PrivacyProxy: Leveraging Crowdsourcing and In Situ Traffic Analysis to Detect and Mitigate Information Leakage

Anonymous Author(s)

ABSTRACT

Smartphone apps often transmit personally identifiable information (PII) without the user's knowledge. To address this issue, we present *PrivacyProxy*, a system that monitors outbound network traffic and generates app-specific signatures that describe the data being shared, all without any modifications to the OS. We use a crowdsourced approach to detect likely PII in an adaptive and scalable manner by anonymously combining signatures from different users. Our design is also privacy sensitive as we do not see users' network traffic and instead rely on cryptographically hashed signatures . We present the design and implementation of PrivacyProxy and evaluate its effectiveness in detecting known and unknown PII through a lab study and a field deployment. Our field study shows that without any user intervention, PrivacyProxy achieves a precision of 0.79 in detecting PII. We also show that the performance overhead of PrivacyProxy is under 15% and majority of the users report no perceptible impact on battery life or the network. [SK: I can't see 15% overhead anywhere in evaluation section. Is this number correct?]

CCS CONCEPTS

• **Operating Systems** → **Security and Protection**; **Privacy**; • **Information Systems Applications** → **Miscellaneous**;

KEYWORDS

Privacy, Crowdsourcing, Android

1 INTRODUCTION

NIST defines personally identifiable information (PII) as "any information that can be used to distinguish or trace an individual's identity" [3]. Past work has found many smartphone apps aggressively collect PII [12, 13, 22, 30, 32, 36, 37]. Some PII is used to directly benefit users, but there are also cases where PII is used to track users' behaviors without their consent, potentially by multiple trackers and across different apps. If PII could be detected and scrubbed before leaving a user's device, many associated privacy concerns can be mitigated.

Focusing on Android, any app can access the unique, per-device Android ID [1] without any permission. Other identifiers include Google Advertising ID (AID) and IMEI number. Also, apps and third-party libraries can generate their own unique identifiers. As such, simply using permissions to block apps from sharing PII is not enough. Furthermore, identifiers may not always have predictable names or formats, making it hard to have a comprehensive blacklist based on regular expressions.

Anonymous Submission to ACM CCS 2017, Due 19 May 2017, Dallas, Texas
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Adding to the complexity of preventing PII leakage is the pervasive use of third-party libraries. According to Lin et al., a typical Android app uses 1.59 (std = 2.82) third-party libraries [28]. Libraries within the same app can access the same PII, but for different purposes. For example, an app might contain two libraries, one using location for maps and the other for targeted ads. This means enforcing PII access control on a per-app basis may not suit the user's needs since she might want to share PII with certain parts of an app. Another finding by Lin et al. is that library use among Android apps follows a power-law distribution, with the top 400 most popular libraries included in more than 90% of the apps analyzed [28]. This implies that denying PII access for one app does not mean an interested third party cannot obtain the same information via a different app, which complicates enforcement of PII access policies.

To address these problems, we present *PrivacyProxy*, a privacy-sensitive approach for automatically inferring likely PII using network analysis and crowdsourcing. Like other proxy-based approaches [35–37], we use a VPN to intercept and analyze smartphone network data, thus offering privacy protections without modifying apps or the OS. However, instead of routing all traffic to a remote server for analysis (like ReCon [36]), PrivacyProxy looks for key-value pairs in HTTP requests on the smartphone itself and only sends [YA: cryptographically hashed signatures] to our servers in the cloud, thereby minimizing potential privacy and security risks. Each signature represents the content of a request to a remote host without revealing actual content, letting us identify potential PII based on how unique a signature is. Additionally, instead of requiring hard-coded rules or regular expressions (like PrivacyGuard [37] or HayStack [35]) or requiring users to label network traces for training (like ReCon [36]), our approach is adaptive and robust against data obfuscation. [SK: Data obfuscation is generally intentional to evade detection, PP can handle changes in keys and formatting changes, not intentional obfuscation] Furthermore, we show that as the user base of PrivacyProxy grows, we can more accurately infer PII. [YA: add that if we have quantitative data comparing to them.] Lastly, since the scan takes place on the user's device, we can also filter out PII before it ever leaves the device while maintaining a small trusted computing base. In summary, this paper makes the following contributions:

- A privacy-sensitive approach that uses network analysis and crowdsourcing to detect likely PII leaked by apps without requiring any changes to the app or OS. Our approach uses a novel and adaptive way of detecting PII without relying on a priori knowledge of names or formats of identifiers. Our approach also does not require labeled network traces or any user intervention for detection of PII.
- A system that lets users filter leaks in real time and replace detected PII with fake data that is similar in structure to the detected PII.

- The results of lab studies, a small field trial, a user survey, and a comparison against two pieces of past work. PrivacyProxy achieved a precision of 0.79 in the field study. We also present examples of how precision increases as number of users increase. [YA: number?] We also show that 85% of users in our survey found no perceptible change in battery life and 68% found no change in network performance.

2 BACKGROUND AND RELATED WORK

Personally Identifiable Information (PII) refers to "any information that can be used to distinguish or trace an individual's identity" [3]. In this paper, we focus on identifiers such as user names, Android ID, IMEI, MAC address, Google Advertising ID, phone numbers, and so on. We are also interested in identifiers generated by apps or libraries, e.g. using Java's UUID class or hashing one's MAC address. Furthermore, we are interested in likely identifiers, such as install time of an app or location data. Here, we limit the scope of identifiers to just single values. We currently do not consider fingerprinting approaches that look at variations in smartphone sensor data (e.g., [20]) or combinations of keys (as has been done with web browsers, e.g. [21]).

Prior work in mobile privacy and PII detection can be broadly grouped into five categories. First, static analysis techniques, which aim to improve privacy behavior of mobile apps by analyzing the apps' source code or binaries. The second category involves modifying the underlying mobile OS to be more privacy preserving by adding new primitives. The third set of approaches look at anonymization of network traffic. The fourth category looks at using network flow analysis to detect and remove PII. The final set of approaches focus on making users more aware of the privacy leaks and mitigating their burden to make privacy decisions.

Static Analysis: Static analysis approaches aim to identify the uses of sensitive information by analyzing an app's source code or binary. FlowDroid [14] and DroidSafe [25] are static information-flow analysis tools that, given a set of sources and sinks, can statically identify possible PII leaks.

AndroidLeaks [24] uses taint-aware slicing to find potential PII leaks. PrivacyGrade [5, 28] infers the purpose behind each permission request for Android apps by decompiling apps and searching for sensitive API calls made by third-party libraries. Wang et.al. [40] used text mining on decompiled apps to infer the purpose of permission use for location and contact lists. These static analysis approaches do not change the functionality of the app at runtime and do not provide PII access policy enforcement, but rather are used to inform users and developers of potential data leaks and relevant information about data leaks. Static analysis techniques also fail to work in the presence of dynamic code loading [43] and reflection techniques. Additionally, these techniques can only detect access to sensitive data using well-defined permissions or APIs rather than arbitrary identifiers that an app might generate.

Instrumenting the OS and APIs: Another approach for enabling better smartphone privacy is to modify the underlying OS and intercept certain OS APIs that access private user data. For example, ProtectMyPrivacy [12] and xPrivacy [11] intercept API calls requesting sensitive information and alert the user. Mockdroid [18] modifies the Android OS so it can substitute private data with mock data. TaintDroid [22] offers a custom version of the Android OS itself that provides dynamic taint-tracking, making it possible to track information flow from sources (requests for sensitive information) to sinks where data is used or sent outside. AppTracer[31] uses binary rewriting to determine if sensitive resource usages happen due to user interactions in applications or not. While these approaches augment native privacy protection features of iOS and Android, their applicability is limited to rooted or jailbroken devices. PrivacyBlocker [4] uses static analysis on app binaries and replaces calls for private data with hard coded shadow data. This approach however requires target apps to be re-written and reinstalled and cannot be done at runtime.

The long term goal of this line of research is to have these primitives make their way into popular smartphone OS's by hopefully influencing manufacturers. In the interim, PrivacyProxy offers an immediately deployable solution to any user with an unmodified device and OS. In addition, PrivacyProxy only relies on VPN functionality, which is likely to be continually supported in the future versions of these OSes. In addition, our approach can detect an expanded set of PIIs, including those generated by Apps themselves, and not just ones that are accessible using well known OS APIs.

Privacy Preserving Network Architecture: Another line of research is network-based anonymization. The Accountable and Private Internet Protocol [33] argues that privacy-preserving Internet protocols should allow senders to hide their network address. Tor [8] uses onion routing to disguise the identity of the sender from all intermediate nodes as well as from the receiver. Orbot [9] is an Android app that tunnels all smartphone traffic through Tor. However, these approaches merely anonymize the source and ultimate destination of network traffic and cannot detect or prevent PII leaks. PrivacyProxy does not focus on this kind of anonymization, but could easily use Tor if desired.

Network Flow Analysis: The closest area of work is network flow analysis, which seeks to detect and remove PII as it leaves a device. Privacy Oracle uses differential black-box fuzz testing to detect leaks [26], feeding different sets of inputs to an app and looking for associated changes in network traffic. However, this approach is hard to scale up because there are millions of smartphone apps, and one must generate multiple sets of inputs per app. In contrast, PrivacyProxy's novel signature-based detection algorithm scales much better and the accuracy of inferences improves as the number of users increase. Also, PrivacyProxy uses valid network requests based on users' regular use of their apps rather than generating inputs, which increases the possibility of finding more PIIs due to increased coverage. Chattering laptops [16] found that many network protocols leak PII often due to service discovery. They proposed having devices remember which networks specific services have been configured to use. [YA: you could remove the last line if space needed.]

PrivacyGuard [37] and Haystack [35] use Android's VPN Service API to intercept network traffic. However, they both rely on regular expressions to find potential PII, limiting the types of PII they can detect. For example, an app might generate its own identifier or use non-standard encodings. Recon [36] uses a VPN to redirect all user

traffic to a proxy in the cloud and uses heuristics to identify potential PII. However, Recon requires full trust in their cloud service. Recon also does not differentiate between flows from different apps, leading to potential false positives. PrivacyProxy offers a hybrid approach, where data is processed locally on the user's smartphone and only hashed data is sent to PrivacyProxy servers. As more users post hashed data about apps, PrivacyProxy identifies the likely PII based on the uniqueness of the data. Additionally, PrivacyProxy server does not see any actual user data, mitigating certain privacy and security concerns. Finally, PrivacyProxy is more robust to changes in apps. Other approaches will stop working if the key names or formatting of private data changes, while PrivacyProxy will continue to work without any intervention.

User Oriented Privacy: [YA: remove section, if space needed] A great deal of past research has focused on techniques to help users in making better privacy decisions. Van Kleek et. al. [39] use network monitoring to map out various Data Controller Indicators (DCIs), that is organizations where data is being sent, their contextual background information, their purposes for collection data, etc. They use various visualizations to display DCIs which can help users consider more potential privacy implications while allowing them to make more consistent privacy decisions with more confidence. Balebako et al [17] showed visualizations of amount and types of data shared, as well as just-in-time notifications when data is shared. They did a study on whether such feedback can help reduce the gap between the users' understanding and actual privacy leakages. However, these tools are not designed to determine which of the data being sent is private. They use specialized methods to detect limited set of PII, thus they will miss many potential PII. Previous work shows that the relevant privacy information can nudge the users towards more privacy conscious decisions. Kelley et.al. [27] investigated how specialized permissions and privacy displays affects users' app-selection decisions. Liu et.al. look at designs to nudge users towards better privacy decisions while reducing their burden [29]. Some approaches use machine learning models based on contextual information to predict user privacy decisions at runtime when users are prompted to grant permissions at runtime [34, 41]. However, none of the above approaches are designed to detect potential PII or filter them.

3 SYSTEM ARCHITECTURE

Our overarching objective with PrivacyProxy is to develop a practical system that increases smartphone users' awareness of and control over PII that their apps send over the network. To achieve this, we had a number of design goals. First, to maximize utility for everyday users, our solution must work on stock, non-rooted/jailbroken devices. Second, we want to detect a broad range of PII, including known trackable IDs (e.g. UUIDs, MAC addresses, etc), previously unknown identifiers dynamically generated by specific apps or libraries, and likely PII such as location. Third, we want to provide users with effective notifications and fine-grained controls to prevent the flow of PII over the network by blocking or anonymizing the data. Fourth, we want PrivacyProxy to impose minimal performance overhead on apps as compared to without using our system. Fifth, we want a solution that is usable in practice by requiring only minimal user interaction for privacy decisions. Finally, the solution should be scalable as the number of users and the apps that they use grow.

Threat Model: [SC: Updated:] For any system whose goal is securing or managing privacy, defining a threat model is important since it is difficult to protect against all threats. The above design goals are based around the assumption that developers are generally honest and are not trying to deliberately obfuscate their activities. For example, if an app uses custom app layer encryption (not SSL, which we do handle), non-standard encodings, or some indirect ways to track users, we will not be able to detect them. Note that, while network blocking based on certain DNS name/IP address may be able to block sending PII to that server, it essentially makes the system equivalent to an ad-blocker, consequently, breaking the revenue model of the current smartphone industry (e.g. using Advertisements).

3.1 Intercepting Network Traffic

To observe what data (including potential PII) is being sent over the network, we need a way to intercept network traffic on *unmodified* smartphones. We also need to separate traffic into different flows of information, taking into account individual apps, hosts where data is being sent to, etc. To achieve this, we build upon the internal Virtual Private Networking (VPN) service provided by most modern smartphone OSes. Traditional VPNs intercept all device traffic and forward it over an encrypted tunnel to another server, which can then process and/or forward the data to the eventual destination. VPNs can operate at multiple layers, including both transport layer (SSL/TLS) and also at the network layer (e.g. IPSec).

We build on PrivacyGuard [37], which does the following: (a) it registers a `Fake VPN service` to intercept all IP packets; (b) implements `TCP-Forwarders` and `UDP-Forwarders` that implement TCP and UDP protocols; (c) instantiates a `LocalServer` on the smartphone which pretends to be the destination for all app-server communication and acts as a man-in-the-middle (MITM) proxy; (d) connects the LocalServer with the actual remote server for each request and does the necessary TCP handshakes; (e) has the LocalServer pass back the response from the real server to the TCP forwarders, which delivers the IP packets back to the app.

To decrypt SSL/TLS, we perform MITM SSL injection. With the user's consent, we add a trusted root certificate on the device that lets PrivacyProxy sign any certificate for end hosts that individual apps contact, so that the LocalServer process described above can finish the SSL handshake and decrypt packets sent by an app on the device itself before sending the packets through a new SSL connection with the actual server.

3.2 Detecting PII

A key design goal of PrivacyProxy is to detect a wide variety of PII. We assume most apps use RESTful APIs that encode data as HTTP requests/responses. Prior work has shown that using HTTP/HTTPS is common for client/server communication in Android [23]. We also confirmed this based on the apps we analyzed, showing that more than 81% use HTTP or HTTPS and have highly structured responses. [YA: how many Apps? What about the other 19%? Can we say something about them?]

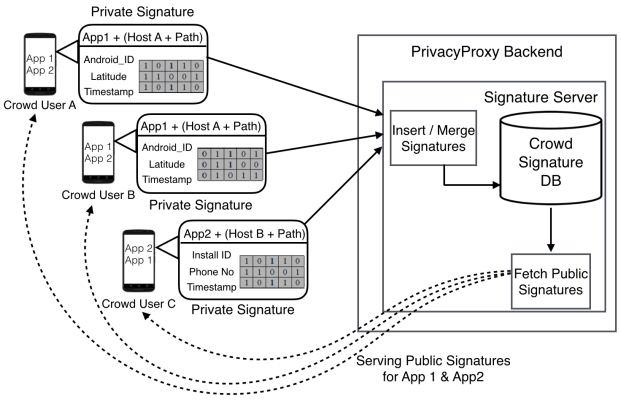

Figure 1: PrivacyProxy detects likely PII using network analysis and crowdsourcing. Users upload to our server anonymized private signatures from key-value pairs extracted from their network traffic. Signatures with the same Signature ID are merged into a public signature. Users fetch all public signatures for the apps they use, letting users see how common (or not) their private signatures are compared to everyone else.

3.2.1 Extracting Key-Value Pairs. After intercepting traffic with the LocalServer, we extract `Key-Value` pairs. Figure 2 shows an example request and key-value pairs that would be extracted. In particular, we extract all arguments in the request URI. We also extract key-value pairs from HTTP headers, ignoring common headers like "Accept-Encoding" and "Content-Type", instead looking for uncommon headers like "X-Secret-Location-Header". We attempt to parse these header values as JSON, XML, or a URL, and if that fails we treat it as text. For the message body we use hints such as the "Content-type" header to parse, failing which we try common formats such as JSON, XML, and URL encoded. If that fails, we treat the entire body as a text value and assign a fixed key to it. Note, our scheme will not work if apps use their own proprietary encoding. In our tests, we observed less than 0.6% of all HTTP requests used some non-standard encoding that we could not parse. Our keys have prefixes such as "U" for URLs, "H" for header, and "B" for body to reduce conflicts. In addition, nested keys are flattened to include all the parent keys.

3.2.2 Signature Generation from Key-Value Pairs. The next step is to detect what is (and is not) PII based on extracted key-value data as described above. A novel aspect is doing this detection by aggregating contributed data from individual devices, anonymously, to generate crowdsourced signatures that identify which data items (per app) are likely PII. The intuition is that key-values that only appear in the data streams of a single or handful of devices are likely PII, while those appearing across multiple devices are not.

Each device contributes `signatures`, comprised of a `Signature ID` (SID) and extracted key-value pairs. A Signature ID, or SID is the tuple `(package-name, app version, method, host, path)`. For example, the HTTP request in Figure 2 was sent from version 1.0 of the io.example.app and would have an SID of (io.example.app, 1.0, GET, www.example.io, /api/v1). SIDs let us associate signatures for the same app/requests from different users' devices. The 'host' and 'path' fields lets us further distinguish where the PII is being sent to.

The other part of a signature is all the key-value pairs with the same SID. For each extracted `key`, we store a count-min sketch [19] which represents the frequencies of all the different `values` associated with a given key, as shown in Figure 2. **Count-min Sketch** is like a 2-dimensional Bloom filter that estimates the frequency of an event. Concretely, each count-min sketch is a collection of r integer arrays of length n (every element initialized to 0) and r hash functions. It supports 2 operations: $increment(v)$ and $count(v)$. The *increment* function takes in a value v and for each array $\alpha_i (1 \leq i \leq r)$, it increments the integer at $\alpha_i[hash_i(v)]$ by one. The count function returns an estimate of the frequency of value v by returning $min_{1 \leq i \leq r}(\alpha_i[hash_i(v)])$.

Note that count-min sketch is extremely space efficient since it only requires $r \times n$ integers. Given any value, it will either return the correct frequency or an overestimate, but never an underestimate. In addition, it does not reveal the actual values that were inserted, which is important from a privacy perspective. Multiple count-min sketches can also be trivially combined by doing a element-wise sum over each array. In our implementation, we use SHA-256 as the hash function and we generate r different hash functions by appending fixed random strings to the value v. Furthermore, for each count-min sketch, we also maintain a counter m which keeps track of the number of times the increment function was called, regardless of the value used. We chose $r = \lceil \ln \frac{1}{0.01} \rceil = 5$ and $n = \lceil \frac{e}{0.05} \rceil = 55$, which theoretically ensures that if the true frequency of v is f_v, then $P[count(v) \leq f_v + 0.05n] \geq 0.99$.

We generate two types of signatures using this approach. *Private Signatures* are generated by each user on their device. Periodically each device uploads their private signatures to the `Signature Server`. Note, since each private signature only contains the SID and the count-min-sketches but not the actual key-values, it is extremely space efficient and there are minimal privacy concerns about what information the PrivacyProxy Signature Server has. By combining these private signatures from different users, the signature server can generate a *Public Signature* which can then be periodically downloaded by devices based on which apps they have. To protect user privacy, we don't authenticate users and signatures. This opens up the possibility of Sybil attacks, which we discuss later.

3.2.3 Identifying Likely PII . Classifying a particular key-value pair as likely PII or not is relatively straightforward. The intuition is that a key-value pair that is unique to and repeatedly seen on a single device is likely PII, while a pair that is common to many devices is not. For example if an app uses a device-based UUID to track users using the key "UserID", its value (the UUID) is likely to be the same for requests from the same user, but different across other users of the same app. We discuss how we handle location and other special cases in Section 4.3.

Let $S_{private}$ and S_{public} represent the private and public signature for a particular SID, respectively. Let $F(S, k, v)$ denote the estimated frequency for the value v associated with key k as reported by signature S. Likewise, let $C(S, k)$ denote the number of values (not necessarily unique) that have been inserted into signature S for key k. For each key-value pair (k, v) extracted from the

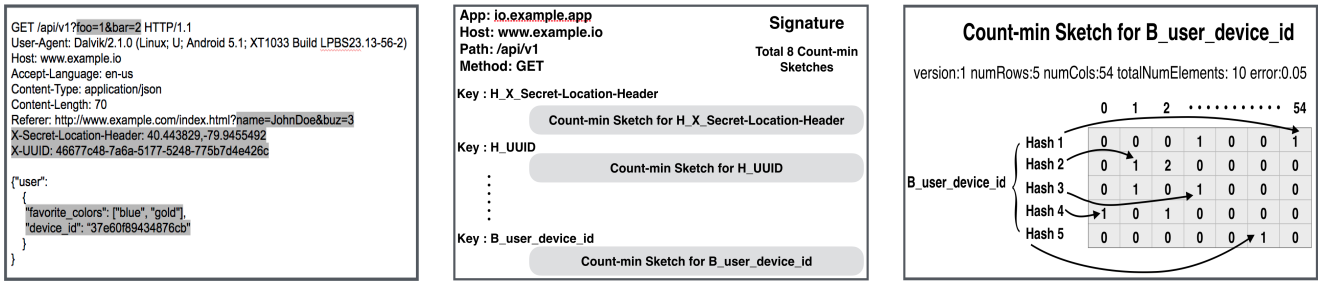

Figure 2: Signature generation process flow: (a) Our Request Parser looks for key-value pairs in app HTTP requests, looking at request URI query parameters, HTTP headers, and body. Extracted key-value pairs are highlighted. (b) Signature structure for the sample HTTP Request. A signature identifies the app used, destination, URL Path, and method, plus count-min sketch for all keys. c) Count-min sketch for key Device_ID. Count-min sketch is a probabilistic and compact data structure for counting the frequency of different values seen for each key.

request, let $P_{private}(k, v)$ be the estimated probability of seeing the pair in the user's own requests and $P_{public}(k, v)$ be the estimated probability of seeing the pair across all users' requests. We can then calculate

$$P_{private}(k, v) = \frac{F(S_{private}, k, v)}{C(S_{private}, k)}$$

Likewise we can do the same for $P_{public}(k, v)$. Let us consider the PII analysis for a given key-value pair (k, v). For brevity, we'll use $P_{private}$ to refer to $P_{private}(k, v)$ and P_{public} to refer to $P_{public}(k, v)$. We will let T be a tunable threshold between 0 and 1. Using these definitions, we can classify (k, v) into 3 categories:

(1) **Application Constant**

$$P_{private} \geq T \quad and \quad P_{private} \leq P_{public}$$

Values in this category are present in most requests of the user. However, they also appear frequently in the requests of other users. As a result we label them as application constants since they are likely to be the same for all the requests with the same SID. A good example would be developer IDs used by ad libraries to identify the apps that made the ad request.

(2) **Context-Sensitive Data**

$$P_{private} < T$$

Values in this category are very uncommon among requests from the user and thus unlikely to be PII or application constants. There are two likely explanations. The first are values dynamically generated by the app. Examples include timestamps, random numbers used by libraries such as Google Analytics to prevent caching [2], and app-specific checksums. The second are values associated with the user's actions. Examples include number of minutes the screen is on or the user's precise GPS location if the user frequently moves.

(3) **Personally Identifiable Information(PII)**

$$P_{private} \geq T \quad and \quad P_{private} > P_{public}$$

For values to be classified as PII, they need to be common for a single user and rare for others. We can further categorize detected PII into three confidence levels (L), Highly Likely ($L \geq 0.8$), Moderately Likely ($L \geq 0.2 \quad and \quad L < 0.8$), and Less Likely ($L < 0.2$), where $L = P_{private} - P_{public}$. From the definition of PII, it is clear that $L > 0$ for all PII values.

The selection of parameter T is flexible, although it should be set such that it prevents context-sensitive values from being misclassified as PII or application constants. Also, the properties of count-min sketch means that we will never underestimate $P_{private}$ and P_{public}. Along with theoretical guarantees on the bounds of $count(v)$, our PII detection algorithm is quite conservative in determining whether a given value is PII or not. By erring on the false positive side, the user can always mark the information as not PII. However, if we err on the true negative side, we risk PII being leaked. In general, it is hard to determine how privacy sensitive a piece of PII is. As such, we have designed PrivacyProxy to detect likely PII, but defer decisions about filtering to the user.

The **crowdsourcing** aspect of PrivacyProxy relies on contributions of private signatures. Early adopters will not gain much since our coverage of apps will be low, though we only need two users using the same app to provide basic PII detection for that app. As more users use PrivacyProxy, not only will our signature coverage improve, but classification will become more accurate as well. Also, this cold start problem will only happen once, and there are many ways to bootstrap our signature database, e.g. downloading the most popular apps onto test phones and having a monkey interact with apps, or just paying a small set of initial participants.

3.3 Mitigating PII Leaks

Filtering PII: We provide UIs to explicitly mark detected values as PII or not. If a user marks a value as PII, we then give her the option to filter this value in subsequent leaks. We generate fake values to replace values marked for filtering by identifying their type and structure using regexes. We currently detect MAC addresses, IPv4, IPv6, phone number, Social Security Number, and email addresses. In case of a match we use a default pre-configured value for each type as fake value that is the same for all users. [YA: same or different across users?] [GS: Same for all users. If different the fake value becomes PII.] If the value does not match any regex, we use a set of zeros maintaining the exact length of the value being filtered. For subsequent requests, we perform a simple search-and-replace to remove PII with the fake value generated before sending the request to the destination on the Internet.

Each time a marked PII is filtered in a request, the user receives a notification showing the actual PII and the fake data it was replaced

with. We solicit user feedback in order to determine whether the app leaking PII continues to work properly post filtering and if the fake data sufficiently protects user privacy.

Impact of Filtering on Apps: [SC: New:]Filtering the requests to replace PII with non-PII is an important feature of PrivacyProxy. However, its possible that doing so may impact the app functionality or in some cases may even cause the app to crash. If the user filters a PII from an app and it breaks, PrivacyProxy sends the HTTP response codes to the server. [YA: you mean if its not an HTTP-OK response code?] If the response codes for multiple users (for the same key-value pair) indicates that filtering broke the app, we add it to a global whitelist (app-crash whitelist) and display a warning to the user that filtering might break the apps. In addition, we get user feedback and ask the users if filtering the key-value pair made the app unusable. Again, we map the user's decision and add the key-value pair of the app to the global whitelist (app-unusable whitelist). We also do random sampling to remove the app from the whitelist, to protect against malicious actors. [YA: Not sure what the last sentence means?] [SC: updated it]

Case Study on Filtering: Authentication To understand the impact of filtering on apps, lets consider the special case when the App wants to actually authenticate a user. If an app is authenticating via OAuth tokens, then PrivacyProxy doesn't detect the token as PII since OAuth tokens are sufficiently different for every session, just like timestamps. On the other hand, if the authentication happens via username and password, PrivacyProxy detects the username and the password as PII. If the user decides to filter the username or password, the app authentication would fail. If the app crashes, the response codes would change and the key-value pair would be added to the global app-crash whitelist. The users would also provide feedback that filtering the username made the app unusable. [YA: Just curious: Do we have an example App?]

Purpose-Based Filtering: We also offer an app-agnostic way of blocking traffic to hosts known to collect PII, letting users select categories they want to block. Presently, PrivacyProxy offers three categories based-on purposes: Targeted Ads, Analytics, and User Tracking. We currently implement a naive approach where we drop packets if a request is sent to domains associated with selected categories. Identifying which hosts belong to what purpose categories is outside the scope of this paper. We currently maintain a table on our server based on data from PrivacyGrade.org [5] to map hosts to their corresponding purpose categories. [YA: Do we acrually do that? Also, somewhere earlier we explicitly say that we don't want to do this since then we become an ad-blocker to differentiate ourselves from network based firewall techniques?]

4 IMPLEMENTATION

We describe our implementation of PrivacyProxy on Android, highlighting how we support our design goals mentioned in Section 3. Note that our approach can be ported to any smartphone OS that exposes a VPN API. We plan to make the source code of PrivacyProxy available once the results have been published.

4.1 PrivacyProxy Android App

We use PrivacyGuard's FakeVPN and LocalServer to intercept traffic and perform man-in-the-middle for SSL/TLS traffic. PrivacyGuard

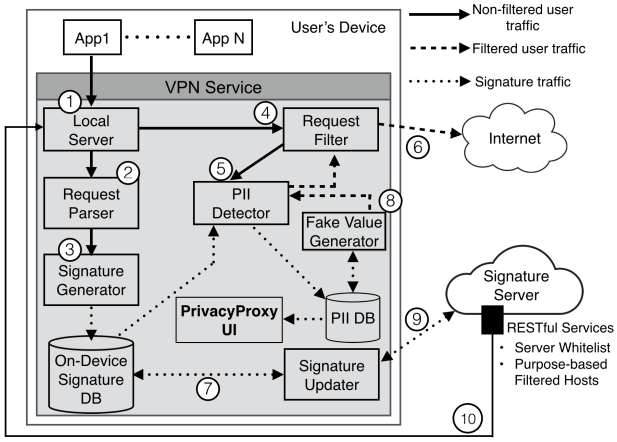

Figure 3: Different components of the PrivacyProxy Android app as well as various processing steps including (1) packet capture, (2) request parsing, (3) signature generation, (4) Request filtering/-marking, (5) PII detection, (6) forwarding to destination, (7) upload-/download signatures, (8) generating fake data for values marked for filtering, (9) getting public signatures, and (10) host-purpose mapping, server-whitelist from the Signature Server.

[37], as well as our implementation of PrivacyProxy, use Android's `VPNService` API (added in API Level 14), which is intended for 3rd party VPN apps. These VPN-based apps need to request the BIND_VPN_SERVICE permission. Note that the Android OS also shows a warning to users that an app may be monitoring their network traffic.

Our PrivacyProxy app works on devices running Android API 15 to API 24, and has been tested on devices running Android 4.3, 5.0, 5.1, 6.0, and 7.0 in the lab. We ported PrivacyGuard to Android Studio 2.2.2 and modified 3324 lines of code to implement new functionality provided by PrivacyProxy, specifically modifications to LocalServer and parsing HTTP requests. The rest of our app (processing logic, UI, XML, and other design elements) added 18,475 lines of code, for a total of 21,799 new lines of code. [YA: this is the code in the APP right? Might want to make it clearer.]

Figure 3 shows the overall flow of data. App requests are intercepted by the FakeVPN Service and sent to the LocalServer (Step 1). Next, our `Request Parser` (Step 2) attempts to parse each request using several known encodings (currently JSON, XML, and URL) and if not marks it as plain text. The output of this step is a set of key-value pairs. Next, our `Signature Generator` (Step 3) calculates private signatures for the `(package-name, app version, method, host, path)` tuples using count-min sketch, as explained in Section 3. These signatures are stored in a `Signature DB` on the device and also sent to the `PII Detector` on the device. The LocalServer in parallel also sends the app data to the `Request Filter` (Step 4). The `PII Detector` (Step 5) looks for PII by comparing private signatures on the device and the public signatures received from the PrivacyProxy server. These detected PII (key, values) are also stored locally on the device in a PII DB, along with generated fake values that they can be replaced with (Step 8). Detected PII are shown to the user in the PrivacyProxy UI, where they can label

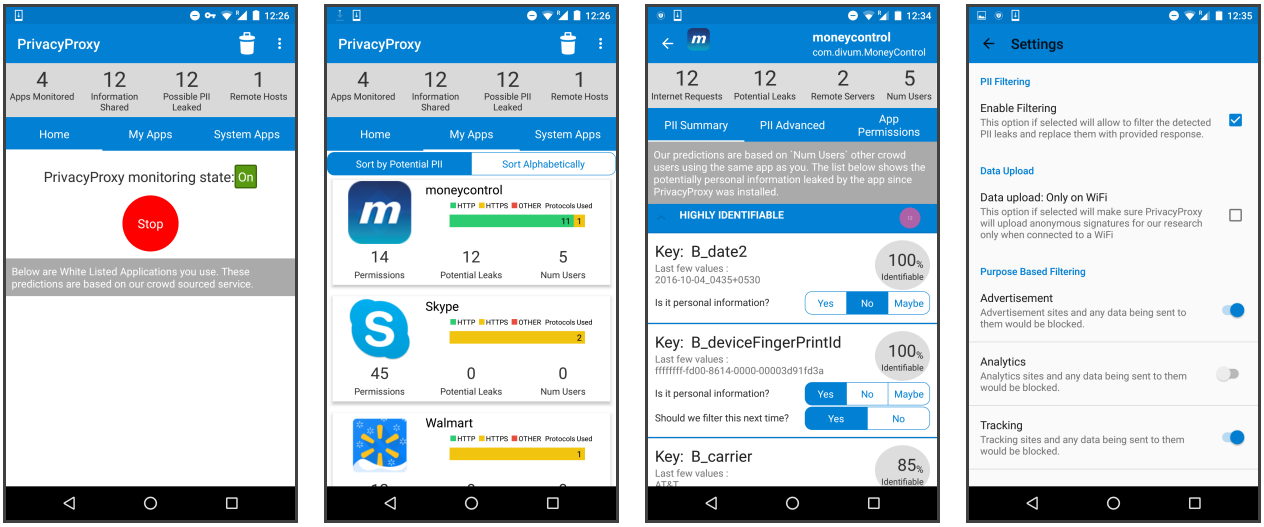

Figure 4: Screenshots of the PrivacyProxy app, from left to right: (a) Home screen showing summary information, (b) a list of monitored apps along with summary information for each app, (c) drill-down information for a specific app, showing key-value pairs that have been sent off the device, (d) Settings Screen for purpose-based filtering

which key-values are actually PII and can choose to filter out these values. Our app stores these decisions in the PII DB and also logs it in an on-device database. Based on the user's decision, either the original app request is unmodified, or PII in the request are modified by replacing original values with anonymized ones, and sent over the network (Step 6). Note, the private signatures stored on the on-device Signature DB are periodically uploaded by a separate `Signature Updater` to the Signature Server (Step 7). The signature updater uploads signatures opportunistically, preferring WiFi over cellular networks and also when the phone is plugged in. The signature server processes these private signatures and sends back public signatures. The local signature updater updates the public signatures from the server. This is done in a battery conserving manner since the PrivacyProxy signature database only requires weak consistency (Step 9). In addition, the PrivacyProxy Signature server has a RESTful interface to fetch the server side whitelists and purpose based filtered hosts (Step 10).

Figure 4 shows the UI for the PrivacyProxy app. The home screen (left) presents an overview of PrivacyProxy. Users can toggle the VPN functionality and see the total #apps monitored, #pieces of information sent to remote servers, #instances of PII detected, and #hosts that apps on their device have contacted. In the My Apps view (center-left), users can see summary info of all non-system apps that have used the network, including the number of potential PII detected and the number of people using the same app. Clicking on a particular app shows the detailed app view (center-right), which shows the associated key-value pairs that have been detected and the last few values for each key. Here, users can label each key-value pair as PII, selecting from "Yes", "No", and "Maybe". If the user chooses Yes, we provide the option to filter that key-value, replacing it with an anonymized value the next time we observe it (shown for the second key in the same screenshot). Finally (right), users can configure various settings, including purpose-based filtering, where data sent for certain purposes (such as to known ad providers, analytics, and tracking) can be automatically filtered across all apps.

Performance and Energy Optimization:

Optimizing battery usage when the device is idle/standby is important since most users' devices are in this state with periods of sporadic activity[15]. The PrivacyProxy app is on the critical path of all network traffic, which affects network performance and battery life. We implemented several optimizations to address these concerns. First, we use Android's `DozeMode`, introduced in API Level 24. While in DozeMode, background services sync and transfer data only during specific time windows. When entering DozeMode, we turn off PrivacyProxy's VPN and turn it back on during that window. Based on our empirical measurements, keeping the VPN running consumes resources even if it does does not monitor any outgoing traffic, and this optimization reduces this overhead.

Second, we use Android's VPN bypass functionality [10] to whitelist certain apps so their traffic does not pass through the VPN. We use this feature to skip apps that we don't look for PII within (e.g. email) or ones that we cannot since they have advanced security features designed to prevent MITM (e.g. Facebook and WhatsAPP use SSL certificate pinning).

Third, we have a **FastPath** option. The idea is to disable VPN interception and packet processing on a per-app basis based on their past request behavior. Apps are added to the FastPath locally on the device and have no connection with the Global Whitelists. Our assumption is that for a particular version of an app that is sending data to a particular host+path combination, if we do not detect a PII leak after a certain number of requests (randomly chosen to be between 500 and 2000, but configurable), we add that app to the FastPath and do not inspect requests from that app/version. Consequently, the apps on the FastPath are dependent on each

user's request and may vary from user to user. Note, however, if this same app then subsequently tries to contact a *new host and path* combination we remove it from the FastPath and start processing all its requests again. [SC: New:]Moreover, whenever PrivacyProxy is started, we randomly remove apps from FastPath and enable sampling their requests again for 200 more requests to make it harder for developers to evade the system. Note, as stated earlier, our assumption is that the developer is not malicious. [YA: this is a bit contradictory, since what we are doing is really for sybil, and those are by definition malicious entities.]

4.2 PrivacyProxy Signature Server

The `Signature Server` collects crowdsourced private signatures and calculates public signatures. More specifically, the Signature Server (SS) combines private signature uploads from clients and updates public signatures for associated signature IDs. These public signatures are stored in the SS database and periodically downloaded by PrivacyProxy clients. Count-min sketch makes combining signatures trivial. In addition, each count-min sketch in a signature can be updated independently, so we don't have to worry about users having slightly different signatures with the same SID. SS is implemented in Go and supports HTTPS. Our server code is just over 1000 lines of Go code, with another 350 lines for processing uploaded signatures. We use the Let's Encrypt SSL Certificates for HTTPS and MySQL as our database. To minimize storage and bandwidth, we use Google's Protocol Buffer library to efficiently serialize all data transfer. As mentioned earlier, our count-min sketch has r rows (currently 5) and n columns (currently 55) see Figure 2 (right). We also use 64 bit integers. This means that each key only takes up $r \times n \times 8 = 2200$ bytes regardless of the number of values.

To protect user privacy, we do not store the IP addresses or any other identifiers of PrivacyProxy clients sending us private signatures. We do ensure that a particular client cannot upload multiple copies of the same private signature by enforcing it in the PrivacyProxy app. In addition, we only allow clients to upload private signatures with at least 10 observations to ensure we get quality signatures.

[SC: Don't know the correct position of this paragraph. We already have a para talking about the global whitelist here but it fits nicely after sys arch.] **Using the crowd to mitigate individual user's burden:** [SC: New:] We leverage the feedback of the initial users to improve the user experience for the subsequent users. First, the global whitelists (filled by the initial users) would allow us to display whether filtering makes the apps crash or unusable. Second, we also leverage our crowd to maintain a global non-PII whitelist. In this scenario, if a large number of users (empirically set to 30) unanimously mark a certain key value pair in a particular app-version as non-PII, we store that particular key-value pair in the global not PII whitelist. New users of that particular app-version would not see non-PII value. As an added protection against Sybil attacks for this form of PII detection, some of the new users do randomly see the key-value pairs from the whitelist. If any one of the new user marks it as PII, the key-value pair is removed from the global whitelist. [SC: end]

Similar to the FastPath optimization earlier, we have also implemented a server-side `WhiteListing` feature that can provide hints to PrivacyProxy clients to reduce performance overhead. The intuition is that if there are enough users for a particular app/version (currently set to 5 users) and if we have not detected PII in all their requests, it's very likely that the app does not leak PII. For this feature, we aggregate data on the number of requests seen and the number of potential PII flagged by PrivacyProxy for a particular app/version. Note, we do not need the actual PII to be shared with our server but only the *number* of PII detected and flagged as TP/FP for each app/version. If we conclude that a particular app does not leak PII, we add it to our whitelist. PrivacyProxy client apps can download this whitelist and add them to the VPN bypass method (Step 10 Figure 3). The key advantage for new users is that they will not observe any performance overhead of PrivacyProxy processing data from these apps. It is possible for an app developer to game the system by not accessing any PII at first. This case can be handled by having a random and small subset of app requests being sampled, though we have not implemented this.

4.3 Handling Special PII Cases

We found in early testing that some forms of PII need special handling. First, location data is potential PII, although it depends on the geographical spread of our users. For example, if many people are in the same exact location using the same app, location data shared by the app is less privacy invasive and would be marked non PII by PrivacyProxy. We handle location by checking if it follows common formats for latitude and longitude and then comparing it to the last known location of the device, as accessed by our app. If the two are similar, we mark it as location PII. Users can then choose to filter it, replacing it with a location which has the latitude and longitude coordinates rounded off to 1 decimal place, thereby getting a location that is somewhat close by. This approach offers some privacy while also allowing location-based searches and maps to work.

David Choffness: 'For example, if many people are in the same exact location using the same app, location data shared by the app is less privacy invasive and would be marked non PII by PrivacyProxy.'That is a conclusion custom-fit to the authors'limitations. If a tracker knows exactly what geographic path I follow through a crowd of people (e.g., during rush hour), that is still very sensitive information most people would call PII.

[SC: The user's PII never leaves the PrivacyProxy app, it is only fetched locally.]

[KK: I think the counter-example provided is a bit unfair since no system I'm aware of can catch that case. Interestingly, suppose an app tracking your trail sends the trail as a list of GPS coordinates, then PrivacyProxy will be able to pick up that the list is unique. So I guess the power of PrivacyProxy depends a lot of the format of the data, which we've pointed out as a limitation.]

Second, timestamps also require special handling. Apps and libraries like Yelp, SpeedTest, and Google Analytics send timestamps for reasons such as caching prevention [2]. Since timestamps are often seen only once, we need to prevent PrivacyProxy from marking them as PII. If a given value is in a common timestamp format and if the timestamp is within an hour of the current time, then we do

not mark it as PII. [KK: Doesn't this 1 hour window just allow app install time to evade the system? As in when I first install an app and use it, the timestamp is going to leak.] This ensures that app install timestamps, which are sometimes used to track users, are still detected as PII. Note that timestamps only need special treatment when the user base is small and the timestamps are non-varying. Otherwise, our system should naturally treat them as non-PII over time since they will keep changing across many users.

[SC: New:]Third, apps can use SSL certificate pinning which prevents PrivacyProxy from monitoring or filtering their traffic. To maintain the usability of PrivacyProxy, we add a set of apps to the global whitelist. These apps are potentially sending data more securely and include the apps by vendors such as Google and Facebook. Furthermore, if an app that uses TLS/SSL does not load correctly on three tries we infer that it has enabled certificate pinning and add it to the whitelist dynamically. [SC: end]

5 EVALUATION

We evaluated PrivacyProxy in four different contexts: a lab study, a small field study, a user survey from the field study, and a comparison of PrivacyProxy with two other related systems - Recon [36] and Haystack [35]. Based on these evaluations, we highlight several key findings. First, we show that we can detect PII accurately with a precision (0.59) and recall (0.99) in our lab experiment and precision (0.79) and recall (1.0) for our field study. Second, we quantify the improvement in detection efficiency as the number of signatures (number of users using the same app) in our database increases. Finally, based on our user survey, we show that 85% of the users in the survey found no perceptible change in battery life and 68% users found no change in network performance.

5.1 Experimental Design

We evaluated PrivacyProxy in two scenarios, described below. For both scenarios, we identified PII using the definition in Section 3.2.3. We calculated Precision, Recall, and F1 scores by manually labeling values as potential PII and then comparing against results from PrivacyProxy. Note that an app might make many requests to the same host/path with the same key-value pair. If we analyze all these requests as separate events, then we will inflate the number of detected PII/non-PII key-value pairs in our experiments. To address this, we only examine unique (SID, key, value) tuples.

Scenario 1: Controlled Experiment: We installed the top 100 apps from the "Free in Android Apps" category on Google Play. For this experiment, the devices were allowed to upload the signatures after just 1 observation (as opposed to 10 in the usual deployment). We manually interacted with each app for 1 minute on an Android device with PrivacyProxy running in the background. Then, we repeated the same process on a different device to get more signatures. Afterward, we interacted with the same apps again on the first device and examined the results. Note that the two devices used were identical (unmodified Nexus 6P phones running Android 6.0.1), since if we used 2 different models, PrivacyProxy might erroneously treat the model number or OS version as PII. To verify our results, we manually inspected each unique key-value pair the proxy extracted.

Scenario 2: PrivacyProxy Field Deployment: In this scenario, we consider our signature database as that created by users who installed our app from the Play Store and from our user study. Nine users found and downloaded PrivacyProxy from Google Play (published in Nov 2016). We also recruited 18 participants from an online portal, asking them to use PrivacyProxy for at least 4 days. This was an IRB-approved User Study and participants were given a $20 Amazon Gift Card. Participants were asked to keep PrivacyProxy running for as long as possible and use their devices as they normally do. After 4 days, all participants were given the option to take a survey from the app.

We collected 1803 signatures from 191 apps. We took the top 100 apps based on most signatures and installed them on a test device. The top five apps were BBC News, IMDb, MoneyControl, Pinterest, and Skout. We used a test device to evaluate PrivacyProxy's PII detection accuracy, obviating the need for actual users to share their original key-value pairs. We manually launched these 100 apps and interacted with them for a minute on our test device. We then examined the key-value pairs marked by PrivacyProxy as PII. [SK: Was the manual interaction done randomly? It may be better to clarify this as it can possibly introduce bias in the experiments.]

We also collected telemetry information from users about the apps which were detected to leak PII. We got app package name, count of detected PII, count of user-marked PII and non-PII values, and the count of protocols (HTTP/ HTTPS/ Other) used by the app. This information enabled us to make decisions on whitelisting the apps on the server side.

5.2 Detection Efficiency

We manually labeled key-value pairs PrivacyProxy classified as PII. We considered values such as unique identifiers and location true positives, and the rest false positives. Note that this is an over approximation of false positives as some of these values might contain app-specific PII we were not aware of. We also inspected potential PII values that were below threshold and hence not classified as PII, labeling these as false negatives. We used a threshold of $T = 0.6$ based on preliminary lab tests. The results of for both scenarios are shown in Table 1. As described in Section 3.2.3, the PII in Table 1 are categorized into three confidence levels: High, Moderate and Low Likelihood.

Scenario 1: Our interactions in scenario 1 led to 989 signatures. We then used the second device to detect PII by interacting with the same apps again. We captured requests from 40 apps, which communicated with 39 remote servers. The other 60 apps did not connect to any remote servers. PrivacyProxy detected 140 potential PII leaks for 23 apps. We labeled the detected PII and calculated True Positive (82), False Positive (58) and False Negative (1) numbers (see row 4 in Table 1). Note that precision is highest for highly likely PII and decreases as our confidence decreases.

Interestingly, GO Weather Forecast and Widgets app (14 True Positives) generated two unique ids. One concatenates install time and Android ID (`14810493757127 cfc44f05470b61a`). The other is a device fingerprint (`deviceFingerPrintId:ffffffff-9333-3215-0000-000049eee8b9`). These examples suggest regex-based approaches can easily miss PII if we do not have a priori knowledge. Additionally, users may not have sufficient knowledge to label values

Scenario		Confidence Level	TP	FP	FN	Precision	Recall	F1 Score
Controlled Experiment (Scenario 1)	1	Highly Likely	65	26	1	0.71	0.98	0.83
	2	Moderately Likely	15	27	1	0.36	0.94	0.52
	3	Less Likely	2	15	1	0.12	0.67	0.20
	4	Overall	82	58	1	0.59	0.99	0.74
PrivacyProxy in wild and User Study (Scenario 2)	5	Highly Likely	51	11	0	0.82	0.90	0.89
	6	Moderately Likely	39	10	0	0.80	1.0	0.89
	7	Less Likely	2	3	0	0.40	1.0	0.57
	8	Overall	92	24	0	0.79	1.0	0.88

Table 1: Summary Table showing detection efficiency in controlled experiment and in wild and user study. The precision rates in our controlled experiment are relatively low although we see from the results of the Scenario 2 as the number users increase the precision improved by 34%.

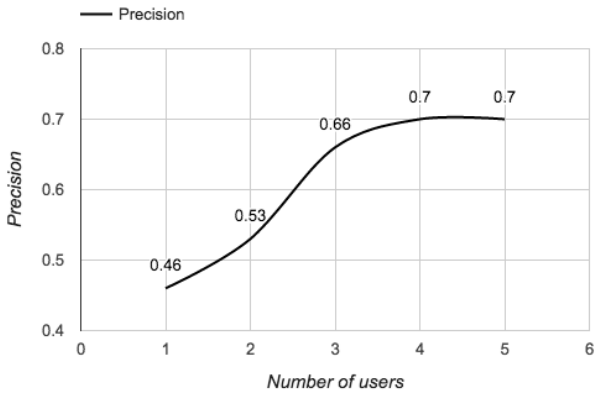

Figure 5: Summary of change in precision of PrivacyProxy with increasing number of users. The precision numbers included are for the set of 5 apps (Waplog, Pinterest, Moneycontrol, Smart Voice Recorder and Yelp) used in this experiment.

like these as PII. Lastly, the current permission-based controls in Android would not be able to control such PII leaks.

The only false negative was from Skout, which embedded multiple key-values inside of a single string value (`"userid":112498803, "datetime":1481051772274, "action":"Click"`). PrivacyProxy marked this value as "Context-Sensitive" since the timestamp makes the string different across different users. The developer put multiple items in a single string, so PrivacyProxy did not detect `userid`. This app was the only one we saw using this app encoding, where various key-value pairs are sent as a value inside another key.

Scenario 2: Our crowdsourced DB had 1803 signatures for 191 apps. In Scenario 2, we interacted with the top 100 apps with the most signatures, and captured requests for 75 apps, communicating with 30 remote hosts. There are two reasons for why we only had data for 75 apps. First, our use of apps for 1 minute did not always reach the threshold of 10 observations before uploading. Second, some apps require phone numbers to use, e.g. Marco Polo and Viber, and our device did not have this. PrivacyProxy detected 116 potential PII leaks in 24 apps. We found 92 True Positives and 24 False Positives, resulting in a precision of 0.79 (see row 8 in Table 1). Analyzing the false positives, we noticed the New Break app retrieves news items by sending encoded URLs (e.g. `url:05QR8Z1i01`). Different users read different news, so these key-value pairs were misclassified as PII. We found other interesting cases of PII leaks, e.g. "GO Weather Forecast & Widgets" sends IMEI to 't.appsflyer.com' (AppsFlyer), a mobile app tracking and attribution analytics library. The "Gaana" music app sends location to 'checkout.freecharge.in' (payments app). Both are cases where PII use does not align with the purpose of the app.

Crashlytics is a popular library for crash reporting and was detected by PrivacyProxy to send (Android ID, Google AdID, Unique Installation ID, OS Build version) PII. This library was included in 24 apps (e.g. Skout, Tinder, Venmo) out of 191 apps in our signature DB. Interestingly, Crashlytics provides a feature on their web portal for developers so that users get a notification before sending a crash report (which contains PII). However, during our manual interactions with these apps we did not receive any such notifications, indicating that none of the developers enabled this feature.

The number of users for the 100 apps in scenario 2 ranged from 1 to 10 users. 54 apps had only 1 user, but 3 apps had 6 users (Moneycontrol, Airdroid, News Break), 1 app had 8 users (System App: Android Backup Service), and 1 app had 10 users (System App: Google Services Framework). With the results from Table 1 we note that the precision improved significantly (34%) although there were only 46 apps with more than 1 user.

Our hypothesis was that detection accuracy would increase considerably as our user base increases. To test this, we designed another experiment where we simulated a progressive increase in the number of users. We started with 1 device and interacted with a set of five apps for a minute each (Waplog, Pinterest, Moneycontrol, Smart Voice Recorder and Yelp). These five apps were selected randomly from the top 20 Apps used in Scenario 2. We then repeated the process on 5 more devices (6 total). Two devices were identical while the others were different Android smartphones.

Figure 5 shows how PII detection (precision) changes as we add the number of devices using these 5 Apps ranging from 0.46 for 1 device, to 0.7 for 5 additional devices. Note that reported PII detection accuracy is achieved without users explicitly marking any values as PII. If we used that information, the precision numbers should be even better. This observation also applies to the accuracy (Precision/Recall) numbers presented in Table 1 for Scenario 1 and Scenario 2.

		Recon	HayStack	PrivacyProxy
Waplog com.waplog.social	True Positives	Username[2], Password, Tracking identifier (Android ID)	Android ID	Android ID, Latitude, Longitude, Google Advertising ID (Also sent as Advertising ID and IDFA), Crashlytics Installation ID, App/Library Specific IDs (Device Fingerprint ID, Hardware ID, Attribution ID, Identity ID, Generated ID, Placement ID)
	False Positives[1]	Gender	Time-Zone, Build Number	Timestamp, Screen Resolution, Build Number
Pinterest com.pinterest	True Positives	Tracking identifier (Android ID)	Android ID	Android ID, Pinterest Install ID
	False Positives[1]	-	Hardware Info, Build Fingerprint	Event Timestamp

Table 2: Comparison of PrivacyProxy, Recon, and HayStack in detecting unknown PII. [1]We categorize false positives based on the strict definition of PII we are using for this paper. For example, Gender and Time-Zone alone cannot identify a person, though can if combined with other values, as discussed in Section 6.2. [2]PrivacyProxy did not flag `username` because the Waplog app put the `Username` in the URL path rather than query parameter.

Type Of PII (Based on Request Key)	Apps with corresponding PII (Total Apps = 191)
Standard Identifiers (A) (IMEI, Advertisement ID(AdID), Android ID, Crashlytics Install ID)	57 29.8%
Location(B) (Latitude, Longitude, Zipcode)	21 11.0%
App Generated Unique Identifiers(C) (Device Fingerprint ID, Install ID, Identity ID, Generated ID, UID (User ID), Attribution ID, Hardware ID, Placement ID)	19 10.0%
All categories above (A+B+C)	70 36.7%

Table 3: Summary of our findings from Request key analysis. Analyzing the 1803 signatures for 191 apps from our crowdsourced signature database we found 31% of apps leak at least one of the PII types.

Comparison with Recon and HayStack: We also compared PrivacyProxy's PII detection with Haystack [35] and Recon [36]. We installed Lumen Privacy Monitor (HayStack) on December 5th 2016 and reported results from their app. We were unable to use Recon and so used data from Recon Reports [6]. We used the same device on the same network, and used HayStack and PrivacyProxy one after the other to report PII flagged by them for 10 apps from scenario 2. For brevity, we have included a few representative apps from our test setup in Table 2 and we include the total PII detected by those apps. Our goal was to evaluate whether there are certain PII that approaches based on regex matching (PrivacyGuard), user supplied labeling (ReCon), and deep packet inspection (Haystack) would miss as compared to PrivacyProxy or vice-versa. [SK: Where is the data of comparison with PrivacyGuard?] As noted in table 2, PrivacyProxy identified many more correct PIIs than either approach (True Positives), but also has some false positives. Considering true positives, PrivacyProxy found 31 PIIs while HayStack could find only 12 PIIs for these 10 apps.

[GS: New:]**Taxonomy of Keys containing PII:** Since our signatures are created using key-value pairs from the HTTP requests, it is possible to look for known PII (such as IMEI and Android ID) and identify all associated keys across different apps. For example, some of the keys associated with Android ID are `Android-ID`, `X-CRASHLYTICS-ANDROID-ID`, `DeviceID`. Analyzing the 1803 signatures for 191 apps from the signature server database, we discovered that 31% of apps leak at least one of the PII types (see Table 3). Since many apps use the same libraries, we can potentially improve PrivacyProxy's coverage for apps we have not seen before if they use keys known to be associated with PII leaks. While there are some challenges here, for example reliably identifying libraries based on names of keys, success here could help mitigate the cold start problem for apps not yet in our database.[GS: End]

5.3 Battery Consumption

In Section 4, we discussed the various optimizations we did to improve the energy overhead of running PrivacyProxy, which we measured to be 30% higher without our techniques. To quantify impact on battery life, we installed the top 20 apps from "Free in Android Apps" on our test devices. For the first test, we launched these apps (in the background) and then put the device in standby mode (screen off). In the second test, we used the MonkeyRunner script to open each of these 20 apps one by one, and interacted with them by performing 100 clicks on the screen (screen is on in this case). To measure overhead, we calculate the time it took for the battery to go from from 100% to 85% with and without PrivacyProxy enabled. Note, to mimic typical user settings, we enabled the WiFi radio, LocationServices, etc on the test device and ran each test thrice to report data. Table 4 presents the energy overhead results. Our data shows that for standby, PrivacyProxy reduces battery lifetime by 9.7% - 11.0%, while for the active use case the overhead can be up to 14.2%.

5.4 User Survey

We conducted an in-app user survey to measure perceived usability and performance overhead of PrivacyProxy. The survey was delivered after 3 days of using PrivacyProxy. All questions were on a 5-point Likert scale. For battery consumption, 85% (23/27) of survey participants said that they "Noticed no perceptible change". 67% (18/27) said the same about network performance. Although there is a performance overhead of using PrivacyProxy, 54% (14/26) of participants agreed that the performance overhead was worth the

Scenario	Samsung S7 Edge		Nexus 5x	
	StandBy	Monkey Runner	StandBy	Monkey Runner
Without PrivacyProxy	463 min	52 min	650 min	70 min
With PrivacyProxy	416 min 10.2%	46 min 11.5%	594 min 8.6%	61 min 12.8%
With PrivacyProxy and Filtering	412 min 11.0%	45 min 13.4%	587 min 9.7%	60 min 14.2%

Table 4: Battery Overhead Evaluation. The percentage depicts the reduction in battery life, which varied from 12.8% to 14.2% in our 2 test devices.

privacy improvement while 31% (8/26) were neutral. Overall, 59% (16/27) of users were able to use PrivacyProxy without noticing any perceptible change to apps.

6 DISCUSSION AND LIMITATIONS

6.1 Security

R3: I don't think if I agree with the proposed solution in 6.1. This solution can cause problems when the system is bootstrapping the database.

[KK: Seems like the best way to solve this is to include a device-identifying ID in the SID. However, this breaks the privacy guarantees of the server. We could consider blocking multiple updates from the same IP in a short period of time. That way, an attacker needs to amass a bunch of different devices to poison the database in a short time. This doesn't stop them from slowly poisoning it in the long run though.]

[SC: We can also use a one-way hash of the device id to uniquely identify the devices. We do have a unique one way hash 'survey id' to uniquely identify the survey from the users. Same thing can be done to identify malicious attackers in a privacy preserving way.]

[KK: I guess the one-way hash would also need a nonce generated on device so we will never be able to map a hash to a particular user.]

A potential security concern for PrivacyProxy is Sybil attacks where a malicious actor poisons our signature database. [GS: New:] [GS: Sybil attacks on recommender systems is a well studied field. There are suitable defenses available for different recommender systems depending on the context. DSybil [42] particularly suits our requirements for defense against sybil attacks. DSybil focus on scenarios where (1) the objects to be recommended are either good or bad. (2) the lifespan of the users is not overly short so that they can build trust; and (3) the users aim to find some good objects to consume instead of exhaustively finding all good objects. The three applicable scenarios of DSybil conforms with PrivacyProxy requirements of (1) to determine whether a uploaded signature is honest or not before merging to public signatures. (2) the honest users of PrivacyProxy are fairly long lasting, and uploading signatures periodically; and (3) to provide good detection PrivacyProxy doesn't require to consider all good signatures for an app.] DSybil provides strong provable guarantees that hold even under the worst-case attack and are optimal [42]. A quantitative analysis of implementing DSybil [SK: Check the last line, it is incomplete. I removed a bracket due to latex error]

One way to mitigate such an attack is to compare the frequency estimates from the uploaded signatures to those existing in the database. If the uploaded signature differs substantially, we can reduce their relative weight when combining signatures or hold off using them until we see similar signatures for the same SID. We could also use security techniques like Certificate Pinning (we use SSL already to send signatures/data) as well as app level encryption in our PrivacyProxy app, making it harder for an attacker to reverse engineer our communication protocols and poison our database.

6.2 Limitations

We currently do not parse the request URI path for PII, as it is not a structured key-value pair. However, we observed that some apps send username and email as part of the path. We have the path as part of our SID, and so it is possible to add values from the path to our analysis, though it is still an open question as to the best way of using this data.

There are also several privacy issues outside the scope of our current work, which we defer to future work. First, some values by themselves are not highly identifiable, but could be when combined with other values (e.g. see [38]). Second, managing user privacy can involve more than just preventing PII leaks. As reported in [7], a sex toy was sending intimate information about when it was used to the manufacturer. This usage information is not PII, but still a privacy issue. Generalizing the issue, sensor values might not be PII, but can still paint an intimate portrait of an individual. Third, our detection approach does not allow us to identify the exact type of the detected PII as we do not access various OS/Android identifiers on the device ourselves (e.g. MAC Address, UUID, phone etc). This is an intentional design choice in PrivacyProxy to follow principles of least privilege and privacy-awareness. Fourth, as mentioned in Section 4, we currently bypass the apps which have enabled certificate pinning. Fifth, PrivacyProxy may not be able to successfully detect PII if we fail to properly parse the request and extract the key-value pairs. In practice, we have not found many instances of this. [KK: What's the latest number on number of apps we can't parse? It would be nice to have this in the evaluation section too] Finally, we only monitor the network traffic of apps and not of mobile browsers. PII leaks can also happen via web browsers. However, since people visit different websites, we cannot collect enough requests per site to detect PII with our crowdsourced approach. [KK: I don't think this is a good statement. There are popular websites and unpopular apps. We should say that we can adapt PrivacyProxy to monitor web browser too by making the app ID the domain name.]

7 CONCLUSIONS AND FUTURE WORK

PrivacyProxy is a scalable system for detecting and mitigating information leaks from devices running stock Android OS. We utilize a unique signature structure to efficiently summarize all requests leaving the user's device without revealing actual content. In addition, we propose a novel crowdsourced PII detection algorithm that relies on user-generated signatures instead of heuristics. This

approach lets us detect both traditional forms of PII as well as app-specific identifiers. By running on the user's device, we can filter out PII before it leaves the device, and we have optimized our app to minimize network overhead and battery consumption.

We conducted four evaluations of PrivacyProxy, including a lab study, a small field trial, a survey of participants from the field trial, and a comparison against two other systems with similar goals. We found that precision increases as more users use PrivacyProxy. In addition we show that PrivacyProxy can find several types of PII that other approaches cannot. Furthermore, we show that perceived impact on battery life as well on the network performance is low as reported by users in our study.

REFERENCES

[1] Android Developers. Settings.Secure. Android ID. https://developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID.

[2] GoogleAnalytics.TrackingCodeOverview.. https://developers.google.com/analytics/resources/concepts/gaConceptsTrackingOverview#gifParameters. Accessed: 2016-11-29.

[3] Guide to Protecting the Confidentiality of Personally Identifiable Information (PII). https://doi.org/10.6028/NIST.SP.800-122. Accessed: 2016-11-29.

[4] PrivacyBlocker. http://privacytools.xeudoxus.com/. Accessed: 2016-11-30.

[5] PrivacyGrade: Grading The Privacy Of Smartphone Apps. http://privacygrade.org/. Accessed: 2016-11-29.

[6] Recon Reports. Information Leaked by Apps. https://recon.meddle.mobi/app-report.html. Accessed: 2016-11-29.

[7] This sex toy tells the manufacturer every time you use it. http://fusion.net/story/334603/sex-toy-we-vibe-privacy/. Accessed: 2016-11-30.

[8] Tor.. https://www.torproject.org. Accessed: 2016-11-30.

[9] Tor on Android.. https://www.torproject.org/docs/android.html.en. Accessed: 2016-11-29.

[10] VpnService.Builder.. https://developer.android.com/reference/android/net/VpnService.Builder.html. Accessed: 2016-11-30.

[11] XPrivacy - The ultimate, yet easy to use, privacy manager. https://github.com/M66B/XPrivacy. Accessed: 2016-11-29.

[12] Yuvraj Agarwal and Malcolm Hall. 2013. ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*. ACM, New York, NY, USA, 97–110. https://doi.org/10.1145/2462456.2464460

[13] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj Agarwal. 2015. Your Location Has Been Shared 5,398 Times!: A Field Study on Mobile App Privacy Nudging. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 787–796. https://doi.org/10.1145/2702123.2702210

[14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[15] Andrius Aucinas, Narseo Vallina-Rodriguez, Yan Grunenberger, Vijay Erramilli, Konstantina Papagiannaki, Jon Crowcroft, and David Wetherall. 2013. Staying Online While Mobile: The Hidden Costs. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. ACM, New York, NY, USA, 315–320. https://doi.org/10.1145/2535372.2535408

[16] Tuomas Aura, Janne Lindqvist, Michael Roe, and Anish Mohammed. 2008. Chattering Laptops. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies (PETS '08)*. Springer-Verlag, Berlin, Heidelberg, 167–186. https://doi.org/10.1007/978-3-540-70630-4_11

[17] Rebecca Balebako, Jaeyeon Jung, Wei Lu, Lorrie Faith Cranor, and Carolyn Nguyen. 2013. "Little Brothers Watching You": Raising Awareness of Data Leaks on Smartphones. In *Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS '13)*. ACM, New York, NY, USA, Article 12, 11 pages. https://doi.org/10.1145/2501604.2501616

[18] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. 2011. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile '11)*. ACM, New York, NY, USA, 49–54. https://doi.org/10.1145/2184489.2184500

[19] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58 – 75. https://doi.org/10.1016/j.jalgor.2003.12.001

[20] Anupam Das, Nikita Borisov, and Matthew Caesar. 2016. Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*.

[21] Peter Eckersley. 2010. How unique is your web browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.

[22] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[23] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. 2010. A First Look at Traffic on Smartphones. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*. ACM, New York, NY, USA, 281–287. https://doi.org/10.1145/1879141.1879176

[24] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST'12)*. Springer-Verlag, Berlin, Heidelberg, 291–307. https://doi.org/10.1007/978-3-642-30921-2_17

[25] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe.. In *NDSS*. Citeseer.

[26] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 279–288. https://doi.org/10.1145/1455770.1455806

[27] Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. 2013. Privacy As Part of the App Decision-making Process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3393–3402. https://doi.org/10.1145/2470654.2466466

[28] Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I. Hong. 2014. Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings. In *Symposium On Usable Privacy and Security (SOUPS 2014)*. USENIX Association, Menlo Park, CA, 199–212. https://www.usenix.org/conference/soups2014/proceedings/presentation/lin

[29] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, SA Zhang, Norman Sadeh, Alessandro Acquisti, and Yuvraj Agarwal. 2016. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Symposium on Usable Privacy and Security*.

[30] Yabing Liu, Han Hee Song, Ignacio Bermudez, Alan Mislove, Mario Baldi, and Alok Tongaonkar. 2015. Identifying Personal Information in Internet Traffic. In *Proceedings of the 2015 ACM on Conference on Online Social Networks (COSN '15)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/2817946.2817947

[31] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Michelle L. Mazurek, and Jeffrey S. Foster. 2017. User Interactions and Permission Use on Android. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 362–373. https://doi.org/10.1145/3025453.3025706

[32] Suman Nath. 2015. MAdScope: Characterizing Mobile In-App Targeted Ads. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*. ACM, New York, NY, USA, 59–73. https://doi.org/10.1145/2742647.2742653

[33] David Naylor, Matthew K. Mukerjee, and Peter Steenkiste. 2014. Balancing Accountability and Privacy in the Network. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 75–86. https://doi.org/10.1145/2740070.2626306

[34] Katarzyna Olejnik, Italo Ivan Dacosta Petrocelli, Joana Catarina Soares Machado, KÃľvin Huguenin, Mohammad Emtiyaz Khan, and Jean-Pierre Hubaux. 2017. SmarPer: Context-Aware and Automatic Runtime-Permissions for Mobile Devices. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[35] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. 2015. Haystack: In Situ Mobile Traffic Analysis in User Space. *CoRR* abs/1510.01419 (2015). http://arxiv.org/abs/1510.01419

[36] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, New York, NY, USA, 361–374. https://doi.org/10.1145/2906388.2906392

[37] Yihang Song and Urs Hengartner. 2015. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '15)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2808117.2808120

[38] Latanya Sweeney. 2000. Simple demographics often identify people uniquely. *Health (San Francisco)* 671 (2000), 1–34.

[39] Max Van Kleek, Ilaria Liccardi, Reuben Binns, Jun Zhao, Daniel J. Weitzner, and Nigel Shadbolt. 2017. Better the Devil You Know: Exposing the Data Sharing Practices of Smartphone Apps. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 5208–5220. https://doi.org/10.1145/3025453.3025556

[40] Haoyu Wang, Jason Hong, and Yao Guo. 2015. Using Text Mining to Infer the Purpose of Permission Use in Mobile Apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 1107–1118. https://doi.org/10.1145/2750858.2805833

[41] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. 2017. The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences. In *38th IEEE Symposium on Security and Privacy*.

[42] Haifeng Yu, Chenwei Shi, Michael Kaminsky, Phillip B Gibbons, and Feng Xiao. 2009. Dsybil: Optimal sybil-resistance for recommendation systems. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 283–298.

[43] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. 2015. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY '15)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2699026.2699105