# PRIVAID: Differentially-Private Event Frequency Analysis for Google Analytics in 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 data about app performance and user behaviors. However, there are significant privacy concerns about the collection and use of such data. In a world with increasing privacy threats and demands, it is important to develop better definitions and enforcement of the trade-offs between data analytics benefits and the loss of privacy.**

**We propose PRIVAID, a conceptual approach and related software components for achieving such trade-offs in app analytics for Android apps. Our proposal employs *differential privacy* (DP), a powerful and rigorous privacy definition and algorithmic framework. In essence, a DP solution perturbs the results of a data analysis in order to achieve a quantifiable notion of privacy. We highlight the challenges of applying DP techniques to existing analytics frameworks for Android apps. We then describe how these challenges are addressed by the design of PRIVAID and its components for app analysis, rewriting, and profiling. Next, we develop an instance of PRIVAID for DP collection of event frequency information in apps that use the popular Google Analytics framework. Our experimental evaluation demonstrates that practical trade-offs between privacy, accuracy, and cost can be achieved for real-world Android apps.**

## I. INTRODUCTION

Mobile apps often use analytics infrastructures provided by companies such as Google and Facebook [1]. The data gathered from such analytics can be used for offline 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 [2].

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]. However, there are significant privacy concerns about the collection and use of such data. Over the last few years, there has been increasing awareness and scrutiny of data gathering performed by various companies. There are legislative efforts and societal demands for increased transparency and for well-defined trade-offs between the utility of data gathering and the corresponding loss of privacy.

One promising technique for achieving such trade-offs is *privacy-preserving data analysis*. Such analysis is designed with guarantees about the loss of privacy and the accuracy of analysis based on the collected data. The last decade has witnessed the rise of a rigorous theory to deal with this challenge. This theory is centered around a meaningful and robust mathematical definition for privacy, known as *differential privacy* (DP) [3]. A powerful algorithmic framework for differential privacy has been developed over the years, and led to numerous practical and efficient algorithms with strong and provable privacy guarantees for various problems. Differential privacy has recently been adopted by industry—for example, in the Chrome browser [4] and in iOS-10 [5], [6].

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 collect such analytics data has clear benefits to users. The app developer also benefits from such mechanisms: the information they gather provides analytics value while at the same time the app creator can claim, with confidence, that users are provably protected against leaks of their sensitive data due to rogue employees, legal proceedings, unethical business practices, or security breaches. Such claims by developers make their product more attractive to users. In addition, they may be able to provide legal protection for the business, which may be valuable due to the increasing interest taken by legislatures around the world in protecting the privacy of their citizens.

***Challenges.*** Despite significant advances in DP theory, applying DP solutions to analytics frameworks for mobile apps faces several 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 capabilities in a given app with little or no effort from the app developer. Ideally, the developer would write their app without any DP considerations, and a subsequent automated code rewriting step would introduce DP-enforcing code. Such separation of concerns is highly valuable for software development, testing, debugging, and evolution.

The third challenge is to allow a developer to understand and fine-tune their data collection. The performance of DP analyses depends on various parameters and it is difficult to understand how these parameters affect the trade-offs among privacy, analytics accuracy, and time/memory/communication costs. Developing trade-offs for these factors under realistic usage scenarios is a major challenge for an app developer who has decided to deploy a DP-based analytics solution.

***Our proposal.*** To address these challenges, we propose the PRIVAID approach for differentially-**priv**ate **a**nalytics for Andro**id** apps. The approach consists of a conceptual design for DP-based mobile app analytics together with several software components that instantiate the approach and can be used by app developers. To the best of our knowledge, this is the first work that attempts to introduce DP in the behavioral analysis of Android apps.

PRIVAID is specifically designed to address the three challenges outlined above. First, the approach does not expect any changes to the infrastructure provided by Google, Facebook, etc. Rather, by using its own data pre/post processing, it "fools" the DP-unaware analytics infrastructure to behave in DP manner. The data 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 addresses the second challenge described earlier, by allowing a developer to focus on the business logic of the app without directly creating or manipulating DP-related code. Finally, PRIVAID provides infrastructure for testing and profiling the resulting DP app, to help understand the interplay between analysis parameters and to enable the developer to make informed deployment decisions.

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

- We propose a design for PRIVAID and describe the significance of its individual components
- We define an instance of PRIVAID for collecting event frequency information for apps that use the popular Google Analytics infrastructure
- We develop static code rewriting and run-time support needed to implement this instance of PRIVAID
- We describe an experimental evaluation that demonstrates PRIVAID's feasibility and performance

This work is a first step in defining a research agenda for DP analytics for mobile apps. Such apps are deployed on billions of devices. There is fundamental tension between the privacy of device users and the business needs of app developers. In this context, PRIVAID is the first effort to employ differential privacy to establish a well-defined space of trade-offs between privacy and analytics accuracy, together with a practical demonstration for real apps using Google Analytics. This work also enables future research on the increasingly-important subject of user privacy, in an environment where widespread use of data analytics is rapidly becoming the norm.

## II. BACKGROUND: GOOGLE ANALYTICS FOR ANDROID

There are several analytics infrastructures that offer full-stack solutions to an app developer. Google Analytics (GA) [7] and its successor Firebase [8] are among the most popular ones. They allow a developer to collect and conduct various analyses against users' data. Facebook [9], Yahoo [10], and several others also provide similar services. Even though their policies [11]–[14] 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 [15], [16].

A recent study of thousands of Android apps [2] has identified that Google Analytics was used by 38% of the analyzed apps. This popularity motivates our focus on privacy-preserving analysis for GA; however, the underlying theoretical machinery and program analyses are general and should be applicable to other analytics libraries as well. One of the key features of GA is to provide the *frequency of events*. Such frequency information helps a developer understand how users interact with her app, by quantifying user engagement and behaviors. An *event* is defined by the developer to track specific actions that users take within an app. 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.

### *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 some basic GA capabilities, we use the example in Figure 1. The `ParKing` app navigates users to parking places, records history of parking locations, and reminds users about parking time. It has over 100K installs via Google Play. Figure 1a shows a snippet of decompiled code from the app. Class `ParKingApplication` maintains a global context and its instance is retrieved via `getApplication` (line 13). 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 Figure 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.

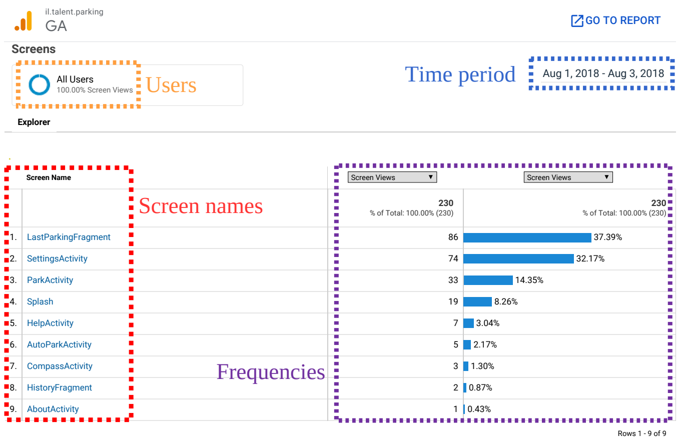
Activities are the core components in Android apps. An activity displays a window containing GUI widgets. Class

```
 1 class ParKingApplication extends Application {
 2   Tracker t;
 3   Tracker a() {
 4     if (t == null) {
 5       GoogleAnalytics i =
 6         GoogleAnalytics.getInstance(this);
 7       t = i.newTracker(R.xml.global_tracker); }
 8     return t; } }

 9 class AutoParkActivity extends Activity {
10   Switch s;
11   Tracker t;
12   void onCreate(...) {
13     t = ((ParKingApplication)getApplication()).a();
14     s = (Switch)findViewById(...);
15     s.setOnCheckedChangeListener(
16       new OnCheckedChangeListener() {
17         void onCheckedChanged(boolean isChecked) {
18           MyUtils.a(isChecked, t); ... } }); ... }
19   void onResume() {
20     t.setScreenName("AutoParkActivity");
21     ScreenViewBuilder b = new ScreenViewBuilder();
22     t.send(b.build()); } }

23 class MyUtils {
24   static void a(boolean z, Tracker t) {
25     EventBuilder b = new EventBuilder();
26     if (z) { b.setAction("Activated"); ...; }
27     else   { b.setAction("Deactivated"); ...; }
28     Map<String,String> m = b.build();
29     t.send(m); } }
```

(a) Decompiled code

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

(b) `global_tracker.xml`



| | 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`.

`AutoParkActivity` shows a screen with settings for automatic parking detection. It contains a switch widget `s` (line 10) to turn this function on/off. The `onCreate` callback is called when an activity is created. It retrieves and stores the `Tracker` to field `t` at line 13. An `OnCheckedChangeListener` is associated with widget `s` at line 15–16. Whenever the switch is pressed, method `onCheckedChanged` at line 17 will be invoked and `MyUtils.a` will be called at line 18. GA uses the builder pattern for creation of events. At line 25, an `EventBuilder` is created. Lines 26–27 set the action for the event to "Activated" or "Deactivated" according to the `boolean` parameter `z` indicating whether auto-parking function is on/off. `EventBuilder.build` at line 28 returns a helper map `m` containing the event action together with other GA-internal data. This map is used as a parameter of `Tracker.send` at line 29, which uploads the event to Google's backend servers.

When an activity comes to the foreground, its `onResume` callback is invoked. Lines 20–22 create a *screen view* event and send it to the GA servers. Screen views track the views of a particular screen in the app. Conceptually, they are identical to pageviews for web analytics. 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 20 records a name for the current screen as "AutoParkActivity". `ScreenViewBuilder` at line 21 is the helper class to build the event. `ScreenViewBuilder.build` functions similarly to `EventBuilder.build`, returning a map for upload with action type "screenview". The call to `Tracker.send` at line 22 sends the screen view to the remote 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 in the app code: "AboutActivity", "AutoParkActivity", "CompassActivity", "HelpActivity", "HistoryFragment", "LastParkingFragment", "ParkActivity", "SettingsActivity", "Splash", "TransparentActivity", and "ZoneEditorActivity". One of the challenges for PRIVAID 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. Figure 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, as discussed in the next section, even seemingly-innocent information that has been anonymized can lead to real privacy leaks when combined with additional information from other sources. In a societal/legal environment where privacy is increasingly valued and enforced, this state of affairs is unacceptable.

## III. BACKGROUND: DIFFERENTIAL PRIVACY

Differential privacy is a general approach for protection against a wide range of privacy attacks. In such scenarios, there is release of some data and an adversary attempts to learn private individual information from the 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). These attacks are different from security attacks, in which unauthorized access is gained to sensitive data. In security, data access by an adversary should be denied. In privacy, certain data is intended to be released (to government, researchers, or businesses) but it is assumed that an adversary will also gain access to this data. 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, user data shared with some organization could be subpoenaed in legal proceedings despite the user's expectations.

Anonymizing or removing personally-identifiable information cannot guarantee user privacy while keeping the usefulness of collected data. Researchers have demonstrated various attacks in this setting [15]–[18]. A prominent example is work [15] that identified individual records from Netflix's collection of anonymized viewing histories by linking with another movie database.

Differential privacy (DP) [19] has emerged as a prominent approach for protection against privacy attacks. With DP analysis, anyone seeing the analysis results will essentially make the same inference about any individual's private information, whether or not that individual's private information is included in the input to the analysis [3], [20]. There are many techniques for achieving DP. As one simple example, the output of a non-DP analysis could be perturbed using random noise with Laplacian distribution to achieve provable DP guarantees.

We will not attempt a detailed description of this rich field of research; extensive overviews are available elsewhere [19], [20]. DP solutions have been deployed by several companies (e.g., Google [4], [21], Apple [5], [6], and Uber [22]). The U.S. Census Bureau will use differential privacy to protect the results from the 2020 census [23]. Given the rapid emergence of large-scale data analytics and machine learning, and their detrimental effects on privacy, the importance and urgency of privacy solutions (including differential privacy) will continue to increase in the foreseeable future.

There are two major models for defining DP problems. In the *centralized model*, individuals' data is provided to a trusted curator, where DP analysis is performed and its results are released to (untrusted) analysis clients. In the *local model*, the curator is not trusted: raw data that reaches the curator can be observed by an adversary (e.g., the curator itself could be an adversary). For such *locally differentially private* (LDP) problems, each user performs local data perturbation before releasing any information to the curator. The LDP model is particularly well suited for app analytics for mobile devices. 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 client—that is, to the app developer. This model provides privacy guarantees to the app user regardless of the (unpredictable) actions of an analytics company such as Google or of the app development company.

## A. Exemplar LDP Analysis: Frequency Estimates

To make our discussion more concrete, we will employ a fundamental problem in data analytics: constructing (estimates of) *event frequencies*. This exemplar problem is closely related to many 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) [24], [25], distribution estimation [26], regression, clustering [27], and classification.

Consider a community of $n$ app users and assume that each user is identified by an integer id $i \in \{1, \dots, n\}$. Each user has a single data item $v_i$. For now, assume that all data items belong to some data dictionary $\mathcal{D}$ that is pre-defined by the app developer based on her analytics needs. For any value $v \in \mathcal{D}$, its frequency is $f(v) = |\{i \in \{1, \dots, n\} : v_i = v\}|$. The app developer's goal is to obtain the histogram defined by $f(v)$ for all $v \in \mathcal{D}$. This problem definition is an abstraction of a typical analytics task. All analytics frameworks for mobile apps support such tasks. For example, frequency histograms are one of the major outputs provided by GA to the app developer, as illustrated in Figure 1c.

An LDP solution to this problem will apply a *local randomizer* to each user's data item. More precisely, an $\epsilon$-LDP protocol applies an $\epsilon$-local randomizer $R : \mathcal{D} \to \mathcal{Z}$ to each user's item $v_i$. The resulting $z_i = R(v_i)$ is sent to the server. 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 $v \in \mathcal{D}$.

Two important properties of an $\epsilon$-LDP protocol are:

- *Privacy:* The privacy is due to the $\epsilon$-local randomizer: $\forall v, v' \in \mathcal{D}, z \in \mathcal{Z} : Pr[R(v) = z] \leq e^{\epsilon} Pr[R(v') = z]$. Here $Pr[a]$ denotes the probability of $a$.
- *Accuracy:* Measured based on the difference between the vector of actual frequencies and the vector of estimates—specifically, the largest value of $|\hat{f}(v) - f(v)|$ over all $v$.

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 a third party) 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* $\epsilon$ 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.

There are various ways to design the local randomizer $R$. For example, the popular RAPPOR approach [4] used in Google Chrome represents the data of user $i$ as a bit vector of length $|\mathcal{D}|$, where only bit $v_i$ is set to 1. In the simplest version of RAPPOR ("basic one-time" [4]), each bit is randomly inverted using a biased coin. The server processes the resulting bit vectors and accounts for the randomization when computing the frequency estimates $\hat{f}(v)$ for all $v \in \mathcal{D}$.

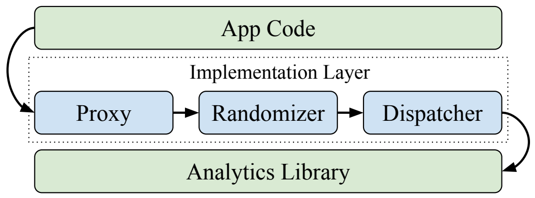


Fig. 2: PRIVAID on app user's device.

There is a large body of work on this problem. As a baseline, we describe a very simple standard solution. Our explanation is a re-formulated version of the basic one-time RAPPOR descriptions available elsewhere [4], [28]. There are more advanced solutions with better theoretical accuracy and cost (e.g., [24], [29]), but they require significant "white box" re-design of the analytics infrastructure and cannot be easily deployed in the near future. Instead, we consider an approach that keeps the current analytics infrastructure intact.

The solution is based on a local randomizer $R : \mathcal{D} \to \mathcal{P}(\mathcal{D})$ where $\mathcal{P}(\ldots)$ denotes a powerset. Given an item $v_i$ held by an user, the user's data "I have $v_i$" is randomized to "I have the set of items $R(v_i)$". This is an instance of the classic *randomized response* technique that has been used in social sciences to gather sensitive data (e.g., about illegal behaviors). The randomizer is defined as follows:

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

It is easy to show that this is an $\epsilon$-local randomizer.

Given sets $R(v_i)$ reported by all users, the server computes standard frequency counts to obtain a histogram value $h(v)$ for each $v$. Here $h(v)$ is the number of occurrences of $v$ in all $R(v_i)$. These values have to be adjusted to account for the effects of randomization. The resulting frequency estimates are

$$\hat{f}(v) = \frac{(1 + e^{\frac{\epsilon}{2}})h(v) - n}{e^{\frac{\epsilon}{2}} - 1} \tag{1}$$

where $n$ is the number of users. It is easy to see that the expected value of the estimate $\hat{f}(v)$ is equal to the real frequency $f(v)$. Thus, $\hat{f}(v)$ is an unbiased estimator of $f(v)$.

## IV. DESIGN OF PRIVAID

Current analytics frameworks for mobile apps—e.g., Google Analytics, Facebook Analytics, and Yahoo Flurry—do not provide LDP features and there is no prior work on performing LDP analytics in them. Introducing LDP mechanisms in these frameworks would provide rigorous and quantifiable trade-offs between data gathering utility and loss of privacy. However, doing this *without* any changes to the underlying analytics infrastructure is a challenge. Figure 2 presents the design of PRIVAID that addresses this challenge.

On top of the standard analytics libraries running on the user's device, PRIVAID introduces 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., the calls to `Tracker.send` in Figure 1) can be automatically redirected to corresponding proxy APIs (e.g., `Proxy.send`). This redirection can be achieved with an automated code rewriting tool. Our concrete implementation uses the Soot code analysis/rewriting framework [30], but many other choices are also available. The proxy API methods coordinate the remaining PRIVAID components. The randomizer component 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. 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.

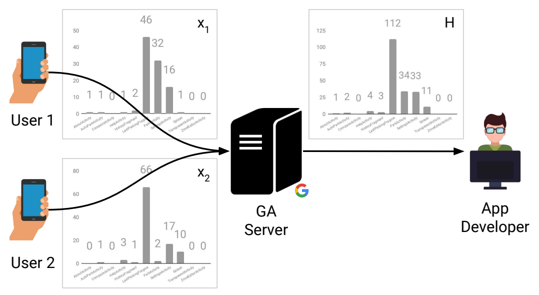
The implementation layer can be built at the time the app is created, and distributed as part of the app code when the app is published in an app market. Most of the code in this layer would be open-source and created by a trusted party (e.g., privacy researchers), while app-specific functionality would be added by the app developer. 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 and analysis. After the server reports its results, they are post-processed by the app developer to obtain the actual estimates of the desired analytics data. Such a "black box" solution is easy to deploy today.

## V. EVENT FREQUENCIES FOR GOOGLE ANALYTICS

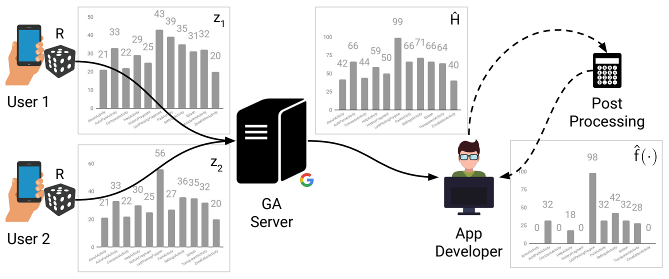
As a proof of concept, we have defined an instance of PRIVAID that collects event frequency information for apps that use Google Analytics. Our current implementation focuses on screen view events, but the underlying theoretical machinery and implementation infrastructure are general and could be used for other categories of events. Recall from Section II-A that each screen in the app has a string identifier and the app uses GA API calls to send such events to the server (lines 20–22 in Figure 1a). The GA event frequency reports, similar to the one shown in Figure 1c, can be created using local randomization and then post-processed to account for its effects. Next, we describe the building blocks of this solution.

### A. LDP Event Frequency Estimates

The problem we define is a generalization of the exemplar problem from Section III-A. While in that problem each user reports a single data item, now 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



(a) Regular GA workflow



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

Fig. 3: Regular GA vs PrivAid, with two app users.

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.

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 Figure 1c.

***Example.*** Figure 3a illustrates this scenario with two app users. The data was obtained by running the popular Monkey GUI testing tool [31] 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 $H$ to the app developer. ⧫

An $\epsilon$-LDP solution applies an $\epsilon$-local randomizer $R : \mathbb{N}^d \to \mathbb{N}^d$. 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.*** Figure 3b illustrates the steps of the LDP solution. Each user $i$, independently of any other users or the server, applies the local randomizer $R$ 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 PrivAid to the real GA events in `ParKing` that were used for $x_i$ in Figure 3a. Value $\epsilon = \ln(9)$ is used in prior work [4] 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 Figure 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 VII presents an extensive experimental evaluation of the accuracy that can be achieved in realistic scenarios. ⧫

As standard with DP analyses, including the exemplar problem from Section III-A, privacy is achieved by ensuring that for any output of the randomizer $R$, many inputs are possible with high probability. One standard choice for the desired properties of $R$ is based on the notion of *user-level privacy* [19]. In our context, the definition is as follows. Consider two 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 is defined as follows: for all $z \in \mathbb{N}^d$ and all pairs $x, x' \in \mathbb{N}^d$, $Pr[R(x) = z] \leq e^\epsilon Pr[R(x') = z]$. 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. The exemplar problem discussed earlier is an instance of this definition for $k = 1$.

### B. Baseline Local Randomizer

We first define a baseline randomizer $R$ for use in PrivAid, similar to the single-event randomization from Section III-A. Refinements of this baseline solution are described later.

For any $v \in \mathcal{D}$, let $r(v)$ be a subset of $\mathcal{D}$ defined as follows: (i) $v$ is included in $r(v)$ with probability $(e^{\frac{\epsilon}{2}})/(1+e^{\frac{\epsilon}{2}})$, and (ii) $v'$ is included in $r(v)$ with probability $1/(1+e^{\frac{\epsilon}{2}})$ for any $v' \neq v$. This element-level randomizer is the same as the one defined in Section III-A. 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 $R(x_i)$. Figure 3 shows examples of $x_i$ and $z_i$.

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 inside PrivAid is shown in Figure 4. The call

```
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.

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 $R : \mathbb{N}^d \to \mathbb{N}^d$. 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 $\hat{H}(v)$. This is done similarly to the single-event case described in Section III, Equation 1: the value of $(1 + e^{\frac{\epsilon}{2}})\hat{H}(v) - nk$ is divided by $(e^{\frac{\epsilon}{2}} - 1)$. Here $n$ is the number of users and $k$ is the number of real events for each user. The expected value of an estimate $\hat{f}(v)$ is the real frequency $f(v)$.

***Example.*** Consider the $v$ for which $\hat{H}(v) = 71$ in Figure 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 Figure 3b. If the estimate becomes negative, the reported frequency is 0. For example, the first bar for $\hat{H}$ in Figure 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. ⧫

One technical detail is that this post-processing depends on the number $nk$ of real events across all users. If some users have fewer than $k$ events, the actual number of events should be used in the $\hat{f}(v)$ computation. Each user can send to the server his number of real events. Alternatively, the number of real events across all users can be easily estimated from the number of events observed at the server (i.e., by the total size of $\hat{H}$) by considering the probabilities in the definition of $R$. In our experience, these estimates are very accurate.

### C. 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 per user), achieving useful user-level privacy presents a significant challenge.

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* [24], [25]. 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. PRIVAID uses 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 observed event $v_i^j$, index $j$ is checked against this set of $t$ values. The event is ignored if the index 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. These benefits come at the expense of decreased accuracy. The worst-case accuracy is reduced by a factor of $\sqrt{k/t}$ [24], [25]. Despite this worst-case behavior, 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.

## VI. PRIVAID FOR GA EVENT FREQUENCIES

The conceptual approach from the previous section was implemented with the help of several components, following the design from Figure 2. First, we built a proxy for GA APIs, as well as a code rewriting tool 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 the code in our implementation layer, is then packed back into a new APK which can later be installed on a user's device.

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 $\mathcal{D}$. While this dictionary could be provided by an app developer who utilizes PRIVAID, it is also possible to construct it automatically. We built a static analysis that considers relevant API calls (e.g., `setScreenName` at line 20 in Figure 1) to determine the screen names that flow into calls to `send` (e.g., at line 22). For the running example, this analysis identifies the 11 screen names described in Section II-A.

For each `GoogleAnalytics.newTracker` call site, the static analysis creates an artificial object representing the corresponding `Tracker` instance. The analysis 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 flow-insensitive, context-insensitive, field-based points-to analysis [32].

The static analysis also creates an object for each `new ScreenViewBulder` site. These objects are propagated to `build` calls (e.g., line 22 in Figure 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 Figure 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. 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 in maps during the propagation. This information is then packed in the app's APK as part of the PRIVAID implementation layer.

It is possible that the randomizer does not have the complete definition of $\mathcal{D}$ ahead of time, and occasionally observes run-time events that are not in $\mathcal{D}$. This could happen, for example, if the app developer's dictionary definition is incomplete, or the static analysis is not sound. It is possible 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.

## VII. Experimental Evaluation

Our PRIVAID instance for GA screen view events was implemented via app analysis and rewriting with Soot [30]. In the near future, we plan to release this implementation together with the subject apps and the testing/profiling infrastructure used in its evaluation. 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 [31].

### 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 the dictionary $\mathcal{D}$ of string identifiers for screens. Apps with small dictionaries (less than 10 elements) were excluded because we wanted to study the overhead of the approach; recall that this overhead depends on dictionary size.

The code rewriting tool then took as input the original APK of each app to generate a repacked APK with our implementation layer bundled with the app. We installed each APK and ran Monkey for 5 minutes on 10 emulators. Each of the 10 runs used a different seed for Monkey's random generation of GUI events, and thus triggered a different GUI behavior (and, as a result, a different sequence of GA API calls). Apps for which the entire dictionary could not be determined statically were excluded from further consideration. Apps for which Monkey achieved low coverage of the dictionary were also excluded; this usually happens when an app requires sign-up or sign-in for use. From the resulting set of apps, we selected 15 subjects that covered a representative range of values for $|\mathcal{D}|$.

TABLE I: Study subjects.

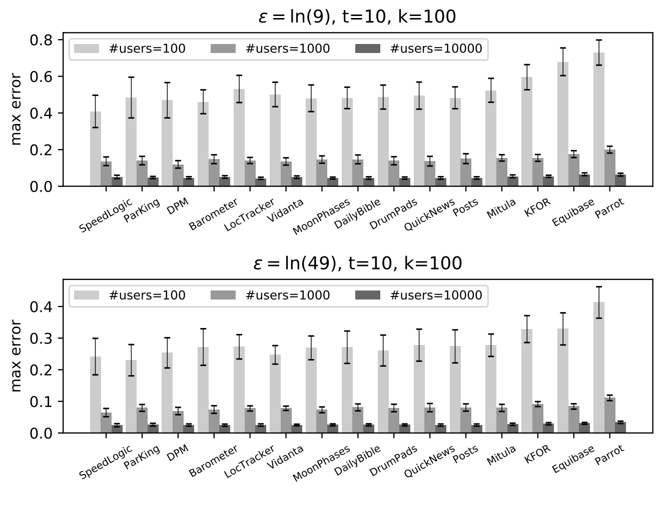
| App | #Classes | #Stmts | $\lvert\mathcal{D}\rvert$ | Time (s) |
|---|---|---|---|---|
| SpeedLogic | 119 | 5881 | 10 | 0.51 |
| ParKing | 543 | 37478 | 11 | 1.84 |
| DPM | 10859 | 939666 | 12 | 80.2 |
| Barometer | 668 | 52252 | 13 | 1.91 |
| LocTracker | 269 | 24575 | 14 | 1.10 |
| Vidanta | 2652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

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 average cost is 5.78 seconds per 100K Jimple statements.

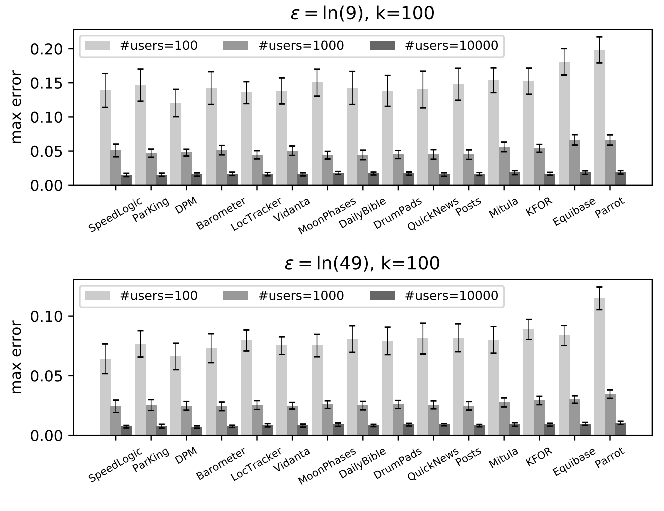
### B. Accuracy

For our evaluation, we extended PRIVAID 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 [31] 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 in PRIVAID 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



(a) With sampling



(b) Without sampling

Fig. 5: Accuracy of PRIVAID.

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 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.

Figure 5 shows the accuracy of PRIVAID with and without sampling, with different values of $\epsilon$ and number of users $n$. The values of $\epsilon$ match those used in prior work [4]. Recall that the 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 $f = \sum_{v \in \mathcal{D}} f(v)$. For each measurement, the $y$-axes show the mean of 20 separate repetitions, with 95% confidence interval. The $x$-axes show the names of apps sorted by $|\mathcal{D}|$.

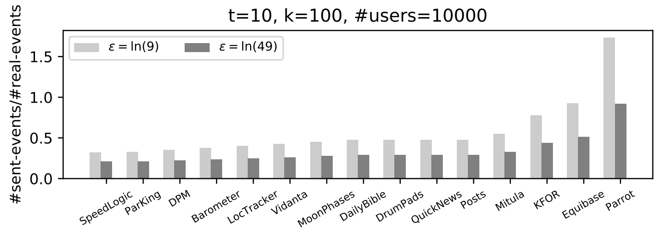
***More users result in better accuracy.*** The nature of differential privacy requires sufficiently large number of users to be useful. With small values of $n$, high accuracy cannot be achieved in our setting (and, in all likelihood, in similar settings). Figure 5 shows the dramatic improvement in 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 Figure 5a 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 $\frac{3}{4}$ and each other element of $\mathcal{D}$ is reported with probability $\frac{1}{4}$. 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 Figure 5b.

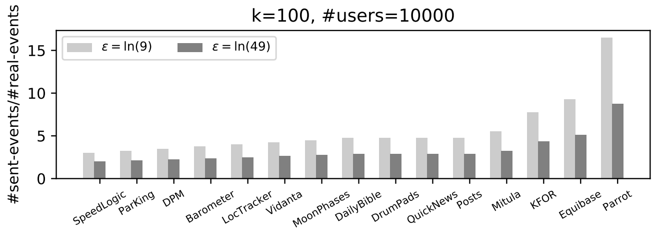
***Sampling does not hurt accuracy.*** Sampling achieves both lower overhead and increased user-level privacy. However, a natural concern is whether sampling will reduce accuracy. The trend exhibited by our results is that high accuracy can be achieved with large number of users. For the analysis parameters used in our evaluation, the estimates are reasonably accurate when $n = 10000$: in Figure 5a, 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*

Figure 6 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



(a) With sampling



(b) Without sampling

Fig. 6: Overhead of PRIVAID.

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 Figure 6b meet this expectation. For example, for `ParKing` we have $d = 11$ and PRIVAID sent about 3 events per actual event when $\epsilon = \ln(9)$. Sampling can reduce this cost by a magnitude of $k/t$, as we are sending only $t$ events per $k$ actual events. Figure 6a 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 Figure 5a, these experiments indicate that practical trade-offs between accuracy, overhead, and privacy can be achieved when the number of users $n$ is reasonably large.

## VIII. RELATED WORK

Privacy has gained growing attention in various fields of software engineering [33] such as testing [34]–[37] and defect prediction [38]–[40]. Budi et al. [35] propose $k$-anonymity-based generation of new test cases while preserving their original behaviors. Their following work [37] extends the approach to be applicable to evolving programs. MORPH [38] preserves data privacy of software defects in a cross-company scenario, by perturbing instance values. CLIFF+MORPH [39] removes dominant attributes for each class before perturbation. Li et al. [40] 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 earlier. There also exist several practical realizations of LDP for data analytics. Google's RAPPOR combines randomized responses and Bloom filters to encode and identify popular URLs in the Chrome browser without revealing users' browsing habits [4], [21]. Apple applies DP for gathering analytics data for emoji and quick type suggestions, search hints, and health-related usage [5], [6]. Samsung proposed the Harmony LDP system to collect data from smart devices for both basic statistics and complex machine learning tasks [41]. Microsoft uses LDP to collect telemetry data over time across millions of devices [42]. We are not aware of any efforts to apply these techniques to analytics for Android apps. One significant challenge is that, unlike this prior work, we need to assume that the analytics infrastructure is LDP-unaware. The solution presented in this paper achieves LDP for Google Analytics event frequencies without any changes to the GA libraries on the users' devices, or to the GA servers.

In addition to frequency estimation, many other analytics problems have been considered: for example, heavy hitters [24], [25], distribution estimation [26], clustering [27], learning [43], and convex optimization [44]. This rich body of work presents interesting opportunities for sophisticated LDP analytics for mobile apps.

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; for example, Apple collects "details about app or system crashes, freezes, or kernel panics" from their macOS 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.

## IX. CONCLUSIONS AND FUTURE WORK

We demonstrate that LDP features can be added to existing app analytics in Android apps without changes to the underlying analytics infrastructure. The proposed PRIVAID approach increases user privacy, requires little effort from app developers, and does not sacrifice analytics accuracy.

There are many interesting software analytics problems for mobile apps. Rather than simple frequency counts, more powerful analyses could be performed [24]–[27], [43], [44]. Solutions to these problems should achieve practical analyses of real app data, which requires both theoretical and empirical understanding of trade-offs between privacy, accuracy, and cost. There should also be strong tool support for app developers to enable them to deploy such solutions with ease and confidence. In a world with increasing privacy threats and demands, a research agenda focusing on these problems defines an important direction for future work.

## References

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

[2] Exodus Privacy, "Most frequent app trackers for Android," https://reports.exodus-privacy.eu.org/reports/stats/, Aug. 2018.

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

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

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

[6] A. G. Thakurta, A. H. Vyrros, U. S. Vaishampayan, G. Kapoor, J. Freudiger, V. R. Sridhar, and D. Davidson, "Learning new words," in *Granted US Patents 9594741 and 9645998*, 2017.

[7] Google, "Google Analytics," https://analytics.google.com, Aug. 2018.

[8] ——, "Firebase," https://firebase.google.com, Aug. 2018.

[9] Facebook, "Facebook analytics," https://analytics.facebook.com, Aug. 2018.

[10] Oath, "Flurry," http://flurry.com, Aug. 2018.

[11] Google, "Measurement protocol/SDK/user ID policy," https://developers.google.com/analytics/devguides/collection/android/v4/policy, Aug. 2018.

[12] ——, "Google Analytics for Firebase use policy," https://firebase.google.com/policies/analytics, Aug. 2018.

[13] Facebook, "Facebook platform policy," https://developers.facebook.com/policy, Aug. 2018.

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

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

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

[17] L. Sweeney, "Weaving technology and policy together to maintain confidentiality," *The Journal of Law, Medicine & Ethics*, vol. 25, no. 2-3, pp. 98–110, 1997.

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

[19] 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.

[20] K. Nissim, T. Steinke, A. Wood, M. Altman, A. Bembenek, M. Bun, M. Gaboardi, D. O'Brien, and S. Vadhan, "Differential privacy: A primer for a non-technical audience (preliminary version)," *Vanderbilt Journal of Entertainment and Technology Law*, 2018.

[21] G. Fanti, V. Pihur, and Ú. Erlingsson, "Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries," *PoPETs*, vol. 2016, no. 3, pp. 41–61, 2016.

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

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

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

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

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

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

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

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

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

[33] 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.

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

[35] 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.

[36] 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.

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

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

[39] 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.

[40] 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.

[41] 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.

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

[43] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, and A. Smith, "What can we learn privately?" *SICOMP*, vol. 40, no. 3, pp. 793–826, 2011.

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

[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.