Introducing Differential Privacy Mechanisms for Mobile App Analytics of Dynamic Content

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Sufian Latif

Graduate Program in Computer Science and Engineering

The Ohio State University

2021

Dissertation Committee:

Atanas Rountev, Advisor

Raef Bassily

Michael D. Bond

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 potentially 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 differentially-private scheme for mobile app analytics of such content. We first present a conceptual design for this 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 the differential privacy guarantee. We then instantiate 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. Our experimental evaluation demonstrates that, with sufficient number of app users, high-accuracy frequency estimates can be obtained using the proposed techniques.

The second contribution of this work is a refined version of the above data collection. Our goal now is to ensure that information about which items were retrieved by the app is not shared with the analytics server, since this information could convey sensitive knowledge about the user. To achieve this, we need a randomization scheme which operates without knowledge of the entire set of content items that were produced (or will be produced in the future) by the content server. We design a randomization approach which maps each item into a smaller pre-defined domain, using a well-known hashing-based data summarization approach referred to as count sketch. We next develop a technique for efficient randomization of the count sketch data, by randomizing the accumulated contributions of different events rather than randomizing each of the events separately. Our evaluation shows that, despite the mapping to a hashing-based data structure which in general may reduce accuracy, frequency estimates can still be obtained with high accuracy, especially for items with high frequency.

The third contribution is a generalization of the previous contribution, where the data collection technique using a count sketch is explored under variable space budgets. The effect of space constraints on the count sketch mechanism is observed for individual content items as well as for the pairs of items reported by the users. We provide a characterization study of designing differentially-private sketching data structures for frequent items and itempairs. The experimental evaluation of the study demonstrates the trade-off between space and accuracy of the scheme.

To my family and friends

Acknowledgments

This work could not be completed without the support and assistance from a number of individuals and organizations.

First and foremost, I would express my sincerest gratitude towards my advisor, Prof. Atanas Rountev. This dissertation would not have been possible without his support, guidance and patience. I consider myself fortunate to have him as my mentor, and I could not ask for a better supervisor for my PhD.

I would also like to thank Prof. Raef Bassily and Prof. Mike Bond for being in my candidacy and dissertation committees. Also thanks to the Department of Computer Science and Engineering of The Ohio State University, for the continuous support throughout my stay here.

I have received immense help from the members of the PRESTO research group: Yu Hao, Yan Wang, Haowei Wu, and of course Hailong Zhang. They helped me in numerous ways since the first day in the lab till the end.

I am grateful to my family– my mother Sufia Khatun, my father Abdul Latif Molla, my sister Shaon Sharmin, and my brother-in-law Khalid Hussain. Their love, support, and sacrifices paved the path of my life - not just up to this point, but also into the future. I should also mention my niece Nameera Nurjahan, as I missed to be with her when she grew up from a toddler into a wonderful kid while I was away. A special thanks goes to my wife Swagota Islam for her love and patience.

Last but not least, I would like to thank the Bangladeshi graduate students of OSU. This community created a family outside my family, which turned Columbus into a home away from home.

The material presented in this dissertation is based upon work supported by the National Science Foundation under Grant CCF-1907715. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Vita

April 2012 ..... B.Sc., Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

Publications

Research Publications

Yu Hao*, Sufian Latif*, Hailong Zhang, Raef Bassily, and Atanas Rountev. Differential Privacy for Call Chain Coverage Analysis of Deployed Software (*co-leads with equal contributions). In *European Conference on Object-Oriented Programming (ECOOP)*, July 2021.

Hailong Zhang, Yu Hao, Sufian Latif, Raef Bassily, Atanas Rountev. Differentially-Private Software Frequency Profiling Under Linear Constraints. In *Object-Oriented Programming, Systems, Languages and Applications*, November 2020.

Sufian Latif, Yu Hao, Hailong Zhang, Raef Bassily, and Atanas Rountev. Introducing Differential Privacy Mechanisms for Mobile App Analytics of Dynamic Content. In *IEEE International Conference on Software Maintenance and Evolution*, September 2020.

Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev. Differentially-Private Control-Flow Node Coverage for Software Usage Analysis. In *USENIX Security Symposium*, August 2020.

Hailong Zhang, Yu Hao, Sufian Latif, Raef Bassily, and Atanas Rountev. A Study of Event Frequency Profiling with Differential Privacy. In *ACM SIGPLAN International Conference on Compiler Construction*, February 2020.

Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev. Introducing Privacy in Screen Event Frequency Analysis for Android Apps. In *International Working Conference on Source Code Analysis and Manipulation*, September 2019.

Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev. Differentially-Private Software Analytics for Mobile Apps: Opportunities and Challenges. In *International Workshop on Software Analytics*, November 2018.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Programming Language and Software Engineering	Prof. Atanas Rountev
Machine Learning and Data Mining	Prof. S. Parthasarathy
Security and Privacy	Prof. R. Bassily

Table of Contents

Page

List of Tables

List of Figures

Chapter 1: Introduction

Android apps commonly use app analytics infrastructures provided by companies such as Google and Facebook. For example, Google Firebase [27] is used by 48% of the thousands of apps investigated in a recent study [21]. 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* [17]. 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.

1.1 Overview and Outline

The goal of this dissertation is to study several problems related to *differentially-private mobile app analytics*. We aim to formulate these novel problem definitions as well as design and evaluate effective and efficient solutions for them. Next, we present the overview and outline of this dissertation.

Background. Chapter 2 provides necessary background on mobile app analytics. In particular, we focus on the key distinction between app-specific data and dynamic user-specific data, and argue that user-specific data is more revealing and thus a natural target for privacy-protection mechanisms. We then describe briefly the key ideas of differential privacy and illustrate them with a classic exemplar problem related to frequency estimation.

Privacy for dynamic content in mobile app analytics. Chapter 3 defines our first problem related to mobile app analytics. We consider a scenario where dynamic content items are retrived by an app and the user interacts with some subset of these items. This scenario is motivated by our studies of a number of real-world Android apps, whose operation follows this pattern. We then formulate the problem of differentially-private data collection to gather the population-wide frequencies of such events. We then define a data randomizer that achieves the differential privacy guarantees. To achieve ease of practical use, we propose

code rewriting techniques to introduce the privacy-preserving code into the code of an app that already uses Google Firebase. The rewriting introduces calls to our run-time wrapper around Firebase libraries. This makes it easy to add the privacy-related functionality to an existing app and to evolve it with the evolution of the app. Finally, we develop techniques to characterize the space of tunable parameters for the approach. We aim to provide insights to app developer who deploy our approach, by simulating different execution scenarios and characterizing the resulting trade-offs between privacy and accuracy. Our experimental evaluation demonstrates that, with sufficient number of app users, high-accuracy frequency estimates can be obtained using the proposed techniques.

Stronger privacy via count sketch. Chapter 4 presents a refined version of the above data collection. The approach from Chapter 3 shares with the analytics server the set of items that were retrieved by a user's app. In some scenarios this could leak sensitive information—for example, that the user engaged with the app in a certain time period, or that that particular user-defined searches of content were initiated to retrieve the data from the content server. Our next goal is to ensure that information about retrieved items is not shared with the analytics server. This requires a randomizer which operates without knowledge of the entire set of content items that were produced (or will be produced in the future) by the content server. To solve this problem, we design a randomization approach which maps each item into a smaller pre-defined domain, using a well-known hashing-based data summarization approach referred to as *count sketch*. To randomize the count-sketch data, one could randomize each individual contribution to the sketch, which occurs every time an event is observed. However, this approach has high run-time cost. Instead, we propose an efficient approach which first accumulates all contributions to the sketch, and

then randomizes the result by drawing random values from the Binomial distribution. We evaluated these techniques on data from the same Android apps that were used in the experimental evaluation from Chapter 3. Despite the expected loss of accuracy due to the use of the hashing-based count sketch, our results indicate that frequency estimates can still be obtained with high accuracy, especially for items with high frequency.

Analysis of frequent items and frequent itempairs. For real-world deployment of the data collection described in Chapter 4, there are limitations on the amount of data that can be transferred from a user to the analytics server. In Chapter 5 we consider the design of differentially-private sketching under a given space budget for the sketching data structure. This design is explored for the analysis of frequent items described in Chapter 4. In addition, we consider the problem of identifying pairs of frequently co-occurring content items, which is an instance of the more general problem of frequent itemset mining [3]. We present a characterization study that provides insights needed to design an effective scheme for differentially-private sketching data structures for frequent items and itempairs. Based on the results of this study, we propose a design for such frequency analysis under given space constraints for the sketches. Finally, we present an experimental evaluation of the proposed design and identify its intrinsic trade-offs between space and accuracy.

Chapter 2: Background

2.1 Mobile App Analytics

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) [27]. Based on recent statistics of thousands of popular apps, Firebase is used by 48% of the analyzed apps [21]. We focus our work on Firebase, but the core techniques we develop also apply to other app analytics frameworks such as Facebook Analytics [22] and Flurry [42]. In the following discussion we consider *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 studies. 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 [25]. 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

```
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");
  ...
 }
}
```

Figure 2.1: Code derived from the `cookbook` app.

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., [39, 40]).

2.1.1 Example

Figure 2.1 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 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 2.1. 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.

2.2 Differential Privacy

Differential privacy [18] is a rigorous theoretical approach that allows systematic design of privacy-preserving data collection. Both theoretical foundations [17] and practical applications in industry/government [4, 13, 20, 54] 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 [5, 20, 55]. (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^{ε}, where ε is a parameter defining the strength of indistinguishability. Values of ε in prior work range from 0.01 to 10 [32]. In related work that uses the style of randomization we employ, exemplar values are $\ln(3)$, $\ln(9)$, and $\ln(49)$ [20, 55]; for example, the last two values are used in the first stage of a two-stage randomizer [20]. Larger values of ε 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 [20, 55]. 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 probability $1/(1+e^{\varepsilon/2})$. 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 $h(v) = |\{i : v \in R(v_i)\}|$ for every v. This frequency is then calibrated to account for the presence of noise over all n users. This produces an unbiased estimate of the true frequency of v: $\left((1+e^{\varepsilon/2})h(v) - n\right)/(e^{\varepsilon/2}-1)$. 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.

Chapter 3: Introducing Differential Privacy Mechanisms for Mobile App Analytics of Dynamic Content

3.1 Introduction

As discussed in Chapter 2, 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 [58, 59, 61].

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 earlier in Chapter 2, 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?

3.1.1 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.

3.1.2 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 3.2.3). 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 3.3.1). 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 3.3.3). By simulating different execution scenarios and characterizing the resulting privacy/accuracy trade-offs, our analysis provides critical pre-deployment insights to app developers.

3.2 Problem Definition and Solution Design

3.2.1 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.

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 [5], estimates of distributions [16], and clustering [41]. 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 3.1. Here $\mathcal{V} = \{1, \ldots, 9\}$. The figure shows the set C_i of content item ids retrieved by each user; for example, $C_2 =$

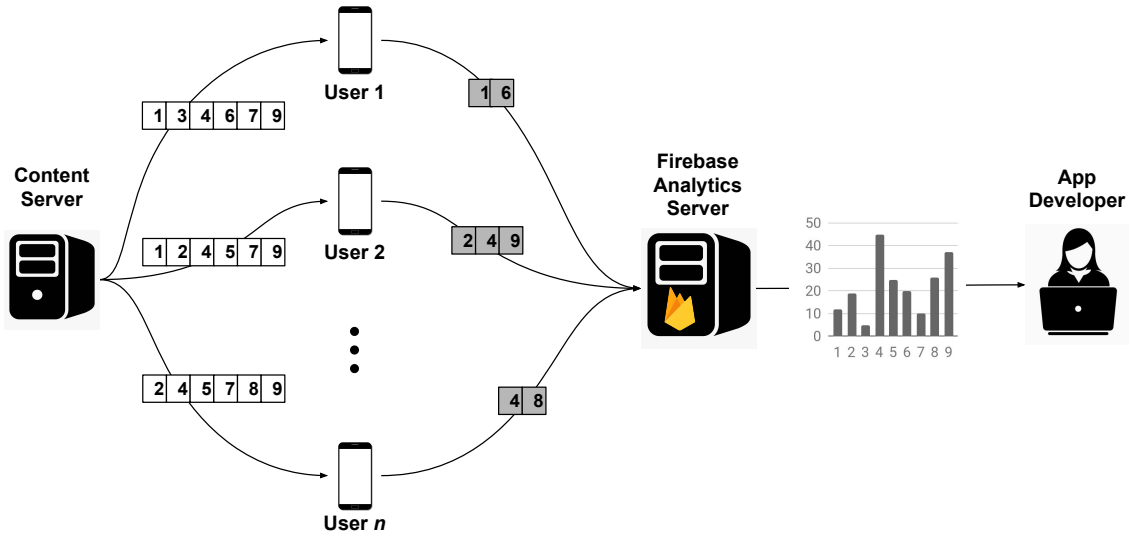

Figure 3.1: Data collection using Firebase, without differential privacy.

$\{1,2,4,5,7,9\}$. 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 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 $f(c)$. This estimated frequency $\hat{f}(c)$ may sometimes turn out to be negative

or sometimes larger than the number of users n. In such cases, any negative estimates of $\hat{f}(c)$ are set to 0, and any estimates of $\hat{f}(c)$ larger than n are set to n.

3.2.2 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 ε used by it; as typical in differential privacy, the same ε 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.

3.2.3 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

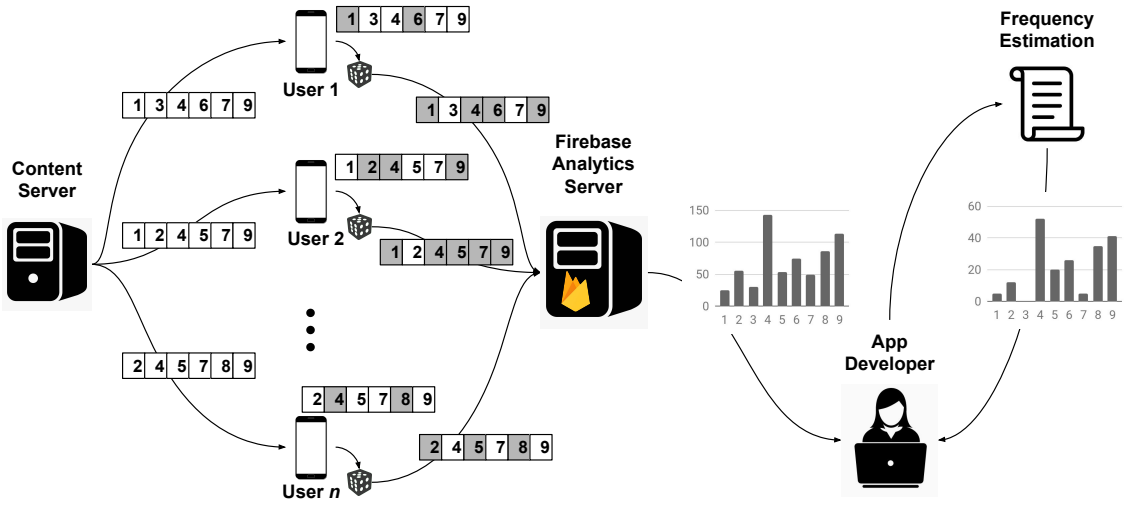

Figure 3.2: Data collection using Firebase, with differential privacy.

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 $p = e^{\varepsilon}/(1+e^{\varepsilon})$ and inverts it with probability $1-p$. This approach provably provides ε-indistinguishability between any two vectors that differ in a single bit.

Example. The randomization process is illustrated in Figure 3.2. For each user, sets C_i and E_i are the same as shown earlier in Figure 3.1. 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 ε. 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^{ε}. 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 3.2. 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 $\hat{f}(c) = ((1+e^{\varepsilon})m_c - n_c)/(e^{\varepsilon}-1)$. 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 ε. 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.

3.2.4 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 Chapter 4 we discuss refinements of our approach that 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.

3.3 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.

3.3.1 Overview

The instrumentation invokes three helper functions defined and implemented by us. These functions are described in Algorithm 3.1. 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 3.1.

The other state change is of the form "an event is observed on some $c \in C$". Function `event` in Algorithm 3.1 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

Algorithm 3.1: Randomization of observed events

```
Function init():
    C ← ∅
    E ← ∅
    R ← ∅
    num_events ← 0
Function retrieve(c):
    C ← C ∪ {c}
Function event(c):
    if c ∈ E then
        return
    E ← E ∪ {c}
    with probability p, R ← R ∪ {c}
    num_events ← num_events + 1
    if num_events = k then
        for c ∈ C \ E do
            with probability (1 − p), R ← R ∪ {c}
        report C and R
```

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

[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.

Figure 3.2, 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 3.2.3.

3.3.2 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 3.1, the specification describes the corresponding program points where instrumentation should be inserted.

For example, for the call to `logEvent` in Figure 2.1, 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 2.1, a call to our implementation of `retrieve(c)` from Algorithm 3.1 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 [50]. 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.

3.3.3 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 ε. 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 ε 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 3.4.5.) By considering these and similar questions, developers can fine-tune the data collection before releasing/updating the app.

3.4 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 [50] and were run on Android device emulators. To implement the test cases, we used a Python wrapper [1] for the Android testing framework UI Automator [26]. Our implementation, subjects, and data are available at `http://web.cse.ohio-state.edu/presto/software`.

3.4.1 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 3.1 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 3.3.3. 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

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

Table 3.1: Study subjects

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.

3.4.2 Simulating User Behavior

As described in Section 3.3.3, 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 3.1). 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.

3.4.3 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 [24]). In addition to $n = 100$, we also used $n = 1000$, $n = 10000$, and $n = 100000$ as described in Section 3.3.3. In related work that uses similar kind of randomization, typical values for ε are $\ln(3)$, $\ln(9)$, and $\ln(49)$ [20, 55, 58, 59]. We collected data for all three values of ε.

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 3.3, the x-axes show the names of the apps and y-axes show the mean relative error

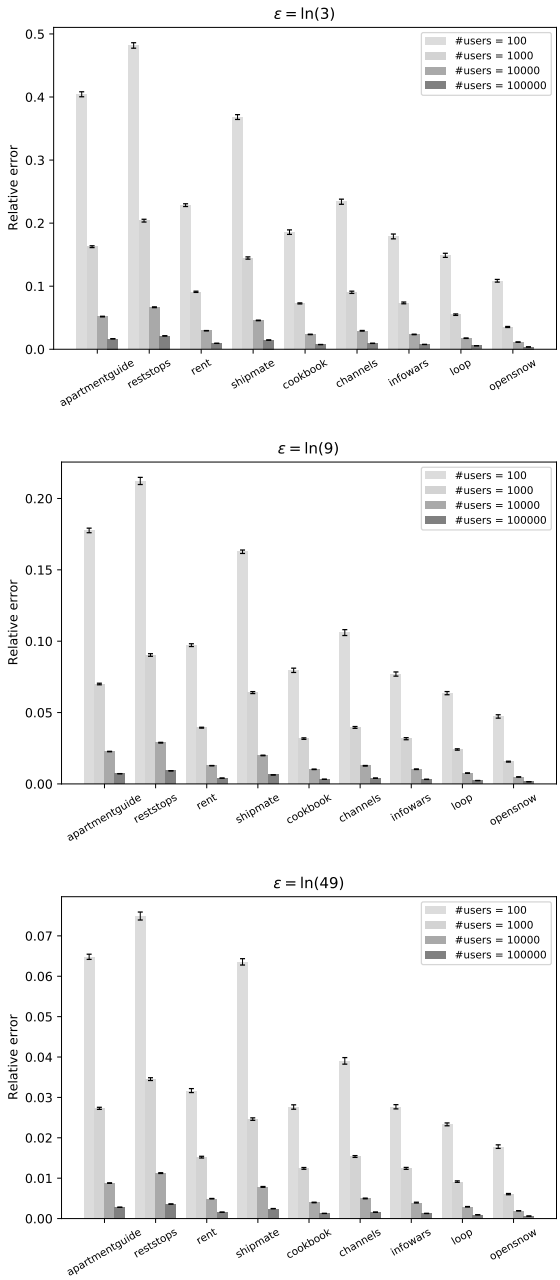

Figure 3.3: Accuracy of frequency estimates. Shown are the mean values of the relative error from 30 runs, together with the 95% confidence interval.

calculated over 30 runs. The 95% confidence intervals are small and are barely visible on top of the mean value bars.

3.4.4 Precision and Recall for Hot Items

The relative error is a good measurement of the accuracy of this differentially-private scheme since it reflects how close the estimated frequencies are to the ground-truth ones. However, it does not demonstrate how accurately the approach can detect the hot items. Finding the most popular contents is an important purpose of collecting analytics data. To determine how accurately this algorithm can detect the most frequent items, we calculated the precision and recall of the "estimated hot" elements. We computed the sets H and $\hat{H}$ of content items visited by at least 10% of the users, based on the ground-truth and estimated frequencies respectively. The precision and recall of the estimated hot items are

$$Precision = \frac{|H \cap \hat{H}|}{|\hat{H}|} \quad Recall = \frac{|H \cap \hat{H}|}{|H|}$$

The average precision and recall calculated over 30 runs are shown in Figures 3.4 and 3.5 with 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 ε, but even then with 10000 users the error is around 5% or less. With $\varepsilon = \ln(9)$ and this same number of users, the error is around 2.5% or less, and with $\varepsilon = \ln(49)$ the error becomes around 1%. The same effect can be seen on precision and recall of identification of the most frequent items. With 10000 users, this scheme can identify the estimated hot items with more than 95% precision and recall for every app.

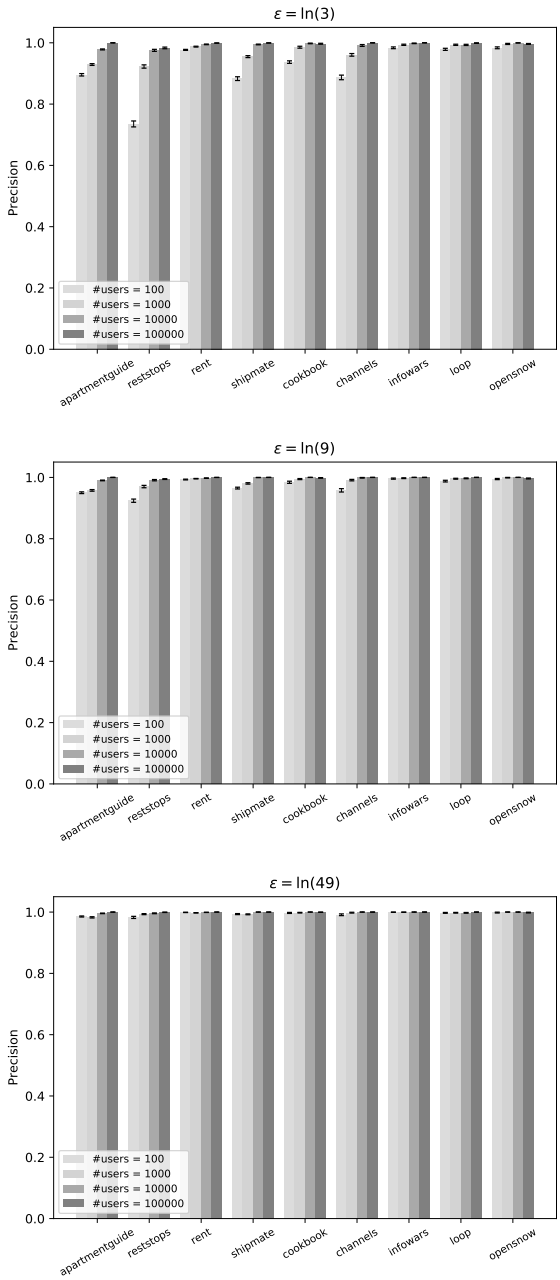

Figure 3.4: Precision of identification of hot items.

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

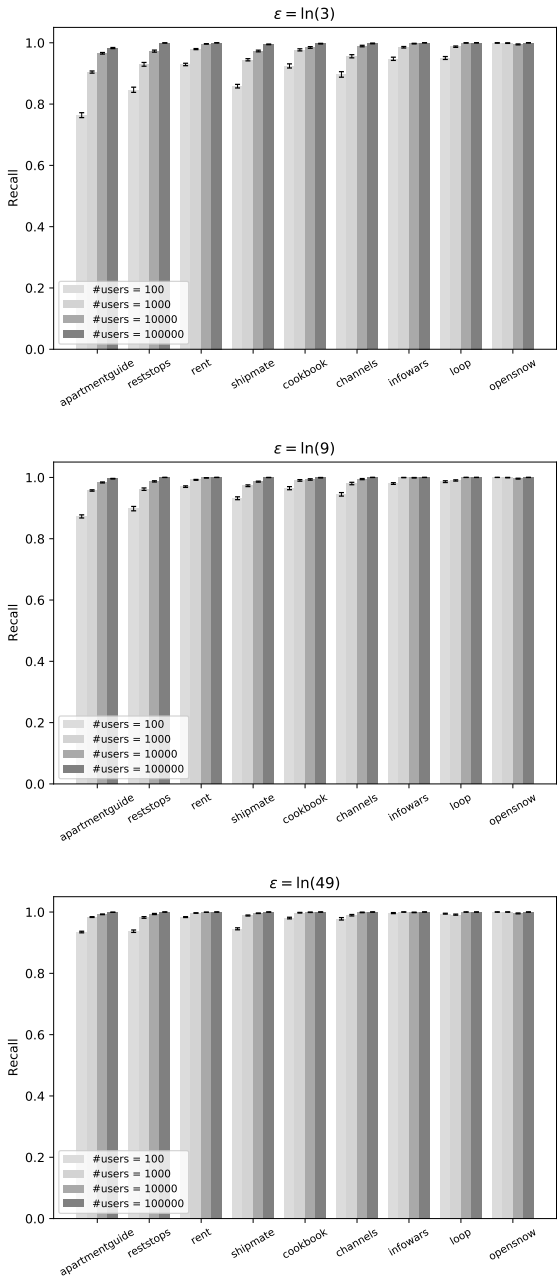

Figure 3.5: Recall of identification of hot items.

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.

3.4.5 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 3.3.3 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

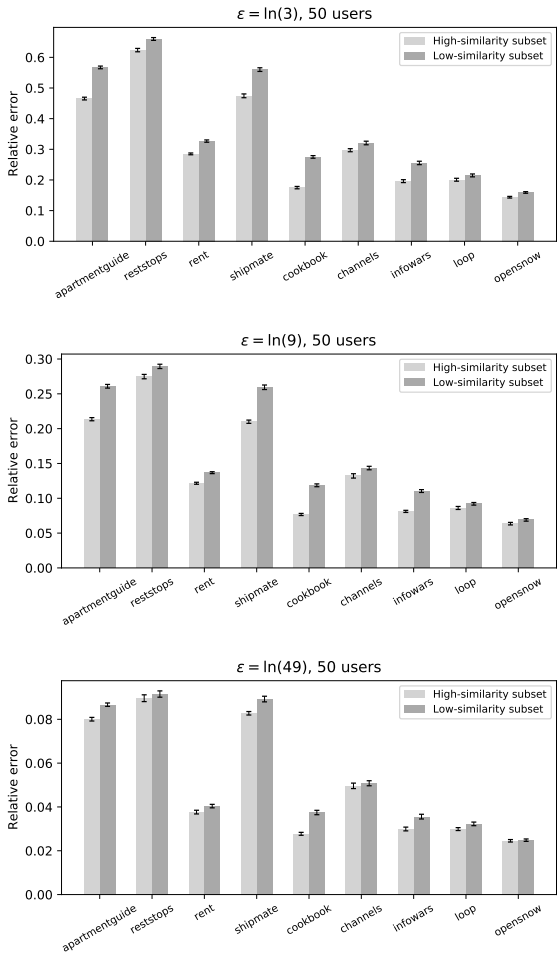

Figure 3.6: Accuracy for high-similarity and low-similarity subsets of users

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 3.6 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.

3.5 Summary

We consider an important category of mobile app analytics, 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. The conclusion from our experimental studies is that with sufficient number of app users, our approach produces high-accuracy frequency estimates.

Chapter 4: Stronger Privacy for Dynamic Content in App Analytics via Randomized Sketches

The previous chapter describes an approach to collect differentially-private software analytics data. This approach keeps track of two separate sets for every user u_i: the set C_i of dynamically retrieved content items and the set $E_i \subseteq C_i$ of items acted upon by the user. To achieve differential privacy, a local randomizer R randomizes E_i and creates a randomized response which is reported to the analytics server. The randomizer R requires both C_i and E_i to add the statistical noise to the actual data. The server requires the randomized version of E_i (i.e., $R(E_i)$) as well as *the plain unprotected data* from C_i to construct the server-side population-wide estimates.

A desirable goal for increased privacy is to hide C_i from the analytics infrastructure. By sharing the unprotected C_i, the software user "leaks" information. For example, from C_i an adversary could infer that the user engaged with the app in a certain time period (based on the lifetime of content items in C_i), or that particular user-defined searches of content were initiated to retrieve the data from the content server: for example, the user may have requested data to be retrieved based on geographic location (e.g., postal code) or specific properties (e.g., only vegetarian recipes). In addition, it may become possible to make inferences about user settings, profiles, or past behaviors.

Problem statement. Our goal is to define a differentially-private data collection scheme that does not require the release of C_i. In essence, we now have to assume that C_i contains all content items that were ever published (or will be published in the future) by the content server. It is important to note that the differential privacy indistinguishability guarantee is also affected by this new problem statement. While in Chapter 3 we ensured indistinguishability between conclusions "$c \in E_i$" and "$c \notin E_i$" for every *c in the local set* C_i *of retrieved items*, now we will achieve this *for any c that was ever published by the content server.*

4.1 Challenges and Contributions

Data representation and randomization. The randomization described in the previous chapter cannot be directly applied in this setting, as it requires randomization not only for elements of E_i, but also for all elements of $C_i \setminus E_i$. The first contribution of our approach is the design of a scheme for data representation and randomization to solve this problem. In particular, we employ an approach that encodes each element of E_i into a pre-defined smaller domain. The randomization is then performed in this domain. Similar ideas have been used in prior work (e.g., [5, 20]) for scenarios where each user holds a single data item from some extremely large domain such as the set of all possible URLs. In fact, as seen in our studies, URLs are one example of unique identifiers for content items, and thus we are faced with a similar problem. However, in our case we have several items per user, and thus our approach needs to handle this more general case.

The mapping to a smaller domain is achieved using *count sketch* [9]. This well-known approach creates a "sketch": a fixed-size representation of the frequencies of items in streaming data. In our setting, the sketch provides estimates of frequencies of content

items. Since a count sketch is designed to work without *a priori* knowledge of the set of possible items, it is well suited for the goal of representing and randomizing the local set E_i. This local information is now simply treated as being a subset of the (unknown) set C of all content items, and knowledge of the local set C_i of retrieved items is not used at all. Moreover, multiple count sketches can be combined easily, so this approach is suitable to use when data is collected from multiple app users. To achieve differential privacy, randomization is added over the frequency estimation provided by the count sketch.

Efficient randomization. Naive randomization of the count sketch data would randomize each individual contribution to the sketch whenever an event is observed. However, this approach has high run-time overhead. As a second contribution of our work, we propose an efficient approach for per-sketch randomization, rather than per-event randomization. The key idea is to first accumulate all contributions to the sketch, and then to randomize the result by drawing random values from the Binomial distribution. While one prior related effort has employed an approach of a similar nature [59], our problem is more complex as it has to be applied to sketch update operations. With the help of this technique, we achieve significant reduction of run-time randomization costs, making this approach suitable for real-world deployment in mobile apps.

Study of privacy/accuracy trade-offs. Accuracy vs privacy trade-offs are a key concern for privacy-preserving algorithms. We present an experimental study to characterizes these trade-offs in several dimensions. Broadly, our conclusions are that with large number of software users, both practical accuracy and practical privacy can be achieved, especially for high-frequency items.

4.2 Background

A *count sketch* [9] is a data structure originally designed to store the approximate frequencies of items in a data stream. Theoretically, the sketch aims to provide accurate estimates for high-frequency items. Suppose a data stream contains elements from the set $\mathcal{V} = \{1, \ldots, d\}$ where d can be very large. Keeping count of every item in $\mathcal{V}$ would require $O(d)$ space. In a software system where each user u_i reports randomized counts for items from $\mathcal{V}$, such data collection would require $O(d)$ time and $O(d)$ memory for each user, which is infeasible for any practical setting. By using the small-size representation in a count sketch, this cost can be made practical. Further, every element of $\mathcal{V}$ is not equally significant from the perspective of the software developers. In a situation where the developers are interested in the items with high frequencies, they can expect to obtain high accuracy estimates for those items while benefiting from the reduced running time and memory consumption of the analysis.

To illustrate a simple scenario of using count sketch for mobile app analytics without any randomization, suppose that each software user u_i holds a single item $v_i \in \mathcal{V}$. The count sketch algorithm works with two hash functions: $h : \mathcal{V} \to \{1, \ldots, m\}$ and $g : \mathcal{V} \to \{+1, -1\}$. The element v_i held by user u_i can be represented as a d-dimensional bit-vector with a 1 at position v_i and 0s at all other locations. Hash function h maps v_i to a value in $\{1, \ldots, m\}$, which transforms the aforementioned d-dimensional bit-vector into an m-dimensional bit-vector containing a 1 at position $h(v_i)$ and 0 at each other position. The second hash function g assigns a sign to the bit in the m-dimensional bit-vector. Thus, after applying both hash functions, the d-dimensional vector is transformed to an m-dimensional vector with a $+1$ or -1 at one position and 0 at each other position. This vector is the *local sketch* of user u_i. Each user u_i reports her local sketch to the analytics server. The server aggregates these

sketches via element-wise addition to produce a *global sketch*. A frequency estimate $\hat{f}(v)$ for any element $v \in \mathcal{V}$ can be obtained by taking the value at position $h(v)$ in the global sketch and multiplying it by $g(v)$.

This approach is likely to result in inaccurate estimates due to collisions in the hash functions. To get a more accurate estimate, the approach uses t pairs of independent hash functions $(h_1, g_1), \ldots, (h_t, g_t)$ where $h_k : \mathcal{V} \to \{1, \ldots, m\}$ and $g_k : \mathcal{V} \to \{+1, -1\}$ for $1 \leq k \leq t$. These functions are the same for all users u_i. After applying these pairs of hash functions on v_i, instead of a signed m-dimensional vector, a $t \times m$ matrix is created. Each matrix row k contains a single non-zero element, at position $h_k(v_i)$ and with value $g_k(v_i)$. These local sketches are sent to the server where they are aggregated via element-wise addition to create a global sketch S, which is also a $t \times m$ matrix. From this global sketch, a frequency estimate $\hat{f}(v)$ for any $v \in \mathcal{V}$ can be obtained by calculating the median of $S[1, h_1(v)] \times g_1(v), \ldots, S[t, h_t(v)] \times g_t(v)$.

4.3 Randomized Count Sketch for Dynamic Content Frequencies

The count sketch approach described earlier provides non-randomized frequency collection when each user holds a single data item. Existing work [5] has considered how to introduce differential privacy in this setting. This prior work provides theoretical insights but does not bring clarity on how to solve our target problem: practical use of count sketch on analytics data obtained by real app executions, when the local information is a set of content items instead of a single item. Building on ideas for that work, we develop a solution for differentially-private frequency analysis of dynamic content in mobile apps, using randomized sketching. The presentation below starts by describing the design and implementation

Algorithm 4.1: Building a local count sketch without and with randomization

Function `init` (*t, m*) **:**
  $m \leftarrow 2^{\lceil \log_2 m \rceil}$
  $M \leftarrow$ matrix of size $t \times m$, initialized with $0s$
Function `hash` (*k, item*) **:**
  $v \leftarrow$ SHA256 hash of $concat(k, item)$
  $h \leftarrow \log_2 m$ most significant bits of v
  $b \leftarrow (1 + \log_2 m)$-th most significant bit of v
  $g \leftarrow \begin{cases} -1, & \text{if } b = 0 \\ +1, & \text{if } b = 1 \end{cases}$
  return h, g
Function `add` (*item*) **:**
  for $k \leftarrow 1$ **to** t **do**
    $h, g \leftarrow$ `hash` (*k, item*)
    $M[k,h] \leftarrow M[k,h] + g$
Function `add_private` (*item,* ε) **:**
  for $k \leftarrow 1$ **to** t **do**
    $h, g \leftarrow$ `hash` (*k, item*)
    for $j \leftarrow 1$ **to** m **do**
      if $j = h$ **then**
        $M[k,j] \leftarrow \begin{cases} M[k,j] + g, & \text{with probability } \frac{e^\varepsilon}{1+e^\varepsilon} \\ M[k,j] - g, & \text{with probability } \frac{1}{1+e^\varepsilon} \end{cases}$
      else
        $M[k,j] \leftarrow \begin{cases} M[k,j] + 1, & \text{with probability } 0.5 \\ M[k,j] - 1, & \text{with probability } 0.5 \end{cases}$

of a non-private version of our solution. Next, we introduce a privacy-preserving version which achieves differential privacy.

4.3.1 Count Sketch Without Privacy

When the local data for user u_i is a set E_i of content items, the local sketch accumulates the contributions of all items in this set. This process is described in Algorithm 4.1. The sketch is a $t \times m$ matrix M, initialized with 0 elements. For efficiency of hashing, our

Algorithm 4.2: Server-side frequency estimates

```
1 Function estimate(S, item):
2     A ← ∅
3     for k ← 1 to t do
4         h, g ← hash(k, item)
5         A ← A ∪ {S[k,h] × g}
6     return median(A)
7 Function estimate_private(S, item, ε):
8     E ← estimate(S,item) × (e^ε + 1)/(e^ε − 1)
9     return trim(E, 0, n)
```

approach uses values of m that are powers of two, as indicated in function `init`. Every element $v \in E_i$ is added to the sketch using function `add`. For each row k, a hash is calculated by applying the SHA256 hash algorithm on the concatenation of the string representation of k and v. From the value of this hash, a column $h = h_k(v)$ is selected where $h_k(v)$ is the number constructed by the $\log_2 m$ most significant bits of the hash. The counter at $M[k,h]$ is updated by adding $g_k(v) \in \{1,-1\}$ to it, which is chosen using the next most significant bit the hash.

After adding every element in E_i to the sketch, the resulting local sketch for user u_i is sent to the server. Element-wise addition of these local sketches produces the global sketch S. For any v that was ever published by the content server, an estimate $\hat{f}(v)$ of the true frequency $f(v) = |\{i : v \in E_i\}|$ can be obtained by computing the median of $S[1,h_1(v)] \times g_1(v), \ldots, S[t,h_t(v)] \times g_t(v)$. Function `estimate` in Algorithm 4.2 describes this process.

Example. An example of a local sketch is shown in Figure 4.1. The content items $\{c_1, ..., c_{10}\}$ are 10 recipe IDs collected from the set of recipes visited in the app `cookbook` by a simulated user. The example illustrates the algorithm with a sketch matrix with $t = 3$

Item	$h_1(c), g_1(c)$	$h_2(c), g_2(c)$	$h_3(c), g_3(c)$
$c_1 = 51354$	$6, 1$	$6, 1$	$1, -1$
$c_2 = 10972$	$1, 1$	$4, -1$	$1, -1$
$c_3 = 121$	$1, 1$	$2, 1$	$6, 1$
$c_4 = 6$	$6, -1$	$6, 1$	$3, 1$
$c_5 = 244033$	$1, -1$	$6, 1$	$8, -1$
$c_6 = 1083139$	$4, 1$	$2, -1$	$8, 1$
$c_7 = 353278$	$8, -1$	$3, -1$	$7, -1$
$c_8 = 4$	$4, -1$	$5, -1$	$7, -1$
$c_9 = 239$	$5, 1$	$8, 1$	$8, -1$
$c_{10} = 1972875$	$7, -1$	$5, 1$	$6, 1$

Local Sketch

1	0	0	0	1	0	-1	-1
0	0	-1	-1	0	3	0	1
-2	0	1	0	0	2	-2	-1

Figure 4.1: Count sketch illustration, with $m = 8$ and $t = 3$

and $m = 8$. Each item c is hashed into a value $h_k(c) \in \{1, \ldots, m\}$ by applying the SHA256 hash algorithm on the concatenation of the row number and the ID itself. In this way, the same ID produces a different hash in each row. The 3 most significant bits of the hash is used to select a column in the matrix. Additionally, another hash function $g_k(c)$ produces a $+1/-1$ value depending on the value of the 4th most significant bit of the aforementioned SHA256 hash. These two hash values for each of the 10 IDs are shown in the table in the left part of Figure 4.1.

The table in the right part of the figure shows the sketch matrix created after counting these 10 items using their corresponding hash values. For every item c, a column is selected on row k by the value of $h_k(c)$, and the value of $g_k(c)$ is added to that cell. For example, counting the item c_2 adds a $+1$ to the cell $[1, 1]$, a -1 to the cell $[2, 4]$ and a -1 to the cell $[3, 1]$. It also shows some hash collisions for some items. For example, counting c_1 adds a $+1$ to the cell $[1, 6]$ and counting c_4 adds a -1 to the same cell, hence cell $[1, 6]$ contains a 0. But due to the design of the hash function h, c_1 and c_4 do not collide on the second and third rows.

An estimate for each item can be obtained from the cells modified by counting that item. For example, counting item c_2 contributes to cells $[1,1]$, $[2,4]$ and $[3,1]$. The values stored in these cells are 1, -1, and -2. Multiplying these values by their corresponding g hash values (1, -1 and -1 respectively) we get 1, 1, and 2. The median of these values is 1, which is the correct frequency of c_2. Of course, this estimate could be different from the true frequency; in this example, the estimated frequency of c_1 is 2 and the estimated frequency of c_6 is 0 while the true frequency is 1 for both of them.

4.3.2 Count Sketch With Privacy

4.3.2.1 Processing for Each User

A differentially-private count sketch algorithm can be constructed by introducing probabilistic counting of elements of the non-private count sketch. First, consider the randomization of a single content item c. The non-randomized contribution of this item to row k of the sketch is a vector of length m, containing the value of $g_k(c)$ in position $h_k(c)$ and values of 0 in the remaining positions. This vector is randomized as follows. First every 0 element is changed to -1 with probability $\frac{1}{2}$ and to $+1$ with probability $\frac{1}{2}$. This is done independently for each such position; thus, this process requires $m-1$ "coin flips". Position $h_k(c)$ is randomized as follows: with probability $\frac{e^{\varepsilon}}{1+e^{\varepsilon}}$ the value is preserved and with probability $\frac{1}{1+e^{\varepsilon}}$ the value's sign is inverted (e.g., $+1$ is replaced with -1). As a result, the randomized $1 \times m$ vector contains only $+1$ and -1 values. This process in described in function `add_private` in Algorithm 4.1.

The indistinguishability property holds between any two $1 \times m$ input vectors, each containing a single $+1/-1$ value and 0 values in all remaining positions. Specifically, consider any two such vectors x and y. Let z be any $1 \times m$ vector containing $+1/-1$ values in all positions. Let R be the randomizer applied to the input vectors. Then the probability

Randomized Local Sketch

-2	0	4	4	2	-2	0	-2
2	-2	-2	-2	4	4	0	-4
-8	-2	-8	-4	0	6	0	0

Figure 4.2: Randomized Count sketch illustration, with $\varepsilon = \ln(3)$, $m = 8$ and $t = 3$

that $R(x) = z$ and the probability that $R(y) = z$ differ by at most a factor of e^{ε}. To see this, consider that $P[R(x) = z] = \prod_{1 \leq j \leq m} P[R(x_j) = z_j]$. Here x_j denotes the value at position j in the vector. Similarly, $P[R(y) = z] = \prod_{1 \leq j \leq m} P[R(y_j) = z_j]$. Without loss of generality, suppose that the non-zero element in x is x_1 and the non-zero element in y is y_2. Then the ratio of the two probabilities is $P[R(x_1) = z_1]/P[R(y_2) = z_2]$, which is maximized when $P[R(x_1) = z_1] = \frac{e^{\varepsilon}}{1+e^{\varepsilon}}$ and $P[R(y_2) = z_2] = \frac{1}{1+e^{\varepsilon}}$.

Now, consider the randomization of a set of content items E. For every $c \in E$, the processing in function `add_private` is performed separately. We can consider the cumulative result to be produced by set-level randomizers R_k for each row k, such that $R_k(E)$ is the resulting row in the randomized sketch. The indistinguishability achieved by applying R_k is as follows. For any E and any content items $c \in E$ and $c' \notin E$, let F be the set obtained from E by replacing c with c'. Then E and F are indistinguishable in the differential privacy sense, since $P[R_k(E) = G]$ and $P[R_k(F) = G]$ can differ by at most a factor of e^{ε} for all G.

Example. An example of randomized local sketch is illustrated in Figure 4.2 as a continuation of the example shown in Figure 4.1. Here, after determining the hash values $h_k(c)$ and $g_k(c)$ for an item c, the value of $h_k(c)$ is preserved with probability $\frac{e^{\varepsilon}}{1+e^{\varepsilon}}$ (which is 0.75 in this example with $\varepsilon = \ln(3)$) or inverted with probability $\frac{1}{1+e^{\varepsilon}}$ (which is 0.25). This value is added to the cell $[k, h_k(c)]$. A -1 or $+1$ is added at random (i.e. with probability 0.5) to

each of the other cells of row k. The effect of this randomization, for one particular run of this randomized algorithm, is shown in the sketch matrix in Figure 4.2.

4.3.2.2 Processing at the Server

The result of the randomization, obtained by applying function `add_private` to every $c \in E_i$, is a local randomized sketch that is reported to the analytics server. The server accumulates these sketches to produce a global sketch S for all users u_i. The estimates from this global sketch need to be scaled, as described by function `estimate_private` in Algorithm 4.2. This scaling ensures that the resulting estimates are unbiased. To see this, consider the expected value of cell $S[k, j]$. Suppose there would have been n contributions of $+1$ to this cell without randomization, across all users. The expected value of these contributions after randomization is $n\frac{e^{\varepsilon}}{1+e^{\varepsilon}} - n\frac{1}{1+e^{\varepsilon}}$. Similar reasoning is applied to the -1 contributions. To obtain unbiased estimates, scaling with the inverse of $\left(\frac{e^{\varepsilon}}{1+e^{\varepsilon}} - \frac{1}{1+e^{\varepsilon}}\right)$ is necessary; that is, the scaling factor should be $\frac{e^{\varepsilon}+1}{e^{\varepsilon}-1}$. The scaled value is then trimmed to be in the interval $[0, n]$: if negative, it is set of 0 and if greater than n it is set to n.

4.3.3 Efficient Construction of the Randomized Local Sketch

In the non-private version of the count sketch, counting an item involves modification of one sketch cell in each row (i.e., adding $-1/+1$ to the cell). Thus, the overall processing cost is $O(t)$, assuming that hashing has unit cost. But in the private version, counting an item requires additional operations of adding $-1/+1$ at random to every other cell in a row. This makes the overall processing cost $O(|E_i| \times t \times m)$. To reduce this cost, we designed a modified version of the randomized count sketch, as described in Algorithm 4.3.

Algorithm 4.3: Reduced-cost local count sketch algorithm

Function `init(`*t, m*`)`**:**
- $m \leftarrow 2^{\lceil \log_2 m \rceil}$
- $M \leftarrow$ matrix of size $t \times m$, initialized with $0s$
- $C_{-1} \leftarrow$ matrix of size $t \times m$, initialized with $0s$
- $C_{+1} \leftarrow$ matrix of size $t \times m$, initialized with $0s$
- $total\_events \leftarrow 0$

Function `hash(`*k, item*`)`**:**
- $v \leftarrow$ SHA256 hash of $concat(k, item)$
- $h \leftarrow \log_2 m$ most significant bits of v
- $b \leftarrow (1 + \log_2 m)$-th most significant bit of v
- $g \leftarrow \begin{cases} -1, & \text{if } b = 0 \\ +1, & \text{if } b = 1 \end{cases}$
- **return** h, g

Function `add(`*item*`)`**:**
- $total\_events \leftarrow total\_events + 1$
- **for** $k \leftarrow 1$ **to** t **do**
  - $h, g \leftarrow$ `hash(`*k, item*`)`
  - **if** $g = -1$ **then**
    - $C_{-1}[k,h] \leftarrow C_{-1}[k,h] + 1$
  - **else**
    - $C_{+1}[k,h] \leftarrow C_{+1}[k,h] + 1$

Function `finalize_private(`ε`)`**:**
- **for** $k \leftarrow 1$ **to** t **do**
  - **for** $j \leftarrow 1$ **to** m **do**
    - $pos \leftarrow 2 \times binomial(C_{+1}[k,j], \frac{e^{\varepsilon}}{1+e^{\varepsilon}}) - C_{+1}[k,j]$
    - $neg \leftarrow 2 \times binomial(C_{-1}[k,j], \frac{e^{\varepsilon}}{1+e^{\varepsilon}}) - C_{-1}[k,j]$
    - $nz \leftarrow total\_events - C_{+1}[k,j] - C_{-1}[k,j]$
    - $zero \leftarrow 2 \times binomial(nz, 0.5) - nz$
    - $M[k,j] \leftarrow pos - neg + zero$

The main idea of the approach is to first accumulate counts for various per-cell contributions in the non-randomized version, and then draw random values from the Binomial distribution to apply randomization to each accumulated count. Two categories of counts are maintained per sketch cell: matrix C_{+1} records the number of $+1$ contributions, and matrix C_{-1} records the number of -1 contributions. In addition, a counter of the total number of "add item" events is maintained. This is used to infer the number of times a 0 would have been added to a cell in the non-randomized sketch.

In the basic private count sketch, every time an event is counted as a -1 in a cell of M, it is preserved with a probability $p = e^{\varepsilon}/(1+e^{\varepsilon})$ and changed to a $+1$ with probability $(1-p)$. If the event is counted n times, this is equivalent to counting the number of successes in a sequence of n independent trials with a probability p of success. The Binomial distribution provides such probabilities. This property is utilized in the `finalize_private` procedure of the modified algorithm. After counting the number of -1, $+1$ and 0 values added to each cell in the non-private version, the randomized counts are determined by drawing random values from the Binomial distribution, denoted by $binomial(n,p)$. First, consider all pre-randomization $+1$ contributions to some cell $M[k,j]$. The non-optimized randomization would have contributed $binomial(C_{+1}[k,j], \frac{e^{\varepsilon}}{1+e^{\varepsilon}})$ values of $+1$ and $\left(C_{+1}[k,j] - binomial(C_{+1}[k,j], \frac{e^{\varepsilon}}{1+e^{\varepsilon}})\right)$ values of -1. Thus, the total contribution *pos* (i.e., positive increments) of these pre-randomization $+1$ values to the final cell value is the difference between these two counts. The total contribution *neg* (i.e., negative increments) of the pre-randomization -1 values to the final cell value is computed similarly. Finally, the 0 pre-randomization values will also contribute $+1$ and -1 counts, which are determined using $binomial(nz, 0.5)$. The overall cost of this computation is $O(t \times m)$, assuming that random values are generated at unit cost.

4.4 Experimental Evaluation

The experimental evaluation of the proposed approach was performed on the same data collected for the evaluation from the previous chapter. Recall that 9 Android apps and user interactions with them were simulated to create sets E_i and C_i. Since our new approach does not depend on the set of retrieved items C_i, only the set of events E_i from every user u_i was used. Details on the study subjects can be found in Table 3.1.

4.4.1 Synthesizing User Data

The user data was collected by running a test script 100 times on 9 instrumented apps. Since collecting this data takes a large amount of time, the data collected from 100 simulated users were used to simulate behaviors of more users. We followed the same approach for synthesizing additional user data as mentioned in the previous chapter. Since the content dictionary is not needed in this analysis, the synthesizing process was applied on the sets E_i only. As described earlier, to obtain the synthesized data for 1000 users we selected 900 pairs of users (u_i, u_j) at random. For each pair, we computed the set of events $E_i \cup E_j$, and then drew $\frac{|E_i|+|E_j|}{2}$ items from this set without replacement. Together with the original data from 100 users, this set of synthesized data produced sets of events for 1000 simulated users. The same approach was used to synthesize the sets of events for 10000 and 100000 users.

4.4.2 Efficiency of Optimization using Binomial Distribution

To observe the effect of efficient construction of the randomized count sketch using binomial probability distribution as described in Section 4.3.3, both the basic (Algorithm 4.1 and the modified (Algorithm 4.3) version of the count sketch algorithm were applied on the dataset. The results of the optimization are summarized in Table 4.1. The data in

App	Average $\lvert E_i \rvert$	Basic count sketch	Modified count sketch
apartmentguide	101.09	8.847378	0.086971
reststops	70.62	6.167492	0.063994
rent	112.13	9.784922	0.096058
shipmate	76.83	6.236660	0.061835
cookbook	55.87	4.364689	0.045915
channels	46.94	3.659719	0.039744
infowars	38.96	3.052576	0.036958
loop	54.91	4.267905	0.049260
opensnow	85.08	6.628039	0.061553

Table 4.1: Average time taken (in seconds) to count all events in E_i for a user with $\varepsilon = \ln(9)$

the table shows the average number of events triggered by an user and average of total time to count all events with a basic and a modified count sketch. The data was collected from the 100 simulated users of each app. As can be seen from these results, the modified approach is significantly faster, and clearly suited for practical use in mobile apps. The reason behind this is, the $O(t \times m)$ amount of work required for randomization *per event* in the basic algorithm is replaced by the $O(t \times m)$ amount of work *per user* in the reduced-cost algorithm.

4.4.3 Accuracy of Frequency Estimates

The modified count sketch algorithm (Algorithm 4.3) was executed in 30 independent trials to calculate the frequency estimates $\hat{f}(c)$ for each $c \in C_i$. We used $t = 256$ and $m = 256$ for our initial set of experiments. The chosen value of t is similar to that used in prior work [5]. The error of the estimates is calculated using the relative error (RE) metric. In this experiment, the relative error is defined as the normalized L_1 distance between the ground truth frequency vector $f(c)$ and the estimated frequency vector $\hat{f}(c)$ for each $c \in \cup_i C_i$:

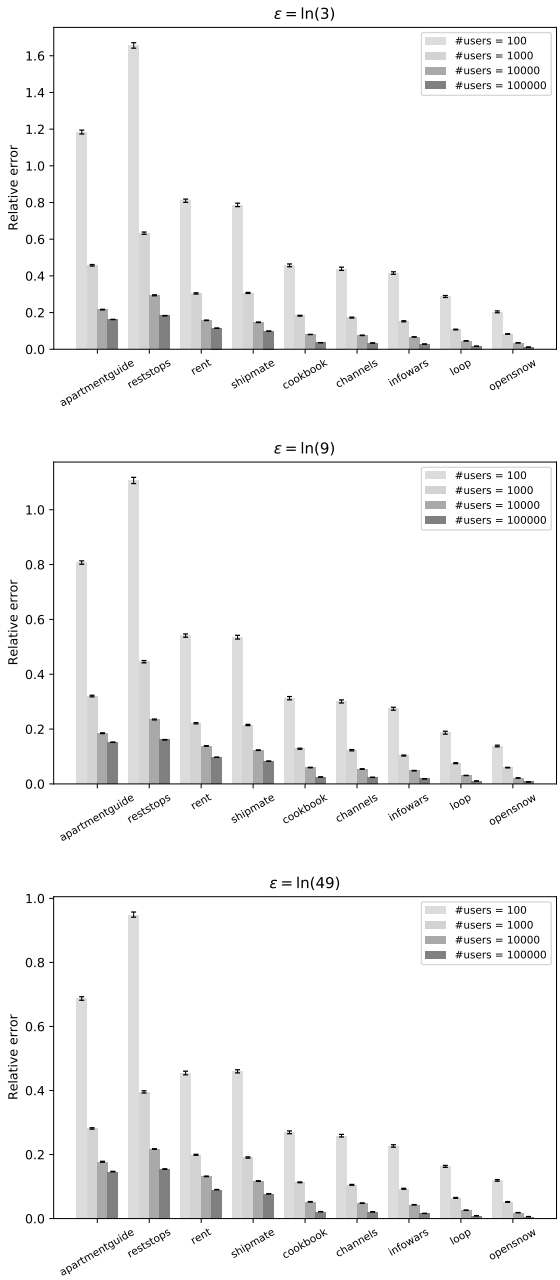

Figure 4.3: Accuracy of private count sketch over items retrieved by all users ($\cup_i C_i$).

$$RE = \frac{\sum_{c \in \cup_i C_i} |f(c) - \hat{f}(c)|}{\sum_{c \in \cup_i C_i} f(c)}$$

Figure 4.3 shows the accuracy of the estimated frequencies for the three values of ε used in the previous chapter: $\ln(3)$, $\ln(9)$ and $\ln(49)$. For this and all subsequent metrics we report the mean and 95% confidence interval of the metric over 30 independent trials.

The relative error calculated over $\cup_i C_i$ can be quite large, especially with fewer users. This error includes the frequency estimates for the content items not visited by any user (i.e. $c \notin E_i$ for $1 \leq i \leq n$, therefore $f(c) = 0$). For these items the estimated frequency $\hat{f}(c)$ accounts for the random noise only. With larger number of users the contributions of these noise -1s and $+1$s are "cancelled out" more, which results in smaller relative error.

We also considered a modified metric to measure the accuracy of this scheme that only includes the items visited by at least one user. In this case, the relative error is measured over $\cup_i E_i$. It produces lower relative error due to discarding noise from zero-frequency elements, especially with the apps with larger $\cup_i C_i$. Figure 4.4 shows these results.

The frequency estimates can be more accurate for the most frequent items in the event sets than they are for the entire dataset. Count sketch is a data structure designed to accurately estimate the frequencies of the high-frequency elements in a stream, and this property is evident in this experiment as well. The same experimental setup was applied to "estimated hot" content items, which is defined as the set of items visited by at least 10% of the users based on their estimated frequencies. The relative error for the hot items is quite large for 100 users because the total number of events is very small. But for more users, it is even less than the error for the non-zero-frequency items. These results are shown in Figure 4.5.

It is important to note that the set of estimated hot items can be determined directly by querying the global sketch for each content item ever published by the server, since the published items are known to the analysts and the number of such items is expected to be practically small (e.g., less than a million). In contrast, the general problem of identifying

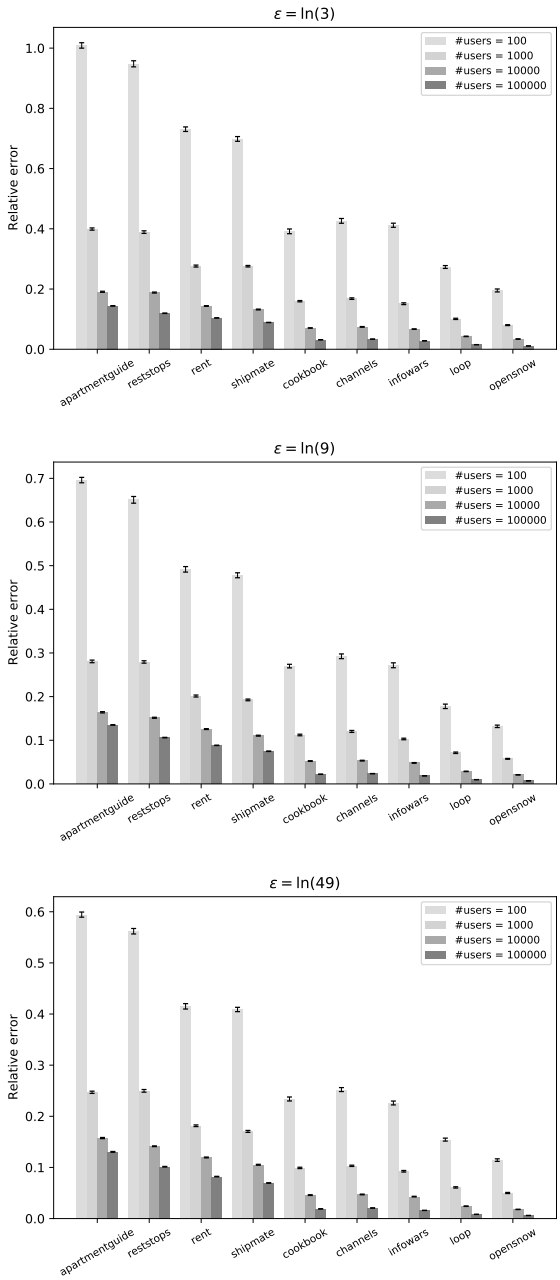

Figure 4.4: Accuracy of private count sketch over items observed by all users ($\cup_i E_i$).

estimated hot items ("heavy hitters") in a very large domain (e.g., the domain of all 64-bit binary strings) requires more advanced techniques [5, 56].

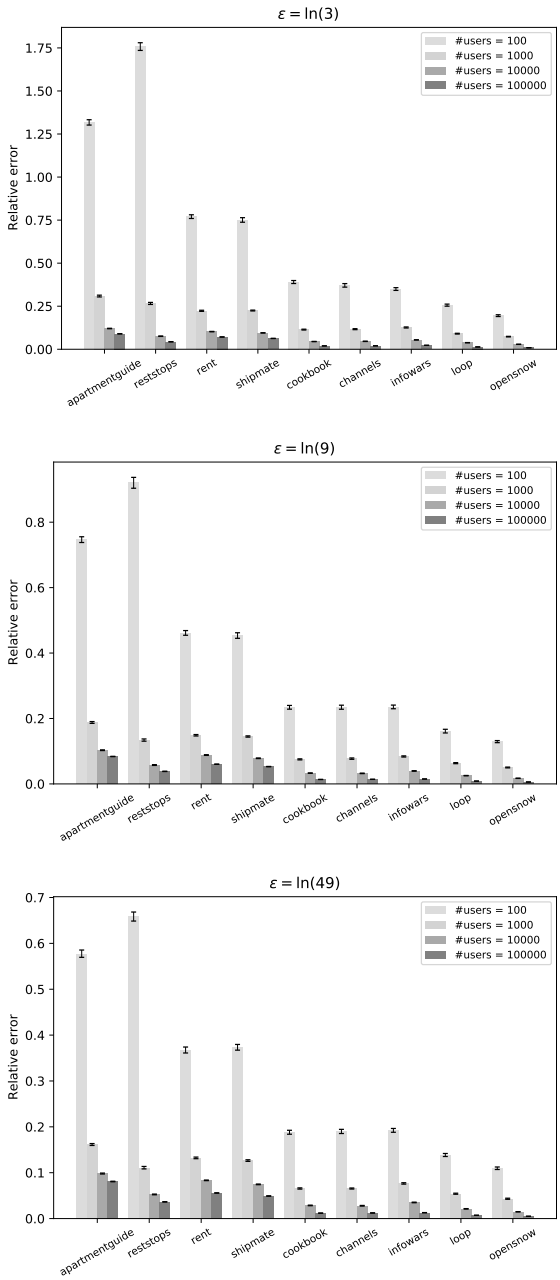

Figure 4.5: Accuracy over hot items: those with estimated frequency $\geq 10\%$ of n.

Summary of results. From the results, it can be observed that the accuracy of the scheme improves with larger values of ε and larger number of users. As expected, the accuracy

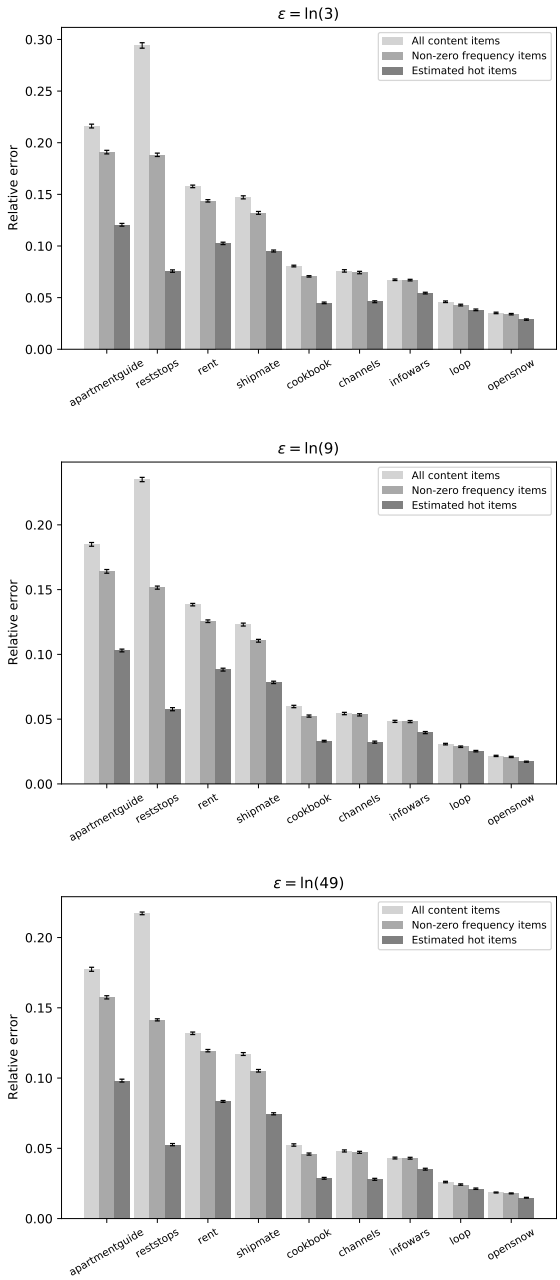

Figure 4.6: Accuracy of private count sketch for 10000 users over three sets of items.

improves with the high-frequency subset of the items. A comparison of the accuracy of the scheme for different groups of items is shown in Figure 4.6. The chart shows the

relative error computed for 10000 users over all content items, non-zero-frequency items, and estimated hot items. We observe that the relative error for the set of estimated hot items is typically less than 10%. These results indicate that for realistic numbers of app users (e.g., at least 10000 users), the frequencies of items that are estimated to be hot are close to their true values.

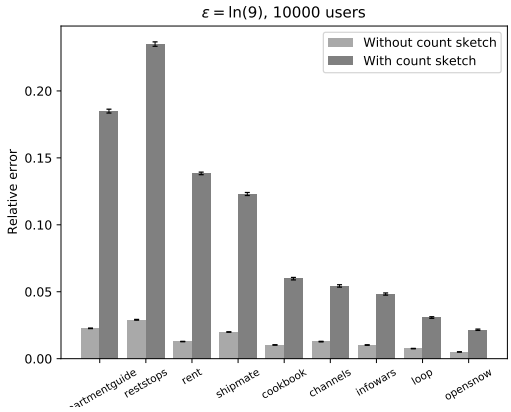

Figure 4.7: Accuracy of frequency estimation vs the scheme defined in Chapter 3.

A comparison between the accuracy of frequency estimation using a deferentially private count sketch and the scheme described in Chapter 3 is shown in Figure 4.7 for one particular combination of n and ε. Not surprisingly, the relative error is larger with the count sketch approach, as the underlying data representation before randomization is based on hashing and hash collisions between elements could occur.

4.4.4 Precision and Recall for Hot Items

Similar to Chapter 3, we measured the precision and recall of the "estimated hot" items identified by the private count sketch. The definition of the estimated hot items remains the

same- an item is considered hot if at least 10% of all users interact with it (based of the frequency estimates). We computed the sets H and $\hat{H}$ of content items visited by at least 10% of the users, based on the ground-truth and estimated frequencies respectively. The precision and recall of the estimated hot items are

$$Precision = \frac{|H \cap \hat{H}|}{|\hat{H}|} \quad Recall = \frac{|H \cap \hat{H}|}{|H|}$$

The measurements of precision and recall are shown in Figures 4.8 and 4.9. It is evident that both the precision and recall of the estimated hot items are typically above 90% with 10000 or more users.

4.4.5 Effects of Sketch Size on Accuracy

Besides the randomization, another source of errors in this scheme is hash collisions. In the data presented so far, all the events were counted in a sketch matrix with 256 columns. This means, with a larger set of content items, more items are hashed into the same cell of a matrix. It increases the relative error, as well as effects the precision and recall of the estimated hot items. To observe the effect of number of columns in the sketch matrix, we conducted another experiment with $\varepsilon = \ln(9)$ and 10000 users using sketch matrices with 128, 256 and 512 columns. This experiment is focused on the hot items as their estimates are most reliably accurate. We have measured the relative error, precision and recall over the estimated hot items (items visited by at least 10% of the users), and the results are presented in Figure 4.10.

From the figures it can be seen that larger number of columns affects the accuracy of the estimates as the relative error for the estimated hot items becomes smaller. Similar effects

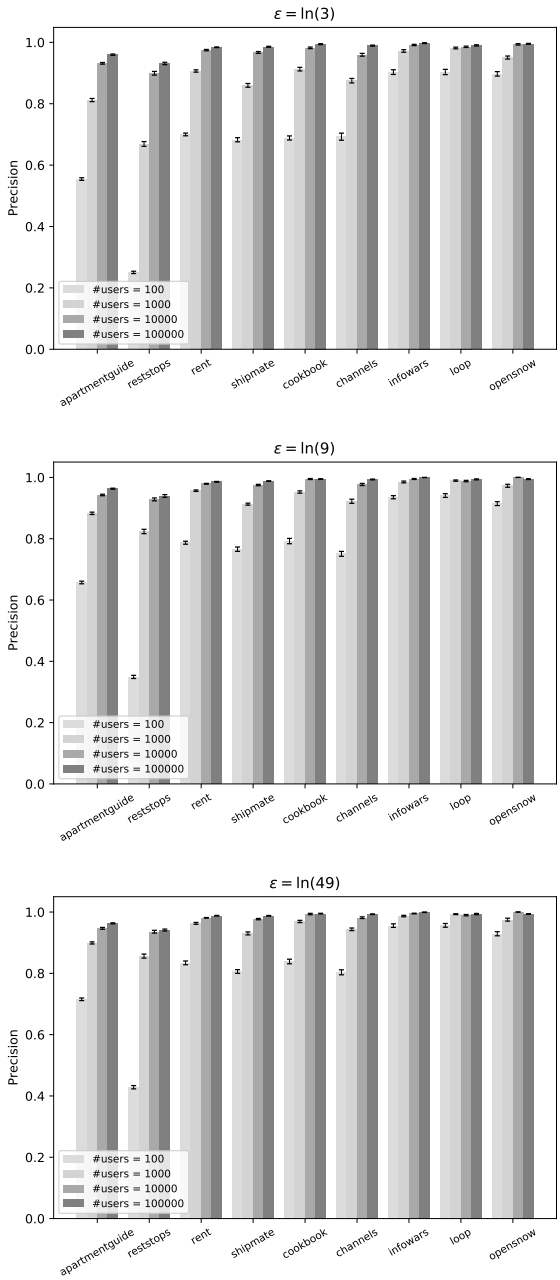

Figure 4.8: Precision of identification of hot items.

can be seen for the identification of hot items as the values of precision and recall typically increase with larger number of columns.

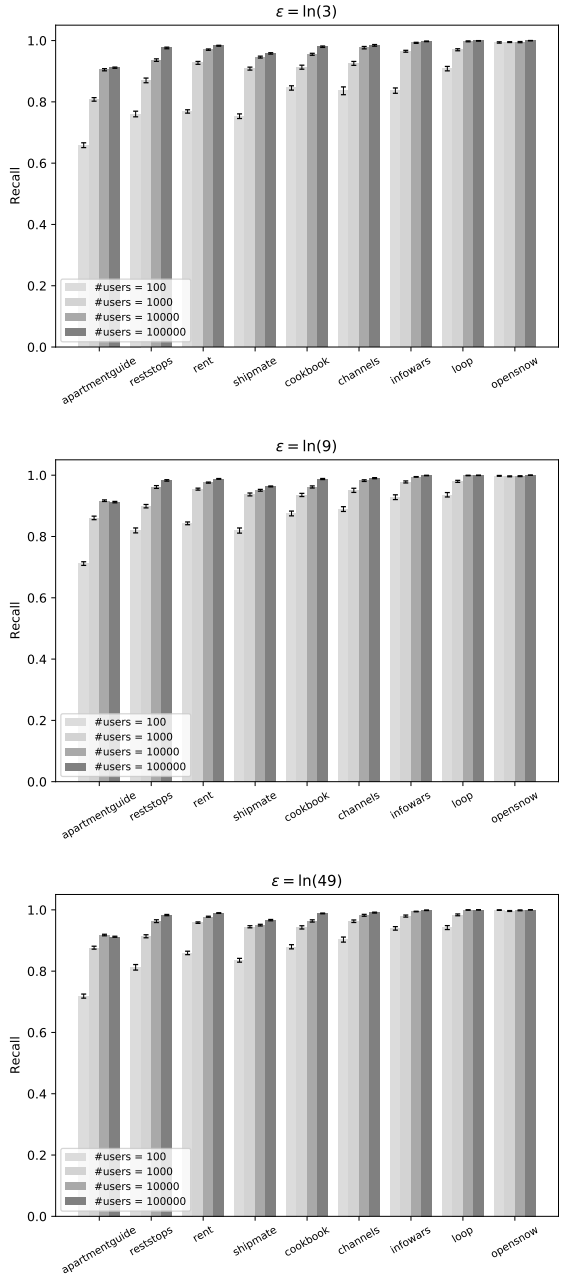

Figure 4.9: Recall of identification of hot items.

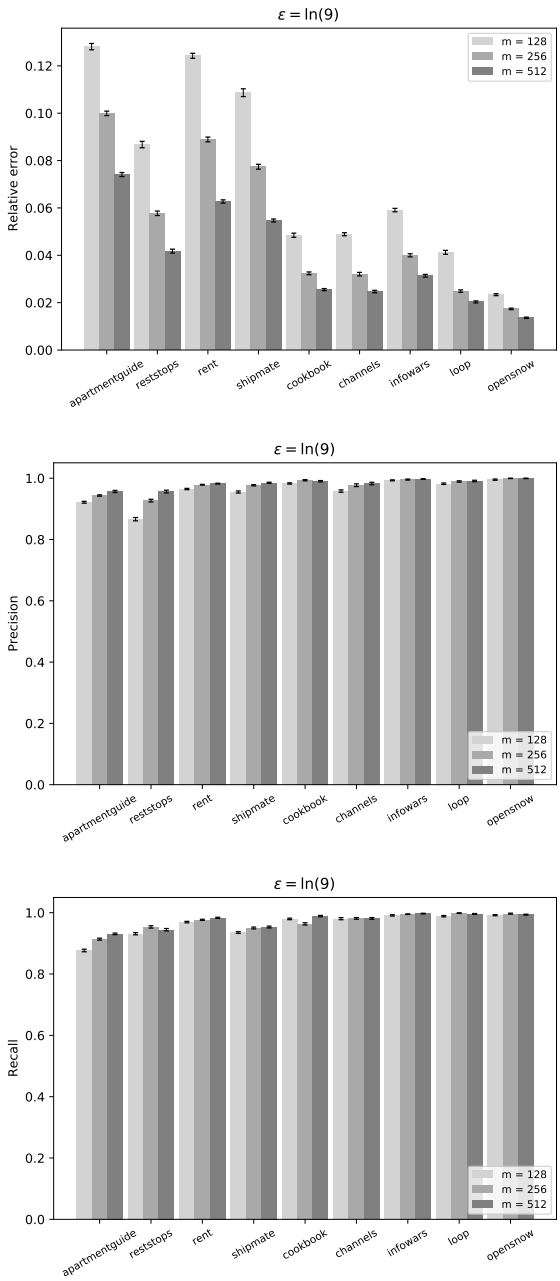

Figure 4.10: Comparison of different numbers of sketch columns, with 10000 users.

4.5 Summary

In this chapter we presented an approach for collecting analytics data from mobile apps using a differentially-private count sketch. This approach does not require sending the set of retrieved content items from the users to the analytics server. We illustrated how a differentially-private count sketch can be used to collect analytics data, how the randomization can be performed efficiently, how the data can be accumulated on the server side and be used to estimate the event frequencies. We have also explored the effects of tuning the parameters of the scheme on the accuracy of estimation. With a significantly large number of users, this new approach can achieve stronger privacy of analytics data maintaining a reasonable level of accuracy, especially with the most frequently accessed elements.

Chapter 5: Differentially-Private Analysis of Frequent Items and Frequent Itempairs Using Randomized Sketches

The previous chapter explored the use of differentially-private count sketch to collect frequency information about content items across a population of mobile app users. However, several questions need to be addressed before such data collection can be deployed in realistic scenarios. First, in practical deployment there are limitations on the amount of data that can be transferred from a user to the analytics server. In many scenarios, the app users have to pay for such data transfer (e.g., with certain mobile Internet plans). One question we address in this chapter is the following: *How should one design differentially-private sketching under a given space budget for the sketching data structure?* In particular, we need to consider the question of selecting the number of rows vs. the number of columns in the sketch. The work in the previous chapter assumes that the numbers of rows and columns are pre-defined values. In this chapter we reconsider this choice.

Identifying (likely) frequent content items is an important analytics task, but there are many other questions that are of interest to content providers. As an exemplar of such questions, we consider the following problem: *How can pairs of frequently co-occurring content items be identified with high accuracy and low cost?* This is an instance of the more general problem of frequent itemset mining. As discussed later, there is a fundamental difference between this problem and the analysis of item frequencies considered so far: the

domain of pairs of items is significantly larger than the domain of items, which makes the problem harder.

The contributions presented in this chapter are as follow. First, we present a characterization study that provides insights needed to design an effective scheme for differentially-private sketching data structures for frequent items and itempairs. Next, based on the results of this study, we propose a design for such frequency analyses under given space constraints for the sketches. Finally, we present an experimental evaluation of the proposed design and identify its intrinsic trade-offs between space and accuracy.

5.1 Analysis of Frequent Items

5.1.1 Characterization Study of Sketch Size and Shape

For the differentially-private count sketch presented in the previous chapter, we used fixed parameters $t = 256$ (number of sketch rows) and $m = 256$ (number of sketch columns). However, the selection of these parameters is an important consideration for achieving high accuracy under a given space budget. To get further insights into this issue, we performed a study with two settings: number of users $n = 1000$ and $n = 10000$. As indicated by the results in the previous chapter, one would expect non-trivial differences in accuracy between these two settings. We consider $n = 100$ to be impractical due to its bad accuracy. We do not consider $n = 100000$ since results for this setting are likely to be close to those for $n = 10000$; further, collecting data from such a large number of users may be impossible for many apps that are not very popular and serve niche markets.

We use measurements for the `apartmentguide` app to illustrate the results of the study. Measurements for the remaining apps are similar and are not shown here. For the characterization, we considered combinations of number of sketch rows and number of

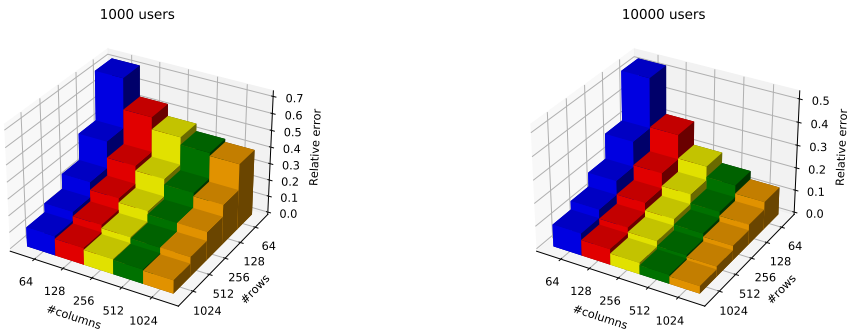

Figure 5.1: Relative error of frequency estimates over the set of estimated hot items for app `apartmentguide`, with 1000 and 10000 users and different numbers of sketch rows and columns.

sketch columns selected from set $\{64, 128, 256, 512, 1024\}$. For each combination of values, we executed the approach to determine the set of estimated hot pairs (as was done in the previous chapter) and to compute the relative error for this set, for $\varepsilon = \ln(9)$. The results are shown in Figure 5.1. Since the measurements from the previous chapter indicate that the variance across multiple runs is small, we ran each experiment three times and collected the average of the three measurements.

As can be observed from these results, increasing the size of the sketch does reduce the error of estimates. This is expected since the effects of hash collisions are reduced. We also observe that the increase in the number of sketch rows is more beneficial than the increase in the number of columns. As presented below, these results guide our design of a randomized sketch under space constraints.

5.1.2 Sketch Shape under Space Constraints

Several considerations are relevant for the selection of sketch size (i.e., number of sketch elements) and sketch shape (i.e., number of rows and columns) in the analysis of frequent items. First, we assume that a pre-defined limit on sketch size is imposed by analysis

designers based on financial and technical constraints. Such constraints, for example, could stem from the cost of transmitting data (which in some circumstances could be as high as several USD per Gb) and the bandwidth of the connection. For our experiments, we have selected 256Kb for this limit, which is comparable to the size of a low-resolution image.

Given this limit, we consider the following approach for selection of sketch size. First, we select a number R of sketch rows that is close to the total number of unique content items that could be included in any sketch (local or global). We assume that this number is known to the analysis designer—for example, because they know the approximate number of unique content items that would be available from the content server over the duration of mobile app analytics data gathering. This is a reasonable assumption as we expect that the mobile app developers and the content provides are collaborators. We select the number of rows R to be the smallest power of 2 that is not smaller than the number of content items. Table 5.1 shows these numbers for the apps considered in our experiments. Column "all items" shows the size of the union of sets C_i (the content items retrieved by users), which we use as an estimate of the total number of content items available from the content server. Column "rows" shows the corresponding number of sketch rows. The number of sketch columns is determined as $256\text{Kb}/(2R)$. The factor 2 in the denominator reflects an assumption that each sketch element can be stored in 2 bytes, which is a reasonable assumption for local sketches.

App	#all items	#rows	#columns
apartmentguide	1375	2048	64
reststops	1858	2048	64
rent	902	1024	128
shipmate	712	1024	128
cookbook	358	512	256
channels	294	512	256
infowars	226	256	512
loop	186	256	512
opensnow	168	256	512

Table 5.1: Number of all content items and number of sketch rows/columns for 256Kb space budget.

5.2 Analysis of Frequent Itempairs

Frequent itemset mining is a technique for finding associations and correlations among items in a dataset. The objective of such mining is to find sets of items that appear together in a significant number of transactions carried out over a set. The most common application of this technique is determining the products bought together frequently. It is also useful for product placement, cataloging, and cross-selling.

In this dissertation we consider a particular case of frequent itemset mining: the identification of frequently-occurring pairs of items (which will be referred to as "itempairs" from now on). Our goal is to identify pairs $\langle c, c' \rangle$ of content items c and c' (where $c \neq c'$) such the number of sets E_i containing both c and c' is no less than a pre-defined threshold. Specifically, as in the earlier chapters, we consider a scenario where user u_i interacts with a set E_i of content items. Each such set E_i is considered to be a transaction in the terminology of frequent itemset mining.

5.2.1 Design of LDP Analysis of Frequent Itempairs

Suppose that we have a pre-defined threshold θ, such that $0 < \theta \leq 1$. For any itempair $\langle c, c' \rangle$, its frequency $f(\langle c, c' \rangle)$ is

$$f\left(\langle c, c' \rangle\right) = |i : c \in E_i \wedge c' \in E_i|$$

An itempair $\langle c, c' \rangle$ is *hot* if $f(\langle c, c' \rangle) \geq \theta n$; here n is the number of software users. As in the previous chapters, in our experiments we use $\theta = 0.1$.

Estimated hot items. To achieve differential privacy for mining of frequent itempairs, we can use the approaches defined in the earlier chapters as building blocks. Specifically, we can use those approaches to differentially-privately estimate the set of individual items whose frequency is no less than θn. This set can then be used to estimate the frequent itempairs. The collection of (estimated) hot items can be done either with the randomization approach from Chapter 3 (which reports C_i and the randomized version of E_i), or with the sketch-based approach from Chapter 4 (which reports only the randomized sketch of E_i). As the second approach provides stronger privacy protections, we focus on it for the rest of this chapter. Let $\hat{H} = \{c : \hat{f}(c) \geq \theta n\}$ be the set of "estimated hot" items, where the estimates are obtained as described in Chapter 4, using the same hotness threshold θ and employing the sketch size/shape selection approach described in Section 5.1.

Estimated hot itempairs. To determine the most frequent itempairs, the analytics server first computes set $\hat{H}$. A key property is that, in the ground truth, each hot itempair must contain two hot items. Since $\hat{H}$ approximates the ground-truth set of hot items, the LDP analysis of hot pairs aims to collect and report all pairs of elements of $\hat{H}$ that appear in sets E_i, from which the hot pairs can be estimated. To achieve this, the set $\hat{H}$ of estimated hot items is communicated to each user. In a second pass of the communication, the pairs of

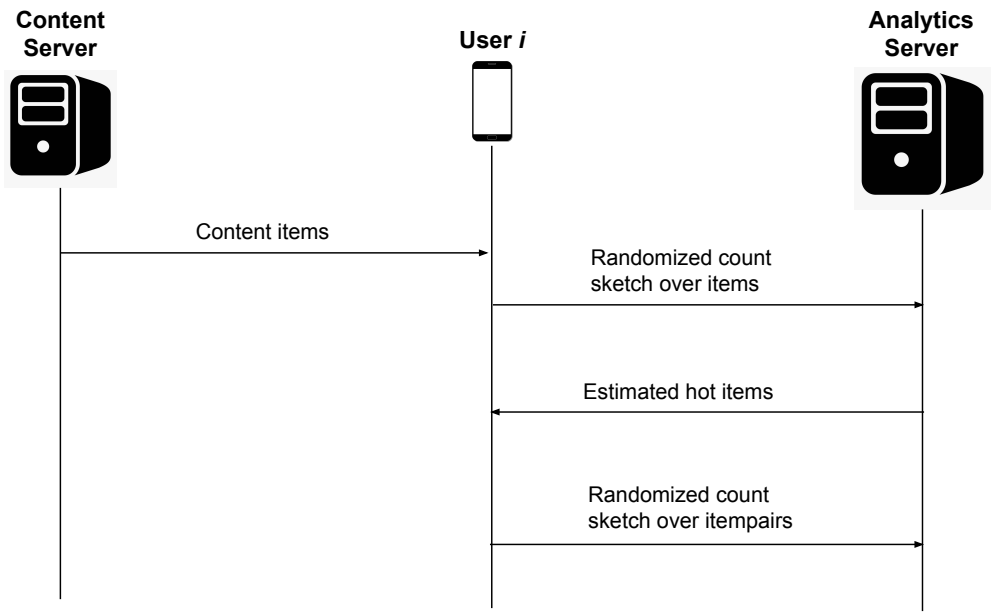

Figure 5.2: Workflow for determining the most frequent itempairs.

estimated hot items are reported by each user to the analytics server. The server identifies the most frequent itempairs from these reports. The details of this approach are described below.

Solution design with randomized sketches. The same differentially-private count sketch algorithm described in Chapter 4 can be applied to determine the estimated hot itempairs. Each user u_i receives the set $\hat{H}$ of the estimated hot items from the server. From this set, all pairs $\langle c, c' \rangle$ are constructed where c and c' are elements of E_i, and both items also appear in $\hat{H}$. The set of such itempairs is defined as $E_i^2 = \{\langle c, c' \rangle : c, c' \in E_i \cap \hat{H}\}$. The pairs in E_i^2 are then counted in a differentially-private count sketch, similarly to the counting done for the individual items in Chapter 4, and the randomized sketches of itempairs are reported to the analytics server. The workflow is illustrated in Figure 5.2.

At the server side, all randomized sketches from the users are accumulated into one sketch. From this sketch server estimates the frequency $\hat{f}(\langle c, c' \rangle)$ of any itempair $\langle c, c' \rangle$ such that both c and c' are estimated hot items (i.e., belong to set $\hat{H}$). A pair $\langle c, c' \rangle$ is considered an estimated hot pair if $\hat{f}(\langle c, c' \rangle) \geq \theta n$.

One interesting special case is when set E_i^2 is empty, which means that E_i does not contain any elements of $\hat{H}$. In such a scenario the user would contribute an empty sketch, which in practice means the user would not send any information back to the server. This behavior does leak information: it indicates to an adversary that no element of E_i is in $\hat{H}$. We consider this to be an acceptable privacy leak and do not attempt to obfuscate it.

Implementation. An itempair is considered to be a size-2 set, which implies that the order of the two items is not significant. In the implementation, the pair of items c and c' is represented by concatenating their identifiers with "`|`" as a separator character in their lexicographic order. For example, if a user interacts with items `244033` and `1083139` in the `cookbook` app, this pair of items is represented with the identifier `1083139|244033` in the analysis of itempairs. The character "`|`" is used as the separator symbol because it does not appear as a part of any item identifiers in any app. The lexicographic order of concatenation of two identifiers prevents counting $\langle c, c' \rangle$ and $\langle c', c \rangle$ as two different pairs. Given these string identifiers for itempairs, the approach proceeds as described above: given the (global) set $\hat{H}$ of estimated hot items provided by the server, each user u_i constructs the set E_i^2 as a set of itempair identifiers and then records them in a local differentially-private count sketch, as described in Chapter 4. The local sketches are then accumulated by the server to construct a global sketch. For each element of $\hat{H} \times \hat{H}$ (using string concatenation in lexicographic order), its frequency is estimated using the global sketch and the itempair is reported as "estimated hot" if the estimate is $\geq \theta n$.

5.2.2 Selecting Sketch Size and Shape

In the selection of sketches for analysis of frequent items, we assumed a space budget of 256Kb. For sketching of frequent itempairs, the number of elements in set E_i^2 can be significantly larger than the number of elements in set E_i. Thus, we need to allow a larger space budget in order to achieve useful accuracy. However, due to practical constraints on communication cost, we cannot allow the space budget to increase quadratically. Rather, we select a space budget that is about an order of magnitude larger than the one for frequent items. Specifically, we use a space budget of 4Mb for the local sketch used to count the elements of set E_i^2. Later we explore experimentally the impact of this choice on the accuracy of the analysis of frequent itempairs.

Next, we select the number of rows in the sketch to be proportional to the number of (unordered) pairs of elements drawn from set $\hat{H}$. This is an upper bound on the number of elements in any set E_i^2. Thus, the analytics server computes this bound R as the smallest power of 2 not smaller than $|\hat{H}|(|\hat{H}|-1)/2$. However, if R very large, it makes the cost of computing the local sketch impractical. This is because the number of rows in the sketch determines the number of times hash functions need to be applied. We have observed that when many hash values need to be computed, the cost of constructing the local sketch becomes impractical. Thus, we heuristically limit the number of sketch rows to 2^{14}—that is, $R = \min(2^{14}, 2^{\lceil \log_2 |\hat{H}|(|\hat{H}|-1)/2 \rceil})$. Given this number of rows R, the number of columns is set to be $4\text{Mb}/(2R)$.

Table 5.2 shows the number of rows and columns in the sketches used to estimate the frequency of itempairs. This is the result of one representative run of the LDP data collection. Column "#estimated hot items" shows the number of items estimated as hot items from that particular run, that is, the size of set $\hat{H}$. Column "#possible pairs" shows the number

App	#estimated hot items	#possible pairs	#rows	#columns
apartmentguide	512	130816	16384	128
reststops	218	23653	16384	128
rent	455	103285	16384	128
shipmate	334	55611	16384	128
cookbook	164	13366	16384	128
channels	147	10731	16384	128
infowars	170	14365	16384	128
loop	147	10731	16384	128
opensnow	125	7750	8192	256

Table 5.2: Number of possible pairs constructed from the estimated hot items and the number of sketch rows/columns for a space budget of 4Mb.

$|\hat{H}|(|\hat{H}|-1)/2$ of unordered pairs that can be constructed from $\hat{H}$. Columns "#rows" and "#columns" shows the number of rows and columns determined using 4Mb space budget. For the first four apps the number of possible pairs is quite large, so sketch matrices with 2^{14} rows were used as described above.

5.3 Experimental Evaluation

5.3.1 Analysis of Frequent Items

The proposed analysis of hot items was applied on the 9 apps with randomized sketches under 256Kb space constraint. The number of rows and columns used for each app is shown in Table 5.1. Table 5.3 shows results of the analysis for 1000 users. Column "#estimated hot items" shows the number of items identified as hot items. Column "RE" shows the relative error computed over the estimated hot items. Columns "Precision" and "Recall" shows the precision and recall of identifying the hot items. As it is evident from the results in Chapter 4 that the variance of these values are small, the results are presented as the average of three runs.

App	#estimated hot items	RE	Precision	Recall
apartmentguide	508.6	0.085467	0.946351	0.927425
reststops	213.6	0.061417	0.912699	0.928571
rent	464.3	0.082769	0.974890	0.975575
shipmate	339.3	0.080018	0.957950	0.961538
cookbook	163.3	0.050544	0.973721	0.963636
channels	142.6	0.053381	0.960317	0.971631
infowars	168.0	0.078670	0.992178	0.974659
loop	146.3	0.056796	0.993166	0.975391
opensnow	127.0	0.044244	0.981708	0.997333

Table 5.3: Number of estimated hot items, and relative error, precision and recall of identification of estimated hot items with a 256Kb space budget and 1000 users.

Table 5.3 shows that even a small space budget of 256Kb and a relatively small number of users can ensure high accuracy of the frequency estimates of the hot items. The relative error of frequency estimates are below 10% for every app. This analysis also demonstrates high precision and recall – for two apps the precision and recall is above 0.9 and they are above 0.95 for the other apps. This indicates most of the true hot items were identified correctly and the number of falsely identified hot items is also low. Table 5.4 shows the results for 10000 users under the same space budget. With more users, the relative error over the estimated hot items is lower for every app. It also shows an improvement in precision and recall. The effects of larger number of users on precision and recall is shown in Figure 5.3.

5.3.2 Analysis of Frequent Itempairs

As described in Section 5.2, the analysis of hot itempairs propagates the set $\hat{H}$ of estimated hot items to each user u_i, where set E_i^2 is computed and then stored in a local sketch. The global sketch constructed by the analysis server from these local sketches is then used to produce frequency estimates for itempairs. The performance of this approach is

App	#est-hot-items	RE(hot)	Precision	Recall
apartmentguide	511	0.070372	0.958257	0.932698
reststops	211.6	0.049128	0.929594	0.950081
rent	464.6	0.059715	0.97993	0.981322
shipmate	347.6	0.057013	0.982774	0.970644
cookbook	165	0.025235	0.993939	0.987952
channels	142.3	0.025136	0.976784	0.985816
infowars	171	0.029349	0.996113	0.996101
loop	146.3	0.019695	0.990899	1
opensnow	125.6	0.011948	0.997354	0.994709

Table 5.4: Number of estimated hot items, and relative error, precision and recall of identification of estimated hot items with a 256Kb space budget and 10000 users.

illustrated in Table 5.5 for 1000 users. The results were obtained with 4Mb space budget for the local sketch. The averages of three runs are presented in the table. In each run the analysis of hot items is first performed to compute set $\hat{H}$, followed by the analysis of hot itempairs. The first two columns show the number of true hot itempairs and the number of estimated hot itempairs, respectively. Column "RE" shows the relative error computed over the estimated hot itempairs. Columns "Precision" and "Recall" shows the precision and recall of identifying the hot itempairs. Since app `apartmentguide` does not have any true hot itempairs, recall is undefined and precision is 0; measurements for this app are added for completeness and will not be discussed further. As can be seen from these results, for six of the eight apps the accuracy of estimated frequencies is high.

5.3.2.1 Factors Affecting the Accuracy of Estimates

Next, we present an investigation of the effect of number of users and sketch size on the accuracy of estimates. The same itempairs analysis was performed on the apps over 10000 users under a 4Mb space budget, and over 1000 users under a 16Mb space budget. The

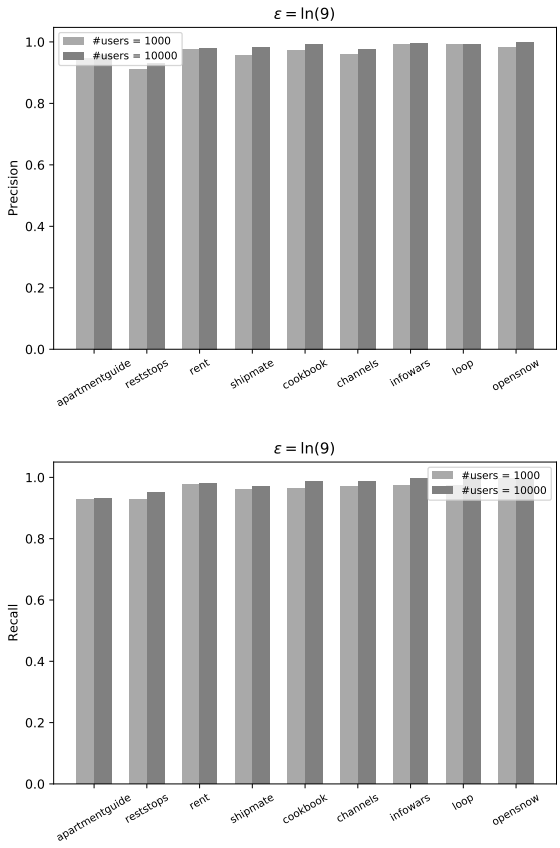

Figure 5.3: Precision and recall of frequency estimates of the estimated hot items under a space budget of 256Kb.

results are presented in Tables 5.6 and 5.7. These results are obtained from one representative run of each analysis. Comparisons between this data and the results in Table 5.5 are shown by the charts in Figure 5.4 and Figure 5.5.

Effects of number of users. Figure 5.4 shows a comparison of precision and recall of identification of the hot pairs for 1000 and 10000 users under the same space budget 4Mb. Similar to the analysis in previous chapters, larger number of users results in better accuracy. This effect is most significant for the two apps exhibiting low accuracy of estimates. Still, for those two apps, the precision and recall do not show a large degree of improvement.

App	#true hot pairs	#est hot pairs	RE	Precision	Recall
apartmentguide	0	312	1.183008	0	–
reststops	2002	2146.3	0.090108	0.839151	0.899434
rent	3703	11478.3	0.531024	0.208933	0.647493
shipmate	185	930.3	0.407134	0.144274	0.724324
cookbook	5024	5004	0.08326	0.927222	0.923501
channels	2851	2932.7	0.070268	0.901713	0.927394
infowars	2015	1958.7	0.074059	0.897977	0.872787
loop	6222	6324.7	0.089719	0.914513	0.929605
opensnow	7626	7660	0.084761	0.993304	0.997727

Table 5.5: Number of true and estimated hot pairs, and relative error, precision and recall of identification of estimated hot pairs with a 4Mb space budget and 1000 users.

Effects of space budget. Figure 5.5 shows a comparison of precision and recall using different space budgets. The same analysis was performed on the data for 1000 users with two different space budgets: 4Mb and 16Mb. The number of rows and columns of the sketch matrix were determined as described in Section 5.2.2. An increased space budget implies using a sketch with larger number of columns, so with a 16Mb space budget, the number of sketch columns is quadrupled the number of columns of a sketch under a 4Mb budget. An overall conclusion that can be drawn from these experiments is that increasing the number of columns in the sketch does not result in substantial improvements.

5.3.2.2 Study of Cases with Low Accuracy

As can be seen from these results, apps `rent` and `shipmate` achieve significantly lower precision and recall compared to the rest of the apps. To study further the underlying causes, we performed an experiment where the analysis of frequent itempairs used a 4Mb budget and 1000 users, but did not perform any randomization of the generated local sketches. This approach is not differentially private and is only used to establish a comparison baseline. For

App	#true hot pairs	#est hot pairs	RE	Precision	Recall
apartmentguide	0	1	0.288998	0	–
reststops	1942	2003	0.050692	0.910135	0.938723
rent	2761	5154	0.221269	0.38475	0.718218
shipmate	114	224	0.138198	0.397321	0.780702
cookbook	5022	5024	0.060253	0.948447	0.948825
channels	2702	2708	0.044968	0.932792	0.934863
infowars	1675	1690	0.044112	0.905325	0.913433
loop	6099	6162	0.064824	0.940604	0.950320
opensnow	7626	7675	0.064390	0.992834	0.999213

Table 5.6: Number of true and estimated hot pairs, and relative error, precision and recall of identification of estimated hot pairs with a 4Mb space budget and 10000 users.

app shipmate, RE is 0.090673, precision is 0.552239, and recall is 0.8. Compare this with the results in Table 5.5, where RE is 0.407134, precision is 0.144274, and recall is 0.724324. For app rent, RE is 0.161917, precision is 0.504046, and recall is 0.740211. In Table 5.5 the corresponding measurements are 0.531024, 0.208933, and 0.647493 respectively. As can be seen from these comparisons, the effects of randomization do contribute significantly to the inaccuracy. However, even when the effects of randomization are eliminated, overall precision and recall for these two apps are still low; ideally, we would like to see their values being close to 0.9, which is typically the case for the remaining apps even with randomization.

The reason for these results is that the number of itempairs added to the global sketch (without randomization) is significantly larger for these two apps compared to the rest of the apps. To illustrate this, we determined the total number of itempairs being counted in the global sketch on one representative run. These measurements are shown in Table 5.8. Column $|\cup_i (E_i^2)|$ shows the total number of itempairs over all of 1000 users for each app. It is evident that the numbers of itempairs reported in these two apps are significantly higher

App	#true hot pairs	#est hot pairs	RE	Precision	Recall
apartmentguide	0	267	1.103211	0	–
reststops	2002	2171	0.080876	0.84339	0.914585
rent	3703	10145	0.459942	0.239724	0.656765
shipmate	185	751	0.343826	0.170439	0.691892
cookbook	5024	5046	0.066214	0.940151	0.944268
channels	2851	2911	0.058881	0.917898	0.937215
infowars	2015	2050	0.067583	0.888780	0.904218
loop	6222	6336	0.071106	0.929451	0.94648
opensnow	7626	7651	0.064235	0.995948	0.999213

Table 5.7: Number of true and estimated hot pairs, and relative error, precision and recall of identification of estimated hot pairs with a 16Mb space budget and 1000 users.

App	$\lvert \cup_i (E_i^2) \rvert$
apartmentguide	130802
reststops	23428
rent	109272
shipmate	57624
cookbook	14027
channels	10438
infowars	14535
loop	11026
opensnow	8128

Table 5.8: Total number of itempairs reported over 1000 users.

than that in the other apps. Thus, the space budget is not sufficient to achieve as accurate results as for the other apps.

To explore this observation further, we ran the analysis on app shipmate with various sketch dimensions and no randomization. The results (relative error, precision, and recall) are shown in Table 5.9. The table also shows the average time taken (in seconds) to report all itempairs for one user; these measurements are in column "Average time per user (s)"

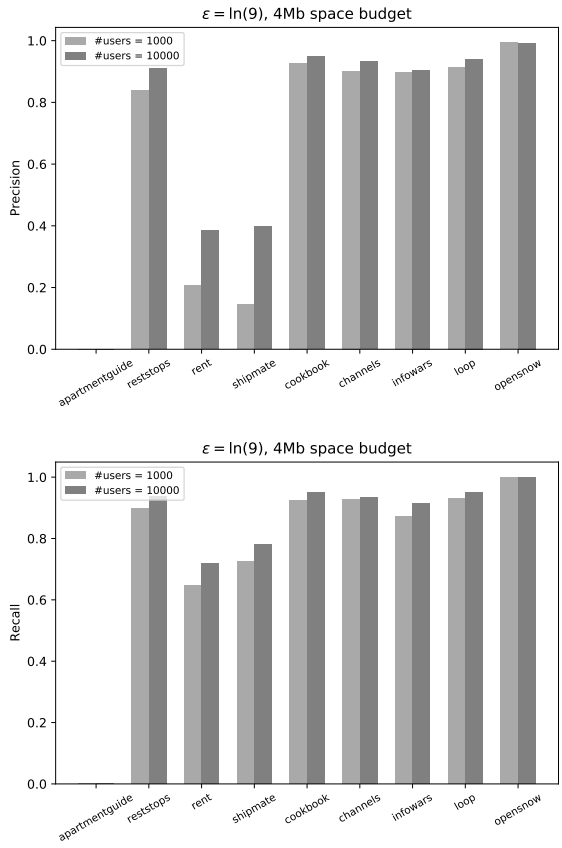

Figure 5.4: Precision and recall of frequency estimates of the estimated hot pairs under a space budget of 4Mb.

column. From these results it can be seen that metrics can improve significantly with a larger space budget, but the number of rows directly affects the processing time. Thus, there is a trade-off between the space budget and processing time: under the same space budget, increasing the number of columns instead of the number of rows can result in an improvement of the results and shorter computation time. As discussed earlier, we consider 16384 rows to be the largest number of rows that is practical to use, as larger numbers of rows lead to local computation times that are too large.

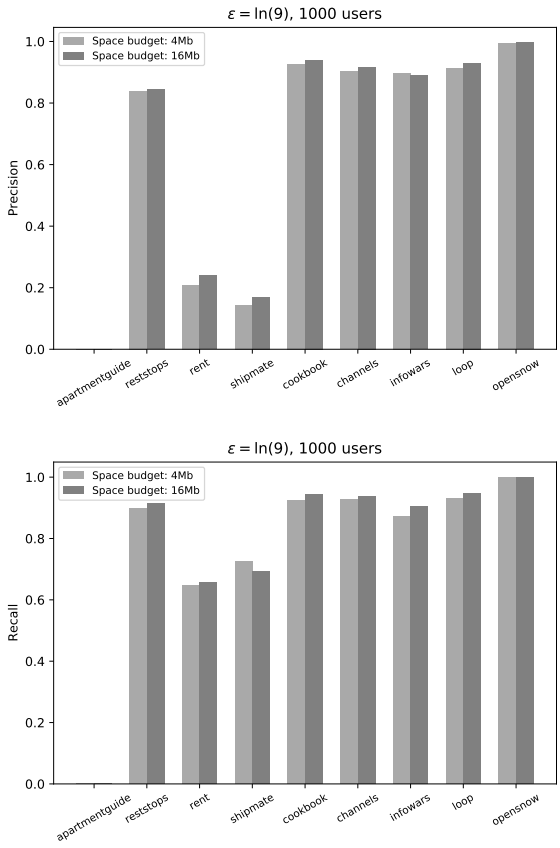

Figure 5.5: Precision and recall of frequency estimates of the estimated hot pairs under space budgets of 4Mb and 16Mb.

Another exploratory experiment was run on `shipmate` with randomization using $\varepsilon = \ln(9)$ (similar to all other randomized experiments in this chapter) and the number of sketch columns increased to 512 and 1024, i.e., using 4 and 8 times as much space budget compared to the experiments run with a space budget of 4Mb. The results are presented in Table 5.10. It can be seen from these measurements that with an increased space budget, the proposed analysis can achieve relative error, precision, and recall comparable with the results achieved with smaller space budget and no randomization.

#rows	#columns	RE	Precision	Recall	Average time per user (s)
16384	128	0.090673	0.552239	0.800000	114.16
16384	256	0.052068	0.694064	0.821622	106.96
32768	128	0.054879	0.716216	0.859459	229.73
32768	256	0.032728	0.833333	0.891892	218.95

Table 5.9: Relative error, precision and recall for 1000 users of `shipmate` with various sketch sizes and no randomization.

#rows	#columns	RE	Precision	Recall
16384	512	0.068202	0.648276	0.824561
16384	1024	0.05983	0.660256	0.903509

Table 5.10: Relative error, precision and recall for 1000 users of `shipmate` with various sketch sizes and randomization for $\varepsilon = \ln(9)$.

To summarize the results of our study: for the two apps that show low accuracy of estimates in Table 5.5, the underlying reasons are both the randomization effects and the small sketch size. Randomization effects become less pronounced with increased numbers of users, as illustrated in Figure 5.4. Increasing sketch size is beneficial as well, but the number of rows is limited by practical considerations of sketch computation time.

5.3.3 Summary of Experimental Evaluation

Our results indicate that analysis of frequent items can be done with high accuracy even with a small space budget and a small number of users. However, the analysis of itempairs is more challenging. For some of the analyzed applications, using a 4Mb space budget (which is still practical) and 1000 users, the achieved accuracy is good. The accuracy improves further when the number of users is increased. However, for some apps there is significant

inaccuracy of the estimates. To distinguish between these two categories, a set of opt-in users could be used. Such users can opt into the data collection and the analysis of accuracy can be performed for their raw data. Such analysis can consider the practical trade-offs between computation time, sketch size, and achieved accuracy.

Chapter 6: Related Work

Differential privacy. A few examples of prior work on differential privacy were already discussed briefly [5, 20, 55, 58, 59]. In particular, Zhang et al. [58] target the now-deprecated Google Analytics for mobile apps and use randomization to perturb each event to achieve differential privacy for event frequency reporting. Their follow-up work [59–61] extracts and applies consistency constraints on frequencies to improve accuracy and/or privacy. 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 [5, 20, 55] and from the mobile app frequency analysis for fixed and static app data. 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 [5, 7], estimates of unknown distributions [16], and clustering such as k-means [41]—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 [4, 13, 15, 20, 38, 54]. 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. [36] focus specifically on analytics libraries and propose the Alde tool for static and dynamic analysis of the data collection. Chen et al. [10] take advantage of the vulnerabilities in two analytics libraries to manipulate user profiles to control ad delivery. Seneviratne et al. [51] study tracking libraries in popular paid apps and find that more than half of these apps contain at least one tracker. LinkDroid [23] tackles unregulated aggregation of app-usage behaviors. Han et al. [30] 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 [2, 52, 57]. 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 [29] 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 [6, 8, 12, 19, 28, 35, 37, 48, 49, 53]. 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.

Remote software analysis. Many prior efforts have studied the remote analysis of deployed software. Coverage information from software users has been used for testing in a study on residual coverage monitoring [47]. In another study, GAMMA [45] demonstrates data collection from users across program instances. Some other projects addressed placement of profiling probes [14, 43], failure reproduction and debugging [11], and analysis of post-deployment failure reports [44] using data collected from deployed software. Privacy in remote software analysis has also been studied in prior work. There are techniques for anonymization of collected data [12, 19]. There are also studies that show that anonymization does not guarantee strong privacy [39, 40]. We consider the privacy protection provided by local differential privacy. There is prior work on impact analysis and regression testing [46] and failure reports [31, 33, 34] which could potentially benefit from adopting differentially-private techniques.

Chapter 7: Conclusions

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, described in Chapter 3, 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 different problem parameters affect the accuracy of the produced frequency estimates.

While Chapter 3 focuses on collection of randomized analytics data along with the distributed content from the users, Chapter 4 provides another approach to deal with the same problem with more privacy. The use of a differentially-private count sketch allows the approach to hide the set of content items retrieved by a user and thus provides higher privacy, at the expense of somewhat lower accuracy of the frequency estimates. This approach works particularly well to identify the most frequent elements in a differentially-private way, which is a major goal of collecting analytics data.

Chapter 5 considers another aspect of collecting analytics data: finding frequent items and itempairs under space constraints. We have extended the analysis in Chapter 4 to identify the most frequent items and pairs of items, while using a fixed practical amount of space and maintaining the same privacy guarantees. We found that even with a small number of users and a small space budget for randomized sketches, highly-accurate estimates can be obtained for the hot items and their frequencies. Our results further indicate that in many cases frequent itempairs can also be identified accurately, but practical limits to this accuracy are imposed by the number of app users and the space budget.

Bibliography

[1] UI Automator. https://github.com/xiaocong/uiautomator.

[2] Y. Agarwal and M. Hall. ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *MobiSys*, pages 97–110, 2013.

[3] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 207–216, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915925. doi: 10.1145/170035.170072. URL https://doi.org/10.1145/170035.170072.

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

[5] R. Bassily, K. Nissim, U. Stemmer, and A. Thakurta. Practical locally private heavy hitters. In *Advances in Neural Information Processing Systems*, pages 2285–2293, 2017.

[6] A. Budi, D. Lo, L. Jiang, et al. kb-anonymity: A model for anonymized behaviour-preserving test and debugging data. In *Programming Language Design and Implementation*, pages 447–457, 2011.

[7] M. Bun, J. Nelson, and U. Stemmer. Heavy hitters and the structure of local privacy. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 435–447, 2018.

[8] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–328, 2008.

[9] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

[10] T. Chen, I. Ullah, M. A. Kaafar, and R. Boreli. Information leakage through mobile analytics services. In *HotMobile*, pages 15:1–15:6. ACM, 2014.

[11] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *International Conference on Software Engineering*, pages 261–270, 2007.

[12] J. Clause and A. Orso. Camouflage: Automated anonymization of field data. In *International Conference on Software Engineering*, pages 21–30, 2011.

[13] 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`, Sept. 2017.

[14] M. Diep, M. Cohen, and S. Elbaum. Probe distribution techniques to profile events in deployed software. In *International Symposium on Software Reliability Engineering*, pages 331–342, 2006.

[15] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *Advances in Neural Information Processing Systems*, pages 3571–3580, 2017.

[16] J. Duchi, M. Jordan, and M. Wainwright. Local privacy and statistical minimax rates. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 429–438, 2013.

[17] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[18] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284, 2006.

[19] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *International Symposium on Software Testing and Analysis*, pages 65–75, 2004.

[20] Ú. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1054–1067, 2014.

[21] Exodus Privacy. Most frequent app trackers for Android. `https://reports.exodus-privacy.eu.org/en/reports/stats`, 2020.

[22] Facebook. Facebook analytics. `https://analytics.facebook.com`, 2020.

[23] H. Feng, K. Fawaz, and K. G. Shin. Linkdroid: reducing unregulated aggregation of app usage behaviors. In *USENIX Security Symposium*, pages 769–783, 2015.

[24] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. page 57–76, 2007.

[25] Google. Google Analytics for Firebase use policy. https://firebase.google.com/policies/analytics, .

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

[27] Google. Firebase. https://firebase.google.com, 2020.

[28] M. Grechanik, C. Csallner, C. Fu, and Q. Xie. Is data privacy always good for software testing? In *International Symposium on Software Reliability Engineering*, pages 368–377, 2010.

[29] 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*, 23(1):259–289, 2018.

[30] 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.

[31] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 146–155, 2005.

[32] 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*, pages 398–410, 2014.

[33] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *International Conference on Software Engineering*, pages 474–484, 2012.

[34] W. Jin and A. Orso. F3: Fault localization for field failures. In *International Symposium on Software Testing and Analysis*, pages 213–223, 2013.

[35] 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. *IEEE Transactions on Software Engineering*, pages 1–21, 2017.

[36] X. Liu, S. Zhu, W. Wang, and J. Liu. Alde: Privacy risk analysis of analytics libraries in the android ecosystem. In *SecureComm*, pages 655–672, 2016.

[37] D. Lo, L. Jiang, A. Budi, et al. kbe-anonymity: Test data anonymization for evolving programs. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 262–265, 2012.

[38] Microsoft. New differential privacy platform co-developed with harvard's opendp unlocks data while safeguarding privacy. `https://blogs.microsoft.com/on-the-issues/2020/06/24/differential-privacy-harvard-opendp/`, June 2020.

[39] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, pages 111–125, 2008.

[40] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *IEEE Symposium on Security and Privacy*, pages 173–187, 2009.

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

[42] Oath. Flurry. http://flurry.com, 2020.

[43] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit. Optimizing customized program coverage. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 27–38, 2016.

[44] P. Ohmann, A. Brooks, L. D'Antoni, and B. Liblit. Control-flow recovery from partial failure reports. In *Programming Language Design and Implementation*, pages 390–405, 2017.

[45] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. GAMMA system: Continuous evolution of software after deployment. In *International Symposium on Software Testing and Analysis*, pages 65–69, 2002.

[46] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 128–137, 2003.

[47] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *International Conference on Software Engineering*, pages 277–284, 1999.

[48] F. Peters and T. Menzies. Privacy and utility for defect prediction: Experiments with MORPH. In *International Conference on Software Engineering*, pages 189–199, 2012.

[49] F. Peters, T. Menzies, L. Gong, and H. Zhang. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering*, 39(8): 1054–1068, 2013.

[50] Sable. Soot – A framework for analyzing and transforming Java and Android applications. https://soot-oss.github.io/soot, 2020.

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

[52] 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 *International Conference on Software Engineering*, pages 25–36, 2016.

[53] K. Taneja, M. Grechanik, R. Ghani, and T. Xie. Testing software in age of data privacy: A balancing act. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 201–211, 2011.

[54] Uber. Uber releases open source project for differential privacy. `https://medium.com/uber-security-privacy/differential-privacy-open-source-7892c82c42b6`, July 2017.

[55] T. Wang, J. Blocki, N. Li, and S. Jha. Locally differentially private protocols for frequency estimation. In *USENIX Security Symposium*, pages 729–745, 2017.

[56] T. Wang, N. Li, and S. Jha. Locally differentially private heavy hitter identification. *IEEE Trans. Dependable Sec. Comput.*, 2019.

[57] 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 *International Conference on Software Engineering*, pages 37–47, 2018.

[58] H. Zhang, S. Latif, R. Bassily, and A. Rountev. Introducing privacy in screen event frequency analysis for Android apps. In *International Working Conference on Source Code Analysis and Manipulation*, pages 268–279, 2019.

[59] H. Zhang, Y. Hao, S. Latif, R. Bassily, and A. Rountev. A study of event frequency profiling with differential privacy. In *ACM SIGPLAN International Conference on Compiler Construction (CC)*, Feb. 2020.

[60] H. Zhang, Y. Hao, S. Latif, R. Bassily, and A. Rountev. Differentially-private software frequency profiling under linear constraints. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), Nov. 2020.

[61] H. Zhang, S. Latif, R. Bassily, and A. Rountev. Differentially-private control-flow node coverage for software usage analysis. In *USENIX Security Symposium*, pages 1021–1038, 2020.