiable information (*e.g.* location, passwords, and phone numbers) over the Internet without our knowledge [5, 35, 39]. These third parties that gather information about users' Web and app usage are commonly called *Trackers* [19].

In the fixed-line environment most tracking is performed through Web browsers, a topic that is the focus of the Mozilla LightBeam project [5], also explored by Roesner *et al.* [35]. In the mobile environment, the problem worsens, mainly because mobile devices make significant amounts of PII readily available to apps (*e.g.* location, phone number, contacts, etc); early studies showed information such as location, usernames, passwords and phone numbers were leaked to third parties by popular apps [39]. A recent study [12] used TaintDroid along with notifications of privacy leaks to understand the impact of user awareness of privacy violations.

Several efforts systematically identify privacy leaks from mobile devices, and develop defenses against them. These approaches generally fit into information flow analysis, both through dynamic and static analysis, and network flow analysis.

Dynamic Analysis. One approach, dynamic taint tracking, modifies the device OS to track access to PII at runtime [17] 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, depending on the implementation 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. 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 [25].

Static Analysis. An alternative 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 [11, 16, 27, 44]. This approach can avoid runtime overhead by performing analysis before code is executed, but state-of-the-art tools suffer from inprecision [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. For example, F-Droid [4] is an app store that statically analyzes the apps and warns users about tracker libraries used by the apps. Static analysis is also limited by obfuscation, and tends not to handle reflection and dynamically loaded code [46]. A recent study [32] finds dynamically loaded code is increasingly common, comprising almost 30% of the latest goodware app load code at runtime.

These approaches, and follow-up work that extends them [10, 18,23,29,33,43,45], can improve mobile privacy but depend on a deployment model that restricts their impact to custom OSes or app stores. Privacy leakage, however, is an evolving target affecting all users and all platforms.

The approach taken in this work is to analyze network flows for PII leaks. Previous studies using network traces gathered inside a mobile network [19, 40] and in a lab setting [30] 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.*

2.3 Meddle

Existing solutions to address privacy in mobile systems are limited because they do not have visibility into network flows from mobile devices, the ability to modify them, and/or a deployment model that facilitates large-scale adoption to ensure broad impact. At first glance, addressing all these limitations seems to impose a high barrier to success, as it may require custom OSes and/or privileged access to mobile carrier networks.

The key insight that enables this work is that we can in fact achieve these goals *today*, without requiring any privileged access to networks or OS modifications. Using a system we call *Meddle*, we achieve 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 (top of Fig. 1). Once traffic arrives at the proxy server, we use software middleboxes to enable users and researchers to exert control over mobile-device traffic (middle of Fig. 1).

Meddle supports a plugin infrastructure for custom flow processing. Each plugin takes as input a network flow and outputs a (potentially modified or empty) network flow. When a packet arrives at *Meddle*, a software-defined switch [7] determines the ordered set of plugins that the corresponding flow will traverse. Plugins support a variety of features such page speed optimization and content blocking (*e.g.* for blocking ads).

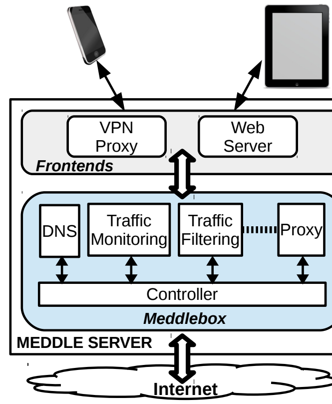


Figure 1: *Meddle* **Architecture**. *Mobile devices (top) communicate with a Meddle frontend (VPN proxy and/or Web proxy). VPN proxy traffic is forwarded to a meddlebox, which provides software middlebox services to monitor and/or interpose on network traffic.*

Meddle supports controlled experiments on mobile device network traffic, and a key advantage of this approach is that it facilitates *in situ* measurement and experimentation via an end-user deployment. Toward this latter goal, *Meddle* presents a number of incentives that appeal to a wide range of users, including improved security through encrypted tunnels and device-wide content filters often used for ad-blocking. In this paper, we describe how to use this platform to enable *privacy revelations* [41] via the *ReCon* tool, which allows users to see how they are being tracked by apps, and allows them to customize how this information is shared.

Meddle is easy enough to install that even non-expert users can run it. Configuring a VPN on iOS is done via installing a configuration file which we provide, and on Android requires filling out five fields. We are hosting a cloud-based deployment that is free for users, to support large numbers of flows for *in situ* experimentation. The initial prototype has reasonably low overheads (typically between 5-30 ms of extra delay, and a 10% increase in power consumption) [34]. As evidence of the potential to recruit users, a related approach [3] that uses APNs and in-network proxy-based performance optimizations attracted tens of thousands of users.

2.4 Protecting User Privacy

An important concern with a *Meddle* user study is privacy. Using an IRB-approved protocol [6], we encrypt and anonymize all captured flows before storing them. The secret key is stored on a separate secure server and users can delete their data at any time.

We will make the *Meddle* software publicly available. For those who want to run their own *Meddle* instance (*e.g.* if they

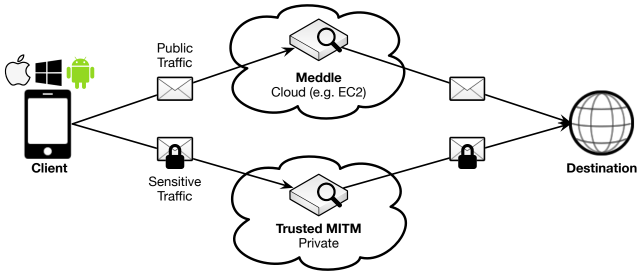


Figure 2: *Meddle* **Deployment Options**. *Users concerned about the privacy of their sensitive traffic can deploy their private Meddle instance and e.g. redirect plaintext traffic to our EC2 deployment and SSL traffic to their trusted instance.*

do not want to participate in our study), the *Meddle* server requires only that a user has root on a modern Linux OS. *Meddle* 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, or VM in the cloud, as depicted in Figure 2. Specifically, the "Trusted MITM" in the figure refers to a trusted *Meddle* instance (*e.g.* running in the user's home network) that can man-in-the-middle HTTPS connections to identify PII leaked over encrypted flows. *Meddle* is currently in private beta with dedicated-server, EC2, and Aliyun deployments in the US, France, and China.

3. MOTIVATION AND CHALLENGES

In this section, we describe how 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 three 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.

3.1 Methodology

Our goal with controlled experiments is 1) to obtain ground-truth information about network flows generated by apps and devices, and 2) to characterize the network activity for a large variety of apps in a lab setting. We use this data to understand how to model apps' network behavior, how to map network flows to the app that generated them, and how to identify PII in those network flows. We use SSL bumping [8] to decrypt and inspect SSL flows only during our controlled experiments where *no user traffic is intercepted*.

Device Setup. We conducted our controlled experiments using two Android devices (running Android 4.0 and 4.3) and an iPhone running iOS 6. 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 Android apps in the *Google Play* store and 209 iOS apps from the iOS *App Store* on April 4, 2013. For each app, we install it, enter user credentials, interact with it for up to 10 minutes, and uninstall it. 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. The second set of controlled experiments consist of fully-automated experiments on 922 Android apps from the free, third-party Android market *AppsApk.com* [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 100,000 actions using *Monkey* [9], and finally uninstall the app and reboot the device to end any lingering connections. We limit the automated tests to Android devices because iOS does not provide equivalent scripting functionality.

Andrubis/TaintDroid. The Andrubis sandbox uses TaintDroid to identify PII leaks from Android apps during dynamic analysis. They find that data leaks to the network increased from 13.45% to 49.78% of all submitted apps between 2010 and 2014 [32]. To understand PII leaks based on their analysis, we submit the above 922 Android apps in our dataset to Andrubis and report the results. Andrubis was able to report the results for only 770 of the 922 apps because some apps do not generate network traffic and others exceed the file-size limit for Andrubis.

Analysis. We use *tcpdump* and *bro* to analyze the network traffic generated during our experiments, 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]. We further augment this list with the domains that received PII during our controlled experiments (discussed in §3), and recent research on mobile ads [25,31].

3.2 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 detailed 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 Android and iOS apps in the plaintext. The device identifiers (specifically the Android ID tied the Android device) that can be used to track user's behavior, are most frequently leaked PII by Android apps. Table 1 shows that other PII—user identifiers, contacts, location, and *credentials such as username and password*—are also leaked in plaintext.

OS	Store	Testing Technique	# of Apps Tested	# Apps leaking a given PII Device Identifier	User Identifier	Contact Information	Location	Credentials
iOS	App Store	Manual	209	4 (1.9%)	4 (1.9%)	13 (6.2%)	20 (9.5%)	6 (2.8%)
Android	Play Store	Manual	100	34 (34%)	2 (2%)	3 (3%)	10 (10%)	1 (1%)
Android	AppsApk	Auto. (Monkey)	922	287 (27.5%)	1 (0.11%)	12 (1.3%)	12 (1.3%)	0(0%)
Android	AppsApk	Auto. (Andrubis)	770	89 (11.5%)	N-A	0 (0%)	7 (0.9%)	N-A

Table 1: **Summary of PII leaked in plaintext (HTTP) by Android and iOS apps.** *The popular iOS apps tend to leak the location information in the clear (20 iOS apps leak location info.) while Android apps leak the device identifiers in the clear.*

We also observed that the information leaked by an app depends on the OS. Of the top 100 apps for Android, 26 apps are available on both iOS and Android. Of these 26 apps, 17 apps leaked PII on at least one OS: 12 apps leaked PII only on Android, 2 apps leaked PII only on iOS, while only one app had the same data leakage in both OSes. Of the remaining 2 apps that leaked PII, one app leaked the device identifiers (specifically the Android ID and IMEI) in Android and the credentials (username) in iOS, while the other app leaked the device identifier (Android ID) in Android and location in iOS. The difference in the PII leaks is primarily due to the different privileges that the underlying OS provides these apps, and that iOS was tested manually (and thus facilitated entering username and passwords in the appropriate fields).

During our experiments, we observed that PII is also sent over encrypted channels. We observe that three of the top 5 domains that receive PII over SSL are trackers. Our observations highlight the limitations of current mobile OSes with respect to controlling access to PII via app permissions. In particular, it is unlikely that users are made aware that they are granting access to PII for tracker libraries embedded in an app that serves a different purpose. This problem is pervasive: of the 77 domains that received some PII in the clear or over SSL during our controlled experiments, 18 domains were trackers.

We note that our observations are a conservative estimate of PII leakage because we cannot detect PII leakage using obfuscation (*e.g.* via salted hashing). Regardless, our study shows that a significant number of PII leaks are visible from *Meddle*.

3.3 Summary and Challenges

While the study in the previous section 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. As a strawman solution to this challenge, consider an extension to the above approach where we automatically run every app in every app store. 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.

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.

Obfuscation of PII. 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 [36]. 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. This work focuses on identifying PII leaks in plaintext network traffic, so our approach 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).

In addition to using encryption, the parties leaking PII may use obfuscation to hide their information leaks. In our experiments, we found little evidence of this (§ 6.2.5). 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. In the ensuing cat-and-mouse game, we envision automating this process of reverse engineering obfuscation.

4. 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 *Med-*

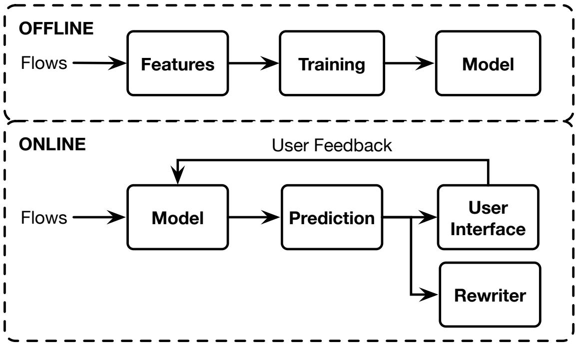


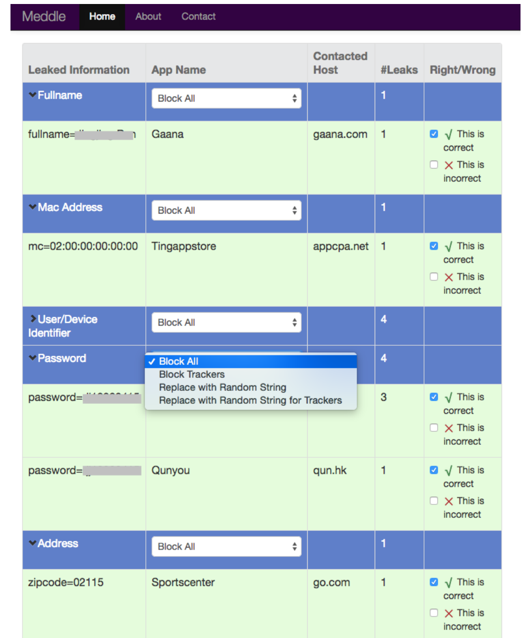
Figure 3: *ReCon* **Architecture**. *We conduct feature selection and model training on labeled network flows, then use this model to predict whether new network flows are leaking PII. Based on user feedback, we retrain our classifier.*

dle application to visualize how users' information is shared with various sites, and allow 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 sub-goals of *ReCon* are as follows:

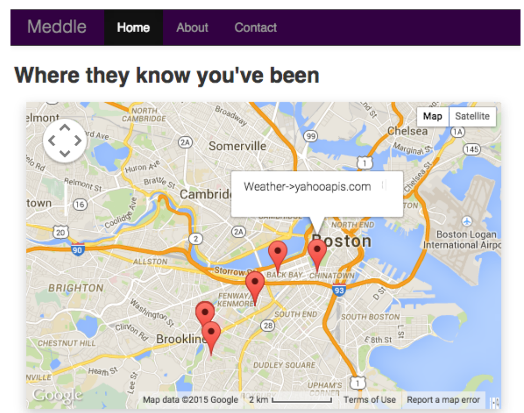
- Accurately identify PII in network flows, *without* requiring knowledge of users' PII *a priori*.
- Improve awareness of PII leaks by presenting this information to users.
- Automatically improve the classification of sensitive PII based on user feedback.
- Enable users to change these flows by modifying 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 study, we are investigating how to design 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 3 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 we can use to predict whether PII is leaked. When new network flows enter *ReCon* via *Meddle* (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* user interface (Fig. 4). 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 (top). In addition,



(a) Screen capture of the *ReCon* tool



(b) Screen capture of the Map View

Figure 4: *Screen capture of the* ReCon *tool, allowing users to view how their PII is shared with third parties, and to validate the suspected PII leaks, and create custom filters to block or modify leaks.*

ReCon supports a map view, where we display the location information information that each domain is learning about the user (bottom). A demo of *ReCon* is available at `http://goo.gl/v52tbg`.

To support modification/blocking of PII, *ReCon* allows users to instruct the system to replace the PII with other text (or nothing) for future flows that leak PII (see the drop-down boxes in Fig. 4(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 are investigating how to support *interactive* PII labeling and filtering, potentially using push notifications or other channels to notify the user of leaks immediately when they are detected (as done in a related study [12]).

Non-Goals. Note that *ReCon* is not intended as a blanket replacement for existing approaches to improve privacy in the mobile environment. For example, identifying privacy leaks from mobile devices may be reliably addressed using information flow analysis [17]. However, due to the overheads of this approach, it is difficult to deploy to users and at scale. In contrast, *Meddle* allows us to identify and block unobfuscated PII in network flows from arbitrary devices without requiring OS modifications or taint tracking.

5. RECON IMPLEMENTATION

We now discuss several key aspects of our system implementation. We evaluate our design decisions in the following section.

5.1 Detecting PII from Network Flows

The first step in our pipeline is to identify PII leaks from network flows. We use *bro* to parse packet traces into logs for protocol-specific analysis, focusing on HTTP flows because most PII leaks occur over HTTP. These logs are passed to a machine learning classifier for labeling as a PII leak or not.

5.2 Machine Learning Techniques

We use the *weka* data mining tool [21] to train classifiers that predict PII leaks. We train our classifier by extracting 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,[1] so we use the "bag-of-words" model [24]. 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 appear in a flow is set to 0.

A key challenge for feature extraction in network flows is that, unlike in many documents, 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`). Amusingly, one domain (Yahoo) even leaks PII using SQL: "`...from XXX where (lat = 42.33N and lo = 71.09W) and lang='en-GB'`". 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 bag-of-words model is simple, but produces far too many features to be useful for training accurate classifiers that can make predictions within milliseconds (so we can intercept PII leaks in-band with traffic). To help 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. A key challenge is how to select a reasonable threshold value without sacrificing classifier accuracy, so we select the threshold empirically based on accuracy and classification time for labeled data (discussed in Section 6.2.3).

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 general classifier. Because this general classifier tends to have low numbers of labeled PII leaks, we are 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.

5.3 Automatically Extracting PII

A machine learning classifier indicates whether a flow contains PII, but does not indicate *which content in the flow is*

[1] Here, we treat a network flow as a document, and its structured data as words.

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:

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

where *type* is the PII type (*e.g.* IMEI, email 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 $P_{\text{type,key}}$ 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 and true positives for our dataset. Because some users may want more or less sensitivity, we will make this available as a configurable threshold in *ReCon* (*e.g.* if a user wants to increase the likelihood of increasing true positives at the potential cost of increased false positives).

In the second step, we leverage the 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.

6. 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.

6.1 Dataset and Methodology

To evaluate *ReCon*, we need app-generated traffic and a set of labels that indicate which of the corresponding flows leak PII. For this analysis, we reuse the data from controlled experiments presented in Section 3.1; Table 2 summarizes this dataset in terms of the number of flows generated by the apps, and fraction that leak PII. We identify that more than 1,000 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.

	Manual tests		*Automated tests*	
OS (Store)	iOS (App Store)	Android (Google Play)	Android (AppsApk)	
Automation type			*Monkey*	*Andrubis*
Apps tested	209	100	922	770
HTTP flows (total)	10066	14055	4207	5640
Flows leaking PII	110	697	260	305
Flows to trackers	1740	3715	1105	2604
Flows leaking PII to trackers	9	353	115	245

Table 2: **Summary of HTTP flows observed during our controlled experiments.** *Popular Android apps in the Google Play store tend to leak PIIs to known trackers more frequently compared to iOS apps. We train our classifier using these HTTP flows.*

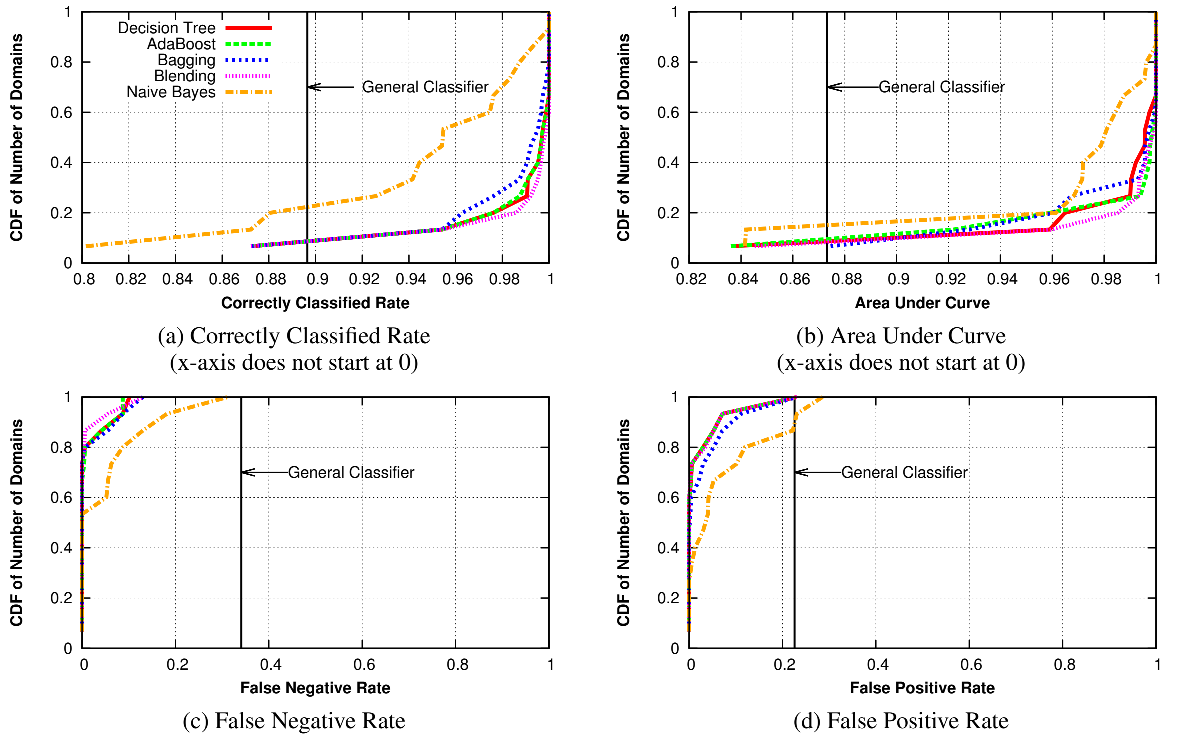
- **Correctly Classified Rate (CCR)**, the sum of true positive (TP) and true negative (TN) samples divided by the total number of samples. $CCR = (TN + TP)/(TN + TP + FN + FP)$.
  A good classifier has a CCR value close to 1.
- **False Negative Rate (FNR)**, the number of false negatives divided by the number of positive samples. $FNR = FN/(FN + TP)$
- **False Positive Rate (FPR)**, the number of false positives divided by the number of negative samples. $FPR = FP/(FP + TN)$
  A good classifier has FNR and FPR values close to 0.
- **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 the class. *A good classifier has a AUC value close to 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 a variety of machine learning approaches. Specifically, we use Naive Bayes, C4.5 Decision Tree (DT) and several ensemble methods atop DTs (AdaBoost, Bagging, and Blending).

6.2 Lab Experiments

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



(a) Correctly Classified Rate
(x-axis does not start at 0)

(b) Area Under Curve
(x-axis does not start at 0)

(c) False Negative Rate

(d) False Positive Rate

Figure 5: **CDF of per-domain classifier accuracy, for alternative classification approaches.** *For the 16 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 results in significantly worse performance.*

6.2.1 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 key 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 Figure 5. We plot the accuracy using a CDF over the domains that we use to build per-domain classifiers as described in Section 5.2. Focusing on the top two graphs (overall accuracy via CCR and AUC), we note 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).

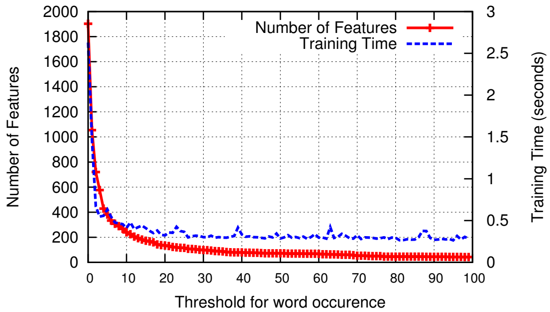
Among the ensemble methods, Blending with DTs and k-nearest-neighbor (kNN) yields the highest accuracy; however, the resulting accuracy is not significantly better than a simple DT. Importantly, a simple DT takes significantly less time to train than ensemble methods. For ensemble methods, the training time largely depends on the number of iterations for training. When we set this value to 10, we find that ensemble methods take 6.43 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 and FNs for the majority of domains. The domains with the largest false predictions 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` sometimes corresponds to an Android ID value and other times to some other unknown identifier.

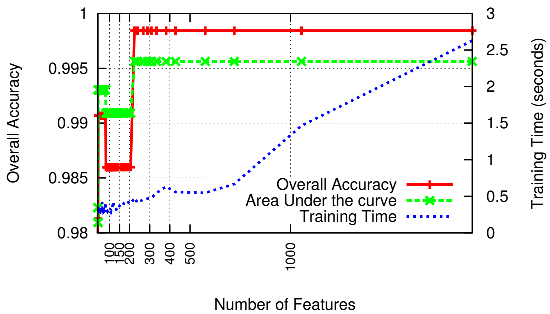
6.2.2 Per-Domain Classifiers

We now evaluate the impact of using per-domain classifiers instead of using one classier for all flows. We build per-domain classifiers for all domains with greater than 100 samples, 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 will use a common structure for leaking PII. Once *ReCon* acquires sufficient labeled data (*e.g.* from users or controlled experiments) for a destination domain, we create a per-domain classifier.

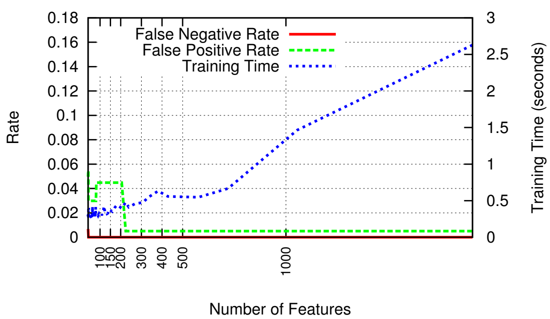
We evaluate the impact of per-domain classifiers on overall accuracy in Figure 5. The vertical lines in subgraphs are corresponding values for the general classifier, *which is trained using all flows from all domains*. We can see that the general classifier has lower accuracy (88% CCR) than >90% of the per-domain classifiers. Further, training such general classifiers is expensive in terms of runtime: it takes *minutes*



(a) #features changes as threshold changes



(b) accuracy and training time over #features



(c) false negative and false positive rate and training time over #features

Figure 6: **Feature Selection** *for the tracker domain flurry, the experiment varies threshold of word occurrence, which causes the change of number of features, and measure the trends of Overall Accuracy, False Negative and False Positive Rate and Training Time.*

to train per-domain classifiers for thousands of flows, but it takes *hours* to train general classifiers for the same flows.

6.2.3 Feature Selection

The accuracy of the classifiers described in the previous sections 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 6 to illustrate this problem and how we address it. Here, we focus on statistics for the tracker domain `flurry.com` (441 flows out of 642 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 5.2, we filter out features from words that appear infrequently. Fig. 6(a) shows the impact of this decision on training time, with the x-axis representing the minimum number of appearances for a word to be included as a feature, and the y-axis representing 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 1,903 to 101 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. 6(b), where the x-axis is the number of features, the left y-axis is accuracy (note that 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 200 does not improve performance at all (but does increase training time). We see a similar effect on the FP rate in Fig. 6(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.

6.2.4 PII Extraction Strategies

As discussed in Section 5.3, 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 the naive strawman, and

	# leaks detected	Type of PII being leaked Device Id.	User Id.	Contact Info.	Loca-tion	Creden-tials
Andrubis	plaintext (A)	195	N-A	15	0	N-A
	obfuscated (B)	44	N-A	8	0	N-A
	incorrect (C)	156	N-A	8	0	N-A
	Total (A+B+C)	396	N-A	31	0	N-A
ReCon	True Positive	117	21	16	24	1
	False Negative	77	0	0	0	0

Table 3: **Comparison with Andrubis (which internally uses TaintDroid).** *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.*

the FP and FN rates are sufficiently low to correctly extract PII the vast majority of the time.

6.2.5 Comparison with TaintDroid

While the previous sections evaluate the accuracy of our approach using ground-truth information based on searching for known PII in network flows, our labeled dataset may miss PII leaks that are obfuscated or otherwise hidden from our analysis. We now evaluate our approach by comparing with an approach that is resilient to such issues: information flow analysis via TaintDroid.

We use Andrubis [32] to conduct this study. Andrubis is a public Android app analysis sandbox that leverages TaintDroid as part of its analysis to capture data leaks. We submitted the apps in our dataset from the third party, *AppsApk.com* [2] store to Andrubis.

Andrubis installs each app in an emulated Android environment and monitors its behavior during a default runtime of 240 seconds. Besides calling all the apps registered components and simulating common events, such as incoming SMS and location changes, it uses Monkey [9] to generate approximately 8,000 pseudo-random streams of user events. After each experiment, it provides a detailed analysis report including all detected data leaks, as well as the recorded network packet traces.

We analyzed the reports and packet traces, and compared the result with sending the same packet traces through *ReCon*. Note that TaintDroid may generate false positives (particularly for arrays and IMSI values), due to propagating taint labels per variable and IPC message [17]. 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 3 shows the results of our analysis, grouped by PII type.

We use the plaintext and obfuscated 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 (0.65%), but the false negative rate was relatively high (21.3%). The vast majority of false negative flows were IMEI leaks (12/289 are obfuscated and 130/289 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 early 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 TaintDroid. These include user credentials (username and password), gender, birthdays, ZIP codes, and e-mail addresses.

System	# HTTP flows	PII prediction time (ms/flow) min.	average	max.
iOS (13 devices)	107255	0.05	0.24 (std.dev=0.88)	5.82
Android (6 devices)	39457	0.01	0.11 (std.dev=2.2)	6.47

Table 4: **PII prediction time per flow by OS.**

6.3 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 20 days, interacted with our system via the UI, and completed a follow-up survey. Note that our user study was biased by the fact that most participants 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 can 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 16 users in total, with 13 iOS devices and 6 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.

6.3.1 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. Table 4 shows the results for minimum, average, and maximum delays. Importantly, the combined cost of these steps never exceeds 6.47 milliseconds per flow and is typically less than one millisecond. We believe this is sufficiently small compared to end-to-end delays 10s or 100s of milliseconds in mobile systems.

6.3.2 Impact of User Feedback

		User feedback on accuracy		
Leak Type	**predicted**	**correct**	**incorrect**	**unlabeled**
Device Id.	1158	16	74	1068
User Id.	93	66	0	27
Contact Info.	8	5	1	2
Location	397	392	5	0
Credentials	22	21	1	0

Table 5: **Summary of leaks found across all device OSes in IRB-approved user study (146,712 HTTP flows).** *Device identifiers, location, and user identifiers are the most commonly leaked PII .*

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.	3
I learned to keep location service off unless needed.	2
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.

Participants in our study were asked to view their PII leaks via the *ReCon* UI, and label them as correct or incorrect. As of May 5, 2015, our study covers 146,712 flows, of which 1,678 were predicted to contain PII. Of those, we have 500 TP flows, 81 FP flows and 1,097 unlabeled flows. Table 5 shows the results across all users. *Importantly, the users in the study found few cases when ReCon incorrectly labeled PII leaks.*

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.

6.3.3 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?" The results are summarized in Table 6. Although our sample set is small, the survey shows that the majority of users found the system useful and changed their habits related to privacy when using mobile devices. This is in line with results from Balebako et al. [12], where the authors found that users "do care about applications that share privacy-sensitive information with third parties, and would want more information about data sharing."

6.3.4 PII Leak Characterization

We now investigate what information was leaked in the user study. As Table 5 shows, the most commonly leaked information includes device identifiers, likely used by advertising and analytic services. The next most common leak is location information, which typically occurs for apps that customize their behavior based on user location. We also find significant amounts of user identifiers being leaked (*e.g.* name and gender), suggesting a deeper level of tracking than simply anonymous device identifiers. Depressingly, even in our small user study we found 22 cases of credentials being leaked in plaintext (21 verified). These results highlight the negative impact of closed mobile systems—even basic security is often violated by sending passwords in plaintext over untrusted channels.

We further investigate the leaks according to OS (Table 7). We find that iOS users in our study collectively experienced more data leaks than Android users, and particularly experience higher relative rates of device identifier, location, and credential leaks.

We investigated the leaks described above to identify the apps responsible for suspicious leaks and present the results in Table 8. For example, the ABC Player app is inferring and transmitting the user's gender. The Brainscape app leaks user credentials, including password, in plaintext. Additionally, we observe that All Recipes–a cookbook app–is tracking user locations even when there is no obvious reason for it to do so.

			Feedback on leaks		
	Leak Type	**predicted**	**correct**	**incorrect**	**unlabeled**
iOS	Device Id.	1101	14	74	1013
	User Id.	82	55	0	27
	Contact Info.	6	3	1	2
	Location	343	338	5	0
	Credentials	16	16	0	0
Android	Device Id.	57	2	0	55
	User Id.	11	11	0	0
	Contact Info.	2	2	0	0
	Location	54	54	0	0
	Credentials	6	5	1	0

Table 7: **Summary of leaks predicted by OS.** *We observe a higher number of leaks for iOS because the number of iOS devices (13) is more than the number of Android devices (6).*

		Feedback on leaks		
App (PII Leaked)	**#leaks predicted**	**#leaks correct**	**#leaks incorrect**	**#leaks unlabeled**
ABC Player (Gender)	3	3	0	0
Brainscape (Password)	3	3	0	0
All Recipes (Location)	24	3	0	21

Table 8: **Examples of PIIs leaked by Apps.** *The ABC Player leaked the users' gender (category User Identifier), the Brainscape app leaked the password (category Credential), while the All Recipes app leaked the users' location (category Location) in the clear.*

7. DISCUSSION

Privacy and Incentives. We are using *ReCon* for an IRB-approved study 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. Last, we note that users must trust our system with their network data; to address this we will make our code open-source to build this trust and to enable users to deploy *ReCon* on a server under their control.

Outside of the context of our IRB-approved study, we propose the following privacy protections and incentives. First, we will make our software source code publicly available, and allow 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.

In our second deployment model, we have IRB approval 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.

Other Deployment Models. *ReCon* is currently implemented as part of a VPN proxy, but the general techniques apply to any environment with access to user network flows. As such, it can be integrated into the mobile device OS or carrier networks. In this work, we focused on the *Meddle*-based deployment because it offers the ability to deploy our solution today, without any OS vendor or carrier support.

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.

Dataset Limitations. While our study showed *ReCon* has high accuracy in our experiments, we need more data to ensure our results apply to a broad set of users. Access to large network traces could benefit our system, as will a larger user deployment to both gather a better understanding of PII leaked during user interaction and to obtain a larger set of user-labeled data. In such an environment, we must also develop techniques to deal with mislabeled crowdsourced data.

8. 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 the vast majority of PII leaks occur over the network, detecting these leaks at the network layer is a natural fit. 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. As next steps, we will incorporate additional static and dynamic analysis tools to improve our classifiers by identifying information leaks that are difficult to detect from network flows alone.

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

9. REFERENCES

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

[2] AppsApk.com. `http://www.appsapk.com/`.

[3] AwaZza. `http://www.awazza.com/web/`.

[4] F-Droid. `https://f-droid.org/`.

[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] Open vSwitch: An open virtual switch. `http://openvswitch.org/`.

[8] Squid-in-the-middle SSL Bump. `http://wiki.squid-cache.org/Features/SslBump`.

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

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

[11] 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.

[12] 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.
[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] 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.
[16] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proc. of NDSS*, 2011.
[17] 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.
[18] 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.
[19] 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.
[20] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proc. of WISEC*, 2012.
[21] 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.
[22] 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.
[23] 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.
[24] Z. Harris. Distributional structure. *Word*, 10(23):146–162, 1954.
[25] 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.
[26] M. Huber, M. Mulazzani, S. Schrittwieser, and E. Weippl. Appinspect: Large-scale Evaluation of Social Networking Apps. In *Proc. of ACM COSN*, 2013.
[27] 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.
[28] 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.
[29] J. Kim, Y. Yoon, K. Yi, and J. Shin. SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *Proc. of MoST*, 2012.
[30] B. Krishnamurthy and C. Wills. Privacy Diffusion on the Web: A Longitudinal Perspective. In *Proc. of ACM WWW*, 2009.
[31] 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.
[32] 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.
[33] 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.
[34] 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.
[35] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. *Proc. of USENIX NSDI*, 2012.
[36] Sandvine. Global Internet Phenomena Report. `https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/1h-2014-global-internet-phenomena-report.pdf`, 1H 2014.
[37] D. Spinellis. Another Level of Indirection. In A. Oram and G. Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 17, pages 279–291. O'Reilly and Associates, 2007.
[38] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proc. of ACM SIGCOMM*, 2002.
[39] The Wall Street Journal. What They Know - Mobile. `http://blogs.wsj.com/wtk-mobile/`, December 2010.
[40] 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.
[41] D. Wetherall, D. Choffnes, B. Greenstein, S. Han, P. Hornyack, J. Jung, S. Schechter, and X. Wang. Privacy Revelations for Web and Mobile Apps. In *Proc. of HotOS*, 2011.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] 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.