# Introducing Differential Privacy Mechanisms for Mobile App Analytics of Dynamic Content

Sufian Latif
*Ohio State University*
Columbus, OH, USA
latif.28@osu.edu

Yu Hao
*Ohio State University*
Columbus, OH, USA
hao.298@osu.edu

Hailong Zhang
*Fordham University*
New York City, NY, USA
hzhang285@fordham.edu

Raef Bassily
*Ohio State University*
Columbus, OH, USA
bassily.1@osu.edu

Atanas Rountev
*Ohio State University*
Columbus, OH, USA
rountev.1@osu.edu

***Abstract*—Mobile app analytics gathers detailed data about millions of app users. Both customers and governments are becoming increasingly concerned about the privacy implications of such data gathering. Thus, it is highly desirable to design privacy-preserving versions of mobile app analytics. We aim to achieve this goal using differential privacy, a leading algorithm design framework for privacy-preserving data analysis.**

**We apply differential privacy to dynamically-created content that is retrieved from a content server and is displayed to the app user. User interactions with this content are then reported to the app analytics infrastructure. Unlike problems considered in related prior work, such analytics could convey a wealth of sensitive information—for example, about an app user's political beliefs, dietary choices, health conditions, or travel interests. To provide rigorous privacy protections for this information, we design a differentially-private solution for such data gathering.**

**Our first contribution is a conceptual design for data collection. Since existing approaches cannot be used to solve this problem, we develop a new design to determine how the app gathers data at run time and how it randomizes it to achieve differential privacy. Our second contribution is an instantiation of this design for Android apps that use Google Firebase. This approach keeps privacy logic separate from the app code, and uses code rewriting to automate the introduction and evolution of privacy-related code. Finally, we develop techniques for automated design space characterization. By simulating different execution scenarios and characterizing their privacy/accuracy trade-offs, our analysis provides critical pre-deployment insights to app developers.**

## I. Introduction

Android apps commonly use app analytics infrastructures provided by companies such as Google and Facebook. For example, Google Firebase [1] is used by 48% of the thousands of apps investigated in a recent study [2]. Such analytics machinery gathers a wealth of data about the app user, typically without clarity or guarantees on the intended use of this data. Millions of app users are regularly subjected to such poorly-understood/regulated data gathering and analysis. Powerful data mining can be applied to this data and to other sources of information about the same user, giving significant powers of inference and learning to entities whose intentions are unclear at best and malicious at worst. Not surprisingly, both customers and governments are becoming increasingly concerned about the privacy implications of such widespread data gathering.

In this technological and societal context, a promising direction for research and practice is to design *privacy-preserving data gathering*. While many mechanisms have been proposed to achieve this goal, in this work we focus on *differential privacy* [3]. This theoretical approach has emerged as a leading algorithm design framework for privacy-preserving data analysis, due to its rigorous privacy definitions, extensive body of powerful algorithmic solutions, and a number of practical applications in industry and government. With differential privacy, useful statistics can be collected about a population, without revealing details about any individual member of the population. These privacy protections are achieved by adding random noise to the raw data, and reporting and analyzing only this perturbed data. This approach is appealing as it provides well-defined probabilistic guarantees about the privacy protection of individual user's data, even in the presence of unknown additional data about this user, and regardless of any powerful and unanticipated statistical analyses that may be applied to the data by adversarial entities.

### A. Fixed vs Dynamic Data

A common use of mobile app analytics is to track frequencies of *fixed events*—for example, views of GUI screens— which are then reported to the analytics server. The set of such events is fixed ahead of time, before app deployment, and is the same for all app users. Some prior work has considered privacy-preserving designs for such data gathering [4]–[6].

However, there is an even more important category of data that has not been considered in any prior work. In this scenario, *dynamically-created content at a content server* is retrieved by the app and displayed to the app user. User interactions with this content are then reported to the mobile app analytics infrastructure (i.e., to an *analytics server*), and ultimately to the app developers. Unlike fixed events in which app structural information is typically gathered, here *content-related events* can be used to attempt inferences about the app user. For example, as illustrated later via several real apps we studied, this type of analytics could potentially convey a wealth of information about a user's political beliefs, her dietary choices, her health conditions, or her travel interests. Furthermore, this data could be combined with widely-available data from other sources (e.g., public government databases; consumer data from business analytics companies) to draw even more powerful inferences about the user. Note that such inferences could be attempted not only by unknown privacy adversaries, but also by the analytics server and the app developers themselves.

Privacy protection for such content data is arguably more important than protecting fixed events such as GUI screen views. Consider this question: which is more revealing, (1) that the app user tapped a GUI button to label an article as "favorite", or (2) that the user did this for a particular article, uniquely identifiable by a public article id, in which the topic was a sensitive subject such as anti-government protests? Would an app user be equally comfortable with (1) or (2) being shared with the unknown developers of some app and the analytics servers under the control of Google? We believe that the second scenario is much more sensitive, but no existing work has considered how to add privacy protection in mobile apps that gather such data.

**Problem statement.** Our goal is to design a differentially-private solution for such data gathering, in a way that (1) preserves the privacy of individual app users, while at the same time (2) provides accurate statistics over the entire population of users. We consider this to be a software transformation problem: given an app that already uses mobile app analytics of dynamic content, how should it be modified to introduce differential privacy protections?

### B. Challenges

**Challenge 1.** Unlike differentially-private data collection for a pre-defined set of fixed events, the problem we consider has two new features that have not been addressed in existing work. First, the content items retrieved from the content server by one app instance could be different from the ones retrieved by another app instance. Thus, each individual app user locally observes and interacts with a different set of items, compared to other users. Further, the local behavior of an app interleaves two types of state changes: (1) content retrieval from the content server, and (2) user interactions with this content, resulting in event reports to the analytics server. Existing designs for differentially-private data analysis do not handle these two novel aspects of the collection process.

**Challenge 2.** There could be substantial effort to introduce and maintain the code that implements the differential privacy mechanisms. Given an app with mobile app analytics, the introduction of such privacy-preserving code presents a software evolution challenge. When such functionality is introduced for the first time, this may require code changes in various parts of the program, at places where analytics-related code already exists. As the app evolves, changes to privacy-preserving code may need to be introduced to keep it "in sync" with the corresponding analytics code. Such code changes require programmer effort and are error prone.

**Challenge 3.** Effective integration of differential privacy requires pre-deployment analysis and calibration of a fundamental trade-off: accuracy vs privacy. Stronger privacy guarantees require more random noise, which leads to lower accuracy of population-wide statistics. For an app developer who introduces differentially-private data gathering in her app, it is important to characterize and tune the effects of various design choices to achieve practical trade-offs, and to do this with little effort.

### C. Contributions

Our work makes the following contributions to address the challenges outlined above.

**Contribution 1.** We propose *a new differentially-private data analysis for dynamic content in mobile apps* (Section IV-C). The developed conceptual design includes an abstract problem statement and a mathematical definition of how the data is gathered and processed in the app and in the analytics server. The approach handles both problems outlined above: it accounts for the differences in local information for each app user, and incrementally handles the interleaving between content retrieval and user-triggered events on this content.

**Contribution 2.** As a proof of concept, we develop *an instantiation of this design for Android apps that use Google Firebase* (Section V-A). Our approach keeps all differential privacy logic separate from the original app code, and uses code rewriting to automate the introduction and evolution of privacy-related code. The rewriting introduces calls to a separate run-time layer which wraps the Firebase analytics libraries. The resulting solution makes it easy to add differential privacy functionality to an existing app and to evolve it with the evolution of the app.

**Contribution 3.** We develop techniques for *automated design space characterization* (Section V-C). By simulating different execution scenarios and characterizing the resulting privacy/accuracy trade-offs, our analysis provides critical pre-deployment insights to app developers.

## II. Mobile App Analytics

### A. Google Firebase

Developers of Android apps can use several analytics infrastructures to record and analyze run-time app execution data. Currently the most popular such infrastructure is Google Firebase ("Firebase" for short) [1]. Based on recent statistics of popular apps, Firebase is used by 48% of the analyzed apps [2]. While focused on Firebase, the core techniques developed by our work also apply to other app analytics frameworks such as Facebook Analytics [7] and Flurry [8]. The operation of app analytics is illustrated in Figure 2; details of this figure will be discussed shortly. In this paper we focus on *event frequencies*, which are the most basic and popular form of mobile app analytics provided by Firebase and similar infrastructures.

There are two broad categories of data that are collected via app analytics. One category is *app-specific data*. One simple example of such data are events of the form "the app user has viewed screen $s$" where $s$ is a structural element of the app (e.g., an Android activity). The set of all such possible events is known ahead of time, before the app is distributed to users. Frequencies of such events, gathered over a large number of app users, can help the app developers understand what are the most common features of the app, and how users typically navigate through app functionality.

A second category of data—the one studied in our work—is *dynamic user-specific data*. Such data is not known ahead of time before app deployment; it is dynamically created over

time and the user's interactions with it are logged by mobile app analytics. Such data is much more revealing. For example, consider the `infowars` app which was included as a subject in our study (Section VI). The dynamic content here is a set of news articles posted at the controversial `infowars.com` website. Each article has a unique publicly-available identifier inside the app. The articles available at the website changes over time. When the app user views an article retrieved from the website, and clicks the "Favorite" button, an event is sent to Firebase to log this action. This event includes the identifier for the article. Such information can be used to infer the political inclinations of the specific app user being tracked.

As another example, app `cookbook`, which was also used in our study, allows users to browse and view a large collection of recipes. The content items are the recipes. When the user selects a recipe to view its details, this event is sent to Firebase together with the recipe identifier. By observing such information, it is possible to infer information about user's diet (e.g., vegetarian or gluten-free) and underlying health conditions (e.g., high blood pressure, which is correlated with low-sodium recipes). As a last example, consider two of the other apps we studied: `reststops` and `opensnow`. The first one displays details about rest stops along highways. The second one shows information about skiing locations. Using Firebase, the apps collect the ids of viewed content items. This information could potentially be used to infer the user's travel interests and plans.

Firebase does have high-level guidelines to avoid collecting user-identifiable information [9]. However, there is no enforcement of such guidelines. Even if such protections were rigorously defined and enforced, the "leaking" of user-specific information still makes it possible to construct various privacy attacks by unethical business entities, malicious actors, disgruntled employees, or government agencies. For example, techniques such as anonymization cannot provide strong privacy guarantees and are susceptible to privacy attacks that utilize additional sources of information external to the anonymized data collection (e.g., [10], [11]).

### B. Example

Figure 1 shows a code example derived and simplified from the `cookbook` app. Class `SparkRecipesBaseActivity` has a field `f` which stores a reference to a `FirebaseAnalytics` object. When a "select" event happens on a recipe, the app code calls `DoFireBaseSelectContent` and provides the string id of this recipe as parameter `id`. Inside the method, a bundle is created to store this id, associated with a pre-defined constant `ITEM_ID` defined by Firebase. The call to `logEvent` then sends an event of type "select content" to the Firebase analytics server. The recipe id is provided as part of the logged event. Note that all recipe ids provided by the content server are public knowledge and are easily mapped to the actual recipe details.

Many recipes are retrieved from the content server and their summaries/images are displayed in the app, but only a subset of these are selected by the user for detailed view and are recorded by Firebase via `logEvent`. Specifically, the recipe summaries and images are displayed in a `ListView` (Android's GUI widget for a list), and clicking a list item displays the details of the recipe and records the "select" event by calling `DoFirebaseSelectContent`. The data retrieval from the content server is done via HTTP. The actual data uses JSON, as illustrated by method `ProcessMainScreenData` in Figure 1. The parameter of this method is the string representation of the recipe data, obtained via HTTP from the server. The information about individual recipes is retrieved from this data, including the recipe id. This information is then used to populate GUI widgets that display recipe summaries and images.

```
class SparkRecipesBaseActivity  ... {
 FrebaseAnalytics f;
 public void onCreate(...) {
  ...
  f = FirebaseAnalytics.getInstance(this);
 }
 public void DoFireBaseSelectContent(String id) {
  Bundle b = new Bundle();
  b.putString(FirebaseAnalytics.Param.ITEM_ID,id);
  f.logEvent(FirebaseAnalytics.Event.SELECT_CONTENT,b);
 }
}
class MainFragment ... {
 public void ProcessMainScreenData(String data) {
  ...
  JSONObject jsonRecipe = ...;
  long id = jsonRecipe.getLong("recipe_id");
  ...
 }
}
```

Fig. 1. Code derived from the `cookbook` app.

## III. Differential Privacy

Differential privacy [12] is a rigorous theoretical approach that allows systematic design of privacy-preserving data collection. Both theoretical foundations [3] and practical applications in industry/government [13]–[16] have been studied extensively in the last decade. Intuitively, differentially-private data gathering and analysis aim to provide accurate estimates of population-wide statistics, while "hiding", in a well-defined probabilistic sense, data from individuals who are members of this population. As a simple example, with differential privacy, it becomes possible to estimate accurately the total number of app users who have labeled a certain news article as favorite, while it is not possible to assert with high certainty whether any particular app user has done so.

**Example.** We illustrate this approach with a key exemplar problem that has been studied extensively [13], [17], [18]. (The next section describes in detail the more general problem we solve, and the threat model assumed by that solution.) Consider some publicly-known data dictionary $\mathcal{V}$. Suppose we have $n$ users $u_1, \ldots u_n$, and each user $u_i$ has a single private data item $v_i \in \mathcal{V}$. The problem is to determine, for every $v \in \mathcal{V}$, how many users $u_i$ have $v_i = v$. We would like to estimate the population-wide frequency $f(v)$ of each $v$, following the so-called model of *local differential privacy*. In this model, any data shared by the user is considered to be potentially-abused by external observers, including the analytics server.

The differential privacy scheme perturbs the local information of each user. If this perturbation is designed correctly, malicious actions of the analytics server or the clients of this server cannot break the differential privacy guarantee (this guarantee is described shortly). A differentially-private version of this analysis will randomize the local item $v_i$ of user $u_i$ using a *local randomizer* $R : \mathcal{V} \to \mathcal{P}(\mathcal{V})$. Here $\mathcal{P}(\ldots)$ denotes the power set. Thus, the user reports a set of events $R(v_i) \subseteq \mathcal{V}$ to the analytics server. After such data is collected from all users, for every $v \in \mathcal{V}$ the server computes $|\{i : v \in R(v_i)\}|$ and uses it to estimate the true frequency $|\{i : v = v_i\}|$.

The randomizer creates an output from which it is difficult to determine, in a probabilistic sense, what was the randomizer's input. For every possible randomizer output $z \subseteq \mathcal{V}$ and for any two $v_1$ and $v_2$ from $\mathcal{V}$, the probability that $R(v_1) = z$ is close to the probability that $R(v_2) = z$. Thus, anyone observing $z$ cannot distinguish with high probability the case where the real data was $v_1$ from the case where the real data was $v_2$. Such *indistinguishability* is the essence of differential privacy. In the above definition, two probabilities are considered close to each other if their ratio is bounded from above by $e^{\epsilon}$, where $\epsilon$ is a parameter defining the strength of indistinguishability. Values of $\epsilon$ in prior work range from 0.01 to 10 [19]. In related work that uses the style of randomization we employ, exemplar values are $\ln(3)$, $\ln(9)$, and $\ln(49)$ [13], [18]; for example, the last two values are used in the first stage of a two-stage randomizer [13]. Larger values of $\epsilon$ weaken the indistinguishability guarantee, but increase the accuracy of estimates since the randomizer needs to add less noise to ensure this guarantee.

A well-known approach to meet the requirements of this definition is the following [13], [18]. The local data of user $u_i$ is represented as a bitvector, with one bit for each element of $\mathcal{V}$. For the item $v_i$ held by $u_i$, the corresponding bit is 1; the rest of the bits are 0. The randomizer takes this bitvector as input and for each bit, independently from the other bits, inverts the bit with certain probability dependent on $\epsilon$. The resulting perturbed bitvector is the output of the randomizer and is a representation of set $R(v_i)$. This set is shared with the analytics server. When the analytics server receives all user data $R(v_i)$, it computes a global frequency $|\{i : v \in R(v_i)\}|$ for every $v$. This frequency is then calibrated to account for the presence of noise over all $n$ users, which produces the final frequency estimate for $v$. A key observation behind this approach is the following: when data is collected from a large number of users, the individual noises cancel each other out in a probabilistic sense, leaving a final estimate that is close to the actual value being estimated.

## IV. Problem Definition and Solution Design

### A. Problem Statement

A content server (e.g., a news server, a recipe server, a live events server) continuously delivers dynamic content. In our model, this content is a stream of items, each identified by a unique id. Without loss of generality, we will represent the stream as a sequence of integer ids $c_1, c_2, \ldots$ where $c_j$ is the integer id of the $j$-th content item. The app running on the device of user $u_i$ interacts with the server and retrieves a subset of these ids $c_j$. This retrieval could be done, for example, based on time of content publishing (e.g., upon startup, the `infowars` app retrieves the ids and titles of the latest 50 news articles) or based on preset user preferences. The set of content items retrieved by user $u_i$ will be denoted by $C_i$ and will be referred as the *local dictionary* of user $u_i$.

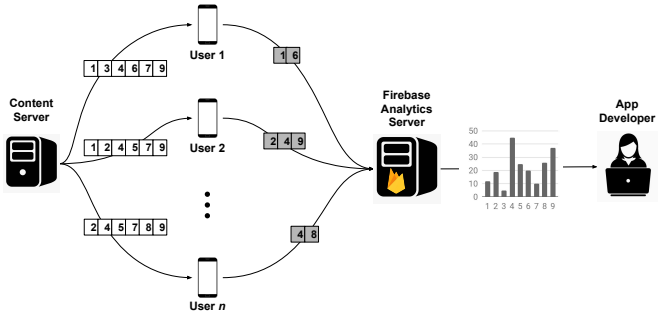


Fig. 2. Data collection using Firebase, without differential privacy.

When the user interacts with the content displayed by the app, user actions can trigger analytics events (e.g., making "favorite" a news article with id $c$, or viewing the details of a recipe with id $c$). To simplify the discussion, we will consider a single type of event; generalizing to multiple event types is trivial. We will abstract the set of app-user-triggered events via a subset $E_i \subseteq C_i$ of the local dictionary. If $c \in E_i$, this means that an event was triggered by the app user on content item $c$. For simplicity, we will often use $c$ to denote both the content item and the event that occurred on it.

**Frequency analysis.** For every item $c$ published by the server, our goal is to estimate the number of users that triggered an event on $c$: that is, the frequency $f(c) = |\{i : c \in E_i\}|$. Such frequency information is useful to the content provider to understand how the user population interacts with published content, for example, which items are most popular. In particular, such data collection is a key functionality of the Android apps we have studied and used for our evaluation: given some set of content items retrieved from a content server, the app reports events related to these items to the Firebase analytics server. App developers (working on behalf of content providers) can then use standard Firebase tools to obtain histograms of this data. Another motivation for considering this problem is that the underlying solution techniques play a key role in other analyses: e.g., heavy hitters [17], estimates of distributions [20], and clustering [21]. Future work could apply these more sophisticated techniques to privacy-preserving analysis of dynamic content in mobile app analytics.

**Example.** The process described above is illustrated in Figure 2. Here $\mathcal{V} = \{1, \ldots, 9\}$. The figure shows the set $C_i$ of content item ids retrieved by each user; for example, $C_2 = \{1, 2, 4, 5, 7, 9\}$. Each user's actions on these items results in a set of events $E_i \subseteq C_i$, each of which is associated with a unique content item id; for example, for $u_2$ we have $E_2 = \{2, 4, 9\}$, shown in gray in the figure. These are shared

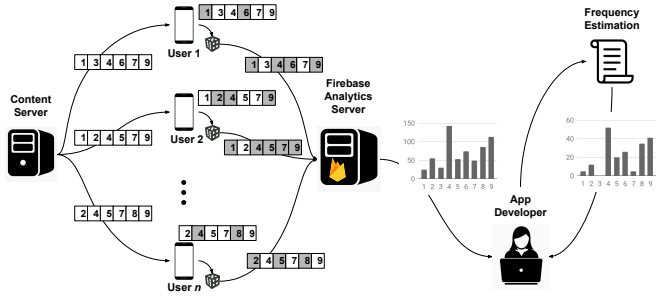


Fig. 3. Data collection using Firebase, with differential privacy.

with Firebase and used to compute population-wide frequencies.

Without differential privacy, sets $E_i$ are simply reported to the analytics server and then used to compute the frequencies $f(c)$ directly. With differential privacy, we introduce randomization: for each $c \in C_i$ (i.e., every element $c$ of the local dictionary of $u_i$), the goal is to provide probabilistic indistinguishability between two conclusions: (1) $c \in E_i$ and (2) $c \notin E_i$. In other words, for every local content item $c$, a privacy adversary should not be able to tell whether the item participated in an event or not. This will be achieved with a local randomizer $R$ such that $R(E_i)$ is reported to the analytics server, as opposed to the raw data $E_i$. Using the set of all reported $R(E_i)$, the analytics server produces estimates $\hat{f}(c)$ of the real frequencies $f(c)$.

### B. Threat Model

The design and implementation of the differentially-private scheme are fixed before the data collection starts and are publicly known to app users and privacy adversaries. This includes knowledge of $R$ and the parameter $\epsilon$ used by it; as typical in differential privacy, the same $\epsilon$ is used for all users. A key assumption is that the app code faithfully implements the design: it performs the randomization as expected, sends the randomized data to the expected analytics server, and does not leak the raw private data in any other way. This can be achieved, for example, by providing open-source implementations or by code certification performed by government agencies or privacy experts. The content server and the analytics server are not trusted. In particular, the content server can track the set of items $C_i$ delivered to a particular user $u_i$, or even provide some specific content chosen as part of a privacy attack. Thus, the approach assumes that for each user $u_i$, the set $C_i$ of retrieved items is publicly known to any malicious party. The privacy guarantee is with respect to the subset of events $E_i \subseteq C_i$ that occurred locally on the user's device. $E_i$ remains private under this model, as defined precisely below. The data shared with the analytics server is $R(E_i)$. From this data the potentially-malicious analytics server, even if colluding with the content server and even if using additional unknown data sources about this user, cannot construct a high-confidence guess as to whether any particular $c \in C_i$ is an element of $E_i$ or not.

### C. Design of a Differentially-Private Scheme

To achieve the desired privacy, we use the following randomizer design. The private local data of user $u_i$ is represented as a a bitvector of length $|C_i|$. Each bit corresponds to some $c \in C_i$. If $c \in E_i$, the bit is 1; otherwise, the bit is 0. This vector is the input to the local randomizer. For each bit, independently from all other bits, the randomizer preserves the bit with probability $p = e^{\epsilon}/(1 + e^{\epsilon})$ and inverts it with probability $1 - p$. This approach provably provides $\epsilon$-indistinguishability between any two vectors that differ in a single bit.

**Example.** The randomization process is illustrated in Figure 3. For each user, sets $C_i$ and $E_i$ are the same as shown earlier in Figure 2. In the randomizer input and output, bits in gray have value 1 and the rest have value 0. Consider, for example, user $u_2$. We have $C_2 = \{1, 2, 4, 5, 7, 9\}$, $E_2 = \{2, 4, 9\}$, and $R(E_2) = \{1, 4, 5, 7, 9\}$. In this particular case, given a bitvector with 1 bits for items 2, 4, and 9, the randomization inverted the bit for item 2. Furthermore, the 0 bits for items 5 and 7 were inverted to 1. The data that leaves the user's device and is shared with the analytics server is the bitvector for $R(E_2)$.

The privacy protection provided by such randomization can be interpreted as follows. Suppose the private local data is set $E_i \subseteq C_i$. As discussed earlier, we assume that a privacy adversary knows the local dictionary $C_i$ and the randomizer output $R(E_i)$, e.g., because the adversary can monitor the traffic to/from the user's device, or because she controls the content server and the analytics server. Furthermore, as done in all differentially-private schemes, we assume that the adversary fully knows how the randomizer is designed, including the value of $\epsilon$. This knowledge could be obtained, for example, through reverse engineering of app code.

Based on this knowledge, what conclusions can the adversary draw about any $c \in C_i$? The indistinguishability property applies in two ways. First, suppose that $c \in E_i$. From the point of view of the adversary, the probability that the randomizer input was $E_i$ is close to the probability that the randomizer input was $E_i \setminus \{c\}$; more precisely, the ratio of these probabilities is bounded by $e^{\epsilon}$. As a second case, now suppose that $c \notin E_i$. In this case the adversary cannot distinguish the case where the randomizer input was $E_i$ from the case where the randomizer input was $E_i \cup \{c\}$. Overall, probabilistically the following two conclusions are indistinguishable from each other: "an event happened on content item $c$" and "an event did not happen on content item $c$". For example, for any particular news article, it is not possible to tell with high certainty whether or not the app user marked this article as favorite. Similarly, for any particular recipe, it is not possible to have high confidence whether the user did or did not view the recipe details.

All bitvectors for $R(E_i)$, for all users $i$, are collected by the server. For any $c$, the number of all occurrences of $c$ in the reported sets $R(E_i)$ is a biased estimator of the real frequency $f(c) = |\{i : c \in E_i\}|$. To obtain an unbiased estimator, additional calibration needs to be performed as shown by the "Frequency Estimation" step in Figure 3. Specifically, for each $c$, let $n_c$ be the number of sets $C_i$ containing $c$, and let $m_c$

be the number of sets $R(E_i)$ containing $c$. The expected value of $m_c$ is $f(c)p + (n_c - f(c))(1-p)$ where $p$ is the probability to preserve (i.e., not invert) a bit in the randomizer's operation. Here $f(c)$ times the randomizers observed a 1 bit for $c$ and preserved it with probability $p$, and $n_c - f(c)$ times observed a 0 bit and inverted it to a 1 with probability $1-p$. Thus, we can estimate $f(c)$ by $\hat{f}(c) = ((1+e^{\epsilon})m_c - n_c)/(e^{\epsilon}-1)$. The accuracy of this estimate depends on the number $n_c$ of users whose local dictionary contains $c$, as well as on the value of $\epsilon$. It is important to characterize the accuracy as a function of concrete values of these two parameters, as part of pre-deployment tuning of the approach. Later we provide further details on how to perform this characterization.

### D. Limitations

While this approach achieves differential privacy for the targeted problem, it is important to understand its limitations. As described earlier, it is assumed that the app code implements the design correctly and does not leak the private data by other means. If an app developer ensures this, she can legitimately claim to have privacy-by-design data collection, which is a significant improvement over the state of practice in mobile app analytics. This not only makes the software more appealing to users, but it may align with government requirements for privacy protection. Providers of mobile app analytics infrastructures (e.g., Google and Facebook) could also benefit: if they collect only randomized data, this provides protection for them against data breaches or unlawful employee actions.

A second limitation is that we focus on an important but narrow problem: obtaining frequency estimates for events. This is a core functionality for infrastructures such as Firebase, but many other interesting analyses could also be considered: for example, user behavior flow analysis, correlation analysis, clustering, etc. Such techniques require more sophisticated differential privacy techniques. While our work may provide some building blocks for such techniques, ultimately the question of how to perform differentially-private mobile app analytics is still open and requires significant follow-up efforts.

Our approach assumes that the local dictionary $C_i$ is publicly known, as the content server can track the data being retrieved by a particular user. However, this information itself could be sensitive—for example, it could be based on user settings, profiles, or past behaviors. In future work it would be useful to introduce privacy protections for the local dictionary as well, for any adversaries that do not collude with the content server.

We only develop a simplified exemplar implementation of this design for Firebase (described in the next section). The implementation does not handle the full complexity of Firebase (e.g., multiple event types) and lacks automated analysis for identification of code locations for retrieval and logging of dynamic content. Such static analysis and subsequent automated code refactoring are important targets for follow-up work. In addition, adapting this approach to other popular app analytics frameworks such as Facebook Analytics is an open problem.

**Algorithm 1:** Randomization of observed events

```
1  Function init():
2      C ← ∅
3      E ← ∅
4      R ← ∅
5      num_events ← 0
6  Function retrieve(c):
7      C ← C ∪ {c}
8  Function event(c):
9      if c ∈ E then
10         return
11     end
12     E ← E ∪ {c}
13     with probability p, R ← R ∪ {c}
14     num_events ← num_events + 1
15     if num_events = k then
16         for c ∈ C \ E do
17             with probability (1 − p), R ← R ∪ {c}
18         end
19         report C and R
20     end
```

## V. Implementation for Firebase Apps

To realize the conceptual design above, we have implemented a proof-of-concept instantiation for Android apps that use Firebase. The implementation considers three kinds of run-time state changes, and reacts to them via our code instrumentation. In essence, we have developed an incremental randomizer through a run-time layer that wraps the Firebase APIs and is called by instrumentation inserted in the app code.

### A. Overview

The instrumentation invokes three helper functions defined and implemented by us. These functions are described in Algorithm 1. The details of the actual instrumentation and how it is inserted will be described in the next subsection.

The first state change is when the Firebase infrastructure is initialized. Function `init` provides a high-level abstraction of the initialization of our implementation. We internally maintain three sets: $C$ is for the local dictionary for this user, $E$ is for the set of events for the user, and $R$ is for the output of the randomizer. Note that we do not maintain bitvectors, but the operations on these sets are equivalent to the processing of bitvectors described earlier. At the end of data collection, $C$ and $R$ are reported to the Firebase server, as described shortly.

After the initialization, two kinds of run-time state changes can be observed, in interleaved fashion. First, there could be a state change of the form "$c$ is added to $C$". This would happen when a new content item is retrieved from the content server. In our earlier example, when the app of user $u_2$ retrieves item 4 from the content server, this item is added to local set $C_2$. This functionality is implemented by `retrieve` in Algorithm 1.

The other state change is of the form "an event is observed on some $c \in C$". Function `event` in Algorithm 1 handles

this state change. The function takes as input the content item on which the event occurred. As we consider $E$ as a set rather than a multi-set, each such item $c$ is recorded once by adding it to $E$ and, with probability $p$, adding it to the randomizer output set $R$. Recall that in our conceptual design $p$ is the probability of preserving (rather than inverting) a bit in the bitvector representing $E$. We also increment a count of the number of events that have been observed so far. In our exemplar implementation, when this count reaches a pre-defined threshold $k$, the data collection completes and the data is sent to the analytics server. This threshold is publicly known, the same for all users, and decided before data collection starts.

Before $C$ and $R$ are sent to the server, all 0 bits in the conceptual bitvector have to be considered and possibly inverted. Equivalently, each $c \in C \setminus E$ should be included in $R$ with probability $1-p$. The resulting sets $C$ and $R$ can then be sent to Firebase item-by-item using the standard `logEvent` API.[1] As a matter of practical implementation, two new event types can be used, one for $C$ and one for $R$, and the items in these sets can be recorded by Firebase under these artificial event types. The post-processing by the app developer, as shown in Figure 3, can use the information recorded by Firebase to reconstruct all sets $C_i$ and $R_i$ for all users $i$, and then compute the estimates $\hat{f}(c)$ as described at the end of Section IV-C.

### B. Code Instrumentation

From the point of view of software evolution and maintenance, it is desirable to avoid the introduction of code specific to our differentially-private data gathering. We aim to easily incorporate our machinery into an existing app via *code instrumentation* inserted by a code rewriting tool. The code locations where the instrumentation should be inserted are defined by a lightweight specification mechanism. For each of the three abstract state changes described in Algorithm 1, the specification describes the corresponding program points where instrumentation should be inserted.

For example, for the call to `logEvent` in Figure 1, the app developer specifies the program location of this call. Our code rewriting tool replaces this call with a call to method `event(c)` defined in our run-time library, which serves as a wrapper to Firebase. Similarly, whenever a content item id is introduced for the first time in the app code, as illustrated by the call to `getLong` in Figure 1, a call to our implementation of `retrieve(c)` from Algorithm 1 is added by the code rewriting. In the current implementation and experiments, since we do not have access to the source code of the subject apps, the specification and instrumentation are at the level of the intermediate representation of the popular Soot tool for code transformation [22]. This approach keeps all privacy-related logic and code separate from the app code base and allows easy introduction/evolution of our solution into an existing app.

[1]Alternatively, $C$ could be determined by the content server and then sent by it to the analytics server. However, this complicates the functionality of the content server and the overall synchronization of data collection. Sending $C$ from the user to the analytics server is a more practical solution.

### C. Pre-Deployment Characterization of Accuracy

Before the app developer releases the differentially-private data gathering as part of her Firebase app, it is important to characterize the potential loss of accuracy. We have built infrastructure to assist with this task, and have used it in our own experiments. The process starts with a test case written by the developer to trigger the relevant content retrieval and Firebase logging. This test case is used to simulate user actions. To ensure diversity of behaviors, the test case should include randomization of GUI actions. For example, our test for the `cookbook` app scrolls a random distance through the list of recipe photos. This scrolling triggers retrieval of data from the recipe server, dependent on the amount of scrolling and the current server state. Then, a random item from the visible portion of the list is clicked, which triggers `logEvent`. Repeating these steps during one execution of the test case produces the set of retrieved items $C_i$ and the set of events $E_i$. In our experience, writing such test cases is straightforward, even for someone (like us) who is not familiar with the app. An app developer can easily create such a test case as a starting point of the characterization process; in fact, it is likely that similar test cases already exist to support correctness testing.

The $i$-th individual execution of the test case produces data for the $i$-th simulated app user, for $i \in \{1, \ldots, n\}$. In our infrastructure, we record the observed sets $C_i$ and $E_i$ in a database, to allow repeated characterization with different parameter values over the same data. From this database, an automated script generates accuracy data in the following two dimensions. First, we generate data for several values of $\epsilon$. The effects of this parameter must be studied carefully, to ensure the desired accuracy-vs-privacy trade-offs. Second, we consider additional synthetic user data. Each run of the test case could take non-trivial time and thus gathering data for a large number $n$ of simulated users is not feasible. Given all $C_i$ and $E_i$ from test case execution, we create additional user data as follows. Two different users $u_i$ and $u_j$ are picked at random. A new user $u_k$ is simulated by drawing $(|E_i| + |E_j|)/2$ random samples from $E_i \cup E_j$ to construct $E_k$. Further, $C_k$ is constructed as the union of $C_i$ and $C_j$. This process is repeated until the desired number of additional users is reached. In our experiments, we used $n = 100$ test case executions to create the initial set of 100 simulated users, and then applied this approach to allow experiments with $n$ equal to 1000, 10000, and 100000.

The script measures and reports accuracy by comparing the ground-truth frequencies $f(c)$ with their estimates $\hat{f}(c)$. Various metrics could be used for this comparison. In our experiments we consider one such choice: a normalized version of the $L_1$ distance (i.e., Manhattan distance) between the frequency vectors: $\sum_c |f(c) - \hat{f}(c)| / \sum_c f(c)$. Other choices are certainly possible and easy to implement.

Given this characterization, the app developer could answer various questions. For example, for some expected number of app users and some targeted accuracy, what value of $\epsilon$ should be used? This value can be automatically inferred from the simulated data and embedded in the app with no effort from the

developer. As another example, how does the accuracy change if the real number of users differs from the expected number? As yet another example, what are the effects on accuracy if data is collected over an extended period of time and thus local dictionaries do not overlap much across users? (This last question is discussed further in Section VI-D.) By considering these and similar questions, developers can fine-tune the data collection before releasing/updating the app.

## VI. Experimental Evaluation

The privacy-preserving analysis for Firebase Analytics event reporting was done by analyzing and rewriting 9 apps. All experiments were performed on a machine with Xeon E5 2.2GHz processor and 64GB RAM. The apps were instrumented with Soot [22] and were run on Android device emulators. To implement the test cases, we used a Python wrapper [23] for the Android testing framework UI Automator [24]. Our implementation, subjects, and data are available at http://web.cse.ohio-state.edu/presto/software.

### A. Study Subjects

We identified a number of popular apps from the Google Play app store that that contain Firebase Analytics API calls. Based on our understanding of app functionality, obtained from testing in an emulator and from examination of decompiled code, we selected 9 representative apps that retrieve their contents at run time from some remote server. We registered these apps to our own Firebase backend project. We also replaced the values of `google_api_key` and `google_app_id` (stored in the app assets as string values) with corresponding values from this backend project as a quick test to ensure the correct event reporting from the apps to the Firebase analytics server.

Table I describes characteristics of the apps and of our run-time apps executions. The table shows the number of classes and methods in the app code in columns "#Classes" and "#Methods". Next, it shows measurements from executing the apps with $n = 100$ simulated users, as described in Section V-C. Recall that each such user $u_i$ has a local dictionary $C_i$. The total number of unique items retrieved from the content server over these users (i.e., the size of the union of sets $C_i$) is shown in column "#All items". Column "Avg #items" contains the average number of items in the dictionaries $C_i$. As can be seen, significant amount of content was retrieved both per user and across all users. The cost of randomization for this content was negligible, around one millisecond or less per user.

TABLE I
Study Subjects

| App | #Classes | #Methods | All items | Avg #items |
|---|---|---|---|---|
| `apartmentguide` | 1166 | 6878 | 1375 | 391.28 |
| `reststops` | 887 | 4768 | 1858 | 319.16 |
| `rent` | 1167 | 6881 | 902 | 218.29 |
| `shipmate` | 4873 | 25904 | 712 | 319.57 |
| `cookbook` | 620 | 3026 | 358 | 89.91 |
| `channels` | 189 | 973 | 294 | 122.88 |
| `infowars` | 2145 | 12483 | 226 | 50.00 |
| `loop` | 2802 | 18953 | 186 | 92.01 |
| `opensnow` | 3498 | 21455 | 168 | 127.04 |

### B. Simulating User Behavior

As described in Section V-C, our infrastructure to characterize the privacy-vs-accuracy trade-offs uses randomized test cases to gather sets $C_i$ and $E_i$ for $i \in \{1, \ldots, n\}$. Each test case is executed in a separate Android emulator and follows a common pattern. It first opens the app and performs GUI actions to the point when a certain `ListView` or `RecyclerView` widget is shown. This widget's children widgets correspond to the content items fetched from the content server. The test case then selects a child widget at random, which triggers event logging, and then goes back to the list. The test case also scrolls through the list which causes the fetching of more content. For all apps except `infowars`, this testing method created an interleaved sequence of (1) fetching new content to dynamically grow the dictionary, and (2) reporting events on elements of the current dictionary. App `infowars` is slightly different by design: instead of retrieving the data on-the-fly, it loads the 50 newest articles every time the app is opened, so the entire dictionary is built at the beginning of the test case.

An execution of a test case was terminated when $k = 100$ events of interest were observed (as shown in Algorithm 1). The `infowars` app was an exception: it was run to log a random number of at most 50 events, due to its design. The test case executions were spread out over several days to diversify the dynamically built dictionaries. Consecutive test executions for the same app resulted in dictionaries with more elements in common, while test executions on different days produced dictionaries with less similarity.

### C. Accuracy of Frequency Estimates

Given sets $C_i$ and $E_i$ for $1 \leq i \leq n$, the construction of all $R(E_i)$ and the computation of frequency estimates $\hat{f}(c)$ was performed in 30 independent trials, in order to characterize the variability of results due to randomizer behavior, with all other parameters being the same. Over these 30 measurements, we report the mean value as well as the 95% confidence interval (as suggested elsewhere [25]). In addition to $n = 100$, we also used $n = 1000$, $n = 10000$, and $n = 100000$ as described in Section V-C. In related work that uses similar kind of randomization, typical values for $\epsilon$ are $\ln(3)$, $\ln(9)$, and $\ln(49)$ [4], [5], [13], [18]. We collected data for all three values, but due to space constraints do not show details for $\ln(49)$.

We use *relative error* to measure the accuracy of estimated frequencies. This is a normalized version of the $L_1$ distance between the vector of ground-truth frequencies $f(c)$ and the vector of the estimated frequencies $\hat{f}(c)$, where $C = \cup_i C_i$:

$$\frac{\sum_{c \in C} |f(c) - \hat{f}(c)|}{\sum_{c \in C} f(c)}$$

Values close to 0 indicate that the two frequency vectors are similar to each other. In Figure 4, the $x$-axes show the names of the apps and $y$-axes show the mean relative error calculated over 30 runs. The 95% confidence intervals are small and are barely visible on top of the mean value bars.

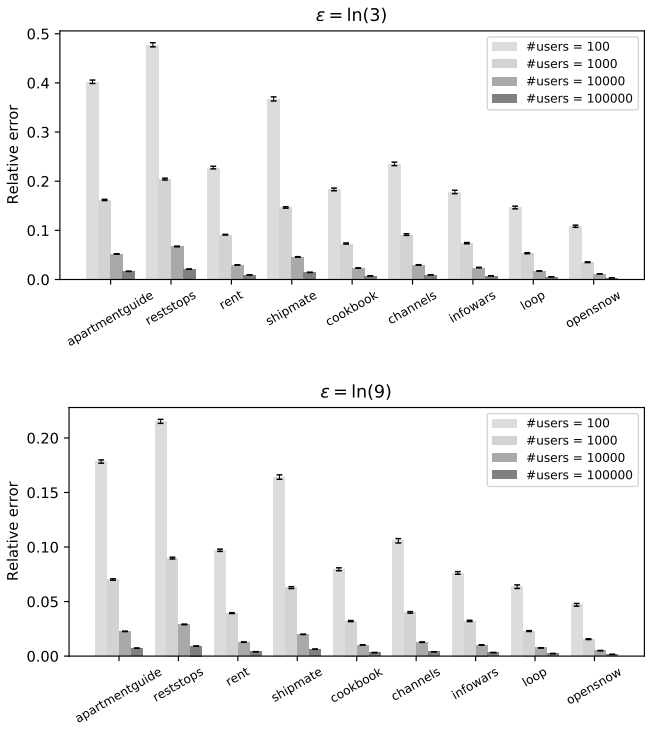


Fig. 4. Accuracy of frequency estimates. Shown are the mean values of the relative error from 30 runs, together with the 95% confidence interval.

**Summary of results.** From these results, the following conclusions can be drawn. With sufficient number of users, the overall accuracy over all parameter settings and all apps is quite high. As expected, the worst accuracy is observed for the smallest value of $\epsilon$, but even then with 10000 users the error is around 5% or less. With $\epsilon = \ln(9)$ and this same number of users, the error is around $2.5\%$ or less, and with $\epsilon = \ln(49)$ the error becomes around $1\%$ (details not shown).

Larger numbers of users can result in significant increase of accuracy. This reflects the fundamental property of differential privacy, in which larger data sets allow the noises from individual randomized contributions to "average out", leading to more accurate estimates. Having large numbers of app users is achievable in practice. For example, almost all of the top most popular apps in each Google Play category have at least 10000 installs. In fact, 5 out of the 9 apps included in our study have a number of installs above one million, and even the least-popular app in our data set has more than 50000 installs.

### *D. Effects of Content Similarity on Accuracy*

The accuracy of estimates for an item $c$ depends on the number of users $u_i$ for which $c \in C_i$. With more such users, the random noise for $c$ can cancel out better. Thus, to characterize accuracy, just considering the total number of users $n$ is not enough—it is also important to consider the degree of similarity among local dictionaries $C_i$. Everything else being equal, higher similarity would result in higher accuracy of estimates. Such effects could be due to the speed of content change in the content server: slow-changing content would result in higher similarity of local dictionaries. Similarly, the similarity is affected by the duration of data gathering, as longer duration provides more opportunities for app users to observe different content at different points of time.

To characterize these effects, we augmented our infrastructure from Section V-C to create and evaluate two subsets of the set of all $n = 100$ users. Both subsets are of size $n/2$, but one of them exhibits higher similarity among local dictionaries compared to the other one. To construct these subsets, we first computed the Jaccard similarity between all pairs of local dictionaries. Recall that the Jaccard similarity of sets $A$ and $B$ is defined as $J(A, B) = |A \cap B|/|A \cup B|$. The overall similarity of a collection $S$ of local dictionaries $C_i$ can be characterized by the average pairwise similarity, which is the average value of $J(C, C')$ for all pairs $C, C' \in S$ such that $C \neq C'$.

To create the subset $S_H$ of high-similarity local dictionaries, we started with the two users $u_i$ and $u_j$ such that $J(C_i, C_j)$ is largest among all pairs of users. The next user $u_k$ to be added was chosen such that the average pairwise similarity of $S_H \cup \{C_k\}$ is maximized. This process was repeated until we had $n/2$ local dictionaries in $S_H$. The subset $S_L$ of low-similarity local dictionaries was created in a similar fashion: starting with $S_L = \{C_i, C_j\}$ such that $J(C_i, C_j)$ is smallest among all pairs, we added $C_k$ to $S_L$ such that $S_L \cup \{C_k\}$ had minimum average pairwise similarity at each step. To ensure that the two subsets were sufficiently different, we compared their average pairwise similarities. Averaged across the 9 studied apps, the similarity of $S_H$ was $67\%$ larger than the similarity of $S_L$. We also measured how many local dictionaries, on average, contain an item $c$ occurring in a subset. Averaged across the apps, this metric was $53\%$ higher for $S_H$ relative to $S_L$. Thus, $S_L$ was significantly more diverse than $S_H$.

The question is, given the higher diversity of $S_L$ compared to $S_H$, how much accuracy loss will result from this diversity? This question is important, for example, in deciding how to gather data across real app users (e.g., for fast-varying vs slow-varying content), and how to interpret the collected data from users if the diversity of their local dictionaries is different from what was expected in pre-deployment tuning. Our characterization infrastructure allows the exploration of such questions. Figure 5 shows the accuracy measured on the two subsets $S_H$ and $S_L$. The conclusion is that higher diversity of content across users does lead to lower accuracy, but this effect is not substantial. Despite the large difference between $S_H$ and $S_L$, overall the error of the estimates does not differ significantly. This result indicates that the accuracy is resilient to the negative effects of local dictionary diversity.

## VII. Related Work

**Differential privacy.** A few examples of prior work on differential privacy were already discussed briefly [4]–[6], [13], [17], [18]. In particular, Zhang et al. [4] target the now-deprecated Google Analytics for mobile apps and use randomization to perturb each event to achieve differential privacy for event frequency reporting. Follow-up work [5] extracts and applies consistency constraints on frequencies

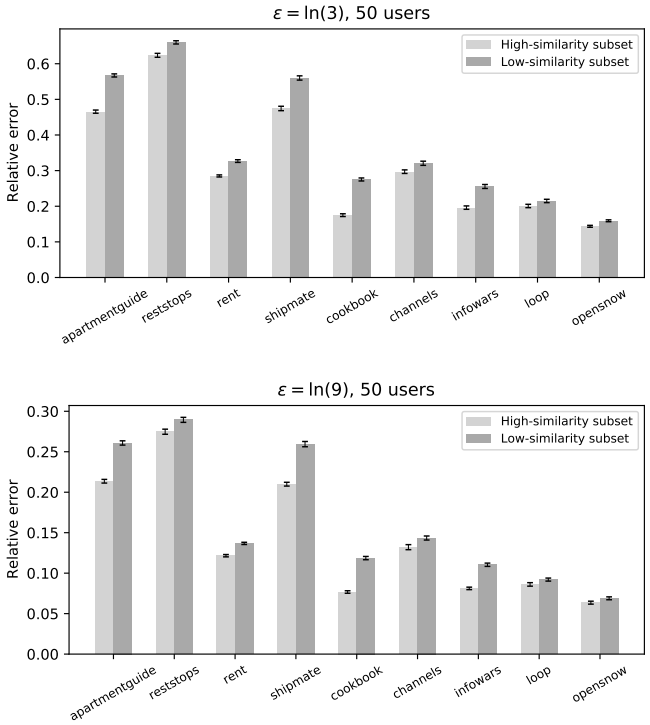


Fig. 5. Accuracy for high-similarity and low-similarity subsets of users

to improve accuracy. In both projects, the underlying data is based on the static structural properties of the program code. As discussed earlier, the problem considered in our work is different from both the single-item-per-user setting in earlier projects [13], [17], [18] and from the mobile app frequency analysis for fixed and static app data [4]–[6]. Our efforts are focused on dynamic content which is more privacy-sensitive, do not assume a pre-defined dictionary, and require handling of on-the-fly updates to local dictionaries interleaved with events on the current dictionary elements. In addition, no prior work defines a systematic way to integrate the privacy-related code with the original app code.

Although theoretical approaches have been developed for differential privacy in other problems—for example, most frequent items [17], [26], estimates of unknown distributions [20], and clustering such as $k$-means [21]—these techniques have not been applied to software analysis in general, and mobile app analytics in particular. Industry and government projects have started to apply the theory of differential privacy in practice [13]–[16], [27]. The success of these real-world efforts provides strong motivation to investigate the application of differential privacy in mobile app analytics. Our work is a step in this direction, focusing on an important category of sensitive data that has not been investigated before.

**Privacy for mobile apps.** Privacy leakage in mobile apps has also been studied extensively. Liu et al. [28] focus specifically on analytics libraries and propose the Alde tool for static and dynamic analysis of the data collection. Chen et al. [29] take advantage of the vulnerabilities in two analytics libraries to manipulate user profiles to control ad delivery. Seneviratne et al. [30] study tracking libraries in popular paid apps and find that more than half of these apps contain at least one tracker. LinkDroid [31] tackles unregulated aggregation of app-usage behaviors. Han et al. [32] employ dynamic information flow tracking to monitor sending of sensitive information. Analysis of privacy policy violations in Android apps has been studied in several projects [33]–[35]. These studies aim to prevent leaks of personal information. Our work, on the other hand, is focused on a trade-off where sensitive data could be collected legitimately over a population of users, but the data of each individual is perturbed with differential privacy guarantees.

**Privacy in software engineering.** Privacy is an important concern in software engineering practice. For example, there is increasing emphasis on privacy-by-design [36] and our work can be thought of as a particular instance of this approach. In software engineering research there is a significant body of work that considers privacy-related aspects of software testing, debugging, and defect prediction [37]–[46]. We are not aware of work in this area that employs differential privacy and benefits from its principled and quantifiable protection of users' data. Further, we focus on data collection by mobile app analytics frameworks, especially the most popular Firebase framework, and consider the dynamic data content an app user interacts with, rather than data specific to testing or debugging tasks.

## VIII. Conclusions and Future Work

The widespread use of mobile app analytics, together with the sensitive nature of the data being collected, provide strong motivation for designing privacy-preserving versions of such analytics. We consider an important but overlooked instance of this problem, where dynamic content is presented to the app user and the resulting interactions are recorded by the analytics infrastructure. Our novel differentially-private solution provides both strong privacy guarantees and high accuracy. Through the use of automated code rewriting, the approach allows practical integration in existing mobile apps and easy maintenance as the app evolves. Our studies illustrate how pre-deployment tuning of the approach can be performed, and how problem parameters affect the accuracy of the produced frequency estimates.

A natural direction for future work is to consider privacy-preserving versions of other aspects of data collection in mobile app analytics, including more complex structured and/or correlated data and more sophisticated data analyses. Such research developments could ultimately be integrated in mobile app analytics frameworks, resulting in increased privacy protections for millions of app users. In addition, complete automation of the app re-engineering effort via automated code analysis and transformation is still an open problem which presents interesting technical challenges for future research.

*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] Google, "Firebase," https://firebase.google.com, May 2020.

[2] Exodus Privacy, "Most frequent app trackers for Android," https://reports.exodus-privacy.eu.org/reports/stats/, May 2020.

[3] 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.

[4] H. Zhang, S. Latif, R. Bassily, and A. Rountev, "Introducing privacy in screen event frequency analysis for Android apps," in *SCAM*, 2019, pp. 268–279.

[5] H. Zhang, Y. Hao, S. Latif, R. Bassily, and A. Rountev, "A study of event frequency profiling with differential privacy," in *CC*, 2020, pp. 51–62.

[6] H. Zhang, S. Latif, R. Bassily, and A. Rountev, "Differentially-private control-flow node coverage for software usage analysis," in *USENIX Security*, 2020, pp. 1021–1038.

[7] Facebook, "Facebook Analytics," https://analytics.facebook.com, May 2020.

[8] Verizon, "Flurry," http://flurry.com, May 2020.

[9] Google, "Google Analytics for Firebase use policy," https://firebase.google.com/policies/analytics, May 2020.

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

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

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

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

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

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

[16] 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.

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

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

[19] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth, "Differential privacy: An economic method for choosing epsilon," in *CSF*, 2014, pp. 398–410.

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

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

[22] Sable, "Soot analysis framework," http://www.sable.mcgill.ca/soot, May 2020.

[23] "uiautomator," https://github.com/xiaocong/uiautomator.

[24] Google, "UI Automator," https://developer.android.com/training/testing/ui-automator.

[25] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *OOPSLA*, 2007, p. 57–76.

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

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

[28] X. Liu, S. Zhu, W. Wang, and J. Liu, "Alde: Privacy risk analysis of analytics libraries in the Android ecosystem," in *SecureComm*, 2016, pp. 655–672.

[29] T. Chen, I. Ullah, M. A. Kaafar, and R. Boreli, "Information leakage through mobile analytics services," in *HotMobile*, 2014, pp. 15:1–15:6.

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

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

[32] S. Han, J. Jung, and D. Wetherall, "A study of third-party tracking by mobile apps in the wild," *Univ. Washington, Tech. Rep. UW-CSE-12-03-01*, 2012.

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

[34] R. Slavin, X. Wang, M. B. Hosserni, J. Hester, R. Krishnan, J. Bhatia, T. Breaux, and J. Niu, "Toward a framework for detecting privacy policy violation in Android application code," in *ICSE*, 2016, pp. 25–36.

[35] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. Breaux, and J. Niu, "GUILeak: Tracing privacy-policy claims on user input data for Android applications," in *ICSE*, 2018, pp. 37–47.

[36] 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.

[37] S. Elbaum and M. Hardojo, "An empirical study of profiling strategies for released software and their impact on testing activities," in *ISSTA*, 2004, pp. 65–75.

[38] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," in *ASPLOS*, 2008, pp. 319–328.

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

[40] J. Clause and A. Orso, "Camouflage: Automated anonymization of field data," in *ICSE*, 2011, pp. 21–30.

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

[42] 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.

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

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

[45] 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.

[46] 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.