# Introducing Privacy in Screen Event Frequency Analysis for Android Apps

Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev
Ohio State University, Columbus, Ohio, USA
Email: {zhang.4858,latif.28,bassily.1,rountev.1}@osu.edu

***Abstract*—Mobile apps often use analytics infrastructures provided by companies such as Google and Facebook to gather extensive fine-grained data about app performance and user behaviors. It is important to understand and enforce suitable trade-offs between the benefits of such data gathering (for app developers) and the corresponding privacy loss (for app users).**

**Our work focuses on screen event frequency analysis, which is one of the most popular forms of data gathering in mobile app analytics. We propose a privacy-preserving version of such analysis using differential privacy (DP), a popular principled approach for creating privacy-preserving analyses. We describe how DP can be introduced in screen event frequency analysis for mobile apps, and demonstrate an instance of this approach for Android apps and the Google Analytics framework. Our work develops the automated app code analysis, code rewriting, and run-time processing needed to deploy the proposed DP solution. Experimental evaluation demonstrates that high accuracy and practical cost can be achieved by the developed privacy-preserving screen event frequency analysis.**

## I. Introduction

Mobile apps often use analytics infrastructures provided by companies such as Google and Facebook [1]: e.g., Google Analytics [2], Google Firebase [3], and Facebook Analytics [4]. The extensive fine-grained data gathered from such analytics can be used for analysis of app performance and user behaviors such as clicks, screen transitions, purchase history, and location data. The libraries implementing the analytics infrastructure are independent modules of apps installed on users' devices. They silently collect information in the background, usually without users' knowledge. The use of such tracking is widespread, as indicated by recent studies [5].

For an app developer, the benefits of such analytics could be substantial. Information obtained from the detailed stream of app-generated events could be used for targeted advertising, behavioral analytics, location tracking, and app improvements [1]. Unfortunately, such benefits come at the expense of reduced privacy for app users. In the broader societal context there are increasingly-strong expectations for better understanding and control of the privacy-related effects of data gathering. In this context, there is very little understanding of such effects in the area of mobile app analytics.

Techniques for privacy-preserving data analysis can be employed to study and evaluate the privacy implications of data gathering from mobile apps. Privacy-preserving analysis techniques are designed with theoretical guarantees about the loss of privacy and the accuracy of analysis results. *Differential privacy* (DP) [6] has been proposed and studied as a broad principled approach for designing privacy-preserving data analyses. Based on a rigorous definition of privacy, this approach has been investigated extensively by researchers and has recently been adopted by industry—for example, in the Chrome browser [7] and in iOS-10 [8].

While there is a large body of work on differential privacy, the practical applications of these techniques in the context of the widely-used analytics frameworks for mobile apps have not been studied. Introducing DP mechanisms in apps that use such frameworks has clear benefits to users. The app developers also benefit from such mechanisms: the gathered information provides analytics value while at the same time the app creators can claim, with confidence, that users are provably protected against leaks of their sensitive data. Such claims by developers make their product more attractive to users. In addition, privacy-by-design protections may be of interest to regulatory bodies.

**Challenges.** Despite significant advances in DP theory, applying DP solutions to analytics frameworks for mobile apps faces two major obstacles. First, the providers of analytics frameworks—companies such as Google, Facebook, and Yahoo—do not supply DP capabilities and are unlikely to provide them in the near future. Any DP solution deployed by an app developer today—e.g., a developer who is using the Google Analytics framework—should work without any changes to the framework implementation and APIs, both locally on the user's device and remotely at Google's servers. This "black box" view is a significant departure from the standard assumption in DP research, where the researchers have complete control over the entire analytics infrastructure and can deploy various sophisticated DP protocols.

The second challenge is to introduce DP in a given app with little effort from the app developer. Ideally, the developer would write their app without any DP considerations, and an automated code rewriting step would introduce DP-enforcing code. Such separation of concerns is highly valuable for software development, testing, debugging, and evolution.

**Our proposal.** Our work focuses on these two challenges for *screen view event frequency analysis*, which is one of the most popular forms of data gathering in mobile app analytics. We propose a privacy-preserving version of such analysis using differential privacy, and instantiate this design for Android apps that use the Google Analytics framework. To the best of our knowledge, this is the first work that attempts to introduce

2470-6892/19/$31.00 ©2019 IEEE
DOI 10.1109/SCAM.2019.00037

DP in the behavioral analysis of Android apps.

Our approach treats the analytics infrastructure as a black box and uses its own event frequency pre/post-processing to achieve DP. This means that the approach can be deployed in the current ecosystem of mobile app analytics frameworks. The event frequency processing, which requires both static code rewriting and run-time data manipulation, is achieved transparently with the help of a code rewriting tool and a run-time support layer. This allows a developer to focus on the business logic of the app without directly creating or manipulating DP-related code.

**Contributions.** The work makes the following contributions:

- We define a privacy-preserving event frequency analysis for apps that use mobile app analytics frameworks. We describe how randomized noise can be added to the collected data in a way that provides well-defined privacy protections for an app user while still reporting high-accuracy analytics measurements to an app developer.
- We develop static code rewriting and run-time support needed to instantiate this analysis for the popular Google Analytics mobile app analytics infrastructure. We demonstrate techniques that are easy to incorporate in the app development process and enable low-effort introduction of DP features in an existing app.
- We describe an evaluation that demonstrates the feasibility and performance of the proposed techniques.

Mobile apps are used pervasively. They have access to a wide range of data that can be used to make inferences about app user features and behaviors. Such data is being collected on a massive scale. This provides strong motivation to study privacy-preserving techniques for mobile app analytics. Our current work, as well as future efforts based on it, could help establish well-founded approaches to address this challenge.

## II. Background: Analytics for Android

There are several analytics infrastructures that offer full-stack solutions to an app developer. Google Analytics (GA) [2] and its successor Firebase [3] are among the most popular ones. They allow a developer to collect and conduct various analyses against users' data. Facebook [4], Yahoo [9], and several others also provide similar services. Even though their policies [10]–[13] require developers to avoid recording (or at least to anonymize) user-identifiable information, users may still be left vulnerable to data breaches and surveillance by infrastructure providers and app developers [14]–[16].

A recent study of thousands of popular Android apps [5] has identified that Google Analytics was used by 42% of the analyzed apps. This popularity motivates our focus on privacy-preserving analysis for GA; however, the theoretical machinery and program analyses are general and should be applicable to other analytics libraries. One of the key features of GA is to provide the *frequency of screen view events*. Such frequency information helps a developer understand how users interact with her app, by quantifying user engagement and behaviors. A *screen view event* is defined by the developer to track a user's visit to a specific screen within an app. Conceptually, it is identical to a pageview for web analytics. GA, as well as other similar analytics libraries, allows a developer to insert API calls to track such events and to send information about them to backend servers (which themselves are maintained by Google, Facebook, etc.) for further analysis.

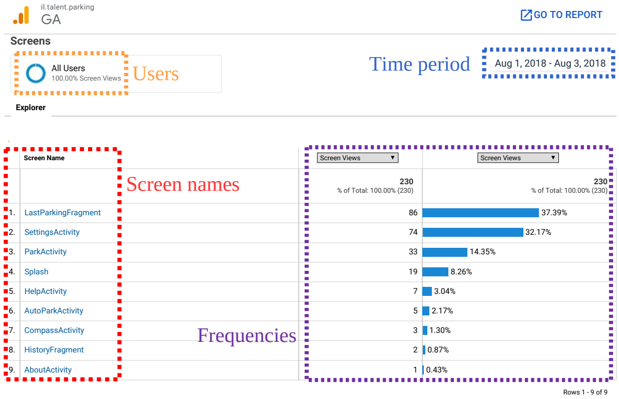
```
class ParKingApplication extends Application {
  Tracker t;
  Tracker a() {
    if (t == null) {
      GoogleAnalytics i =
                  GoogleAnalytics.getInstance(this);
      t = i.newTracker(R.xml.global_tracker); }
    return t; } }

class AutoParkActivity extends Activity {
  Tracker t;
  void onCreate(...) {
    t = ((ParKingApplication)getApplication()).a(); }
  void onResume() {
    t.setScreenName("AutoParkActivity");
    ScreenViewBuilder b = new ScreenViewBuilder();
    t.send(b.build()); } }
```

(a) Decompiled code

```
<resources>
  <string name="ga_trackingId">UA-65112504-3</string>
</resources>
```

(b) `global_tracker.xml`



il.talent.parking

GA

GO TO REPORT

Screens

All Users
100.00% Screen Views

Aug 1, 2018 - Aug 3, 2018

Explorer

| | Screen Name | Screen Views ▼ | Screen Views ▼ |
|---|---|---|---|
| | | 230<br>% of Total: 100.00% (230) | 230<br>% of Total: 100.00% (230) |
| 1. | LastParkingFragment | 86 | 37.39% |
| 2. | SettingsActivity | 74 | 32.17% |
| 3. | ParkActivity | 33 | 14.35% |
| 4. | Splash | 19 | 8.26% |
| 5. | HelpActivity | 7 | 3.04% |
| 6. | AutoParkActivity | 5 | 2.17% |
| 7. | CompassActivity | 3 | 1.30% |
| 8. | HistoryFragment | 2 | 0.87% |
| 9. | AboutActivity | 1 | 0.43% |

Rows 1 - 9 of 9

(c) GA report

Fig. 1. Example derived from `ParKing`.

### A. Running Example

GA is offered by Google to app developers to collect detailed statistics about user information and behaviors including session duration, user-triggered events, etc. For example, GA can track when the user has navigated to a particular screen in the app, or has performed an action such as sharing content with someone from her contact list. As another example, e-commerce data could be gathered, including product clicks, viewing product details, adding a product to a shopping cart, transactions, and refunds. The GA framework is general and the type of data being collected is up to the app developer.

To illustrate GA's capability of screen view event tracking, we use the example in Fig. 1. The `ParKing` app [17] navigates users to parking places, records history of parking locations, and reminds users about parking time. It has over 100K installs via Google Play. Fig. 1a shows a snippet of decompiled code from the app. Class `ParKingApplication` maintains a global context and its instance is retrieved via `getApplication` (line 12). Classes `GoogleAnalytics` and `Tracker` are helper classes to create and send data to Google's remote server. Method `ParKingApplication.a` creates a `Tracker` singleton at line 7 by calling `GoogleAnalytics.newTracker` with the resource ID of an XML file containing a Google-provided tracking ID, which is shown in Fig. 1b. All data recorded for this ID can be accessed by the developer of `ParKing`. We utilize this API in experiments to redirect all tracking data in closed-source apps to our own GA account.

Activities are the core components in Android apps. An activity displays a window containing GUI widgets. Class `AutoParkActivity` shows a screen with settings for automatic parking detection. The `onCreate` callback is called when an activity is created. It retrieves and stores the `Tracker` to field `t` at line 12. When an activity comes to the foreground, its `onResume` callback is invoked. Lines 14–16 create a screen view event and send it to the GA servers. A screen represents some displayed content. Each screen has a unique string name that is used as an identifier. A call to `Tracker.setScreenName` at line 14 records a name for the current screen as "AutoParkActivity". GA uses the builder pattern for creation of events. `ScreenViewBuilder` at line 15 is the helper class to build the event. `ScreenViewBuilder.build` returns a map containing data for the event together with other GA-internal data. The call to `Tracker.send` at line 16 uploads the event to Google's backend servers.

Besides activities, developers can specify other GUI components as screens, e.g., fragments. We inspected the app code and determined that there are 11 screens: "AboutActivity", "AutoParkActivity", "CompassActivity", "HelpActivity", "HistoryFragment", "LastParkingFragment", "ParkActivity", "SettingsActivity", "Splash", "TransparentActivity", and "ZoneEditorActivity". One of the challenges is to identify all possible screen names automatically. Technical details about our solution will be provided in later sections.

Google aggregates data from many app users and provides the result to the app developer. Fig. 1c is a sample report of screen view events in `ParKing` from GA's website. We have replaced the app's tracking ID with ours so that all data from our app runs is sent to our account. GA supports generation of reports for specific types of users (the orange box) within a certain time period (the blue box). The report contains a histogram of all screens, including their names and frequencies, as shown in the red and purple dotted boxes.

### B. User Privacy

GA and similar analytics frameworks provide some rudimentary privacy protection. For example, an app developer can instruct GA to remove the last octet of the IP address being recorded. As another example, Google's guidelines for app developers are to avoid collecting personally-identifiable information such as names, etc. However, to the best of our knowledge, there is no systematic enforcement of such protection mechanisms. Furthermore, even seemingly-innocent information that has been anonymized can lead to real privacy leaks when combined with additional information from other sources [14], [15], [18]. For example, existing analytics frameworks assign a pseudonym to each user. Developers can use the pseudonym to make requests to other services, such as Google Ads and Tag Manager. This mechanism allows developers to track seemingly-independent actions and devices for user identification [16], [19]. Screen view tracking reveals a user's access to certain software components that reflects her interests and patterns using the app. This, together with the user identification threat, could be utilized, for instance, for targeted advertisement. We propose a principled solution to protect users' screen view behaviors while still allowing developers to obtain meaningful results without modifying the underlying analytics infrastructures.

## III. Background: Differential Privacy

### A. Overview

The goal of differential privacy is to protect against a broad class of privacy attacks. Consider a situation where certain data is released intentionally by a party that has that data, in order to serve some data analytics goals. The data is legitimately released to government/business/research entities, but it is assumed that an adversary will also gain access to it. The adversary attempts to learn private individual information from the released data. Examples of such attempts are re-identification of persons, linking of records from different sources, and differencing attacks (e.g., comparing statistics before and after an event of interest). Here "adversary" is used broadly: for example, data released by the user to some company under certain terms may later be obtained by another company (e.g., as part of a corporate takeover) and used in a manner that was not anticipated by the user. As another example, data shared with some organization could be subpoenaed in legal proceedings despite the user's expectations.

Differential privacy (DP) [6], [20], [21] is a powerful method to defend, with formal guarantees, against privacy attacks. An adversary observing the DP analysis output will essentially make the same inference about any individual's private information, regardless of whether or not that information is part of the analysis input. Its strong privacy guarantees make DP an attractive alternative to other privacy-preserving techniques, as elaborated elsewhere [20]. DP solutions have been studied extensively by researchers and been deployed in practice (e.g., by Google [7], Apple [8], Uber [22], and the U.S. Census Bureau [23]). The rest of this section will focus on DP aspects that are relevant to mobile app analytics.

In the *centralized DP model*, individuals' private data is provided to a trusted data curator, where DP analysis is performed and its results are released to untrusted analysis clients. In the *local DP (LDP) model*, the curator is not trusted; in fact, the curator itself could be an adversary. For such LDP problems, each user performs local data perturbation before releasing any information to the curator (and, by extension, to any adversaries). The LDP model is a natural match for mobile app analytics. The app user releases data to the curator (e.g., Google, if the app uses GA). The curator analyzes the data and provides the results to the analysis client—that is, to the app developer. Regardless of the actions of the curator or the app developer, the design of the LDP analysis provides formal privacy guarantees to the app user, as described shortly.

### *B. LDP Frequency Estimates*

We illustrate LDP analysis using a fundamental problem in data analytics: constructing estimates of *data item frequencies*. This problem is closely related to many other analytics problems, including counting (e.g., number of individuals whose data satisfies some predicate), histograms (e.g., counts of data in disjoint categories), heavy hitters (e.g., most frequently occurring items), distribution estimation, regression, clustering, and classification.

Consider a set of $n$ app users. Each user $i \in \{1, \ldots, n\}$ has a single data item $v_i$ from some data dictionary $\mathcal{D}$ that is pre-defined by the app developer based on her analytics needs. For any $v \in \mathcal{D}$, its frequency is $f(v) = |\{i \in \{1, \ldots, n\} : v_i = v\}|$. The app developer's goal is to obtain the histogram defined by $f(v)$ for all $v \in \mathcal{D}$. An LDP solution to this problem will apply a *local randomizer* $R : \mathcal{D} \rightarrow \mathcal{Z}$ to each user's item $v_i$. The resulting $z_i = R(v_i)$ is sent to the LDP curator. Following convention, we will refer to this curator as "server", which is also a terminology well-suited for existing mobile app analytics infrastructures such as Google Analytics/Firebase. The server collects all $z_i$ and uses them to compute a frequency estimate $\hat{f}(v)$ for the real frequency $f(v)$ of each $v \in \mathcal{D}$.

The design of the local randomizer $R$ is parameterized by a *privacy loss parameter* $\epsilon$. Intuitively, this parameter characterizes how much knowledge an adversary could gain by observing the randomizer's output $R(v_i)$. An $\epsilon$-local randomizer $R$ satisfies the following property:

$$\forall v, v' \in \mathcal{D}, z \in \mathcal{Z} : \Pr[R(v) = z] \leq e^{\epsilon} \Pr[R(v') = z]$$

Here $\Pr[a]$ denotes the probability of $a$. The randomization achieves privacy by allowing plausible deniability: if a user has item $v$ and as a result $z = R(v)$ is reported to the server, the observation of that $z$ (by the server or by an adversary) does not reveal "too much" information about which item was actually held by the user. This is because the probability $\Pr[R(v') = z]$ that the item was some other $v'$ is close (by a factor of $e^{\epsilon}$) to the probability $\Pr[R(v) = z]$ that the item was the actual $v$. The privacy loss parameter is used to limit and quantify what can be learned about a user as a result of her data item being included in the analysis. Larger values of $\epsilon$ result in less privacy but higher accuracy.

Clearly, the introduction of random noise affects the accuracy of the frequency estimates reported by an LDP analysis. This accuracy is typically measured based on the $\ell_\infty$ norm of the difference between the vector of actual frequencies and the vector of estimates [21], that is, $\max_{v \in \mathcal{D}} |\hat{f}(v) - f(v)|$.

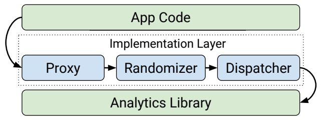


Fig. 2. Design overview.

## IV. Overview of LDP Screen Event Frequency Analysis

Screen view events, introduced in Section II, track each user's views of particular screens in an app. They are one of the most fundamental data items being collected in mobile app analytics. They feature prominently in all popular analytics frameworks for Android apps: Google Analytics, Google Firebase, Facebook Analytics, and Yahoo Flurry. These platforms allow developers to conduct various analyses on such data. For instance, funnel analysis visualizes users' engagement for each screen and identifies screens that many users are not converting through. The analysis results allow potential optimizations in the design and business logic of an app.

In addition to gathering statistics to characterize the entire community of app users, such data is also used for analysis of each individual's behavior patterns—for example, to determine her interests for targeted advertisement, or even for fingerprinting for user identification [16]. Each user is assigned an (anonymized) identifier at her first use of the app and any following data collection will contain this identifier. This enables developers/analysts to link a user's behavior data, e.g., viewing some screens that contain sensitive content, with existing data and models (possibly from third-party sources) to connect seemingly independent actions by the user [19].

Despite the potential for such user identification/tracking, as well as other related privacy concerns [24]–[27], current analytics frameworks for mobile apps do not provide protection for users' screen view data. We propose to address this problem by introducing LDP into the data collection, with transparent pre- and post-processing steps. Existing analytics infrastructures for mobile apps do not have LDP features and there is no prior work on performing LDP analytics in them. Achieving LDP without any changes to the underlying analytics infrastructure is a challenge. We tackle this challenge using the design outlined in Fig. 2.

On top of the standard analytics libraries running on the user's device, we introduce a layer that implements local randomization. To make the discussion more concrete, we describe the components of this layer for an instance of the approach specific to Google Analytics. Similar structure and behavior would be applicable to other analytics frameworks.

Relevant GA API calls (e.g., `Tracker.send` in Fig. 1) can be automatically redirected to corresponding proxy APIs (e.g., `Proxy.send`). This redirection can be achieved with an automated code rewriting tool. As described later, our concrete implementation uses the Soot code analysis/rewriting framework [28], but many other choices are also available. The proxy API methods coordinate the remaining components. The randomizer component, which is described in the next section, applies the local randomization and sends the relevant events to the dispatcher component. The dispatcher maintains a queue of pending events and periodically makes GA API calls to deliver them to the GA layer. The dispatcher also makes unsent events persistent when the app is closed, to allow sending when the app is reopened. Our implementation utilizes `JobService` in Android for scheduling and SQLite for storing events. One GA-specific implementation detail is that the GA layer drops events if they arrive too frequently. The dispatcher ensures that events are forwarded to the GA layer with sufficient delays between them. GA also records event timestamps. Since the dispatcher delays the dispatching of events, perturbation of timestamps is achieved in an *ad hoc* manner. In the future it would be interesting to consider more principled approaches for randomization of event timestamps.

The implementation layer is built at the time the app is created, and distributed as part of the app code when the app is published in an app market. The code in this layer should be open-source and created by a trusted party (e.g., privacy researchers). App-specific configuration parameters (e.g., the privacy loss parameter $\epsilon$) are defined by the app developer at app assembly time. On the user's device, there are no changes to the standard analytics API implementations that process events and send them to the server. The analytics server is completely unaware of the fact that there is any DP aspect to the data collection. After the server reports its results, they are post-processed by the app developer (as described below) to obtain the actual estimates of the desired analytics data. Such a "black box" solution is easy to deploy today.

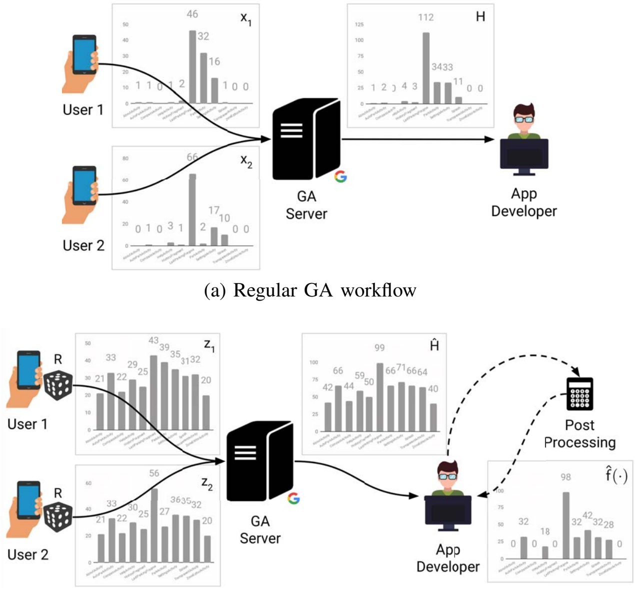
## V. Design and Implementation of LDP Screen Event Frequency Analysis

This section elaborates on the details of the approach outlined in the previous section. Recall that each screen in the app has a string identifier and the app uses GA API calls to send screen view events to the server (lines 14–16 in Fig. 1a). The GA event frequency reports, similar to the one shown in Fig. 1c, can be created using local randomization and then post-processed to account for the effects of this randomization. Next, we describe the building blocks of this solution.

### A. Obtaining LDP Frequency Estimates

The problem we define is a generalization of the frequency problem from Section III-B. While in that problem each user reports a single data item, here each user reports a sequence of items. Each app user has an integer id $i \in \{1, \ldots, n\}$. Such ids are used for the conceptual definition and are not needed in a practical implementation. The set of string identifiers for screens defines a data dictionary $\mathcal{D}$. For example, as described in Section II-A, `ParKing` has 11 strings in its dictionary. For ease of notation, assume that $\mathcal{D} = \{1, \ldots, d\}$. As with user ids, these event ids are used only for explanation purposes.

Data collection is performed for all $n$ users over some period of time. During this period, user $i$ generates a sequence of screen view events $v_i^1, v_i^2, \ldots, v_i^k$ where $v_i^j \in \{1, \ldots, d\}$. To simplify the discussion, assume that each user generates the same number of events $k$. If this is not the case, conceptual "padding" events with no effect can be introduced. The event sequence $v_i^1, v_i^2, \ldots, v_i^k$ can be thought of as a histogram $x_i \in \mathbb{N}^d$—that is, a $d$-dimensional vector of frequencies, where $x_i[v] \in \mathbb{N}$ is the number of occurrences of $v$ in the sequence.



(a) Regular GA workflow

(b) LDP workflow using baseline randomizer $R$ with $\epsilon = \ln(9)$

Fig. 3. Regular analysis vs LDP analysis, with two app users.

In a non-LDP setting, all events $v_i^j$ for all users $i$ are sent to the GA server, which aggregates their counts and produces a histogram $H = \sum_i x_i$, where the summation is element-wise for vectors $x_i$. For any $v \in \mathcal{D}$, its frequency is $H(v)$. The resulting histogram is similar to the one shown in Fig. 1c.

**Example.** Fig. 3a illustrates this scenario with two app users. The data shown in the example was obtained by running the Monkey GUI testing tool [29] on the `ParKing` app. Monkey randomly triggers GUI events that result in GA API calls inside the app code. Two different Monkey runs were used to obtain the data representing the two app users. Each user produces 100 events. In the figure, the corresponding histograms are denoted by $x_1$ and $x_2$. Note that GA does not send the histograms to the server, but rather the individual events that were observed. The server adds up the event counts from the two users and reports the resulting histogram $H$.

The next subsection describes the design of the local randomizer for this problem. As with existing solutions for the single-item-per-user problem from Section III-B (e.g., [7], [30]), our solution for the generalized problem presented in this section is based on the technique of *randomized response* described below. While it may be possible to define more advanced solutions with better theoretical accuracy and cost (e.g., generalization of [31], [32]), they would require significant "white box" re-design of the analytics infrastructure and would not be easily deployed. Instead, we define an approach that keeps the existing analytics infrastructure intact.

Randomized response is a classic technique that has been used in social sciences to gather sensitive data (e.g., about illegal or embarrassing behaviors). For our analysis problem, randomized response would add noise to each observed event $v_i^j$ by probabilistically (1) ”dropping” event $v_i^j$ and/or (2) spuriously reporting other events when $v_i^j$ is observed.

Formally, an $\epsilon$-LDP solution applies an $\epsilon$-local randomizer $R : \mathbb{N}^d \rightarrow \mathbb{N}^d$. The resulting $z_i = R(x_i)$ is sent to the server. The server, which is LDP-unaware, computes a histogram $\hat{H} = \sum_i z_i$ and reports it to the app developer. The app developer performs post-processing of $\hat{H}$ to compute, for each $v$, an estimate $\hat{f}(v)$ for the actual frequency $f(v) = H(v)$ that would have been observed without differential privacy.

**Example.** Fig. 3b illustrates the steps of the LDP solution. Each user $i$, independently of any other users or the server, applies $R$ to her real event sequence to generate a randomized event sequence. (The definition of $R$ will be presented shortly.) All resulting events are sent to the GA server. In the figure, $z_i$ denotes the histogram for this new sequence. The data was obtained by applying our technique to the real GA events in `ParKing` that were used for $x_i$ in Fig. 3a. Value $\epsilon = \ln(9)$ is used in prior work [7] and reused here.

Comparing $x_i$ with $z_i$, it is clear that significant noise was added by $R$ to each user's data. The server adds up the counts of reported events and provides the resulting histogram $\hat{H} = z_1 + z_2$ to the app developer. The developer performs post-processing of $\hat{H}$ (described later) to obtain the $\hat{f}(\cdot)$ frequency estimates. The accuracy of these estimates—that is, their differences from the real histogram $H$ in Fig. 3a—depends on the number $n$ of app users and on the value of $\epsilon$. In this particular example, the number of users is very small ($n = 2$) and the estimates are rather inaccurate. Section VI presents an extensive experimental evaluation of the accuracy that can be achieved in realistic scenarios.

### *B. Design of the Local Randomizer*

One standard choice for the desired properties of $R$ is based on the notion of *user-level privacy* [21]. In our context, the definition is as follows. Consider event sequences $v_1, v_2, \ldots, v_k$ and $v'_1, v'_2, \ldots, v'_k$ and their corresponding histograms $x \in \mathbb{N}^d$ and $x' \in \mathbb{N}^d$. An $\epsilon$-local randomizer satisfies

$$\forall z \in \mathbb{N}^d, \forall x, x' \in \mathbb{N}^d : \Pr[R(x) = z] \leq e^{\epsilon} \Pr[R(x') = z]$$

In essence, the observation of $z$ by an adversary provides very little information about the user's real event sequence, as (with high probability) any other event sequence could have been the user's raw data before randomization.

*1) Baseline Local Randomizer:* We first define a baseline randomizer $R$. Refinements of this solution are described later.

For any $v \in \mathcal{D}$, let $r(v)$ be a subset of $\mathcal{D}$ defined as follows:

- $v$ is included in $r(v)$ with probability $(e^{\frac{\epsilon}{2}})/(1 + e^{\frac{\epsilon}{2}})$
- for any $v' \neq v$, $v'$ is included in $r(v)$ with probability $1/(1 + e^{\frac{\epsilon}{2}})$

Given a sequence of events $v_i^1, v_i^2, \ldots, v_i^k$ which corresponds to a histogram $x_i \in \mathbb{N}^d$, consider the multi-set that is the union of all $r(v_i^j)$. The histogram $z_i \in \mathbb{N}^d$ for this multi-set is $R(x_i)$. Fig. 3 shows examples of $x_i$ and $z_i$.

```
for (String name : D) {
  if (name.equals(observed)) {
    if (rand() <= THIS_PROBABILITY) {
      // probability: exp(epsilon/2)/(1+exp(epsilon/2))
      send(name); }
  } else if (rand() <= OTHER_PROBABILITY) {
    // probability: 1/(1+exp(epsilon/2))
    send(name); } }
```

Fig. 4. Pseudo-code for event randomization.

This approach can be implemented on demand: every time we observe an event $v_i^j$ at run time, $r(v_i^j)$ is computed and the resulting events are sent to the GA server. The pseudo-code for this processing is shown in Fig. 4. The call to `rand` returns a new pseudo-random number $\in [0, 1)$. This processing is executed each time an event is observed. The cumulative effect over the entire event sequence is equivalent to applying a histogram randomizer $R : \mathbb{N}^d \rightarrow \mathbb{N}^d$. However, instead of computing histogram $x_i$ first and then applying $R$, we randomize each event as soon as we see it, which achieves the same effect as directly computing $z_i = R(x_i)$.

To obtain the frequency estimate $\hat{f}(v)$ for each event $v$, the app developer adjusts the server-reported count $\hat{H}(v)$ to ”undo” the effects of the added noise:

$$\hat{f}(v) = \frac{(1 + e^{\frac{\epsilon}{2}})\hat{H}(v) - nk}{e^{\frac{\epsilon}{2}} - 1} \qquad (1)$$

Here $n$ is the number of users and $k$ is the number of real events per user. This equation can be derived by considering the expected value of $\hat{H}(v)$, which is $f(v)(e^{\frac{\epsilon}{2}}/(1 + e^{\frac{\epsilon}{2}})) + (nk - f(v))\,(1/(1+e^{\frac{\epsilon}{2}}))$ since there are $f(v)$ occurrences of $v$ and $nk - f(v)$ occurrences of $v' \neq v$ across all $n$ users. Based on this, the expected value of $\hat{f}(v)$ is the real frequency $f(v)$. Thus, $\hat{f}(v)$ is an unbiased estimator of $f(v)$.

**Example.** Consider the $v$ for which $\hat{H}(v) = 71$ in Fig. 3b. We have $\epsilon = \ln(9)$ and $e^{\frac{\epsilon}{2}} = 3$. After post-processing, $\hat{f}(v)$ becomes $(4 \times 71 - 2 \times 100)/(3 - 1) = 42$, which matches the $\hat{f}(\cdot)$ value shown in Fig. 3b. If the estimate becomes negative, the reported frequency is 0. For example, the first bar for $\hat{H}$ in Fig. 3b has the value of 42. Since $(4 \times 42 - 2 \times 100)$ is negative, the first bar for $\hat{f}(\cdot)$ has zero height.

*2) Achieving Trade-Offs via Sampling:* For user-level privacy, the randomizer $R$ defined above is a $k\epsilon$-local randomizer: the level of privacy protection is worsened by a factor of $k$, where $k$ is the length of the user's event sequence. Intuitively, instead of “hiding” a single event, the randomization now has to hide $k$ events. Given the practical consideration that $k$ would often be large (e.g., hundreds of events), achieving useful user-level privacy presents a significant challenge due to the large value of $k\epsilon$ and the resulting weakened privacy guarantees.

Another challenge is the potential overhead of this approach. For each event $v_i^j$ observed at run time, all events in $r(v_i^j)$ are sent to the analytics server. The expected size of $r(v_i^j)$ is $(d - 1 + e^{\frac{\epsilon}{2}})/(1 + e^{\frac{\epsilon}{2}})$. Thus, the overhead depends on the size of $\mathcal{D}$. For illustration, consider $d = 11$ as in the running example, and $e^{\frac{\epsilon}{2}} = 3$. In this case, for each real event there will be 3.25 events on average reported to the server.

To address these challenges, we employ *sampling* [32], [33]. Each user assigns herself independently and randomly to one of several subsets of $\{1, \ldots, k\}$. Each such subset is of size $t$, where $t$ is a small constant independent of the number of users $n$ and the event sequence length $k$. Instead of considering all of its $k$ real events, the user only considers the $t$ real events whose indices are in the user's subset. These $t$ events are randomized and the results are reported to the server. We develop a simple implementation of this approach which does not require any synchronization with the server or other users. For any user $i$, when the analytics infrastructure is initialized, $t$ independent random values from $\{1, \ldots, k\}$ are drawn (without repetition) and recorded. For any event $v_i^j$, index $j$ is checked against this set of $t$ values. The event is ignored if $j$ is not in this set.

Using this sampling achieves two important goals. First, the privacy guarantees for user-level privacy are significantly improved: instead of having a $k\epsilon$-local randomizer, we now have a $t\epsilon$-local randomizer where $t$ is a small constant. Further, the overhead of extra events is reduced: instead of incurring this overhead $k$ times, we incur it $t$ times. Although sampling could reduce accuracy, our experiments indicate that for real Android apps it is possible to achieve practical accuracy when there is a sufficiently large number of app users.

### *C. Implementation via Code Analysis and Rewriting*

The approach described above was implemented with the help of several components, following the design from Fig. 2. We built a proxy for GA APIs, as well as a code rewriting tool that takes as input an app's APK and automatically replaces GA API calls with calls to the corresponding proxy APIs. This rewritten code, together with our implementation layer, is then packed back into a new APK which can later be installed on a user's device. This approach has the advantage that the app developer does not have to write any DP-related code.

More specifically, the proxy is initialized immediately after the call to `GoogleAnalytics.getInstance`. This initialization includes the creation of the randomizer and the dispatcher, as well as loading of the embedded dictionary $\mathcal{D}$ discussed shortly. We insert code to record each GA tracker created by `GoogleAnalytics.newTracker` for later use in the dispatcher. In our experiments, we replaced it with our own tracking ID so that all traffic was redirected to our GA account. For each call to `Tracker.send`, we replace it with a call to the `Proxy.send` method, in which the screen view event is randomized and sent to the dispatcher. The dispatcher stores every perturbed event into a local SQLite database. Android's `JobService` is then used to periodically fetch unsent events from the database and invoke `Tracker.send` with the recorded tracker and sufficient delays to deliver the event to Google's backend server.

The randomizer is parameterized by $\epsilon$, $t$, and $k$. Only the first $k$ events from a user are considered; this increases privacy by limiting the amount of information released to the server. Randomization also depends on the dictionary $\mathcal{D}$. Our implementation takes as input a dictionary per tracker in XML format and encodes it in the library during code rewriting, together with the parameters described above.

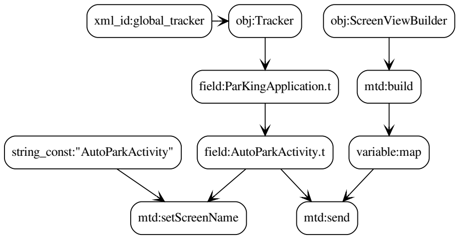


Fig. 5. Simplified constraint graph for the running example.

The dictionary $\mathcal{D}$ is defined by the app developer based on the desired app analytics. To reduce the developer's effort for constructing this dictionary, we built a static code analysis to track the flow of string constants. The output of this analysis can be used by the developer as a starting point for defining the dictionary. Of course, in the general case no static analysis can determine all run-time strings that could be used as screen names. Nevertheless, our experience with several close-sourced apps (described in the next section) demonstrated that our analysis significantly reduced the effort to define $\mathcal{D}$.

The analysis algorithm considers relevant API calls (e.g., `setScreenName` at line 14 in Fig. 1) to determine the screen names that flow into calls to `send` (e.g., at line 16). For each `GoogleAnalytics.newTracker` call site, it creates an artificial object representing the corresponding `Tracker` instance. It then propagates references to these objects, as well as references to string constants, to `setScreenName` calls. This information is used to determine the possible screen names associated with each `Tracker`. There are two types of strings that are considered: string constants in the code and strings defined in XML resources. The propagation is done via a value-flow analysis similar to control-flow-insensitive, calling-context-insensitive, field-based points-to analysis [34].

The static analysis also creates an object for each `new ScreenViewBulder` site. These objects are propagated to `build` calls (e.g., line 16 in Fig. 1). The resulting objects, which are maps storing information about the GA event to be sent, are then propagated to `send` calls. The `Tracker` objects are also propagated to the `send` calls. Note that the example in Fig. 1 shows straightforward propagation of such object references, but the analysis can handle the general case where the propagation is done through a sequence of assignments, parameter passing, and method returns. Since the analysis is field-based [34], it handles aliasing for object references. The resulting analysis solution determines which tracker is responsible for sending which screen view event, as well as what screen names are attached to each event. We record these associations during the static value-flow propagation. This information is then added in the app's APK as part of the implementation layer.

Fig. 5 presents a simplified constraint graph of the running example. Nodes in the graph correspond to objects, methods, variables, fields, and resources such as strings and IDs. Edges correspond to the flow of values. During the analysis, string constant "AutoParkActivity" is propagated to `setScreenName` (node "mtd:setScreenName"), the `ScreenViewBuilder` object (node "obj:ScreenViewBuilder") is propagated to `send` (node "mtd:send"), and the tracker object (node "obj:Tracker") is propagated to the two methods. Using this propagation, we can determine that the tracker with tracking ID stored in `global_tracker.xml` is used to send screen view events with name "AutoParkActivity". This analysis identifies the 11 screen names in the `ParKing` app described in Section II-A.

One technical detail is that the app developer may have provided an incomplete dictionary and as a result the randomizer occasionally may observe run-time events that are not in $\mathcal{D}$. It is easy to recover on-the-fly when an unknown event $v$ is observed on a user's device. First, this event is added to set $\mathcal{D}$ locally. Next, all real events for this user that were already processed and randomized are revisited. For each such past event, $v$ is reported with probability $1/(1+e^{\frac{\epsilon}{2}})$. In effect, this is equivalent to the processing that would have been performed in the past had the dictionary been $\mathcal{D} \cup \{v\}$ at that time.

## VI. Experimental Evaluation

Our implementation of LDP analysis for GA screen view events was based on app analysis and rewriting with the Soot analysis/rewriting framework [28]. The experiments were performed on two machines with Xeon E5 2.2GHz and 64GB RAM. To generate GUI events that simulate user actions (and internally trigger GA events) we utilized the Monkey tool for GUI testing [29]. All experimental subjects, source code for the static analysis and code writing, scripts to manage Monkey runs, and intermediate results are available at http://web.cse.ohio-state.edu/presto/software.

### A. Study Subjects

We analyzed a corpus of the most popular apps in each category in the Google Play store, and identified apps that include GA API calls. The static analysis was used to construct an potential dictionary $\mathcal{D}$ of string identifiers for screens in each app. Since we did not have the app knowledge that the developers had when they added GA screen view tracking, and furthermore the apps were close-sourced, we performed run-time validation and manual code inspection of the decompiled code to ensure that we used the correct dictionary in our experiments. The run-time validation used several independent runs of the Monkey tool to trigger different GUI behaviors (and thus different sequences of GA screen view events). For our experiments, we selected 15 subjects that covered a representative range of values for $|\mathcal{D}|$ and for which Monkey achieved high run-time coverage of the dictionary.

TABLE I
Study subjects.

| App | #Classes | #Stmts | $\lvert\mathcal{D}\rvert$ | Time (s) |
|---:|---:|---:|---:|---:|
| `SpeedLogic` | 119 | 5881 | 10 | 0.51 + 13.3 |
| `ParKing` | 543 | 37478 | 11 | 1.84 + 17.6 |
| `DPM` | 10859 | 939666 | 12 | 80.2 + 67.0 |
| `Barometer` | 668 | 52252 | 13 | 1.91 + 32.5 |
| `LocTracker` | 269 | 24575 | 14 | 1.10 + 21.6 |
| `Vidanta` | 2652 | 162705 | 15 | 12.4 + 36.0 |
| `MoonPhases` | 295 | 44522 | 16 | 1.24 + 18.2 |
| `DailyBible` | 3297 | 332708 | 17 | 11.4 + 57.5 |
| `DrumPads` | 1449 | 126951 | 17 | 5.73 + 60.5 |
| `QuickNews` | 3297 | 332708 | 17 | 12.3 + 57.3 |
| `Posts` | 3297 | 332708 | 17 | 13.3 + 56.8 |
| `Mitula` | 1522 | 120347 | 20 | 5.73 + 29.3 |
| `KFOR` | 3708 | 284581 | 29 | 13.9 + 40.2 |
| `Equibase` | 1697 | 127290 | 35 | 7.85 + 25.3 |
| `Parrot` | 1869 | 120470 | 64 | 6.53 + 25.9 |

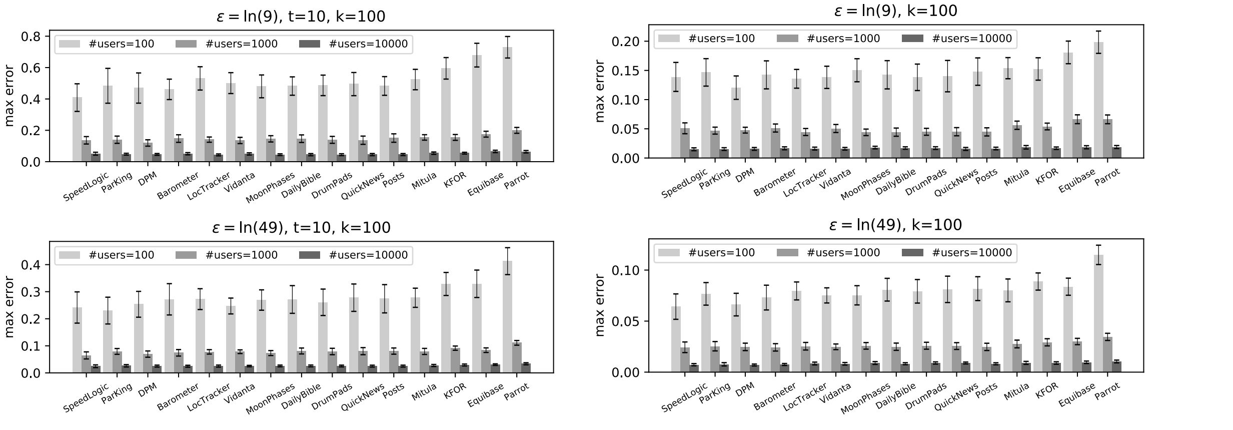
Table I describes the characteristics of these subjects. The names of the studied apps are listed in column "App". Columns "#Classes" and "#Stmts" show the numbers of classes and statements in Soot's Jimple IR, excluding some well-known third-party libraries such as `com.google`, `org.joda` and `org.apache`. The size of dictionary for each subject is shown in column "$|\mathcal{D}|$". As described earlier, we chose apps that cover a representative range of dictionary sizes, in order to study their effect on overhead. Column "Time (s)" shows the running time of the static analysis (the first number) and code rewriting (the number after the "+"). The average cost of the static analysis is 5.78 seconds per 100K Jimple statements. The cost of code rewriting mainly depends on the size of the app and Soot's performance on code transformation and is 22.4 seconds on average across all apps.

### B. Accuracy

For our evaluation, we extended the implementation with testing/profiling infrastructure that allows experimentation with various analysis parameters. That same infrastructure could be used by app developers to understand the characteristics of their LDP app analytics and to fine-tune the parameters of the data collection to achieve the desired trade-offs.

We utilized Monkey [29] to simulate user interactions, by issuing GUI events every 200 ms until $k = 100$ GA screen view events were triggered by those GUI events. For each app, we installed and ran it on 100 emulators in parallel to simulate $n = 100$ users' interactions with that app. Each of the 100 runs used a different seed for Monkey's generation of random GUI events, thus triggering a different sequence of screen view events. Since we had to repeat this process several times per app (e.g., to study the effects of sampling and the choice of $\epsilon$), in each repetition of the 100 Monkey runs we used the same set of 100 seeds for Monkey. Note that this randomization for Monkey is unrelated to the randomization used by the local randomizer $R$ to compute the probabilities for including/excluding events. To account for the variability in the behavior of $R$ under the same Monkey run and under the same choice of all other parameters (e.g., $\epsilon$), every reported metric was measured 20 times, in 20 independent repetitions of the same experiment. The variations among the 20 repetitions were entirely due to $R$. For the metrics, we collected the mean over the 20 repetitions, as well as the $95\%$ confidence interval.

The execution of many thousands of Monkey runs, even with emulators running in parallel, is prohibitively expensive



(a) With sampling

(b) Without sampling

Fig. 6. Accuracy.

for evaluations with a large number of users $n$. To facilitate the simulation of thousands of users, we conducted offline runs based on the traces of real run-time events (i.e., the input to the randomizer $R$) gathered during the Monkey runs for $n = 100$. There were 100 such traces for each app. We ran $R$ offline on all such traces 10 times, all with different seeds for $R$'s internal randomizations, to generate 1000 estimated histograms for each app. This simulates the scenario when an app is used by $n = 1000$ users. The same process was repeated for $n = 10000$. We calculated and reported the accuracy and overhead based on these histograms.

Fig. 6 show the accuracy with and without sampling, with different values of $\epsilon$ and number of users $n$. The values of $\epsilon$ match those used in prior work [7]. As discussed at the end of Section III, accuracy is measured based on the largest difference between the estimated and actual frequencies of all elements in the dictionary, i.e., $\max_{v\in\mathcal{D}} |\hat{f}(v) - f(v)|$. Here we normalize this value by dividing it by the total number of actual screen views $f = \sum_{v\in\mathcal{D}} f(v)$. The $y$-axes show the mean of 20 separate repetitions, with 95% confidence interval. The $x$-axes show the names of apps sorted by $|\mathcal{D}|$.

**More users result in better accuracy.** The nature of differential privacy requires sufficiently large number of users to be useful. With small values of $n$, high accuracy cannot be achieved in our setting (and, in all likelihood, in similar settings). Fig. 6 shows the dramatic improvement in accuracy due to the increase in the number of users, for all possible choices of other parameters. The practical implications are that thousands of users should be included in the data gathering in order to obtain accurate results. While the accuracy also depends on other parameters (e.g., $\epsilon$), the number of users is a primary factor that should be considered carefully. Fortunately, this is not a significant limitation for practical use: most non-trivial apps have non-trivial numbers of users. For example, for 14 out of the 15 apps used in our study, the number of installs according to Google Play is over 100K, with several apps having more than a million installs.

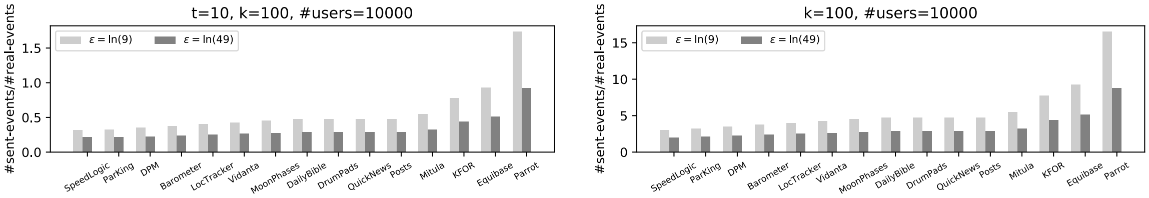
**Larger $\epsilon$ provides better accuracy.** The privacy loss parameter $\epsilon$ controls how much can be learned from a user's data. Larger values of $\epsilon$ result in higher privacy loss and better accuracy. The top chart in Fig. 6a is based on a value of $\epsilon = \ln(9)$ (i.e., $e^{\frac{\epsilon}{2}} = 3$). In other words, when an event is observed, it is reported to the server with probability $3/4$ and each other element of $\mathcal{D}$ is reported with probability $1/4$. The bottom chart in the figure uses a larger value of $\epsilon$ and as a result the max error is reduced: for example, for `ParKing` with $n = 100$, this reduction is from 0.48 to 0.23. The same trend also holds without sampling, as shown in Fig. 6b.

**Sampling does not hurt accuracy.** Sampling achieves both lower overhead and increased user-level privacy. However, a natural concern is whether sampling will reduce accuracy. The trend exhibited by our results is that high accuracy can be achieved with large number of users. For the analysis parameters used in our evaluation, the estimates are reasonably accurate when $n = 10000$: in Fig. 6a, the max error is around $0.05$ for $\epsilon = \ln(9)$ and $0.02$ for $\epsilon = \ln(49)$. Practically, this means that for any event, the estimated relative frequency (as percentage of the total number of events) is a few percentage points off its real value.

### C. Overhead

Fig. 7 shows the number of events sent to the GA server, relative to the number of real events triggered by calls to `send` in the app code. In other words, we consider how many events, on average, are produced by the local randomizer $R$ for each real event. We only report the measurements for 10000 users as the conclusions for other values of $n$ are similar. The $y$-axes show the mean of 20 separate repetitions of each experiment.

**Sampling reduces overhead.** Recall that for each actual event, we expect $(d - 1 + e^{\frac{\epsilon}{2}})/(1 + e^{\frac{\epsilon}{2}})$ events to be sent to the server; here $d$ is the size of the dictionary. The results in Fig. 7b meet this expectation. For example, for `ParKing` we have $d = 11$ and about 3 events were sent per actual event when $\epsilon = \ln(9)$. Sampling can reduce this cost by



(a) With sampling

(b) Without sampling

Fig. 7. Overhead.

a magnitude of $k/t$, as we are sending only $t$ events per $k$ actual events. Fig. 7a shows that, by using sampling, the overhead of additional events introduced by the randomizer $R$ can be controlled well. Together with the accuracy results from Fig. 6a, these experiments indicate that practical trade-offs between accuracy, overhead, and privacy can be achieved when the number of users $n$ is reasonably large.

### *D. Threats to Validity*

The small number of subject apps poses a threat to external validity, which we have attempted to ameliorate by selecting apps with representative dictionary sizes. Apps with login mechanisms are omitted due to low screen coverage, which is another threat to external validity. The static analysis and the run-time processing could suffer from incorrect implementation. This internal validity threat was partially addressed via extensive testing. Another issue is that, although Monkey is a standard tool for large-scale random testing, the "users" in the experiments are simulated and they may not be representative of real-world behaviors. Despite these limitations, we believe that this study presents promising initial evidence that black-box LDP mobile app analytics is feasible.

## VII. Related Work

There is growing attention to privacy in various fields of software engineering [35], e.g., in testing [36]–[39] and defect prediction [40]–[42]. Budi et al. [37] propose $k$-anonymity-based generation of new test cases while preserving their original behaviors. Their following work [39] extends the approach to evolving programs. MORPH [40] preserves data privacy of software defects in a cross-company scenario, by perturbing instance values. CLIFF+MORPH [41] removes dominant attributes for each class before perturbation. Li et al. [42] adopt a sparse representation obfuscation for defect prediction, while preserving privacy of data from multiple sources. Although our overall goal is similar, we aim to protect data gathered by analytics frameworks from mobile apps using differential privacy techniques.

Several examples of prior work on differential privacy were already discussed. There also exist several practical realizations of LDP for data analytics. Google's RAPPOR combines randomized responses and Bloom filters to identify popular URLs in the Chrome browser without revealing users' browsing habits [7]. Apple applies DP for gathering analytics data for emoji and quick type suggestions, search hints, and health-related usage [8]. Samsung proposed the Harmony LDP system to collect data from smart devices for both basic statistics and complex machine learning tasks [43]. Microsoft uses LDP to collect telemetry data over time across millions of devices [44]. We are not aware of any efforts to apply these techniques to analytics for Android apps. One challenge is that, unlike this prior work, we need to assume that the analytics infrastructure is LDP-unaware.

The problem considered in our work is similar in spirit to software analytics [45] which aims to help developers learn from software data such as app store data [46]–[50], code repositories [51]–[56] and bug/security reports [57]–[60]. Many companies utilize error/crash reporting systems to collect various categories of execution information from their users [61]. Lu et al. [62] and Liu et al. [63] leverage an Android-native application management app with over 250M users for app usage pattern mining. Böhmer et al. [64] conduct analysis on usage logs of thousands of users for three popular apps. PMP [65] is deployed to collect users' data protection decisions to help make privacy recommendations for over 90K users. GAMMA [66] continuously gathers and analyzes execution information from a large number of users through lightweight instrumentation. Liblit et al. [67] gather execution data from a large distributed community of users running a program remotely. Their approach samples the data and sends it to a central database for later isolation of bugs.

## VIII. Summary

We demonstrate that LDP features can be added to existing screen event analytics in Android apps without changes to the underlying analytics infrastructure. The proposed approach increases user privacy, requires little effort from app developers, and does not sacrifice analytics accuracy. There are many other interesting software analytics problems for mobile apps. Rather than simple frequency counts, more powerful LDP analyses could be performed [32], [33], [68]–[70]. Future work should consider such analyses as well as the necessary tool support for app developers, including related app code analyses and rewriting, to enable deployment of LDP solutions with ease and confidence.

*Acknowledgments.* We thank the reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1907715.

# References

[1] Yale Privacy Lab, "App trackers for Android," https://privacylab.yale.edu/trackers.html, Nov. 2017.

[2] Google, "Google Analytics," https://analytics.google.com, Jun. 2019.

[3] ——, "Firebase," https://firebase.google.com, Jun. 2019.

[4] Facebook, "Facebook Analytics," https://analytics.facebook.com, Jun. 2019.

[5] Exodus Privacy, "Most frequent app trackers for Android," https://reports.exodus-privacy.eu.org/reports/stats/, Jun. 2019.

[6] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *TCC*, 2006, pp. 265–284.

[7] Ú. Erlingsson, V. Pihur, and A. Korolova, "RAPPOR: Randomized aggregatable privacy-preserving ordinal response," in *CCS*, 2014, pp. 1054–1067.

[8] Apple, "Learning with privacy at scale," https://machinelearning.apple.com/2017/12/06/learning-with-privacy-at-scale.html, Dec. 2017.

[9] Oath, "Flurry," http://flurry.com, Jun. 2019.

[10] Google, "Measurement protocol/SDK/user ID policy," https://developers.google.com/analytics/devguides/collection/android/v4/policy, Jun. 2019.

[11] ——, "Google Analytics for Firebase use policy," https://firebase.google.com/policies/analytics, Jun. 2019.

[12] Facebook, "Facebook platform policy," https://developers.facebook.com/policy, Jun. 2019.

[13] Oath, "Flurry analytics terms of service," https://developer.yahoo.com/flurry/legal-privacy/terms-service/flurry-analytics-terms-service.html, Aug. 2018.

[14] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *S&P*, 2008, pp. 111–125.

[15] ——, "De-anonymizing social networks," in *S&P*, 2009, pp. 173–187.

[16] A. Goldfarb and C. E. Tucker, "Privacy regulation and online advertising," *Management Science*, vol. 57, no. 1, pp. 57–71, 2011.

[17] TalentApps, "ParKing: Where is my car? Find my car - Automatic," https://play.google.com/store/apps/details?id=il.talent.parking.

[18] I. Dinur and K. Nissim, "Revealing information while preserving privacy," in *PODS*, 2003, pp. 202–210.

[19] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *NSDI*, 2012, pp. 12–12.

[20] A. Wood, M. Altman, A. Bembenek, M. Bun, M. Gaboardi, J. Honaker, K. Nissim, D. R. O'Brien, T. Steinke, and S. Vadhan, "Differential privacy: A primer for a non-technical audience," *Vanderbilt Journal of Entertainment and Technology Law*, vol. 21, no. 1, pp. 209–276, 2019.

[21] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3-4, pp. 211–407, 2014.

[22] Uber, "Uber releases open source project for differential privacy," https://medium.com/uber-security-privacy/differential-privacy-open-source-7892c82c42b6, Jul. 2017.

[23] A. Dajan, A. Lauger, P. Singer, D. Kifer, J. Reiter, A. Machanavajjhala, S. Garfinkel, S. Dahl, M. Graham, V. Karwa, H. Kim, P. Leclerc, I. Schmutte, W. Sexton, L. Vilhuber, and J. Abowd, "The modernization of statistical disclosure limitation at the U.S. Census Bureau," https://www2.census.gov/cac/sac/meetings/2017-09/statistical-disclosure-limitation.pdf, Sep. 2017.

[24] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," in *NDSS*, 2018, pp. 1–15.

[25] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee, "The price of free: Privacy leakage in personalized mobile in-apps ads." in *NDSS*, 2016.

[26] H. Feng, K. Fawaz, and K. G. Shin, "LinkDroid: Reducing unregulated aggregation of app usage behaviors," in *USENIX Security*, 2015, pp. 769–783.

[27] S. Seneviratne, H. Kolamunna, and A. Seneviratne, "A measurement study of tracking in paid mobile applications," in *WiSec*, 2015.

[28] Sable, "Soot analysis framework," http://www.sable.mcgill.ca/soot, Aug. 2018.

[29] Google, "Monkey: UI/Application exerciser for Android," http://developer.android.com/tools/help/monkey.html, Aug. 2018.

[30] T. Wang, J. Blocki, N. Li, and S. Jha, "Locally differentially private protocols for frequency estimation," in *USENIX Security*, 2017, pp. 729–745.

[31] R. Bassily and A. Smith, "Local, private, efficient protocols for succinct histograms," in *STOC*, 2015, pp. 127–135.

[32] R. Bassily, K. Nissim, U. Stemmer, and A. Thakurta, "Practical locally private heavy hitters," in *NIPS*, 2017, pp. 2285–2293.

[33] M. Bun, J. Nelson, and U. Stemmer, "Heavy hitters and the structure of local privacy," in *PODS*, 2018, pp. 435–447.

[34] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *CC*, 2003, pp. 153–169.

[35] I. Hadar, T. Hasson, O. Ayalon, E. Toch, M. Birnhack, S. Sherman, and A. Balissa, "Privacy by designers: software developers privacy mindset," *Empirical Software Engineering*, vol. 23, no. 1, pp. 259–289, 2018.

[36] M. Grechanik, C. Csallner, C. Fu, and Q. Xie, "Is data privacy always good for software testing?" in *ISSRE*, 2010, pp. 368–377.

[37] A. Budi, D. Lo, L. Jiang *et al.*, "kb-anonymity: A model for anonymized behaviour-preserving test and debugging data," in *PLDI*, 2011, pp. 447–457.

[38] K. Taneja, M. Grechanik, R. Ghani, and T. Xie, "Testing software in age of data privacy: A balancing act," in *FSE*, 2011, pp. 201–211.

[39] D. Lo, L. Jiang, A. Budi *et al.*, "kbe-anonymity: Test data anonymization for evolving programs," in *ASE*, 2012, pp. 262–265.

[40] F. Peters and T. Menzies, "Privacy and utility for defect prediction: Experiments with MORPH," in *ICSE*, 2012, pp. 189–199.

[41] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," *TSE*, vol. 39, no. 8, pp. 1054–1068, 2013.

[42] Z. Li, X.-Y. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying, "On the multiple sources and privacy preservation issues for heterogeneous defect prediction," *TSE*, pp. 1–21, 2017.

[43] T. T. Nguyên, X. Xiao, Y. Yang, S. C. Hui, H. Shin, and J. Shin, "Collecting and analyzing data from smart device users with local differential privacy," *arXiv:1606.05053*, 2016.

[44] B. Ding, J. Kulkarni, and S. Yekhanin, "Collecting telemetry data privately," in *NIPS*, 2017, pp. 3571–3580.

[45] T. Menzies and T. Zimmermann, "Software analytics: So what?" *IEEE Software*, no. 4, pp. 31–37, 2013.

[46] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "AR-miner: Mining informative reviews for developers from mobile app marketplace," in *ICSE*, 2014, pp. 767–778.

[47] W. Martin, F. Sarro, and M. Harman, "Causal impact analysis for app releases in Google Play," in *ICSE*, 2016, pp. 435–446.

[48] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *ICSE*, 2016, pp. 14–24.

[49] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *TSE*, vol. 43, no. 9, pp. 817–847, Sept 2017.

[50] Y. Z. Ehsan Noei, Daniel Alencar da Costa, "Winning the app production rally," in *FSE*, 2018, pp. 1–12.

[51] P. Devanbu, P. Kudigrama, C. Rubio-González, and B. Vasilescu, "Timezone and time-of-day variance in GitHub teams: An empirical method and study," in *SWAN*, 2017, pp. 19–22.

[52] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov, "The sky is not the limit: Multitasking on GitHub projects," in *ICSE*, 2016, pp. 994–1005.

[53] M. Zhou, Q. Chen, A. Mockus, and F. Wu, "On the scalability of Linux kernel maintainers' work," in *FSE*, 2017, pp. 27–37.

[54] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *MSR*, 2018, pp. 542–553.

[55] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on TensorFlow program bugs," in *ISSTA*, 2018, pp. 129–140.

[56] E. Cohen and M. P. Consens, "Large-scale analysis of the co-commit patterns of the active developers in GitHub's top repositories," in *MSR*, 2018, pp. 426–436.

[57] Y. Zhao, F. Zhang, E. Shihab, Y. Zou, and A. E. Hassan, "How are discussions associated with bug reworking?: An empirical study on open source projects," in *ESEM*, 2016, pp. 21:1–21:10.

[58] F. Peters, T. Tun, Y. Yu, and B. Nuseibeh, "Text filtering and ranking for security bug report prediction," *TSE*, pp. 1–16, 2017.

[59] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, "An empirical study on Android-related vulnerabilities," in *MSR*, 2017, pp. 2–13.

[60] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: A large-scale empirical study," in *MSR*, 2017, pp. 413–424.

[61] Apple, "Share analytics information with Apple," https://support.apple.com/kb/ph25654, Aug. 2018.

[62] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, D. Hao, G. Huang, and F. Feng, "PRADA: Prioritizing Android devices for apps by mining large-scale usage data," in *ICSE*, 2016, pp. 3–13.

[63] X. Liu, X. Lu, H. Li, T. Xie, Q. Mei, H. Mei, and F. Feng, "Understanding diverse usage patterns from large-scale appstore-service profiles," *TSE*, vol. 44, no. 4, pp. 384–411, 2017.

[64] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with Angry Birds, Facebook and Kindle: A large scale study on mobile application usage," in *MobileHCI*, 2011, pp. 47–56.

[65] Y. Agarwal and M. Hall, "ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing," in *MobiSys*, 2013, pp. 97–110.

[66] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "GAMMA system: Continuous evolution of software after deployment," in *ISSTA*, 2002, pp. 65–69.

[67] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003, pp. 141–154.

[68] J. C. Duchi, M. I. Jordan, and M. J. Wainwright, "Local privacy and statistical minimax rates," in *FOCS*, 2013, pp. 429–438.

[69] K. Nissim and U. Stemmer, "Clustering algorithms for the centralized and local models," *arXiv:1707.04766*, 2017.

[70] A. Smith, A. Thakurta, and J. Upadhyay, "Is interaction necessary for distributed private learning?" in *S&P*, 2017, pp. 58–77.