Differential Privacy for Analysis of Software Traces

Dissertation

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

By

Yu Hao

Graduate Program in Computer Science and Engineering

The Ohio State University

2023

Dissertation Committee:

Atanas Rountev, Advisor

Raef Bassily

Michael D. Bond

© Copyright by

Yu Hao

2023

Abstract

Remote software profiling is employed by software developers to learn how their software is used by client users. However, this data collection process does not provide guarantees on the privacy of users' data. *Differential privacy* (DP) is a promising mathematical framework that provides rigorously defined data privacy for users. Some techniques have been developed to incorporate DP as part of remote software profiling. This dissertation is the first to study remote software execution *trace profiling* under differential privacy, where a trace is a sequence of run-time events. Prior related work has not considered DP software trace profiling, but rather the simpler problem of DP profiling for individual events.

The first contribution of this work focuses on the *coverage of software traces*, where each user's data is the set of traces she covers at run time. The goal of the software developers is to estimate how many users cover a given trace, while providing differential privacy guarantees for the coverage data shared by each individual software user. Such DP analysis requires that the domain of possible traces be defined ahead of time. Randomization over such domains is challenging due to their large (or even infinite) size, which makes it impossible to use prior randomization techniques. To solve this problem, we propose to use count sketch, a fixed-size hashing data structure for summarizing frequent items. Our techniques develop a randomization approach for count sketch that achieves the desired DP protections. We also propose an efficient algorithm to identify high-frequency ("hot") traces.

The second contribution generalizes the trace coverage analysis to *trace frequency analysis*, where each user reports not only the set of traces she covers, but also their frequencies, i.e., how many times a trace is covered by her execution of the software. Correspondingly, the developers want to learn the trace frequencies across all users. Our approach to this problem still employs count sketch due to the exponentially large domain. Additionally, to achieve the DP guarantee, the randomization is done by adding random values drawn from the Laplace distribution. The parameters of this distribution are closely related to the desired privacy protections, and we develop several techniques for selecting these parameters, for two distinct protection scenarios: hiding the presence of certain traces, and hiding their "hotness".

The third contribution of this work tackles two major obstacles for practical deployment of the scheme proposed by the second contribution. First, the size of the count sketch has significant effect on the DP protections that are achieved by the approach. We conduct characterization on the effect of number of rows used in the count sketch. Based on this study, we propose to amend the data collection scheme with pre-deployment configuration of sketch rows. As a result, we show that high accuracy of the desired frequency estimates can be achieved while preserving strong privacy guarantees. Second, we study potential under-randomization, which weakens the DP promise. We propose to mitigate this issue by adjusting the amount of noise added to users' raw data between data collection rounds.

Two exemplars of software execution traces are used to demonstrate the proposed approaches. First, we use a call chain analysis in which traces are described through a regular language. Second, we study an enter/exit trace analysis in which traces are described by a balanced-parentheses context-free language. Our experimental studies of call chain

analysis and enter/exit trace analysis indicates that the DP frequency estimates for both trace coverage and trace frequency achieve high accuracy and high privacy.

The growing adoption of differential privacy for practical use, together with its rigorous foundations, provide strong motivation to study DP software analyses. The work described in this dissertation presents promising findings that contribute to broader efforts to integrate privacy-preserving techniques in the analysis of deployed software, in response to growing needs for better privacy of data collection.

To my family

Acknowledgments

I would like to thank especially my advisor, Atanas Rountev, for his continuous support and patient guidance during the Ph.D. program. Without him, this dissertation would not have been possible. I also would like to thank Raef Bassily, Mike Bond, Zhiqiang Lin, and Radu Teodorescu for serving on my defense and/or candidacy committees. I thank Mike Bond, Feng Qin, and Yinqian Zhang for the interesting paper-reading courses. My collaborators in the PRESTO group, Hailong Zhang and Sufian Latif, have always been helpful and supportive. I had two great mentors, Jan Voung and Wontae Choi, during my internship at Google. Lastly, I want to thank my family, friends, and others for being supportive and helping me navigate through difficulties.

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

December 2021 .... M.S., Computer Science and Engineering, The Ohio State University, Columbus, Ohio, US.

June 2018 .... B.S., Computer Science, Beijing Institute of Technology, Beijing, China.

Publications

Research Publications

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

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Programming Language and Software Engineering	Prof. Atanas Rountev
Privacy-preserving Data Analysis	Prof. R. Bassily
Hardware Security	Prof. R. Teodorescu

Table of Contents

Page

Abstract ii

Dedication v

Acknowledgments vi

Vita vii

List of Tables xii

List of Figures xiii

1. Introduction 1
   - 1.1 Overview and Outline 3
   - 1.2 Contributions and Impact 5

2. Background 6
   - 2.1 Differential Privacy 6
     - 2.1.1 Randomized Response 7
     - 2.1.2 The Laplace Mechanism 9
   - 2.2 Count Sketch 10

3. Local Differential Privacy for Coverage Analysis of Software Traces 11
   - 3.1 Background and Problem Statement 11
     - 3.1.1 Software Traces 11
     - 3.1.2 Trace Coverage Analysis for Deployed Software 14
     - 3.1.3 Assumptions 15
   - 3.2 Randomized Count Sketch for Software Traces 16

3.2.1 Count Sketch 17
3.2.2 Sketch Randomization 20
3.2.3 Efficient Randomization 21
3.2.4 Server-Side Processing 22
3.2.5 Selecting Sketch Size 22
3.3 Identification of Hot Traces 25
3.4 Evaluation 28
3.4.1 Accuracy for All Covered Traces 31
3.4.2 Precision and Recall for Hot Traces 32
3.4.3 Accuracy of Estimates for Reported Hot Traces 35
3.4.4 Privacy Loss Parameter 36
3.4.5 Summary of Results 36
3.5 Conclusions 37

4. Local Differential Privacy for Frequency Analysis of Software Traces 38

4.1 Problem Statement 38
4.1.1 Frequency Analysis for Software Traces 38
4.1.2 The Differential Privacy Guarantee 39
4.2 Proposed Approach for Frequency Analysis 41
4.2.1 Randomized Count Sketch with Laplace Noise 41
4.2.2 Data Collection 44
4.2.3 Hiding Trace Information 46
4.2.4 Selecting Sketch Size 47
4.3 Evaluation 48
4.3.1 Hiding The Presence of Traces 49
4.3.2 Hiding The Hotness of Traces 52
4.3.3 Identifying Hot Traces 56
4.3.4 Local Cost 58
4.4 Summary 61

5. Deploying LDP Frequency Analysis of Software Traces 62

5.1 Reducing The Privacy Budget 64
5.1.1 Characterization Study of the Number of Sketch Rows 64
5.1.2 Configuring the Number of Sketch Rows 65
5.1.3 Evaluation 69
5.2 Potential Under-Randomization 71
5.2.1 Mitigating Under-Randomization 73
5.2.2 Evaluation 75
5.3 Summary 84

6. Related Work 85

7. Conclusions 88

Bibliography 89

List of Tables

Table **Page**

3.1 Experimental subjects. 30

4.1 Ratio of the approximate value of τ from test users to the ground truth with varying percentiles for hiding trace presence. The ratios are average values of 30 runs. 51

4.2 Ratio of the approximate value of τ from test users to the ground truth with varying percentiles for hiding the hotness. The ratios are average values of 30 runs. 55

4.3 Cost of building local randomized sketches, averaged over 900 users. 59

5.1 The count sketch size (average of 5 runs) decided by the in-house characterization stage of the amended approach, for frequency analysis using $\varepsilon = 2$ and 50% protection of presence. 70

5.2 Percentage of users with under-randomization for call chain analysis and enter/exit trace analysis with protecting 50% presence and hotness, $\varepsilon = 2.0$. Averaged over 5 runs. 74

List of Figures

Figure **Page**

3.1 Count sketch illustration, with $m = 8$ and $s = 3$ 18

3.2 Error of estimates for all covered traces. 32

3.3 Recall and precision for hot traces. 33

3.4 Recall and precision for hot traces: strict vs relaxed hotness criterion. . . . 34

3.5 Error of estimates for reported hot traces. 35

3.6 Error of estimates for all covered traces for three values of ε. 37

4.1 Data collection scheme for frequency analysis. 45

4.2 Normalized error (NE) for frequency estimates for call chains with varying privacy budget ε and percentage of hidden (presence) traces x. 53

4.3 Normalized error (NE) for frequency estimates for enter/exit traces with varying privacy budget ε and percentage of hidden (presence) traces x. . . . 54

4.4 Normalized error (NE) for frequency estimates for call chains with varying privacy budget ε and percentage of hidden (hotness) traces x. 56

4.5 Normalized error (NE) for frequency estimates for enter/exit traces with varying privacy budget ε and percentage of hidden (hotness) traces x. 57

4.6 Recall and precision for identifying hot traces averaged over 9 combinations of varying privacy budget ε $(0.5, 1.0, 2.0)$ and percentage of hidden traces x $(25, 50, 75)$. 60

5.1 The normalized error of the estimates of all call chains with variant number of sketch rows in frequency analysis. 66

5.2 The normalized error of the estimates of all enter/exit traces with variant number of sketch rows in frequency analysis. 67

5.3 Data collection scheme with configuration of number of sketch rows for frequency analysis. 68

5.4 Normalized error (average of 5 runs) of estimates for test users versus real users in the frequency analysis. 71

5.5 Data collection scheme with incremental update of τ for frequency analysis. 75

5.6 Under randomization (UR) rate (average of 5 runs) by batch for hiding the presence of 50% call chains, $\varepsilon = 2.0$. 76

5.7 Under randomization (UR) rate (average of 5 runs) by batch for hiding the hotness of 50% call chains, $\varepsilon = 2.0$. 77

5.8 Under randomization (UR) rate (average of 5 runs) by batch for hiding the presence of 50% enter/exit traces, $\varepsilon = 2.0$. 78

5.9 Under randomization (UR) rate (average of 5 runs) by batch for hiding the hotness of 50% enter/exit traces, $\varepsilon = 2.0$. 79

5.10 Normalized error (average of 5 runs) by batch for hiding the presence of 50% call chains, $\varepsilon = 2.0$. 80

5.11 Normalized error (average of 5 runs) by batch for hiding the hotness of 50% call chains, $\varepsilon = 2.0$. 81

5.12 Normalized error (average of 5 runs) by batch for hiding the presence of 50% enter/exit traces, $\varepsilon = 2.0$. 82

5.13 Normalized error (average of 5 runs) by batch for hiding the hotness of 50% enter/exit traces, $\varepsilon = 2.0$. 83

Chapter 1: Introduction

This dissertation proposes privacy-preserving mechanisms for remote software trace profiling. More specifically, the proposed mechanisms allow software developers to collect software execution traces from the population of software users in order to gain knowledge of how their software is used in the field, while providing privacy guarantees to participating users. A trace is a sequence of run-time events generated by the deployed software when a user interacts with it. The goal of the mechanisms is to achieve a balance between data utility for software developer and privacy for participating software users.

Remote software profiling has been both studied in academia and deployed in practice extensively. It collects information about executions of software deployed on client users to provide valuable feedback to software developers, facilitating and benefiting the process of debugging [12, 30, 31, 33, 41], performance optimization [3, 5, 51], and testing [10, 45]. While collecting such data from users, it is important to not potentially jeopardise their privacy, either intentionally or accidentally. During the past decade, we see stronger demands and efforts for the protection of data privacy. The importance of reducing the amount of user information collected by business entities has increased. Both societal and legislative pressures have highlighted the need for such reduction. However, for software-generated event information—for example, collected with the help of popular analysis infrastructures for mobile/web analytics (e.g., provided by Google and Facebook)—typically there are no

"built-in" privacy protection mechanisms. The infrastructures themselves collect a wealth of information, including user IP addresses and GUI events. App-specific data collection can provide even more fine-grained knowledge about user's behavior and interaction with the software. For example, trace information can provide details about what paths through the code a user has taken, and what functionality (possibly sensitive) she has executed. This data could potentially be used to infer user-specific habits, interests, and characteristics.

From the point of view of software users, the release of data collected from software executions is often undeclared or obscured. Even if the user is aware of the data collection, they are unlikely to have true appreciation of its implications. What is particularly troubling is that the collected data could be linked with other sources of information about this user (and such linking cannot be prevented even with anonymization [36, 37]) and could be used as part of future larger-scale data mining and machine learning attempts to infer user-specific information. At data collection time, it is impossible to predict what extra data sources will be linked and what future inferences will be possible using the collected data.

Privacy-preserving data analysis aims to develop systematic mechanisms for addressing this problem. Such analysis benefits two categories of stakeholders. First, the privacy of individual users is protected in a well-defined manner. Further, entities performing data collection (e.g., Google and app developers using Google's analytics infrastructure) benefit as well: they are responsive to privacy expectations and do not have access to raw data that can be compromised by unexpected data leaks or unethical business practices. Our work focuses on one particular privacy-preserving mechanism: *local differential privacy* (LDP). This model provides stronger privacy guarantee than its counterpart (the so-called centralized DP) in that the latter requires a trusted "aggregator". On the other hand, LDP does not assume the presence of a trusted aggregator: in essence, software users do not

have to trust software developers or analytics infrastructure providers such as Google and Facebook. In essence, an LDP software analysis adds random noise to the local data of a software user, and then reports the randomized data to the remote software analytics server.

1.1 Overview and Outline

Two different properties—coverage and frequency—with respect to the profiling of software traces are extensively studied in this dissertation. We propose solutions to integrating differential privacy to the data collection process, and design experiments to evaluate the prposed solutions. The rest of the dissertation is structured as follows.

Background on Differential Privacy and Count Sketch. Chapter 2 lays out some background on the theory of differential privacy. We use an example in a simplified setting to illustrate the promise of differential privacy, and describe two classic DP mechanisms, which are used later in our proposed software analyses. Count sketch, a hash-based data-structure, is also briefly described.

Software Trace Coverage Analysis. Chapter 3 presents a novel differentially private mechanism for software trace coverage analysis. Due to the large domain of software traces, naive techniques that performs randomization directly on the original data are not applicable in this setting. We employ a hash-based data structure, *count sketch*, that is used in data science to estimate the frequency of popular data items in a large population. Using the count sketch as a frequency oracle, we developed an algorithm to allow the software analytics server to efficiently identify "hot" traces, i.e., traces whose frequencies are above some pre-defined threshold. We describe two popular exemplars of software traces—call chains and enter/exit traces—and evaluate the approach on them. Our experimental evaluation indicates that the proposed mechanism preserves the privacy of participating users while

allowing developers to learn accurate coverage information about the executed traces in their software. This work appeared in the 35th European Conference on Object-Oriented Programming[27].

Software Trace Frequency Analysis. Chapter 4 considers the frequency analysis of software traces, a generalization of the coverage analysis. Instead of learning how many users cover a given trace, in this scenario, developers aim to learn how many times a trace is covered by the entire population of users. On top of count sketch, the Laplace Mechanism is employed to achieve strong DP guarantee. We demonstrate that a combination of these two techniques indeed achieves the desired LDP theoretical guarantees. Further, by parameterizing the randomization, our approach provides the developers with the flexibility to specify which traces need to be protected and whether their presence or hotness are intended to be hidden. Experiments on the two exemplar trace trace analyses outlined above demonstrate that our approach is efficient and highly accurate.

Deploying LDP Frequency Analysis. Chapter 5 studies two major obstacles for our solution to the frequency analysis that pose challenges for practical deployment. Both problems are related to the balance between accuracy and privacy. First, in the approach from the previous chapter, the number of rows used in count sketch is large, which benefits accuracy but reduces the privacy guarantee. We conduct a characterization study of this effect and then, based on the insights from the study, propose to amend the data collection scheme with pre-deployment configuration of sketch rows. The second problem is the potential under-randomization, which happens when insufficient Laplacian noise is added for some users' frequency data. We propose to mitigate this issue by adjusting the amount of noise added to users' raw data between data collection rounds. Our results show that with

the help of these two techniques, high accuracy of the desired frequency estimates can be achieved while preserving strong privacy guarantees.

1.2 Contributions and Impact

Our work studies how to apply differential privacy to software trace profiling, an area that has not been studied in prior work. While a variety of techniques have been proposed for efficient remote software profiling, protecting user's privacy during the profiling process has not been paid enough attention by the research community. Some recent work has studied other profiling problems such as method frequency profiling and control-flow graph node coverage analysis in the context of DP. This dissertation is the first work that applies DP to the profiling of software execution traces. Novel approaches are designed to resolve the challenges posed by software traces, for both coverage analysis and frequency analysis. Experimental results demonstrate the accuracy, privacy, and efficiency of these approaches. In summary, our work advances the state of the art by expanding the current spectrum of differentially-private remote software profiling with effective novel techniques for software trace analysis.

Chapter 2: Background

2.1 Differential Privacy

Differential privacy is applicable to data analyses where data is being collected from many participants, and some processing of this data produces results that are then made available to untrusted parties. Such untrusted parties could be, for example, government agencies and business entities. Two main models of differential privacy have been considered [16]. In the *centralized* model, a trusted "data curator/aggregator" collects the raw data from participants, performs the data analysis, and releases the results to untrusted entities. As part of the data analysis, some form of randomization is applied to ensure the differential privacy guarantee (this guarantee will be described shortly). In the *local* model, the randomization is performed by each participant, and the resulting modified data is then released to untrusted entities, which perform data analysis on this data. Again, the randomization ensures the differential privacy guarantee. Our work focuses on the second scenario, which is well suited for analysis of deployed software. In the specific problems we consider, the raw data for software user u_i is the set (Chapter 3) or multiset (Chapter 4) T_i of locally-covered traces. The user applies a local randomizer R to this data and then reports $R(T_i)$. We assume a typical setting where the reported data is collected by an untrusted analysis server. This

server analyzes the data from all users and computes an estimate $\hat{f}(t)$ of the true frequency $f(t)$ for a software trace t.

2.1.1 Randomized Response

To illustrate this key *indistinguishability property* promised by differential privacy, we present a classic simplified example. For illustration, suppose that the raw data for user u_i is a single trace $t_i \in \mathcal{T}$, and the goal of the untrusted analysis server is to learn an estimate of the global histogram—that is, an estimate of the true frequencies $f(t) = |\{i : t_i = t\}|$ for all $t \in \mathcal{T}$. A well-know randomization technique is derived from *randomized response*, an approach used in social sciences to handle evasive answers to sensitive questions [57]. The randomizer $R : \mathcal{T} \to \mathcal{P}(\mathcal{T})$ takes as input a single trace t and produces a set of traces, based on the following rules: (1) the input t is included in the output with some probability p, and (2) for every other $t' \in \mathcal{T}$, t' is included in the output with probability $1 - p$. Thus, the real trace could be missing from the output, and any other trace could be part of the output. Note that this approach is applicable only when $\mathcal{T}$ is finite and, practically, the size of $\mathcal{T}$ is relatively small.

By selecting $p = e^{\frac{\varepsilon}{2}}/(1+e^{\frac{\varepsilon}{2}})$, this approach provably achieves ε-indistinguishability: for any set $Z \subseteq \mathcal{T}$ and any two traces $t' \in \mathcal{T}$ and $t'' \in \mathcal{T}$, the probabilities $P[R(t') = Z]$ and $P[R(t'') = Z]$ can differ by at most a factor of e^{ε}. In other words, observing Z means that (1) the raw data that produced Z could have been any trace from $\mathcal{T}$, and (2) no trace from $\mathcal{T}$ is much more likely to have been the input, compared to the remaining elements of $\mathcal{T}$.

In this simplified problem, each user u_i reports $R(t_i)$ to the analysis server; here $1 \leq i \leq n$. The server produces estimates $\hat{f}(t)$ by computing $h(t) = |\{i : t \in R(t_i)\}|$ and then calibrating it in order to create an unbiased $f(t)$ estimate: $\hat{f}(t) = ((1+e^{\frac{\varepsilon}{2}})h(t) - n)/(e^{\frac{\varepsilon}{2}} - 1)$.

Differential privacy guarantee The above local differential privacy mechanism ensures the following differential privacy property: *for every user u_i, an external observer of $R(t_i)$ cannot have high confidence that the hidden raw data is t_i*. In other words, whether the data of user u_i is t_i cannot be ascertained with high probability based only on the observation of $R(t_i)$.

More precisely, let $P[R(X) = Z]$ be the probability that given input X, the randomizer produces output Z. For any Z and any two different $X \in \mathcal{T}$ and $Y \in \mathcal{T}$, the ratio of $P[R(X) = Z]$ and $P[R(Y) = Z]$ should be bounded by e^{ε}. Here X and Y are considered to be "neighbors" in the space of inputs to the randomization algorithm. Because the two probabilities are close to each other, when someone observes any output Z, she cannot have much higher confidence in the statement "the raw data is X", compared to the confidence she can have in the statement "the raw data is not X". Here ε is the *privacy loss parameter*, which is used to tune accuracy/privacy trade-offs. A typical value used in related work is $\ln(9)$ [18, 55, 61]; for example, this value is used in the "basic one-time" version of a popular randomization technique [18]. Larger values of ε improve the accuracy of analysis results, but weaken the privacy guarantee.

A key assumption is that the adversarial observer of $R(t_i)$ knows fully all details of how randomizer R works, for example, because this observer designed the randomizer in the first place, or because she reverse-engineered it from the program code. As part of this assumption, the observer also knows the value ε which was embedded in the randomizer design. Even under such strong assumptions, the differential privacy guarantee makes it impossible to distinguish, in a probabilistic sense, neighbor inputs to the randomizer after the randomizer output is publicly released. Such principled and quantifiable protection is one of the reasons differential privacy has been employed by companies such as Google

[18], Microsoft [35], Apple [4], and Uber [54], as well as by the U.S. Census Bureau [14]. More widespread use of such protection has become possible via recent open-source tools for differentially-private analysis [42].

2.1.2 The Laplace Mechanism

Another way to achieve ε-Differential Privacy is by adding Laplacian noise, known as the *Laplace Mechanism*. Here we give a brief discussion on the algorithm. More details can be found in Section 3 of the classic description by Dwork and Roth [16].

Consider the same scenario as illustrated above where each user u_i contributes a single trace $t_i \in \mathcal{T}$. The local raw data by the user is represented as an one-hot bit vector $\boldsymbol{t_i}$ of length $|\mathcal{T}|$. Each trace in $\mathcal{T}$ is uniquely mapped to a bit in $\boldsymbol{t_i}$. Only the bit mapped to trace t_i is set to 1, and all other bits are 0. The local randomizer R perturbs the raw data by adding random variables drawn from the Laplace Distribution. Formally, the output of the local randomizer is $R(\boldsymbol{t_i}) = \boldsymbol{t_i} + (Y_1, ..., Y_{|\mathcal{T}|})$ where Y_i are independent and identically distributed random variables drawn from $Lap(2/\varepsilon)$. Here $Lap(2/\varepsilon)$ is the Laplace Distribution centered at 0 with scale $2/\varepsilon$.

This approach achieves the same ε-indistinguishability as in the randomized response mechanism described earier. For any output $Z \in \mathbb{R}^{|\mathcal{T}|}$ of the randomizer and any pair of one-hot bit vectors $\boldsymbol{t_i}$ and $\boldsymbol{t_j}$, the ratio of the probabilities $P[R(\boldsymbol{t_i}) = Z]$ and $P[R(\boldsymbol{t_j}) = Z]$ is bounded by e^{ε}. Actually, the scale of the Laplace Distribution is set as $2/\varepsilon$, because $\max \|\boldsymbol{t_i} - \boldsymbol{t_j}\|_1 = 2$, i.e. the maximum ℓ_1 distance between $\boldsymbol{t_i}$ and $\boldsymbol{t_j}$ is 2.

Only the randomized vectors are shared with the analytics server. To get the histogram of frequencies for all traces in $\mathcal{T}$, the server computes the sum of the randomized vectors, i.e. $\boldsymbol{T} = \sum_{i \in \{1,...,n\}} R(\boldsymbol{t_i})$. Each position in $\boldsymbol{T}$ is the estimate of frequency for the trace mapped

to that position. Note that the mappings between positions in the vector and traces are the same at each user's side and the server side.

2.2 Count Sketch

Count sketch [11] is a hash-based data structure originally designed to find frequent items in data streams. Our approaches in both Chapter 3 and Chapter 4 employ count sketch, combined with the differential privacy mechanisms discussed earlier.

Counts sketch is based on s pairs of independent hash functions (h_k, g_k), for $1 \leq k \leq s$, such that $h_k : \mathcal{T} \rightarrow \{1, \ldots, m\}$ and $g_k : \mathcal{T} \rightarrow \{+1, -1\}$. The data in count sketch is represented as a matrix of counters where the number of rows is s and the number of columns is m. Assume each user i holds a set of data items T_i. In our techniques, T_i is used to create a *local count sketch* S_i. This sketch is then randomized in order to achieve the desired differential privacy properties. The randomized local sketch is then shared with the analysis server.

Each user initializes her local count sketch S_i with all zeros. Then the data items in T_i are encoded into the local sketch using the hash functions. For every data item $t \in T_i$, the value of $g_k(t)$ is added to $S_i[k, h_k(t)]$, for every $1 \leq k \leq s$. In essence, for every row k in the matrix, we use hash function h_k to hash t into a value from $\{1, \ldots, m\}$, and then update a counter for the corresponding column with $+1$ or -1 depending on hash function g_k. After receiving the local sketches S_i from all the users, the server constructs a global sketch S_g by element-wise addition of S_i. Finally, for any $t \in \mathcal{T}$, a frequency estimate can be obtained by reporting the median value of $S_g[k, h_k(t)] \times g_k(t)$ over all $1 \leq k \leq s$.

Chapter 3: Local Differential Privacy for Coverage Analysis of Software Traces

This chapter proposes a novel approach for profiling software trace coverage information in a differentially private manner. In the setting of coverage analysis, the goal is to, for each software trace in a domain of interest, get the number of users whose execution covers the trace. Due to the extremely large domain of all possible traces, applying differential privacy mechanism directly on the data items is inefficient or even impossible if the domain is unbounded. Our approach employs *count sketch* to represent the data and proposes an efficient algorithm for the randomization that achieves differential privacy. We describe two exemplars of software traces, which we use to evaluate our approach. The experiments demonstrate that both user privacy and data utility can be achieved. The work presented in this chapter appeared at the 35th European Conference on Object-Oriented Programming [27].

3.1 Background and Problem Statement

3.1.1 Software Traces

We consider software traces, collected over a set of software users u_i for $i \in [1,n]$. Each user u_i executes her own copy of the software. During execution, run-time events are observed and recorded. Let $\mathcal{E}$ be the finite set possible run-time events. This set is defined

before software deployment, as part of the design of the trace analysis. For convenience of definitions, we assume that $\mathcal{E}$ contains an artificial "start" event s denoting the start of a trace. A trace t is a string $t \in \mathcal{E}^+$, starting with s. We will use the notation $t = \langle s, e_1, \ldots, e_k \rangle$ to denote a trace t of length k. (Note that we exclude s when defining trace length.)

Let $\mathcal{T}$ be a domain describing conservatively (i.e., over-approximating) the set of all possible traces that could be observed at run time. We expect this domain to be statically described as part of the design of the trace analysis. In the simplest case, $\mathcal{T} = \mathcal{E}^+$. However, the traces typically have structure that is constrained by the static properties of the software. In particular, one important special case we investigate is when $\mathcal{T}$ is defined inductively through a family of "extension" functions ext_k: $\mathcal{E}^k \times \mathcal{E} \to \mathcal{P}(\mathcal{E}^{k+1})$. Here $\mathcal{P}(X)$ denotes the power set of X and k ranges over the natural numbers. For any $t \in \mathcal{T}$ of length k, $\mathsf{ext}_k(t)$ is the set of all traces $t' \in \mathcal{T}$ of length $k+1$ such that t is a prefix of t'. That is, $\mathsf{ext}_k(t)$ shows all ways in which t could be extended with one more event to form a valid trace. For simplicity, we will omit the subscript k in ext_k when it is clear from the context. As discussed later, this definition of $\mathcal{T}$ enables incremental search for "hot" traces.

Below we discuss two examples of such trace domains $\mathcal{T}$, both with direct connections to popular categories of analyses. These exemplars illustrate how common properties of such analyses can be mapped to the problem definition and solution described in this work. In particular, we define these two domains via well-understood formal languages—a regular language and a balanced-parentheses context-free language—which provides a natural definition for the domain and its extension function. As a result, our approach is directly applicable to other trace analyses where the trace domain has a similar structure.

Both domains are based on a set of events corresponding to entering or exiting a software component (e.g., method, module, or GUI window). We simplify the definition by assuming

that each component is uniquely identified by an integer id from $[1,c]$. In addition, we introduce an artificial component with id 0 which corresponds to the external environment—e.g., the caller of the main method, or the framework code that invokes Android app entry points. The set of events is then $\mathcal{E} = \mathit{Enter} \cup \mathit{Exit}$ where $\mathit{Enter} = \{\mathsf{enter}(i) : i \in [0,c]\}$ and $\mathit{Exit} = \{\mathsf{exit}(i) : i \in [0,c]\}$. The artificial start event s is $\mathsf{enter}(0)$.

3.1.1.1 Exemplar 1: Call Chains

We first describe an exemplar analysis in which the static domain $\mathcal{T}$ of possible traces is defined by a simple regular language. Suppose that we are given a set of static *call edges* $i \to j$ showing that, at run time, the execution of component i may trigger the execution of component j. A finite sequence $i \to j \to k \to \ldots$ of such call edges is a static *call chain*. A call chain denotes a trace of events "i calls j which in turn calls k which in turn calls ...". Equivalently, we can define the domain $\mathcal{T}$ through a regular language containing strings $t = \langle \mathsf{enter}(0), \mathsf{enter}(i_1), \ldots, \mathsf{enter}(i_k) \rangle$ over the alphabet *Enter*. The static call graph can be thought of as the finite-state automaton that defines this language, and the extension function is the transition function of that automaton.

3.1.1.2 Exemplar 2: Enter/Exit Traces

Next we define an exemplar analysis in which $\mathcal{T}$ is based on a balanced-parentheses context-free language. This language captures the standard notion of *interprocedurally valid paths* [48] and is defined by the following grammar:

$$\begin{array}{rcl} \mathit{Valid} & \to & \mathsf{enter}(i)\ \mathit{Valid} \mid \mathit{Balanced}\ \mathit{Valid} \mid \lambda \\ \mathit{Balanced} & \to & \mathsf{enter}(i)\ \mathit{Balanced}\ \mathsf{exit}(i) \mid \mathit{Balanced}\ \mathit{Balanced} \mid \lambda \end{array}$$

where λ is the empty string. Non-terminal *Balanced* defines a sequence of matching enter and exit events. Starting non-terminal *Valid* describes a sequence with some not-yet-matched

enter events. Grammars of similar structure have been used extensively in a wide variety of static analyses (e.g., [48, 50]). For our exemplar analysis we consider the domain of enter/exit traces $\mathcal{T}$ to be strings derived from *Valid* and starting with $\mathsf{enter}(0)$. We further restrict the strings to respect a given set of static call edges $i \to j$. This can be easily encoded in the definition of the corresponding pushdown automaton, as follows. We can define a deterministic pushdown automaton with a single state. The input alphabet is $\mathit{Enter} \cup \mathit{Exit}$ and the stack alphabet is *Enter*, with initial stack symbol $\mathsf{enter}(0)$. The transitions upon observing input event $\mathsf{enter}(j)$ when the top of the stack is $\mathsf{enter}(i)$ is defined only if there is a static call edge $i \to j$. This transition pushes $\mathsf{enter}(j)$ onto the stack. If the input symbol is $\mathsf{exit}(i)$, the transition is defined only if the top of the stack is $\mathsf{enter}(i)$, in which case the stack top is popped. The trace extension function ext, which captures all ways in which a given trace is extended with one more event, is easily derivable from this pushdown automaton.

There are two reasons we use these formalisms to describe our exemplar analyses. First, the underlying structure, defined by a finite-state automaton or a balanced-parentheses pushdown automaton, is commonly observed in a variety other of dynamic analyses. Our machinery can be directly employed for such analyses. Second, the automata naturally provide the definition of incremental algorithms to explore the domain of possible traces. As described later, such algorithms play an important role in our identification of frequently-occurring domain elements.

3.1.2 Trace Coverage Analysis for Deployed Software

When the program is executed by a software user, some subset of $\mathcal{T}$ is actually observed (i.e., covered) at run time. A variety of run-time techniques can be used to determine this coverage (e.g., [2, 7, 52, 65]). We consider such coverage across a large number of software

users, each running her copy of the program. Let there be n software users denoted by $u_1, \ldots, u_n$ and let $T_i \subseteq \mathcal{T}$ be the set of traces covered when user u_i executes the program. We consider the following *trace coverage analysis*: for each $t \in \mathcal{T}$, estimate the frequency of t over the population of users, that is, $f(t) = |\{i : t \in T_i\}|$, while collecting the local data of each user with differential privacy.

Trace information has been used extensively to analyze and optimize software performance [1, 2, 5, 7, 26, 65]. The frequency information defined above can be used to focus such efforts on important user behaviors. Similarly, testing and static checking can be focused on traces that are commonly observed in the user population. Another example is behavior flow analysis in mobile and web analytics frameworks [21, 39], which allows developers to see different paths that users take through the app. The paths can be thought of as a form of traces across GUI components, and the analysis annotates each edge with the number of users who have performed the corresponding transition. A similar example is funnel analysis [19, 21, 22, 39], which visualizes the completion rate of a given task in terms of a series of specific events and helps developers find optimizations in their software design.

3.1.3 Assumptions

Several assumptions need to be explicitly stated before we describe our differentially-private analysis (Section 3.2). As usual in this type of work, it is assumed that the design and implementation of the approach are fixed before any data collection and are publicly known by all stakeholders, including untrusted parties. Another assumption is that the software code correctly implements the design; in particular, it does implement the randomization as publicly announced, and does not try to circumvent it by sending the raw data (or some

version of it) to a malicious party. Although this is a strong assumption, it is no different than what is currently used in remote analysis of deployed software, where the design is typically undocumented and/or obfuscated, and there is no checking of the implementation of the data collection for correctness or presence of malicious code.

If a software developer commits to using the correct design and implementing it as expected, this raises the confidence of software users and watchdog agencies that indeed privacy is protected. Further, several techniques can be used to increase this confidence, including (1) open-source implementations, (2) use of certified and trusted third-party libraries, (3) scrutiny by privacy experts, and (4) code analysis via automated tools. Note that there are no assumptions about the analysis server to which the randomized data is sent. This server could be part of a privacy attack, possibly involving additional external sources of information about the targeted software user. Even with this assumption, the differential privacy guarantee holds [58].

3.2 Randomized Count Sketch for Software Traces

Even if a user's local information contains a single trace, the approach outlined in the previous section is not possible when $\mathcal{T}$ is infinite, since every elements of $\mathcal{T}$ must be visited when randomization is applied. Even if $\mathcal{T}$ is made finite—for example, by using a pre-defined limit on trace length—the approach is still not practical. For illustration, consider call chains for the `localtv` Android app used in our experiments. The alphabet size $|Enter| = 2974$ in this app is close to the median for our set of benchmarks. Even if we only consider chains of at most three methods and count the strings recognized by the corresponding finite-state automaton (as described in Section 3.1.1.1), we have $|\mathcal{T}| = 3,272,137$. Increasing this length by one, the size of $\mathcal{T}$ becomes more than 163

million. A further length increase by one results in $|\mathcal{T}|$ of over 8 billion. The cost of the randomizer described earlier is proportional to the size of $\mathcal{T}$, as each element $t \in \mathcal{T}$ must be visited and a random value must be generated for that t (independently of the processing of the remaining elements of $\mathcal{T}$) in order to decide whether t is included in the randomizer output. Further, the randomizer output, which needs to be sent to the analysis server, has size dependent on the exponentially-large size of $\mathcal{T}$. Clearly, these costs are infeasible.

3.2.1 Count Sketch

To address this problem we employ *count sketch* [11], a data structure originally designed to find frequent items in data streams. Prior work [8] has considered the theoretical analysis of using count sketch for a restricted form of differentially-private data analysis, where each user has a single data item. However, there is no clarity on the practical use of this data structure for analysis of real-world software execution data and for the more general problem we consider, where each user has a set of local traces. Using insights from this prior work, we develop a version of count sketch for our trace analysis and demonstrate its effectiveness on data from actual software executions. Section 2.2 already explains the basics about count sketch. Here we describe it again in the context of software traces. The next subsection shows how randomization can be applied to achieve the differential privacy guarantee.

Counts sketch in our setting is based on s pairs of independent hash functions (h_k, g_k), for $1 \leq k \leq s$, such that $h_k : \mathcal{T} \to \{1, \ldots, m\}$ and $g_k : \mathcal{T} \to \{+1, -1\}$. Here parameters s and m are chosen ahead of time; this choice will be discussed later. To perform analysis without differential privacy, each user would create a *local sketch* and then send it to the analysis server, where a *global sketch* is constructed and used to produce frequency estimates. The

Chain	$h_1(t), g_1(t)$	$h_2(t), g_2(t)$	$h_3(t), g_3(t)$
$t_1 = \langle 0, 473 \rangle$	$3, -1$	$6, -1$	$8, 1$
$t_2 = \langle 0, 93 \rangle$	$1, 1$	$7, 1$	$3, -1$
$t_3 = \langle 0, 473, 83 \rangle$	$5, -1$	$1, -1$	$4, -1$
$t_4 = \langle 0, 473, 472 \rangle$	$5, -1$	$4, 1$	$4, -1$
$t_5 = \langle 0, 473, 83, 1605 \rangle$	$5, 1$	$5, -1$	$2, 1$
$t_6 = \langle 0, 473, 472, 971 \rangle$	$8, 1$	$1, -1$	$7, 1$
$t_7 = \langle 0, 473, 472, 973 \rangle$	$7, -1$	$3, -1$	$4, 1$

Local Sketch

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

Figure 3.1: Count sketch illustration, with $m = 8$ and $s = 3$

local sketch for user u_i is a $s \times m$ matrix S_i initialized with 0 elements. For every locally-covered trace $t \in T_i$, matrix element $S_i[k, h_k(t)]$ is updated by adding to it the value of $g_k(t)$, for every $1 \leq k \leq s$. In essence, for every row k in the matrix, we use hash function h_k to hash t into a value from $\{1, \ldots, m\}$, and then update a counter for that value with $+1$ or -1 depending on hash function g_k. The local sketches S_i for all users are then sent to the analysis server, where a global sketch S_g is constructed by element-wise addition of all S_i. Finally, for any $t \in \mathcal{T}$, a frequency estimate can be obtained by reporting the median value of $S_g[k, h_k(t)] \times g_k(t)$ over all $1 \leq k \leq s$.

Example Figure 3.1 illustrates the local sketch for one user, based on data obtained from our implementation on one of our benchmarks (Android app `drumpads`). We use integer method ids to denote app methods. For example, id 473 corresponds to method `MainActivity.initOnboarding` and id 971 corresponds to `OnboardingView.createImageScene`. For brevity, the example uses the method id to signify an enter event for the corresponding method; id 0 corresponds to the start event.

Suppose that the locally-covered chains for some user are $t_1,\dots,t_7$. We illustrate count sketch with $m=8$ and $s=3$. Thus, each chain t is hashed into a value $h_k(t)\in\{1,\dots,m\}$ using three different hash functions (i.e., $1\le k\le 3$). An additional hash $g_k(t)$ produces a $+1/-1$ value. Accumulating these values, as described above, results in the local sketch shown at the bottom of the figure. For example, the first cell in the second row has a value of -2 because $h_2(t_3)=h_2(t_6)=1$ (i.e., both chains map to this cell), and $g_2(t_3)=g_2(t_6)=-1$ (i.e, both contribute -1 to the value of the cell). This also illustrates that hashing does produce collisions. Using s pairs of hash functions helps ameliorate this problem.

In this particular example the sketch accurately preserves the original information. Consider, for example, chain t_3. The cells for this chain, as determined by hashes h_k, are [1,5], [2,1], and [3,4] in [row,column] format. The corresponding cell values are -1, -2, and -1. The median value of $-1\times g_1(t_3)$, $-2\times g_2(t_3)$, and $-1\times g_3(t_3)$ is 1, which accurately reflects the raw local data.

The advantage of using this approach is that a local sketch S_i for user u_i provides a fixed-sized representation of the arbitrary subset T_i of the set $\mathcal{T}$ of possible traces. Further, randomization of the local sketch, as described shortly, can be performed in time proportional to this $s\times m$ sketch size. Thus, instead of recording the raw data T_i and randomizing it with randomized response to achieve the differential privacy guarantee over $\mathcal{T}$, we will record the sketch S_i and randomize it to achieve the differential privacy guarantee over local sketches. Finally, the count sketch technique is theoretically proven to produce accurate estimates for high-frequency items, which aligns well with our goal to produce information about frequently-occurring traces, as discussed further in Section 3.3.

3.2.2 Sketch Randomization

Trace-level randomization To introduce privacy-achieving randomization, for each locally-covered trace $t \in T_i$ the following actions are performed. First, for each row k in the local sketch S_i, the contribution of t to this row is expressed as a vector of length m (which is the number of columns in the sketch). The vector has the value of $g_k(t) \in \{+1, -1\}$ in position $h_k(t)$, and 0 values in all other positions. Then, the following randomization is applied to this vector:

- for each position with a 0 value, independently of any other positions in the vector, with equal probability the 0 value is replaced by $+1$ or -1
- for the position with the single $-1/+1$ value, the sign of this value is inverted with probability $1/(e^{\varepsilon}+1)$

The resulting randomized vector contains only $+1$ and -1 values. We can think of this process as applying a randomizer $R_k : \mathcal{T} \rightarrow \{+1, -1\}^m$. It can be proven that this approach achieves indistinguishability between t and any $t' \in \mathcal{T}$; the proof is omitted for brevity. Thus, for any vector Z containing m values $+1/-1$, the probabilities $P[R_k(t) = Z]$ and $P[R_k(t') = Z]$ differ by at most a factor of e^{ε}. By observing Z, a malicious observer cannot conclude with high confidence that the underlying trace was t as opposed to any other $t' \in \mathcal{T}$.

Set-level randomization Next we define the complete randomizer: given the local set of traces T_i, $R_k(T_i) = \sum_{t \in T_i} R_k(t)$ where the addition is element-wise. This definition satisfies the indistinguishability property in the following sense. Consider any $t \in T_i$ and any $t' \in \mathcal{T} \setminus T_i$. Let $T_i' = (T_i \setminus \{t\}) \cup \{t'\}$. Thus, T_i' is obtained by replacing t with t'. For any output Z of R_k, the probabilities $P[R_k(T_i) = Z]$ and $P[R_k(T_i') = Z]$ differ by at most a factor of e^{ε}.

Thus, an observer of Z cannot determine with high confidence that a particular trace t was present in T_i, as opposed to any other trace $t' \notin T_i$. The complete randomized local sketch is constructed as a $s \times m$ matrix in which row k is $R_k(T_i)$; we will denote this matrix by $R(S_i)$ where S_i is the non-randomized local sketch. This randomized local sketch is reported to the analysis server.

3.2.3 Efficient Randomization

The approach described above is impractically expensive. Specifically, for any $t \in T_i$ we need to compute s randomized vectors of length m, where each vector element requires drawing a random value. In our experience the cost of such processing could be high for data from actual software executions. Instead, we use an approach that first records the contributions of each t without randomization, and then draws random values from the binomial distribution to implement "one-shot" randomization.

Algorithm 3.1 describes the details of this approach. Consider a cell $[k, j]$ in the sketch. Let $N_{+1}[k, j]$ be the number of traces that contribute $+1$ to the value in this cell, without randomization. Similarly, let $N_{-1}[k, j]$ be the number of traces that contribute -1 to the cell. Our approach first records these counts (function `add`) without randomization. After data collection is completed, `finalize` computes the randomized sketch. With randomization, each of the $N_{+1}[k, j]$ occurrences of $+1$ contributes $+1$ with probability p and -1 with probability $1 - p$. Binomial distribution gives us the probability of y successes in x independent trials, where each trial succeeds with probability p. Let $binomial(x, p)$ denote a random value drawn from this distribution. The randomization will contribute $binomial(N_{+1}[k, j], p)$

values of $+1$ to the cell value; the remaining $N_{+1}[k,j] - binomial(N_{+1}[k,j],p)$ contributions will be -1. Thus, at line 19 of the algorithm we compute the cumulative contribution of the "raw" $+1$ values as the difference between these two quantities—that is, as $2 \times binomial(N_{+1}[k,j],p) - N_{+1}[k,j]$. A similar computation is performed at line 20 for the -1 values. Finally, we also have to account for the randomization of 0 values, which is done at line 21. The combined effect of these three cases is computed at line 22 as the cell value in the randomized sketch. This approach has cost in the order of $s \times m$, while a naive implementation with separate randomization for each observed trace will have cost in the order of $|T_i| \times s \times m$.

3.2.4 Server-Side Processing

The randomized local sketches $R(S_i)$ from all users are collected by the analysis server and their element-wise sum is computed. To obtain unbiased estimates, all elements of the sum need to be scaled by $(e^{\varepsilon}+1)/(e^{\varepsilon}-1)$. The resulting $s \times m$ matrix S_g is the *global sketch* produced by the analysis. For any $t \in \mathcal{T}$, an estimate $\hat{f}(t)$ of the true frequency $f(t)$ can be obtained as the median value of $S_g[k,h_k(t)] \times g_k(t)$ over all sketch rows k. This processing is described in Algorithm 3.2. It is important to note that summing up of the local sketches is essential in order for the randomized noises to "cancel out" across the population of users.

3.2.5 Selecting Sketch Size

The selection of sketch size is important for achieving high accuracy of estimates. In our implementation, both the number of rows s and the number of columns m are powers of 2. Parameter s is set to 256, which is similar to values used in prior work [8]. When selecting the number m of sketch columns, we aim to use a value that would produce a small

Algorithm 3.1: Randomized count sketch

```
   output : S_i: randomized local sketch
 1 Function init():
 2     S_i ← {0}^{s×m}
 3     N_{+1} ← {0}^{s×m}
 4     N_{−1} ← {0}^{s×m}
 5 Function add(t):
 6     T_i ← T_i ∪ {t}
 7     for k ← 1 to s do
 8         if g_k(t) = +1 then
 9             N_{+1}[k, h_k(t)] ← N_{+1}[k, h_k(t)] + 1
10         else
11             N_{−1}[k, h_k(t)] ← N_{−1}[k, h_k(t)] + 1
12 Function finalize():
13     p ← e^ε / (1 + e^ε)
14     for k ← 1 to s do
15         for j ← 1 to m do
16             z ← |T_i| − N_{+1}[k, j] − N_{−1}[k, j]
17             n_{+1} ← 2 × binomial(N_{+1}[k, j], p) − N_{+1}[k, j]
18             n_{−1} ← 2 × binomial(N_{−1}[k, j], p) − N_{−1}[k, j]
19             n_0 ← 2 × binomial(z, 1/2) − z
20             S_i[k, j] ← n_{+1} − n_{−1} + n_0
```

number of hash collisions. One simple choice is to select m to be similar to the total number of unique traces that would be represented in the global sketch—that is, similar to the size of the union of all local sets T_i. The value of m has to be selected ahead of time, before deployment, so that the randomization machinery is included in the distributed code. To make this selection, we use an approach similar in spirit to existing techniques [6, 62]. First, a group of *opt-in* users is used to obtain detailed information in a non-differentially-private manner. Specifically, the set of local traces T_i from each opt-in user u_i is collected and reported to the analysis server. Then, the union of these sets is determined. The value of

Algorithm 3.2: Server-side processing

```
1  Function global_sketch(R(S_1),...,R(S_n)):
2      S_g ← {0}^{s×m}
3      for i ← 1 to n do
4          S_g ← S_g + R(S_i)
5      S_g ← (e^ε+1)/(e^ε−1) × S_g
6  Function estimate(t):
7      E ← ∅
8      for k ← 1 to s do
9          E ← E ∪ { S_g[k, h_k(t)] × g_k(t) }
10     return median(E)
```

m is defined as the smallest power of 2 greater than or equal to the size of this union. This value of m is then used by the *regular* software users, whose copy of the software embeds this m value and only reports the randomized sketch of their local information.

In practice, there are several options for obtaining this opt-in data. First, some users may be willing to share their raw data. Even in this case, instead of the raw data the approach could collect some hashed version of it, which provides some degree of privacy protection (although weaker than differential privacy). Alternatively, such data could also be provided from in-house testing or beta testing. In our experiments, each run of the approach randomly picks 10% of the users as opt-in users, computes m based on their data, and then performs the rest of the experiment on the remaining 90% users.

The size of the sketch produced by this approach depends on the underlying volume of collected data. Suppose, for example, that there are a total of 15 thousand unique traces across all software users, which corresponds to $m = 2^{14}$. Assuming each sketch element is represented as a 2-byte integer, the total sketch size is 8MB, which is a practical amount of data to transfer. However, if the number of unique traces across the population of

software users is many hundreds of thousands, sketch size becomes impractical. If the goal is to achieve high accuracy of estimates while having a reasonably small amount of data communication with the analysis server, our approach would be most suitable for scenarios where the total number of unique traces reported from the user population is in the order of a few thousands to a few tens of thousands. Depending on the intended use of the analysis information, this could be a reasonable constraint. For example, if the analysis data is used to identify common user behaviors for the purposes of manual performance optimization or user interface redesign, it is unlikely that frequency estimates for hundreds of thousands of traces would be of value to software developers. To achieve such data sizes, a simple approach is to use pre-defined limits on the sizes of local sets or the lengths of collected traces. Our implementation limits the length of collected call chains to 10 events and the length of collected enter/exit traces to 20 events. This also bounds the depth of exploration for hot traces, which is described next.

3.3 Identification of Hot Traces

From the global sketch, the analysis server can estimate the frequency of any particular trace $t \in \mathcal{T}$. However, this is not enough for many forms of data analyses, since the size of $\mathcal{T}$ is very large (or even infinite) and obtaining an estimate for each t is not possible. Next we focus on one particular data analysis of significant practical importance: identifying the *hot traces* and estimating their frequencies. Hot traces are useful in identifying common user behavior, which themselves can be used for performance optimization, focused testing and static checking, and application-flow optimization. We consider a trace t to be hot if its true frequency $f(t) \geq h$, where $h = \alpha \times n$ is a "hotness threshold" defined by a parameter α and the number of software users n. The question is, *given the global sketch, how can we*

efficiently and accurately construct an estimate of the set of hot traces? Next, we develop an approach to answer this question.

Exploration of estimated hot traces Our approach takes as input the global sketch S_g, together with the set $\mathcal{E}$ of possible run-time events, the start event $s \in \mathcal{E}$, and the family of extension functions ext. We perform a pruned exploration of the elements of $\mathcal{T}$ defined by $\mathcal{E}$ and ext. The key observation is that if a trace t is *not* hot, any t' that has t as a prefix cannot be hot, since the number of users that covered t' cannot exceed the number of users that covered t. This leads to the following approach: starting with the length-0 trace $\langle s \rangle$, explore the space of possible trace extensions defined by ext. For each explored trace t, estimate its frequency using sketch S_g and stop the exploration if the frequency estimate $\hat{f}(t)$ is below the hotness threshold h. Otherwise, continue the exploration with all traces in $\mathsf{ext}(t)$.

A key assumption of this approach is that for any given trace t, the set of extended traces $\mathsf{ext}(t)$ can be computed efficiently. We chose the two exemplar analyses presented in Section 3.1—call chains and enter/exit traces—to illustrate two common cases where this computation is naturally derived from the definition of the underlying formal language. Such trace structure is not specific to these two examples; other dynamic analyses (e.g., paths in control-flow graphs) have similar properties. For call chains, the traces are strings in a regular language. Thus, the exploration is equivalent to exploring paths in the corresponding finite-state automaton. The extension function is defined by the set of possible transitions from the current state of the automaton. For enter/exit traces, defined by a Dyck context-free language (i.e., a language of balanced parentheses), the corresponding pushdown automaton can be maintained during the exploration of strings, and the extension function is again defined by the possible transitions from the current automaton state. Our implementation of

these exemplar analyses uses exactly this approach. In both cases, the transitions are efficient: the cost of computing $\mathsf{ext}(t)$ is linear in the size of this set. Note that this approach is also applicable in the more general case where $\mathcal{T}$ is defined by an arbitrary context-free grammar, as the corresponding pushdown automaton can be maintained during trace exploration and consulted to decide how to extend the current trace.

Relaxed hotness criterion Our experience indicates that the approach described above has the following disadvantage: sometimes entire groups of hot traces with a common prefix are not discovered because this prefix is misclassified as not being hot due to an inaccuracy of its frequency estimate. As a result, the exploration stops too early. To address this problem, we designed a more robust "relaxed" check for hot traces. If for some explored trace t we have $h/2 \leq \hat{f}(t) < h$, we consider this trace a potentially-misclassified hot trace due to an inaccurate estimate. In such cases, we check whether at least one $t' \in \mathsf{ext}(t)$ has an estimate above the threshold h. If such a t' exists, we take it as strong indication that t itself is hot and treat it as such. The details of the entire approach are presented in Algorithm 3.3.

For illustration, consider an enter/exit trace derived from actual data for the `equibase` app, which is one of our experimental subjects. The trace is $t = \langle \mathsf{enter}(0), \mathsf{enter}(1685), \mathsf{enter}(1678), \mathsf{enter}(910), \mathsf{enter}(805), \mathsf{enter}(10), \mathsf{exit}(10), \mathsf{exit}(805), \mathsf{exit}(910), \mathsf{enter}(1677)\rangle$. The true frequency is $f(t) = 818$. For the hotness cut-off $h = 810$ which was used in that experiment, the trace is hot. However, because of estimate $\hat{f}(t) = 763$, the exploration will stop at this trace if the relaxed criterion is not employed. As a result, 15 hot traces that have t as a prefix would be missed. Using the relaxed criterion, all 15 traces are correctly discovered by Algorithm 3.3.

Algorithm 3.3: Identification of hot traces

```
   output : H: set of estimated hot traces
 1 Function hot_traces():
 2   | H ← ∅
 3   | for t ∈ ext(s) do explore(t)
 4 Function explore(t):
 5   | if hot(t) then
 6   |   | H ← H ∪ {t}
 7   |   | for t' ∈ ext(t) do explore(t')
 8 Function hot(t):
 9   | e ← f̂(t)
10   | if e ≥ h then return true
11   | if e < h/2 then return false
12   | for t' ∈ ext(s) do
13   |   | if f̂(t') ≥ h then return true
14   | return false
```

3.4 Evaluation

For evaluation, we used 15 Android applications that were used by prior related work [60, 61]. We simulated 1000 users interacting with each app using the Monkey tool [23]. Specifically, we performed 1000 independent Monkey runs and considered each Monkey execution as triggered by one simulated user. During this process, for each run, we collected the sequence of method enter/exit events until the total number of enter events reaches $10\times$ the number of methods in the app code. From this sequence we constructed the set of observed call chains for that simulated user u_i—that is, set T_i for call chain analysis. In addition, we also considered the entry methods of the app and collect the subsequences that start at the enter events of those methods; these subsequences form set T_i for enter/exit trace analysis. Thus, for each of the two analyses we gathered sets $T_1, T_2, \ldots, T_{1000}$. We also wanted to study the effects the number of users, but since execution of a large number of

Monkey runs in device emulators takes a very long time, we employed an approach used by others [60]: each of the 1000 sets was replicated 10 times to generate T_i for $n = 10000$.

The instrumentation is based on the Soot code rewriting tool [49]. Given the data collected by the instrumentation, we ran all randomization separately from the executions that gather the traces. This allowed us to conduct each experiment 30 times, in order to report rigorous statistical results that account for the randomness introduced by local randomizers [20]. Experiments were performed for several values of ε used in prior work [18, 55, 60, 61]. For brevity, most results are presented for $\varepsilon = \ln(9)$, but the effects of other values are also discussed. To implement count sketch, we used SHA-256 hashing. In particular, hash functions h_k and g_k used in count sketch were implemented by prepending k to the string representation of the trace (which itself is based on the methods ids), computing SHA-256, and taking the appropriate number of bits from the result. We open sourced the implementation of the proposed approach, the analyzed traces, and the instructions for reproducing the experimental results. These artifacts are available at the following URL: `https://presto-osu.github.io/ecoop21/`.

Table 3.1 shows the details of the subjects used in our experiments. Column "Classes" lists the number of application classes, excluding several well-known third-party Android libraries, e.g., `dagger` and `okio`. The group of columns labeled "Call Chains" describes measurements for the call chain analysis, and the group labeled "Enter/Exit Traces" shows the same measurements for the analysis of enter/exit traces. Column "Total" shows the total number of unique traces across the 1000 local sets T_i. Column "Time$_u$" shows the average time (in seconds) to process the local data of a user, as described in Algorithm 3.1. Column "Time$_s$" contains the time (in seconds) to identify hot traces from the global sketch at the

Table 3.1: Experimental subjects.

App	Classes	Call Chains			Enter/Exit Traces		
		Total	Time_u	Time_s	Total	Time_u	Time_s
`barometer`	379	2765	0.3	25	2717	0.4	6.4
`bible`	1107	1604	0.2	64	2427	0.2	21
`dpm`	272	1272	0.1	4.3	2475	0.2	3.7
`drumpads`	447	926	0.1	6	1289	0.1	4.1
`equibase`	252	773	0.1	3.2	1602	0.3	4.9
`localtv`	716	4037	0.3	42	5285	0.3	12
`loctracker`	198	480	0.1	0.8	1098	0.1	8.9
`mitula`	973	24757	2.8	1784	5614	0.8	27
`moonphases`	166	1755	0.2	3.3	947	0.1	0.6
`parking`	379	1477	0.1	10	2875	0.2	4.6
`parrot`	1099	7575	0.8	427	6499	0.9	63
`post`	1107	2358	0.4	92	3564	0.5	45
`quicknews`	1107	3668	0.4	51	6062	0.7	57
`speedlogic`	86	244	0.0	0.1	304	0.0	0.3
`vidanta`	1608	7811	0.8	833	6687	0.9	124

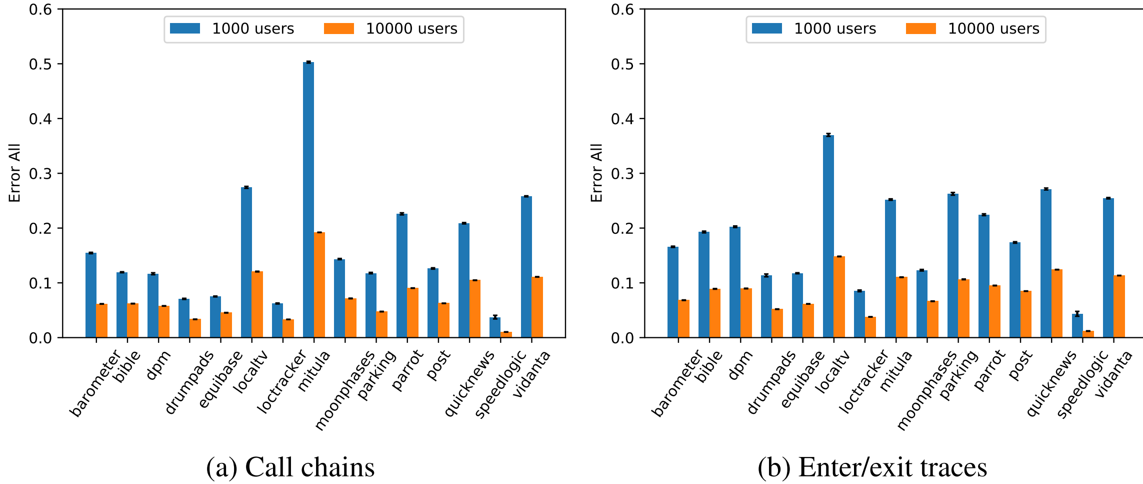
analysis server, using the approach from Algorithm 3.2 for $n = 1000$. For both analyses, the costs are practical and suitable for real-world use.

As mentioned in Section 3.2.3, we initially attempted to perform randomization separately for each covered trace, but incurred high running times for the local randomizer. This led to the development of the optimized approach in Algorithm 3.1. For example, for the `parrot` app, the naive randomization of call chains and enter/exit traces took 264 seconds per user on average, while the optimized one took 1.7 seconds. We typically observed two orders of magnitude improvement in the running time of the local randomizer.

3.4.1 Accuracy for All Covered Traces

The first research question we consider is this: *What is the accuracy of estimates for traces that are covered by at least one user?* Note that, from the data in the global sketch, the analytics server cannot directly determine this set. (We address this issue in the next subsection.) However, the knowledge of this accuracy provides a useful baseline. To answer this question, we use a normalized L_1 distance between the vector of true frequencies and the vector of their estimates. Specifically, for all t that appear in at least one T_i, we compute the error as $\sum_t |f(t) - \hat{f}(t)| / \sum_t f(t)$. Values close to 0 mean that the estimates are overall close to the real frequencies. Figure 3.2 shows these measurements for the two values of n. As described in Section 3.2.5, each run of this experiment (and all later experiments) used a randomly-selected set of size $n/10$ as opt-in users, and then performed the analysis and computed all reported error measurements for the remaining users. For these and all other experiments reported later, we followed a popular approach for statistically-rigorous performance measurements [20]: 30 independent runs of the experiment were performed, and the mean together with the 95% confidence interval are reported. The confidence interval characterizes the variance due to the randomization. In the bar charts, the interval is shown at the top of the corresponding bar. In many cases, the interval is so small (i.e., the variance is so low), that it is hard to see in the figures.

From this data, we reach the following answer to the above question: with sufficiently large number of users participating in the data collection, the estimates are close to the real frequencies. For example, for the call chain analysis with $n = 10000$, the cumulative error over all t is under 20% in all cases, and its average value across the 15 apps is 7.4%. Similarly, for the enter/exit trace analysis with $n = 10000$, the cumulative error over all covered traces is always under 15% and, averaged across the apps, is 8.4%. It is fairly



(a) Call chains

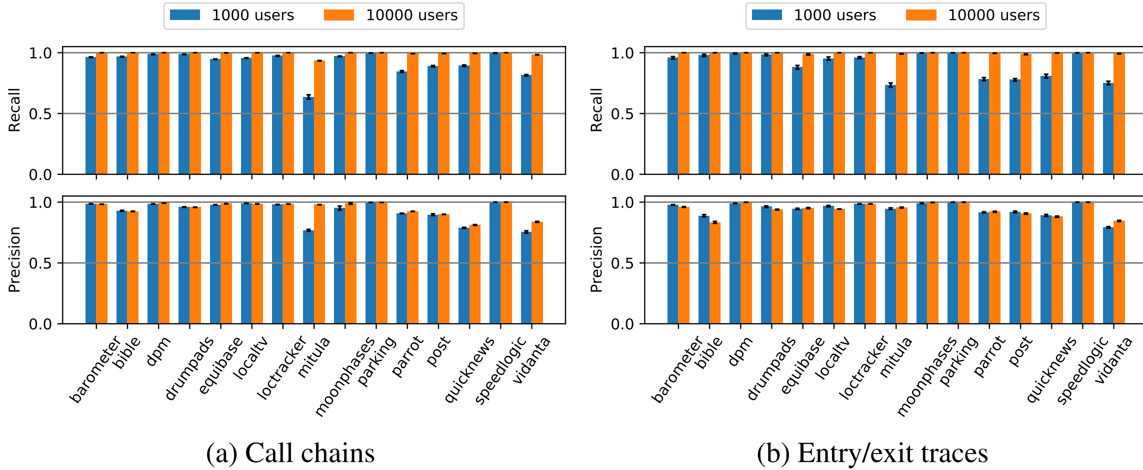
(b) Enter/exit traces

Figure 3.2: Error of estimates for all covered traces.

common for Android apps to have many thousands of users, and popular apps usually have hundreds of thousands of users. Thus, obtaining data from a sufficient number of app users should be feasible.

3.4.2 Precision and Recall for Hot Traces

As discussed earlier, the set of all covered traces is not directly known to the analysis server. Section 3.3 discussed an approach to identify *hot traces*. Our next research question is: *How accurately are the hot traces identified?* The metrics we use to answer this question are recall (what portion of the true hot traces are discovered) and precision (what portion of the reported hot traces are actually hot). We executed Algorithm 3.3 on the global sketch to identify likely hot traces, with hotness threshold $h = 0.9 \times n$. This was done in 30 independent repetitions of the experiment. The mean values from these experiments and their 95% confidence intervals are shown in Figure 3.3.



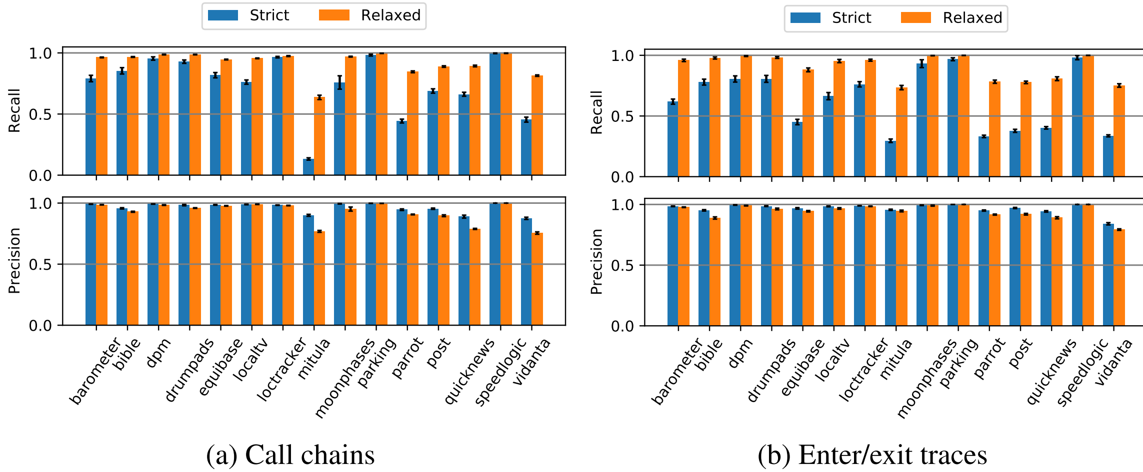
(a) Call chains

(b) Entry/exit traces

Figure 3.3: Recall and precision for hot traces.

Overall, the results of this experiment provide strong indication that hot traces can indeed be identified accurately with a sufficient number of users. For $n = 1000$, the average recall across the 15 apps is 92.1% and the average precision is 92.5% for call chains, and 90.4% and 94.5% for enter/exit traces respectively. For $n = 10000$, the recall for call chains increases to 99.3% and the precision to 95.0%; for enter/exit traces, the recall increases to 99.7% and precision decreases slightly to 94.1%. We investigated the apps with the lowest precision and determined that they have a large number of traces whose true frequencies are slightly below the threshold h; some of these almost-hot traces are misclassified as being hot, leading to the lower precision.

One related question is how the design choices for Algorithm 3.3 affect its precision and recall. In Section 3.3, we discussed two possible criteria for deciding whether a trace should be considered hot. The "strict" criterion is that a trace's estimate $\hat{f}(t)$ should exceed the hotness threshold h. However, if this estimate is inaccurate and too small, the chain and all other hot chains that have it as prefix will be missed. Thus, in the algorithm we use a



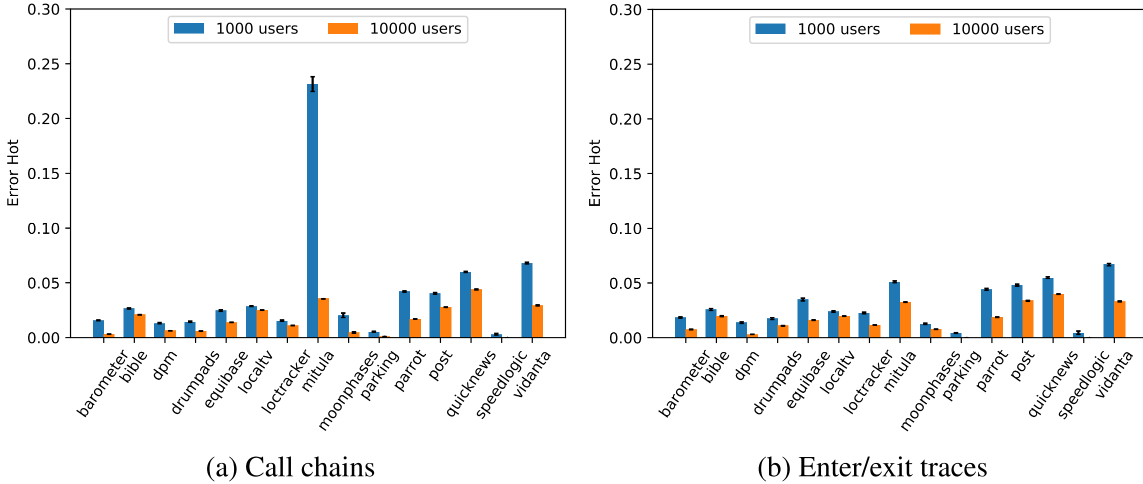
(a) Call chains

(b) Enter/exit traces

Figure 3.4: Recall and precision for hot traces: strict vs relaxed hotness criterion.

"relaxed" criterion which also considers traces t with estimates $h/2 \leq \hat{f}(t) < h$ such that t has at least one extended trace with an estimate that exceeds h. This relaxed criterion was employed when collecting the data in Figure 3.3.

To understand the effects of this choice, we also measured precision and recall using the strict criterion. Figure 3.4 shows a comparison between the two criteria for $n = 1000$; the other value for n leads to similar conclusions. As can be seen from these measurements, using the strict criterion results in lower recall. For example, for call chain analysis, three apps have recall less than 50%. Similarly, for enter/exit trace analysis there are six apps with recall below 50%. As expected, the strict criterion does improve precision, but this effect is not very pronounced. Depending on the intended uses of the analysis, the app developers may prefer higher recall or higher precision. Using these two criteria, or variations of them, allows this trade-off to be adjusted as desired.



(a) Call chains

(b) Enter/exit traces

Figure 3.5: Error of estimates for reported hot traces.

3.4.3 Accuracy of Estimates for Reported Hot Traces

For the set of traces reported by Algorithm 3.3 as likely-hot, we ask following question: *What is the accuracy of estimates for reported hot traces?* Figure 3.5 shows the error of estimates, using a metric similar to the one used in Figure 3.2: the sum of $|\hat{f}(t) - f(t)|$ for all reported hot traces t, normalized by the sum of $f(t)$ for those t. Based on these results, the answer to the question is that high accuracy is achieved for the frequency estimates of hot traces. Combined with the high recall demonstrated earlier, our conclusion is that hot traces and their frequencies can be successfully estimated via our differentially-private analysis.

It is instructive to compare Figure 3.5 with Figure 3.2. Overall, the estimate error for hot traces is smaller than the estimate error for all traces. For example, for $n = 10000$, the average error value in Figure 3.5a is 1.6%, compared to 7.4% in Figure 3.2a, and 1.7% vs 8.4% for Figure 3.5b vs Figure 3.2b. Theoretically, both count sketch and randomized response tend

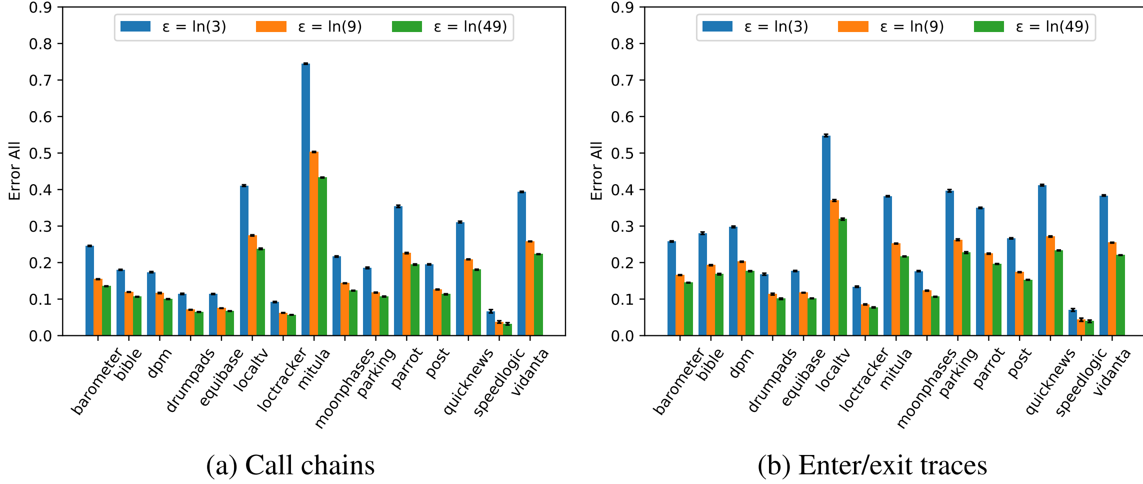
to favor higher-frequency items. The higher accuracy for hot traces demonstrates that this also holds in practice.

3.4.4 Privacy Loss Parameter

As discussed earlier, the privacy loss parameter ε can be used to tune accuracy/privacy trade-offs. We considered the following question: *To what degree does accuracy change with changes in this parameter?* In existing work, ε ranges from 0.01 to 10 [29]. Related work that employs randomized response has used, for example, $\ln(3)$, $\ln(9)$, and $\ln(49)$ [18, 60, 61]. We computed the error for all covered traces for these three values; the results for $\ln(9)$ were already presented in Figure 3.2 and are repeated here. Figure 3.6 shows these measurements for $n = 1000$; similar trends are seen for the other n value. Overall, with increasing ε, the expected accuracy gains are observed but seem to level off. For call chains and enter/exit traces, respectively, the average error across all apps decreases from 25.3% and 28.7% for the smallest value of ε to 16.6% and 19.0% for $\ln(9)$, and then further to 14.5% and 16.5% for the largest value of the parameter. Based on these results, we consider $\ln(9)$ to provide a reasonable trade-off and have used it to present the majority of data in our evaluation.

3.4.5 Summary of Results

Our experimental results can be summarized as follows. First, as illustrated in Figure 3.2, the frequency estimates have high accuracy, for practical values of ε. This results indicates that with good privacy and sufficient number of software users, one can obtain accurate frequency estimates for software traces. Second, based on the results in Figure 3.3, the set of hot traces can be determined with high precision and recall. The relaxed identification of hot traces is important for achieving this result (Figure 3.4). Third, the frequency estimates for



(a) Call chains

(b) Enter/exit traces

Figure 3.6: Error of estimates for all covered traces for three values of ε.

hot traces are accurate and better than those for the remaining covered traces (Figure 3.5). Finally, consider the accuracy/privacy trade-off spectrum: from smaller values of ε (i.e., stronger privacy) and lower accuracy, to larger values of ε and high accuracy. As indicated by Figure 3.6, after a certain point in this spectrum there do not seem to be significant additional improvements in accuracy.

3.5 Conclusions

We develop the design of a differentially-private trace coverage analysis, based on an incremental definition of the trace domain. We employ local count sketches, randomize them efficiently, and analyze them at the server side to obtain frequency estimates and to search for hot traces. The approach is illustrated with a call chain analysis and an enter/exit trace analysis. Our experimental studies present promising findings: with realistic numbers of software users, one can use these privacy-preserving techniques to obtain accurate frequency estimates for trace coverage and to effectively identify hot traces.

Chapter 4: Local Differential Privacy for Frequency Analysis of Software Traces

This chapter generalizes the problem studied in Chapter 3 to the analysis of frequencies of software traces. More specifically, given a trace $t \in \mathcal{T}$, the coverage analysis studied in Chapter 3 answers the following question: how many users' executions cover t? The answer to such a question is in $[0,n]$ where n is the number of users. However, a trace $t \in \mathcal{T}$ may occur more than once in a user's execution of the copy of the program. In this chapter, we answer the following question instead: given $t \in \mathcal{T}$, how many times does it occur among all the users? We define such a profiling problem as frequency analysis of software traces, as a generalization of the coverage analysis described earlier. For some developers, gaining the frequency knowledge about how users are using their program is more pertinent and beneficial than having the coverage information. As will be shown later in this chapter, applying differential privacy to this scenario brings new challenges.

4.1 Problem Statement

4.1.1 Frequency Analysis for Software Traces

Consider a program deployed across n users, denoted by $u_1, ..., u_n$. Each user has her own copy of the program. At run time, a user's execution of the program covers a set of traces (refer to Section 3.1 for the definition of traces), each one of which could be

covered multiple times. To capture this notion, we denote the traces covered by user u_i by a multiset T_i where the underlying set of unique traces is $support(T_i) \subseteq \mathcal{T}$. The multiplicity of $t \in \mathcal{T}$, i.e., the number of times t occurs in T_i, is denoted by $T_i(t)$. Alternatively, the local information for u_i can be thought of as a vector $\boldsymbol{T}_i \in \mathbb{N}^{|\mathcal{T}|}$, with each entry corresponding to one of the traces $t \in \mathcal{T}$. Both notations are used in the rest of this section for the convenience of clear explanation. Without loss of generality, we assume that $\sum_{t \in \mathcal{T}} \boldsymbol{T}_i(t) = k$, where k is a pre-defined parameter—that is, the L_1 norm $\|\boldsymbol{T}_i\|_1$ is equal to k. Further, we assume that this k applies to all users u_i. If a user records fewer than k observations, a unique "dummy" trace could be added to $\mathcal{T}$ to increase the counts to k.

We consider the following two trace frequency analysis problems. First, for each $t \in \mathcal{T}$, to estimate the frequency $f(t)$ of t over the population of users, while collecting the local data of each user with differential privacy. Here $f(t) = \sum_{i=1}^{n} T_i(t)$. Second, to identify a set of hot traces whose frequencies are relatively higher than those of the remaining traces. Formally, we want to identify a set of traces $\mathcal{H} = \{t \in \mathcal{T} : f(t) \geq \alpha kn\}$ where α is a cut-off threshold parameter.

4.1.2 The Differential Privacy Guarantee

For completeness of presentation, we repeat below the key aspects of the differential privacy guarantee. In a differential privacy algorithm, a randomizer is used to add statistical noise on the input data which is the real data from users, and the output which is the randomized data is shared with the analytics server (or any entity not trusted). Differantial privacy guarantees that given any output, one cannot distinguish the real input data a from any of its neighboring data b with high confidence. In a local model of DP, which is the one

used in our approach, a local randomizer is used at the user end to perturb the real data of the user, and only the output of the local randomizer is shared with the data server.

Formally, randomzier R achieves ε-differential privacy if for any pair of neighboring frequency vectors $\boldsymbol{f}, \boldsymbol{f'} \in \mathbb{N}^{|\mathcal{T}|}$ and for any possible output $\boldsymbol{z} \in \mathbb{R}^{|\mathcal{T}|}$ of R, $\frac{Pr[R(\boldsymbol{f})=\boldsymbol{z}]}{Pr[R(\boldsymbol{f'})=\boldsymbol{z}]} \leq e^{\varepsilon}$. This means that the probabilities of the the real data being $\boldsymbol{f}$ or $\boldsymbol{f'}$ are close to each other, so that $\boldsymbol{f}$ and $\boldsymbol{f'}$ are indistinguishable from each other in a statistical sense. Note that the ratio of the two probabilities is bounded by e^{ε} where e is Euler's number which is approximately equal to 2.71828 and ε is a parameter which is also know as the "privacy budget". The value of ε determines the strength of the protection. The smaller the privacy budget, the stronger the protection, but also the less accurate the estimates. Thus, achieving a balance between privacy and utility (i.e. accuracy of estimates) is the goal for all differential privacy algorithms.

4.1.2.1 Defining Neighbors

The definition of neighbors lies in the core of the above definition of ε-differential privacy, because the indistinguishability property is guaranteed only for neighboring input data vectors. We borrow a definition from prior work [62]. Consider any pair of input frequency vectors $\boldsymbol{f}$ and $\boldsymbol{f'}$. The L_1 distance between them is $\|\boldsymbol{f} - \boldsymbol{f'}\|_1 = \sum_{t \in \mathcal{T}} |\boldsymbol{f}(t) - \boldsymbol{f'}(t)|$. Recall that the sum of a frequency vector is equal to k. Therefore, this distance is always in $\{0, 2, 4, ..., 2k\}$. To normalize, we define the distance between two frequency vectors as

$$d(\boldsymbol{f}, \boldsymbol{f'}) = \frac{1}{2}\|\boldsymbol{f} - \boldsymbol{f'}\|_1 = \frac{1}{2}\sum_{t \in \mathcal{T}} |\boldsymbol{f}(t) - \boldsymbol{f'}(t)|$$

The neighbors of a frequency vector $\boldsymbol{f}$ is then defined as

$$Neighbors(\boldsymbol{f}) = \{\boldsymbol{f'} \mid d(\boldsymbol{f}, \boldsymbol{f'}) \leq \tau\}$$

The threshold τ defines how far neighbors can be from each other. In next section, we will talk about how the value of τ is selected in order to achieve the goals of indistinguishability for different scenarios.

4.2 Proposed Approach for Frequency Analysis

Our approach is twofold. As introduced earlier in this section, the analysis of trace frequencies faces the same exponentially large domain problem as in the trace coverage analysis. Thus, count sketch is still used to solve this problem. In addition, the randomization is done by using the Laplace mechanism, instead of the randomized-response-based one as used in Chapter 3. This is because the randomized-response approach could only achieve weak differential privacy guarantee. To see this, consider the following hypothetical adaptation of the approach from Chapter 3: for each trace t covered by user u_i, add it to the local sketch by repeating the process as described in Algorithm 3.1 for $\boldsymbol{f_i}(t)$ times, where $\boldsymbol{f_i}(t)$ is the frequency of t in $\boldsymbol{f_i}$. In this approach, only vectors differing by 2, i.e. their L_1 distance is 2, are guaranteed to be indistinguishable.

4.2.1 Randomized Count Sketch with Laplace Noise

Recall that a local frequency vector $\boldsymbol{f}_i$ for user u_i is in $\mathbb{N}^{|\mathcal{T}|}$ and its size is the size of $\mathcal{T}$ which is the domain of all possible traces up to the pre-defined depth limit l. This means that $\boldsymbol{f}_i$ contains not only frequencies of the traces covered by user u_i, but also zero frequencies for the traces that are not covered. This is needed because in an LDP solution, randomization is defined over a pre-defined data domain, and this domain cannot depend on the specifics of the local data of any particular user u_i (i.e., it cannot depend on the set of covered traces by the user). This is a fundamental requirement for achieving the DP probabilistic guarantees. However, a critical challenge for applying LDP directly on $\boldsymbol{f}_i$ is that the domain $\mathcal{T}$ is

exponentially large. In the simplest possible definition, $\mathcal{T} = \mathcal{E} \cup \mathcal{E}^1 \cup ... \cup \mathcal{E}^l$ (Recall that in Section 3.1, we define $\mathcal{E}$ as the set of possible run-time events. And here $\mathcal{E}^r$ denotes the Cartesian product of r copies of $\mathcal{E}$.) For any non-trivial program and l, the size of this over-approximation of $\mathcal{T}$ could easily reach billions of elements. Even if static analysis is used to reduce domain size (e.g., via context-sensitive call graph analysis), ultimately the exponential growth cannot be avoided.

This large size leads to two problems: (1) the cost of randomization is very high, since each $t \in \mathcal{T}$ requires the processing of an independent random value; (2) the size of the reported randomized data is $|\mathcal{T}|$, which is not practical in any realistic settings. Note that without privacy, the only data that is reported is for t with non-zero frequency measurements. However, after randomization, in order to achieve differential privacy, the randomized data for all $t \in \mathcal{T}$ must be reported.

A solution to this challenge is to first map the data vector $\boldsymbol{f} \in \mathbb{N}^{|\mathcal{T}|}$ into a "compressed" data vector over a significantly smaller domain $\mathcal{T}_c$. This resulting compressed vector $\boldsymbol{fc} \in \mathbb{N}^{|\mathcal{T}_c|}$ is then randomized in a manner that ensures the differential privacy property with respect to the original data vector $\boldsymbol{f}$. Note that the compression provides an additional layer of privacy protection, since many real vectors $\boldsymbol{f}$ could be mapped to the same compressed vector $\boldsymbol{fc}$.

This can be achieved in Count Sketch [11]. To map the large domain to the compressed smaller domain, a pair of hash functions $h : \mathcal{T} \to \mathcal{T}_c$ and $g : \mathcal{T} \to \{+1, -1\}$ is used. For any $t_c \in \mathcal{T}_c$, entry $\boldsymbol{fc}(t_c)$ in the compressed vector $\boldsymbol{fc}$ is the sum of contributions from all $t \in \mathcal{T}$ for which $h(t) = t_c$; that is, all t that are hashed to t_c using the hash function h. Each of those contributions is $g(t) \times \boldsymbol{f}(t)$. Here the value $\boldsymbol{f}(t)$ for element t from the original vector

$\boldsymbol{f}$ is accumulated into $\boldsymbol{fc}(t_c)$ with either a positive sign or a negative sign, depending on the value of $g(t)$ (which is either $+1$ or -1).

We propose to apply Laplace noise to this compressed representation, as follows. Suppose we would like to achieve indistinguishability between a given vector $\boldsymbol{f}$ and all its neighbors $\boldsymbol{f}'$ (i.e., all vectors at distance $\leq \tau$, as defined at the end of the previous section). For each element of $\boldsymbol{fc}$, we draw a random value from the Laplace distribution $Lap(\frac{2\tau}{\varepsilon})$ and add that value to this element. The resulting vector is a randomized version of the compressed vector.

A key question is whether this approach achieves probabilistic indistinguishability between the given vector $\boldsymbol{f}$ and each of its neighbors. Consider any $\boldsymbol{f}, \boldsymbol{f}' \in \mathbb{N}^{|\mathcal{T}|}$ and their corresponding $\boldsymbol{fc}, \boldsymbol{fc}' \in \mathbb{N}^{|\mathcal{T}_c|}$. It can be shown that

$$\sum_{t \in \mathcal{T}} |f(t) - f'(t)| \geq \sum_{t_c \in \mathcal{T}_c} |\boldsymbol{fc}(t_c) - \boldsymbol{fc}'(t_c)|$$

This property can be derived from the standard inequality $\sum_i |a_i| \geq |\sum_i a_i|$. As a corollary, the compressed vectors $\boldsymbol{fc}, \boldsymbol{fc}'$ for two neighbors $\boldsymbol{f}, \boldsymbol{f}'$ are also neighbors. Thus, adding Laplacian noise of magnitude $\frac{2\tau}{\varepsilon}$ to the compressed vector achieves indistinguishability for the original vector.

To increase accuracy, count-sketch uses multiple pairs of hash functions $\langle h_1, g_1 \rangle, \ldots, \langle h_s, g_s \rangle$. These pairs are used to compute s compressed vectors $\boldsymbol{fc}_1, \ldots, \boldsymbol{fc}_s$. For any $t \in \mathcal{T}$, an estimate of the frequency of t can be obtained by taking the median of the s values $g_i(t) \times \boldsymbol{fc}_i(h_i(t))$. Without loss of generality, $\mathcal{T}_c$ can be considered to be the set $\{1, \ldots, m\}$. Thus, the entire count-sketch data structure is a matrix with s rows and m columns, where each row i is the vector $\boldsymbol{fc}_i$ containing m elements.

The overall process is detailed in Algorithm 4.1. Recall that the local data of user u_i is actually a multiset of traces covered by the user at least once, which is denoted as T_i.

The underlying set is denoted as $support(T_i)$ and the frequency of $t \in support(T_i)$ is $T_i(t)$. First, the count sketch S_i is initialized with all zeros. Then, each trace t is added to S_i by updating each row j using the corresponding pair of hash functions i.e. h_j and g_j. Lastly, each element of S_i is added with a random number independently drawn from the Laplace distribution.

Algorithm 4.1: Local Laplace Randomizer

```
Input  : T_i: multiset of traces covered by users u_i
Output : S_i: randomized local sketch
1 S_i ← {0}^{s×m}
2 foreach t in support(T_i) do
3     for j ← 1 to s do
4         S_i[j, h_j(t)] ← S_i[j, h_j(t)] + g_j(t) × T_i(t)
5 for j ← 1 to s do
6     for k ← 1 to m do
7         S_i[j, k] ← S_i[j, k] + Lap(2τ/ε)
```

4.2.2 Data Collection

Figure 4.1 illustrates the overall data collection scheme. The first part of the collection determines suitable values for several configuration parameters; this process will be described shortly. The resulting configuration is instantiated for the n users of the software.

Each user u_i participating in the data collection computes their own local traces T_i , using standard profiling techniques. This information is mapped to a local sketch matrix as described above. Each matrix element is then randomized by adding Laplacian noise (an independent random value is drawn for each matrix element). The resulting randomized sketch is then communicated with an analytics server.

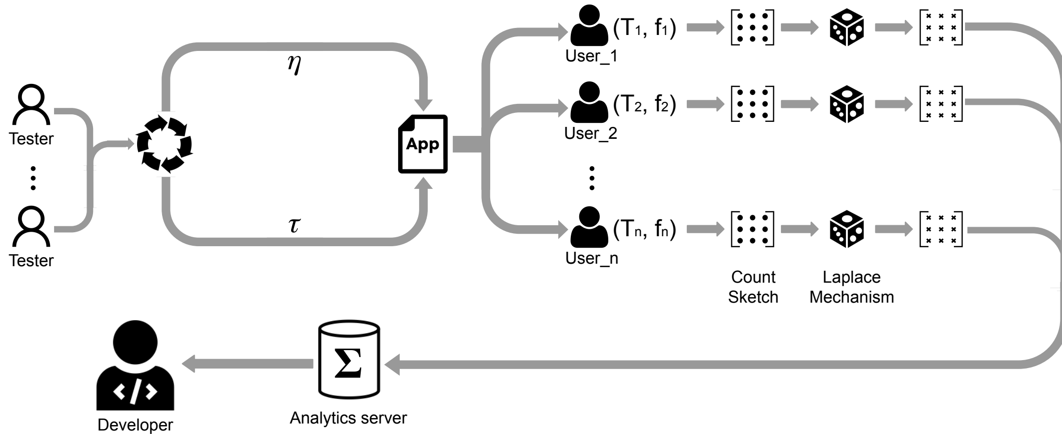


Figure 4.1: Data collection scheme for frequency analysis.

The processing of collected user data is outlined in Algorithm 4.2. The server collects the randomized sketches RS_i of all users u_i and constructs a global randomized sketch G, which is the element-wise sum of the collected local sketches (function `build_sketch`). The resulting global sketch can be used to answer queries of the form: "What is the (estimated) global frequency of a given trace $t \in \mathcal{T}$?" (function `estimate_freq`). To answer the query, for each row i of the global sketch the server computes a value which is the product of $g_i(t)$ and the value at column $h_i(t)$ at that row. The query answer is the median value among the values computed for all sketch rows.

This data collection scheme ensures that (1) the local information of any user u_i is shared with "the outside world" with differential privacy guarantees, and (2) the global frequency of any trace can be estimated by the analytics server based on the collected locally-randomized data.

Algorithm 4.2: Processing of collected user data

```
1 Function build_sketch(RS_1, ..., RS_n):
2     G ← {0}^{s×m}
3     for i ← 1 to n do
4         G ← G + RS_i
5 Function estimate_freq(t):
6     Est ← ∅
7     for i ← 1 to s do
8         Est ← Est ∪ { g_i(t) × G[i, h_i(t)] }
9     return median(Est)
```

4.2.3 Hiding Trace Information

For a pair of frequency vectors $\boldsymbol{f}$ and $\boldsymbol{f}'$ to be neighbors, their distance $d(\boldsymbol{f}, \boldsymbol{f}')$, i.e. $\frac{1}{2}\|\boldsymbol{f} - \boldsymbol{f}'\|_1$, needs to be within τ. Choosing the value for τ determines what traces are guaranteed with the differential privacy indistinguishability, i.e. being "hidden". We propose two concepts for a traces being hidden: its presence and its hotness. A trace t is present in $\boldsymbol{f}$ iff. $\boldsymbol{f}(t) > 0$. Similarly, a trace is "hot" in $\boldsymbol{f}$ iff. $\boldsymbol{f}(t) > \eta$ where η is some pre-defined threshold.

4.2.3.1 Hiding The Presence of A Trace

Given a frequency vector $\boldsymbol{f}$ and some trace t which is present in $\boldsymbol{f}$, i.e. $\boldsymbol{f}(t) > 0$, to hide its presence in differential privacy, there must be some neighboring vector $\boldsymbol{f}'$ where t is absent, i.e. $\boldsymbol{f}'(t) = 0$ and $d(\boldsymbol{f}, \boldsymbol{f}') \leq \tau$. However, there is another constraint that must be considered. Recall that in Section 3.1, the traces are defined inductively by the family of extension functions ext. A trace t is a prefix of t' if $t' \in \mathsf{ext}(t) \cup \mathsf{ext}^2(t) \cup \ldots$. Note that here we use ext^k to denote the composition of the extension function k times. It's obvious that if t is absent, t' should be absent too. Thus, in the neighboring vector $\boldsymbol{f}'$, $\boldsymbol{f}'(t') = 0$ for all t' of

which t is a prefix. It's easy to see that the difficulty of hiding a trace t in a frequency vector $\boldsymbol{f}$ is the sum of the frequencies of t and of all t' for which t is a prefix.

4.2.3.2 Hiding The Hotness of A Trace

The second scenario, hiding the hotness of traces, is similar to hiding the presence. The difference is that a hotness threshold η is introduced. For a hot trace t in vector $\boldsymbol{f}$, i.e. $\boldsymbol{f}(t) > \eta$, to hide its hotness, τ should be at least the minimum distance between $\boldsymbol{f}$ and $\boldsymbol{f}'$ where t is cold, i.e. $\boldsymbol{f}'(t) \leq \eta$. Note that there's not need to consider whether the traces for which t is a prefix are hot or not, because it is possible that a trace is hot while its prefix is cold. Take call chains (one of the exemplars of software traces defined in Section 3.1) as an example. Suppose there are two call chains $c_1 =< m_1, m_2 >$ and $c_2 =< m_1, m_2, m_3 >$. c_1 is a prefix of c_2 because method m_2 calls to method m_3. But suppose that the call statement to m_3 is in a CFG-loop, e.g., a for-loop. Therefore, at run time, it's possible that the frequency of c_2 is larger than that of c_1.

The difficulty of hiding the hotness of t in $\boldsymbol{f}$ is equal to $\boldsymbol{f}(t) - \eta$. If a trace is not hot in a frequency vector, the difficulty of hiding its hotness in the vector is zero.

4.2.4 Selecting Sketch Size

The size of count sketch is an important parameter to tune for achieving balance between accuracy and efficiency, because it is a hash functions based data structure. The output size of hash functions should be chosen carefully such that hash collisions does not become a dominant factor to the analysis error. We set number of rows, i.e. number of pairs of hash functions, as 256. For number of columns, i.e. the output size of hash functions, instead of fixing it as an arbitrarily large number, we configure it to be approximately the same as the number of unique traces covered by users when they run the software. More specifically, a

small set of opt-in users (in our experimental evaluation, we use 10% of all simulated users) are selected to report the traces they have covered, and the number of columns of the count sketch will be configured as the smallest power of 2 greater than or equal to the number of unique traces covered by opt-in users. Our work in Chapter 3 uses the same approach. In reality, this set of opt-in data can come from in-house testing.

4.3 Evaluation

To evaluate the proposed approach, we use the same 15 Android apps and profiling approach used in the coverage analysis. The official Android application testing tool Monkey is used to emulate users. More specifically, we first instrument the apps to record when an app method is entered or exited. The instrumentation is introduced with the Soot rewriting tool [49]. We then run Monkey 1000 times (simulating 1000 users) on each app. During each run, a trace of enter/exit events is collected. The collection is terminated when the total number of enter events in a trace reaches 10$\times$ the number of application methods. Then from these enter/exit events, call chains and enter/exit traces are determined.

In the evaluation of coverage analysis (Section 3.4), we limit the length of call chains to be within 10 and enter/exit traces to be within 20. This limit ensures that the exploration of hot traces (Section 3.3) can be bounded. Such exploration is not possible for frequency information, since a hot trace may have a cold prefix (as opposed to coverage information, where a covered trace can only have covered prefixes). Thus, there is no limit for call chain length for the experiments conducted in this section. However, we do keep the length limit for enter/exit traces, because the number of them gets extremely large when unlimited.

In the analyzed data, the sum of frequencies of all call chains for each user, i.e. k, is equal to 10$\times$ the number of application methods. For enter/exit traces, we aim to have

k equal to 20× the number of application methods. However, here it is possible that k is less than that number, because when the collection of traces is terminated, there can be enter events whose matching exit events are not yet observed. To fix this issue, we append matching exit events to the traces so that the sum of frequencies of all enter/exit traces for each user is always 20× the number of application methods.

Since we limit the length of enter/exit traces to be at most 20, ignoring the traces of length more than 20 causes k not equal to 20x the number of application methods. Instead of discarding their frequencies, we accumulate them into the enter/exit traces of length 20 which are also their prefixes. This essentially makes the frequency of an enter/exit trace of length 20 to not represent the frequency of itself but the sum of frequencies of itself and of all the traces for which it is a prefix. The source code that implements the randomization, the simulated traces, and instructions to reproduce the experimental results presented in this section are publicly available in `https://presto-osu.github.io/dp-trace-freq`.

4.3.1 Hiding The Presence of Traces

This section and the following section evaluate the frequency oracle proposed in Section 4.2, using two different strategies for deciding the parameter τ: hiding the presence and hiding the hotness of traces. This section uses the first problem: hiding the presence of certain traces.

Let $D(\boldsymbol{f},t)$ be the difficulty of hiding the presence of t in $\boldsymbol{f}$, as defined in Section 4.2.3.1. To hide the presence of any trace t that appears in at least one of the n users who are participating in the software trace data collection process, τ should be set as $\max_{1\leq i\leq n} D(\boldsymbol{f_i},t)$.

However, τ must be determined in advance and hard-coded into the software before that software is deployed to the users. The developers cannot predict what the real frequency

vectors $\boldsymbol{f_i}$ will look like. To solve this problem, we leverage an approach similar to the one from the earlier chapter: we assume access to the true local information for some set of opt-in users. In our experiments, we choose randomly 10% of the 1000 simulated users and use their real frequency vectors to approximate the appropriate values used for τ in the experiments, as described below. Only the remaining 90% users are assumed to be participating the differential privacy data collection process. In reality, the developers may collect such real frequency vectors from in-house testing, or from real users who agree to share their data with the developers. Recall that in Section 4.2.4, to decide the appropriate size for the sketch, 10% users are randomly selected as opt-in users. Here we use the same set of users, denoted by "tester" in Figure 4.1 and referred to as "test users" in the rest of this section. We will refer the users participating in the differentially-private data collection as "real users".

Let U' be the set of test users and U be the set of users that participate in the differentially-private data collection (i.e., the 90% of users, as described above). Following an approach similar to prior work [62], let $\tau(t)$ be the value of τ for hiding the presence of t in the frequency vectors of all the users in U, i.e. $\tau(t) = \max_{u_i \in U} D(\boldsymbol{f_i}, t)$. For all traces $\mathcal{T}$ that appear in the frequency vectors for users from U, their difficulties, i.e. $\tau(t)$ for $t \in \mathcal{T}$, are sorted in ascending order. The x percentile of this list (depending on how many traces we intend to cover for DP protection) should be taken as the value for τ. In the proposed approach, $\tau(t)$ and the x percentile is approximated from the set U' of test users. That is, the x percentile is taken from the sorted list of $\tau'(t)$ for all t in $\mathcal{T}'$, where $\tau'(t) = \max_{u_i \in U'} D(\boldsymbol{f_i}, t)$ and $\mathcal{T}'$ is the set of traces covered in the data of users $u_i \in U'$.

Table 4.1 shows the ratio of the value of τ approximated by the test users to the ground truth value, with varying percentiles for each app. Ideally, a ratio close to 1 is the best. A

app	Ratio of the approx. τ to the ground truth					
	Call Chains			Enter/Exit Traces		
	25%	50%	75%	25%	50%	75%
barometer	0.8	0.8	0.8	1.0	0.8	0.8
bible	0.6	0.8	0.8	0.7	0.7	0.9
dpm	0.9	0.9	1.0	1.0	0.9	0.9
drumpads	1.0	0.9	0.9	0.9	1.3	1.3
equibase	0.8	0.8	0.9	1.2	0.9	0.9
localtv	0.8	1.0	1.0	1.1	0.8	1.0
loctracker	0.8	1.0	1.0	1.3	1.1	1.1
mitula	1.0	1.0	1.0	1.0	0.8	0.8
moonphases	0.8	0.6	0.7	3.4	2.4	0.9
parking	0.7	0.7	0.8	0.8	0.8	0.7
parrot	1.0	1.0	1.0	0.9	0.9	0.9
post	1.0	0.9	1.0	1.2	1.1	1.0
quicknews	1.0	0.9	0.9	0.9	1.1	1.0
speedlogic	1.1	1.0	0.9	2.7	1.6	0.8
vidanta	1.1	1.0	0.9	0.9	0.8	0.7

Table 4.1: Ratio of the approximate value of τ from test users to the ground truth with varying percentiles for hiding trace presence. The ratios are average values of 30 runs.

ratio greater than one means more noise is added than necessary, while a ratio less than 1 means insufficient noise is added. As can be seen, most ratios are between 0.8 and 1.2. As mentioned earlier, a similar approach has been used in prior work [62].

Error of frequency estimates. To evaluate the accuracy of the frequency oracle, we measure the error of the frequency estimates. This error is the distance between the estimates and the ground truth, over all traces covered by at least one real user. The error is normalized by nk, which is the sum of all elements in the ground truth. Here n is the number of "real users", i.e. 900. Thus, the reported error metric is $\frac{1}{2nk}\sum_t |\boldsymbol{F}(t) - \hat{\boldsymbol{F}}(t)|$ where $\boldsymbol{F}$ and $\hat{\boldsymbol{F}}$ are the vectors of ground truth and estimates, respectively.

Following prior work [62], the experiments explore values of 0.5, 1.0, and 2.0 for the privacy budget ε, and 25%, 50%, and 75% for the percentage x of traces whose presence is

hidden with deferential privacy. Figure 4.2 and Figure 4.3 show the error for each of the combinations of ε and x for each app. The error is taken as the average of 30 independent runs. Note that we also collected data for protection at $x = 100\%$ (i.e., τ is large enough to hide the presence of every covered chain); as expected, the magnitude of noise is so large that the resulting estimates are essentially meaningless. We observed similar trends in the experiments for both call chains (Figure 4.2) and enter/exit traces (Figure 4.3).

Recall that the random noise being added to sketch elements is drawn from Laplace distribution $Lap(\frac{2\tau}{\varepsilon})$. With increasing ε and decreasing x (which leads to decreasing τ), the error decreases. Values of ε used by related prior work have been typically in the range of 1 to about 4 [18, 55]. As can be seen in Figure 4.2 and Figure 4.3, for $\varepsilon = 2$ our experiments indicate that typically 50% of observed traces can be hidden (i.e., there is "plausible deniability" that they never were executed) while achieving overall normalized error of around 1%. Increasing the protection to 75% of the chains typically increases the error by a few percentage points. Overall, these measurements indicate that such differential privacy protection can be ensured for many call chains and enter/exit traces of individual software users, while still achieving accurate population-wide frequency estimates.

4.3.2 Hiding The Hotness of Traces

This section evaluates the approach for hiding the hotness of traces, using a similar method as the one described in the previous section. The set U' of test users is randomly selected (as 10% of all users) and their real frequency data is used to approximate the value of τ. Recall that a trace is considered hot at user end if its frequency is above threshold η. We set η as $\frac{k}{|T'|}$ where T' is the set of traces that appear in U'. This is approximately the average frequency of a trace. We also observed similar result for other threshold values.

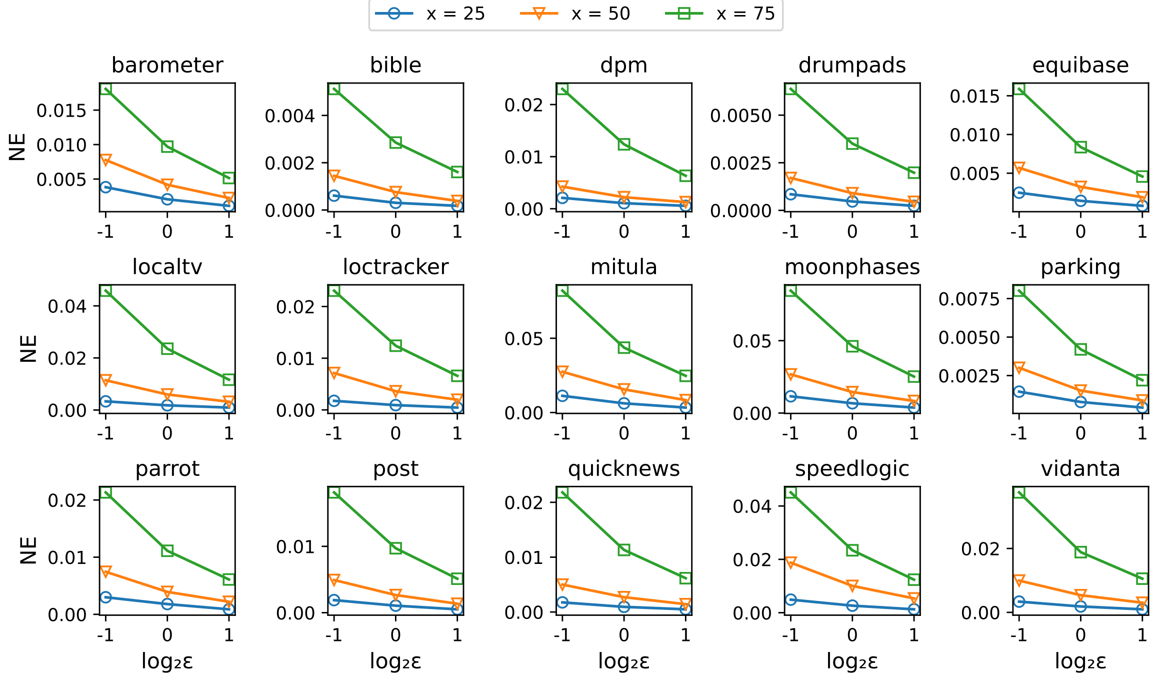


Figure 4.2: Normalized error (NE) for frequency estimates for call chains with varying privacy budget ε and percentage of hidden (presence) traces x.

The value of τ for hiding the hotness of a trace t in all the users in U is approximated as $\tau(t) = \max_{u_i \in U'} D(\boldsymbol{f_i}, t)$, where $D(\boldsymbol{f_i}, t)$ is the difficulty of hiding the hotness of t in $\boldsymbol{f_i}$ as defined in Section 4.2.3.2. Note that if a trace is not hot in any user in U', $\tau(t) = 0$. Then $\tau(t)$ for all t that are hot in at least on user in U' are sorted in ascending order, and the x percentile is taken to be the value of τ. The ratios of the approximated τ to the ground truth are listed in Table 4.2. Most of them are close to 1, while there are some outliers such as 4.8 and 0.3.

Error of frequency estimates. Figure 4.4 and Figure 4.5 show the normalized error of frequency estimates of all the traces covered by at least one real user for hiding the hotness of traces with ε varying in $\{0.5, 1.0, 2.0\}$ and x varying in $\{25, 50, 75\}$. As expected, the

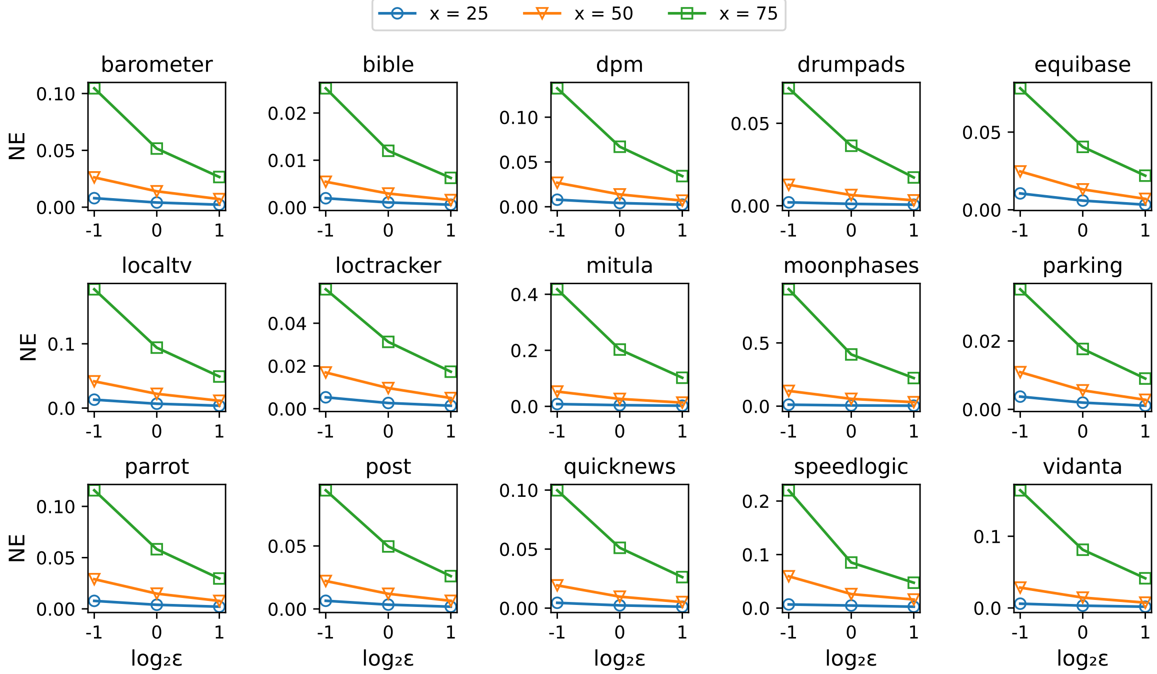


Figure 4.3: Normalized error (NE) for frequency estimates for enter/exit traces with varying privacy budget ε and percentage of hidden (presence) traces x.

trend is similar to that of hiding the presence. The error increases as ε decreases (less privacy budget) or x increases (more traces to hide).

The only exception is `speedlogic`. In Figure 4.4, for $\varepsilon = 0.5$, the error for $x = 75$ is lower than that for $x = 50$. This is because of the under-approximation of τ (the ratio in Table 4.2 for `speedlogic` call chains is 0.5 and 0.3 for $x = 50$ and $x = 75$ respectively). The ground truth τ for $x = 50$ and $x = 75$ is 23 and 43 respectively, while the approximation is only 12 and 11. In Figure 4.5, the error of $x = 25$ is higher than that of $x = 50$ for $\varepsilon = 0.5$. This is caused by the over-approximation of τ: 47 versus 9 (averaged over 30 runs). Also, there is an anomaly for $x = 25$ alone. Instead of going down as ε increases, the error is abnormally lower for $\varepsilon = 1$ than for $\varepsilon = 2$. We investigated and found that the approximation

app	Ratio of the approx. τ to the ground truth					
	Call Chains			Enter/Exit Traces		
	25%	50%	75%	25%	50%	75%
barometer	1.1	0.9	1.2	1.5	1.7	0.8
bible	1.8	0.9	1.5	1.1	1.6	2.1
dpm	0.9	1.0	1.0	1.1	1.0	0.8
drumpads	0.9	1.1	0.8	2.0	2.2	1.5
equibase	1.0	1.0	1.2	0.5	0.8	0.9
localtv	1.2	1.0	0.7	1.3	0.9	0.7
loctracker	0.7	0.9	0.9	0.9	0.9	0.9
mitula	0.9	0.8	0.9	1.0	0.9	1.8
moonphases	0.7	0.9	0.8	1.2	0.7	1.6
parking	1.7	4.8	4.8	1.8	2.0	1.6
parrot	0.8	0.8	0.8	0.8	1.0	1.0
post	1.1	1.0	1.0	0.8	0.8	1.0
quicknews	1.3	1.1	1.0	1.0	1.0	0.8
speedlogic	1.0	0.5	0.3	8.0	1.2	1.0
vidanta	0.8	1.0	1.0	1.1	0.7	0.8

Table 4.2: Ratio of the approximate value of τ from test users to the ground truth with varying percentiles for hiding the hotness. The ratios are average values of 30 runs.

of τ was not stable: over-approximated for $\varepsilon = 0.5$ and 1.0, but under-approximated for $\varepsilon = 2.0$.

Overall, high accuracy is achieved for all values of ε when at most half of hot traces are protected: for both call chains and enter/exit traces, error is less than 3% for most apps and less than 6% for the other three apps. Even for 75% protection rate, for most apps we observe less than 6% error with higher privacy budgets, and less than 10% with the lowest privacy budget.

One open question is how to improve the results obtained by test users. As discussed early, sometimes τ values obtained from this set of users are significantly different from the τ values for the real users. As discussed for app `speedlogic`, this could lead to unreliable frequency estimates. The key issue is that the true frequencies over the test users may

differ significantly from the true frequencies over the users that participate in the actual data collection. Future work should consider techniques for (differentially-private) detection of such divergence, and explaining to software developers the potential impact of the divergence.

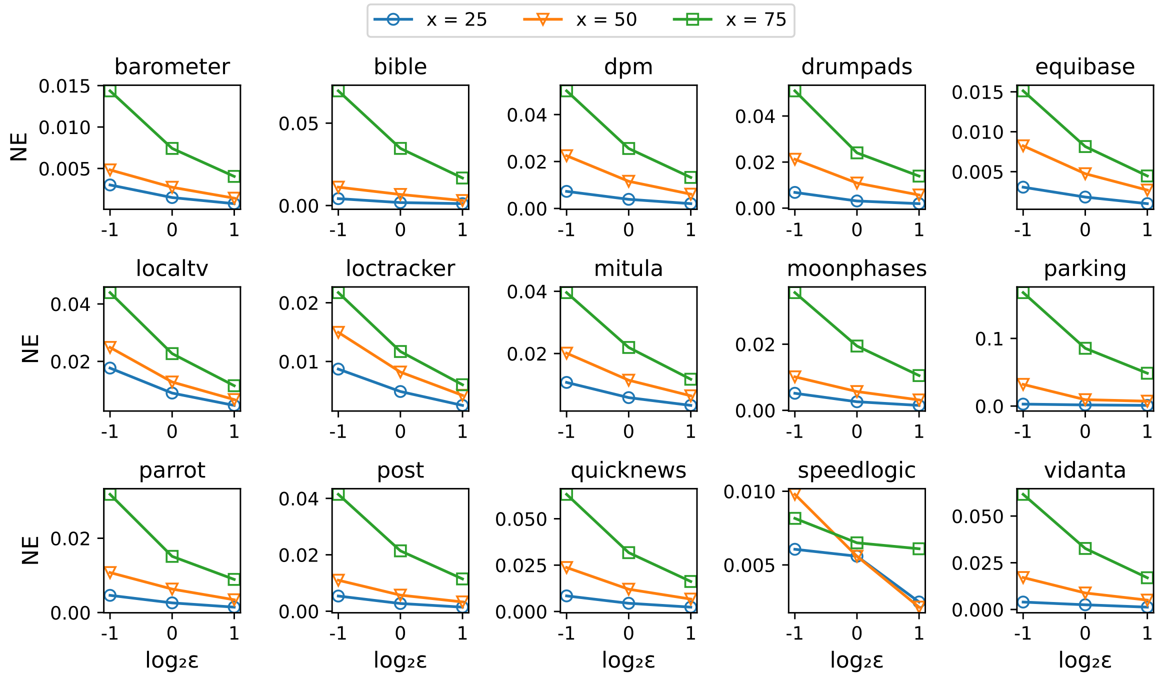


Figure 4.4: Normalized error (NE) for frequency estimates for call chains with varying privacy budget ε and percentage of hidden (hotness) traces x.

4.3.3 Identifying Hot Traces

One of the examples of how the global sketch can be used is to identify the set of "hot" traces whose global frequencies are greater than a pre-defined hotness threshold. This information can be used, for example, for subsequent performance optimizations or software functionality refinements. To identify the set of (esimtated) hot traces, we consider the traces

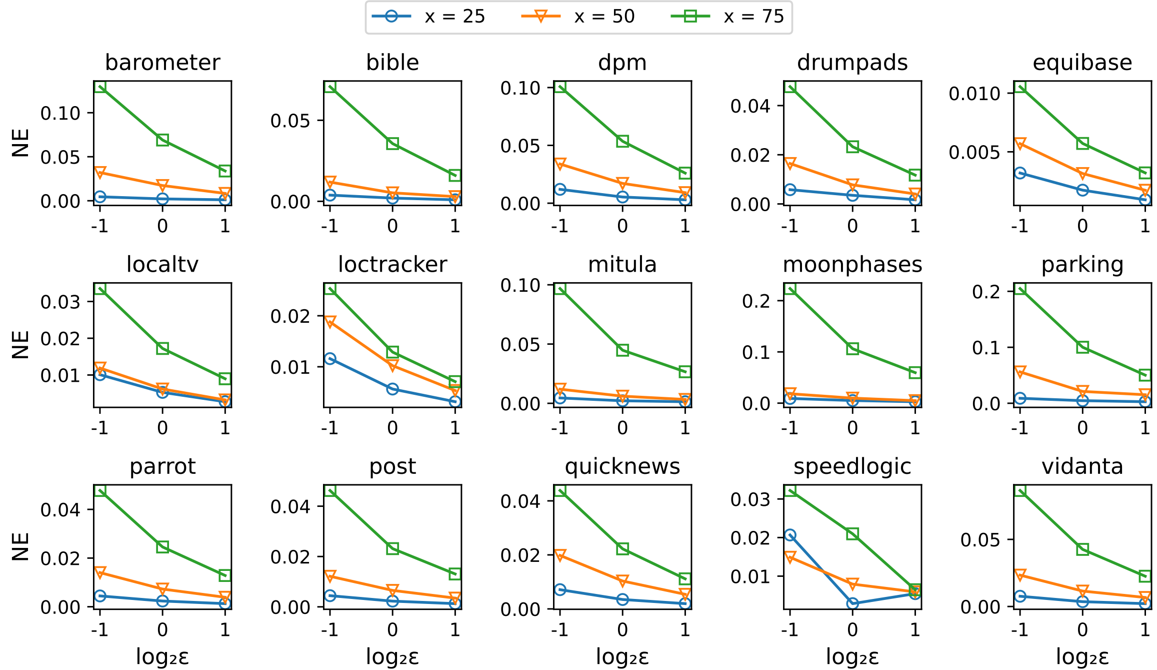


Figure 4.5: Normalized error (NE) for frequency estimates for enter/exit traces with varying privacy budget ε and percentage of hidden (hotness) traces x.

covered by the pre-deployment test data. If the frequency estimate of a trace in the global sketch is above the threshold, the chain is reported as being hot. In our experiments, we choose the hotness threshold to be kn (where n is the number of users supplying randomized sketches, and k is the sum of frequencies per user) divided by the number of unique traces in the test data. This threshold approximates the average frequency of a trace. Note that the threshold here is not the same thing as the local hotness threshold η used for hiding the hotness of traces discussed in Section 4.2.3.2.

Figure 4.6 shows the recall and precision of the identified hot traces. The results are average values of the 9 different combinations of three ε values: $0.5, 1.0, 2.0$; and three x values: $25, 50, 75$. Each of the combination is run 30 times. Figure 4.6a and Figure 4.6b use

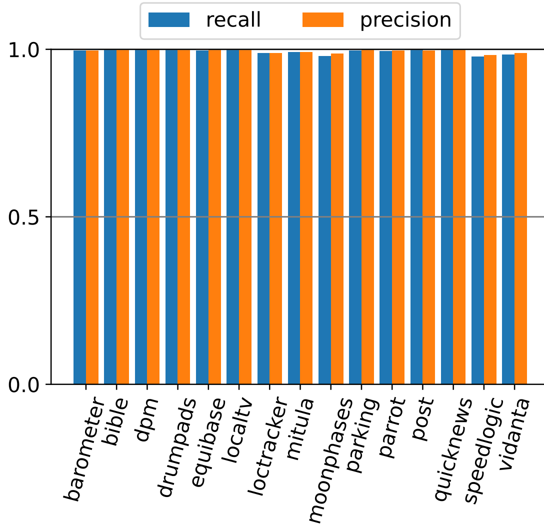
the protection strategy of hiding the presence, while Figure 4.6c and Figure 4.6d use hiding the hotness. As can be seen, the recall in all the four figures are very high (above 0.96). The precision is all above 0.96 except for three apps in Figure 4.6b. This is because their low precision for $x = 75$. For example, the precision of `moonphases` for $\varepsilon = 0.5$ and $x = 75$ is below 0.8. The reason for the low precision is the high error of the frequency estimates, which is shown in Figure 4.3.

4.3.4 Local Cost

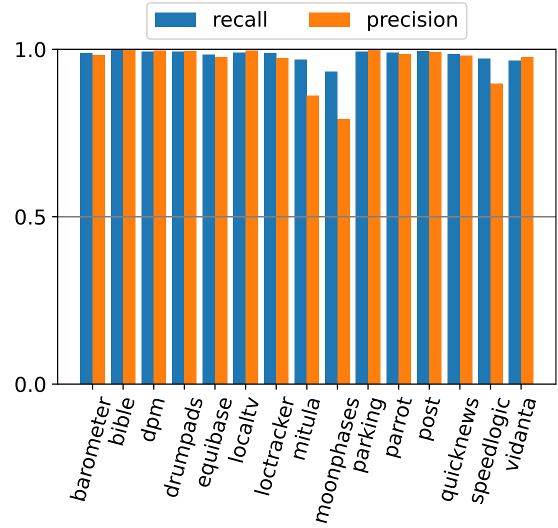
The running time to compute local randomized sketches is part of the expense that a software user incurs when participating in this data collection. The running time for local sketch construction is shown in Table 4.3. This cost includes the hashing of traces to sketch elements, the updates of those elements based on frequency information, and the subsequent addition of randomized noise to each element. As can be seen from these measurements, the local cost is rather low and therefore suitable for practical use.

app	Running time in seconds	
	Call Chains	Enter/Exit Traces
barometer	0.13	0.12
bible	0.07	0.07
dpm	0.06	0.07
drumpads	0.04	0.06
equibase	0.05	0.12
localtv	0.10	0.11
loctracker	0.03	0.07
mitula	1.47	0.28
moonphases	0.09	0.04
parking	0.05	0.07
parrot	0.32	0.32
post	0.14	0.18
quicknews	0.13	0.24
speedlogic	0.01	0.01
vidanta	0.34	0.34

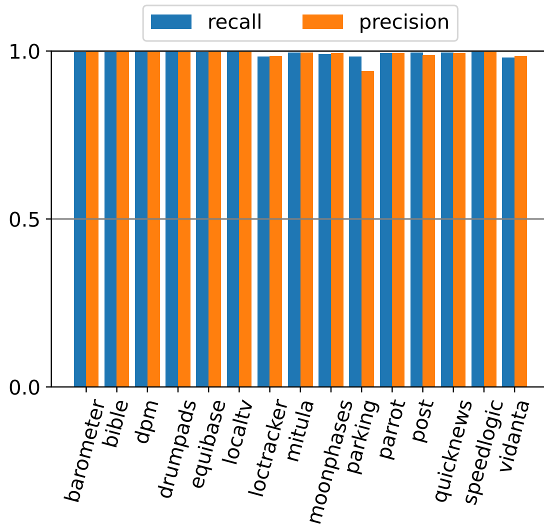
Table 4.3: Cost of building local randomized sketches, averaged over 900 users.



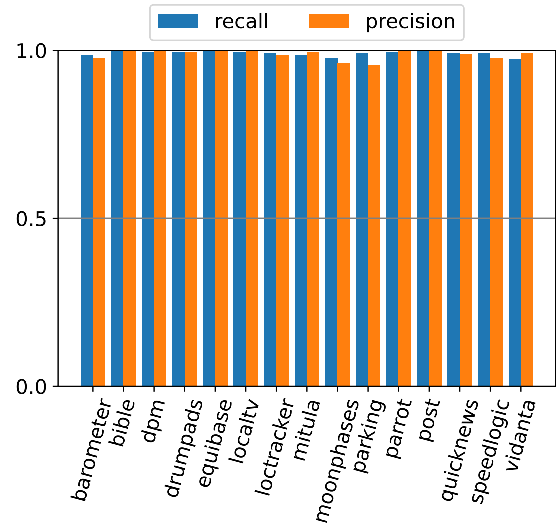
(a) Call Chains with Hidden Presence.



(b) Enter/Exit Traces with Hidden Presence.



(c) Call Chains with Hidden Hotness.



(d) Enter/Exit Traces with Hidden Hotness.

Figure 4.6: Recall and precision for identifying hot traces averaged over 9 combinations of varying privacy budget ε $(0.5, 1.0, 2.0)$ and percentage of hidden traces x $(25, 50, 75)$.

4.4 Summary

This chapter tackles the problem of frequency analysis of software traces, which is a generalization of the coverage analysis studied in Chapter 3. The difference between these two problems is that in frequency analysis, each user's local data is a vector of natural numbers representing the number of times each traces is covered by the user, while in coverage analysis, the local data is a vector of zeros or ones indicating weather a trace is covered.

Same as in the approach for coverage analysis, count sketch is used to handle the issue caused by unbounded domain size of traces. To achieve strong privacy protection, we employ the Laplace Mechanism parameterized by τ which specifies how distant the neighboring inputs are. Only neighboring inputs are guaranteed with differential privacy.

Using the same emulated traces collected in the evaluation of coverage analysis in Section 3.4, we conduct extensive experiments that demonstrate the effectiveness of this approach.

Chapter 5: Deploying LDP Frequency Analysis of Software Traces

The techniques presented in the previous two chapters define and evaluate the core algorithms for LDP coverage/frequency analysis of software traces. However, in order to deploy an actual data collection scheme, several open problems need to be solved. First, *how should an analyst select the tradeoffs between accuracy and privacy before the data collection is deployed?* Recall that the approaches described earlier use count sketch, a hash function based data structure, to share the data between users and the server. To improve accuracy due to potential hash collisions, multiple pairs of hash functions are used, instead of the basic version with a single pair of hash functions. Therefore, the local data at user end is a sketch matrix of $t \times m$ where t is the number of pairs of hash functions and m is the size of the output of hash functions h_i which map the domain of traces to integers in $\{1,...,m\}$. Chapters 3 and 4 propose two different randomization algorithms for Count Sketch to achieve ε-differential privacy. The local randomizers are parameterized with ε, known as the "privacy budget". Given that the randomizer at each row of the local sketch has privacy budget ε, the overall privacy budget for the entire count sketch is thus $t \times \varepsilon$. In the evaluations of Chapters 3 and 4, we set t as 256, and ε as relatively small values used by most other works [4, 18, 62]. This means that the effective privacy budget $t \times \varepsilon$ is large. While this approach achieves high accuracy of estimates, using this fixed large privacy budget is clearly undesirable. Our goal is to understand better the tradeoffs between

sketch shape (i.e., number of rows m and number of columns t) and accuracy, and to define an approach for selecting sketch size before deployment of the LDP analysis. Section 5.1 discusses these problems and the proposed solutions for frequency analysis.

The second issue is the potential under-randomization on users' local data. Recall that in Chapter 4, we proposed the definition of neighbors based on the parameter τ, which essentially specifies the limit on the distance between neighbors for which the differential privacy indistinguishability is guaranteed. The amount of noise added to each user's raw data by the local randomizer is dependent on τ, and its value is estimated by the developers from a set of test users whose raw data is available to them. This process of configuring τ happens ahead of the time when the software is deployed to all users, because randomization of the data is performed at user end so that only each user herself has access to her own raw data and only the randomized version of the data is transmitted to the analytics server which is controlled by the software developers. However, a user's real data (i.e. the frequency vector of software traces) in practice could be different from the data of the test users, such that the value of τ used to achieve the intended protection on traces is much less than the value of τ that should have been used to achieve the same level of DP guarantee for such a user. If the estimated τ is less than the real τ for a user's data, that means insufficient amount of noise is added to the user's raw data and therefore the claimed DP protection does not hold. To assess and improve the scenario when there is deviation of τ in practice, Section 5.2 conducts a series of experiments and proposes an improvement on the approach in Chapter 4 to mitigate under-randomization.

5.1 Reducing The Privacy Budget

To better understand the issue of large privacy budget, we conduct a characterization study on the effect of number of rows used in the count sketch. In the evaluations for the coverage analysis and frequency analysis, the number of rows of the count sketch is both set to 256, which produces low error for the frequency oracle but leads to high privacy budget of the overall process, i.e. $256 \times \varepsilon$ where ε is the privacy budget for each independent row of the count sketch. To prove this statement, consider two neighboring frequency vectors $\boldsymbol{f}$ and $\boldsymbol{f'}$. Let R_i ($i \in \{1,..,256\}$) be the process that maps the input vector to a single row of the count sketch and adds Laplacian noise to it. Let R be the overall process that produces the whole count sketch with 256 rows. For any output count sketch $Z = (Z_1, ..., Z_{256})$ where Z_i is a single row produced by R_i, it has been shown in Section 4.2 that $\frac{P[R_i(\boldsymbol{f})=Z_i]}{P[R_i(\boldsymbol{f'})=Z_i]} \leq e^{\varepsilon}$, where $P[...]$ denotes the probability of some event. Consequently, $\frac{P[R(\boldsymbol{f})=Z]}{P[R(\boldsymbol{f'})=Z]} = \frac{\prod_{i=1}^{256} P[R_i(\boldsymbol{f})=Z_i]}{\prod_{i=1}^{256} P[R_i(\boldsymbol{f'})=Z_i]} \leq e^{256\varepsilon}$.

5.1.1 Characterization Study of the Number of Sketch Rows

In the experiments below, we explore different settings where the number of rows is less than 256. More specifically, we reduce the number of rows while increasing the number of columns to keep the size of the sketch matrix fixed. This essentially makes the size of the count sketch a budget, and keeping the budget fixed gives a fair comparison among different sketch shapes. Other settings of the experiments in Section 4.3 are left unchanged. For simplicity, we focus on the frequency analysis because it is a generalization of coverage analysis and is thus more representative of realistic data collection.

To compare different numbers of rows for the frequency analysis, we repeat the experiments in Section 4.3. The number of rows is set to be a power of two ranging from 1

to 256. The number of columns is increased so that the total size of the count sketch for any of the settings is the same as it is for 256 rows. For example, let m be the number of columns for 256 rows. Then for 128 rows, the number of columns is $2 \times m$; for 64 rows, the number of columns is $4 \times m$, and so on. The same metric of error as in Section 4.3 is used. For simplicity, we choose the scenario for hiding the presence of 50% traces and $\varepsilon = 2$ as an example. Figure 5.1 and Figure 5.2 show the result (average of 5 runs) for call chains and enter/exit traces respectively. The same trend is observed for all apps studied in the experiment. Smaller number of rows leads to worse error. As the number of rows gets closer to 1, the error gets higher faster.

In count sketch, multiple rows are used to reduce the error caused by hash collisions. However, since we choose the number of columns (the range of hash functions) to be close to the number of traces covered by the test users and the number of columns are increased exponentially as the number of rows is decreased, the effect of hash collisions is not the reason causing this trend. Recall that the final estimate of the frequency of a given trace is the median of the estimates retrieved from each row of the count sketch matrix. Each of the estimates is a randomized value whose expected value is the real value because it is an unbiased estimate as achieved by both the randomized response mechanism in Chapter 3 and the Laplace mechanism in Chapter 4. Taking the median of those estimates reduces the error.

5.1.2 Configuring the Number of Sketch Rows

The characterization study shows that for most of the analyzed apps, using fewer rows in the count sketch still produces low error for the estimates. Based on this insight, we propose to amend the overall data collection process for frequency analysis in Figure 4.1 with the

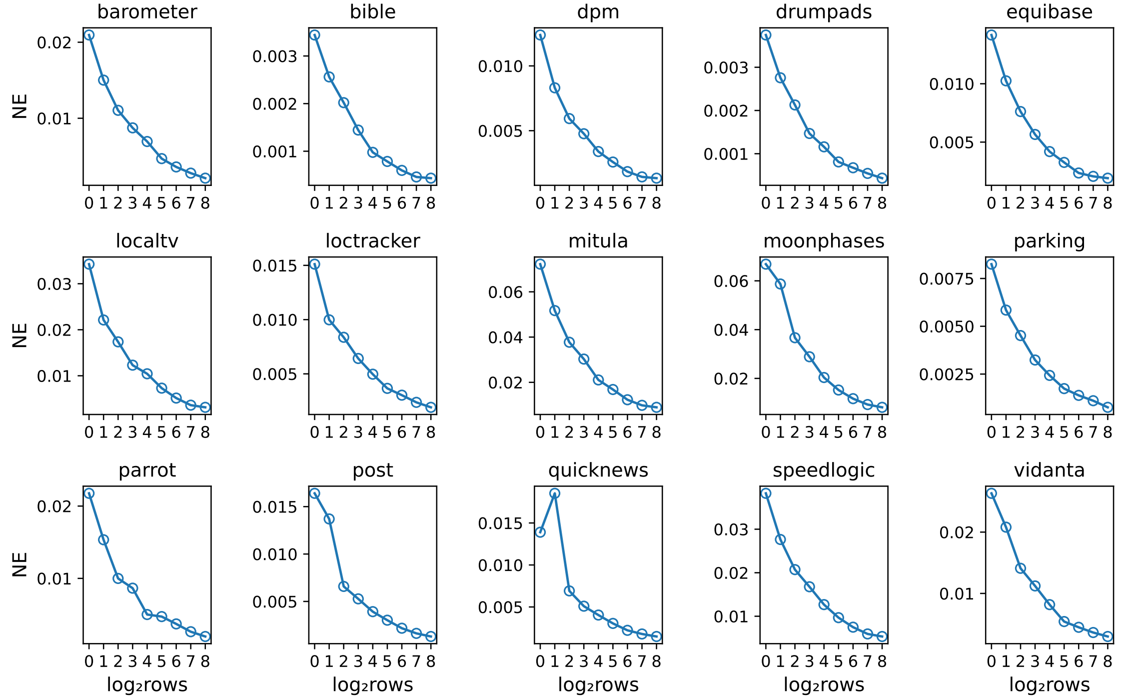


Figure 5.1: The normalized error of the estimates of all call chains with variant number of sketch rows in frequency analysis.

configuration of the number of rows of count sketch. The amended approach is illustrated in Figure 5.3. Compared to the original process, the data of test users are also used to perform a series of experiments to decide the number of rows in the count sketch. In addition to τ and η, which are the parameters for the amount of noise added by the local randomizer and the hotness threshold for hiding the hotness of traces respectively, the test users' data is also used for determining the appropriate number of columns in the count sketch. The details of how this process works are described next.

The software developers first analyze the real data of the test users to determine the value of τ depending on the level of protection they are aiming for, and the value of η if they are choosing the scenario where the hotness of traces are protected instead of presence. This

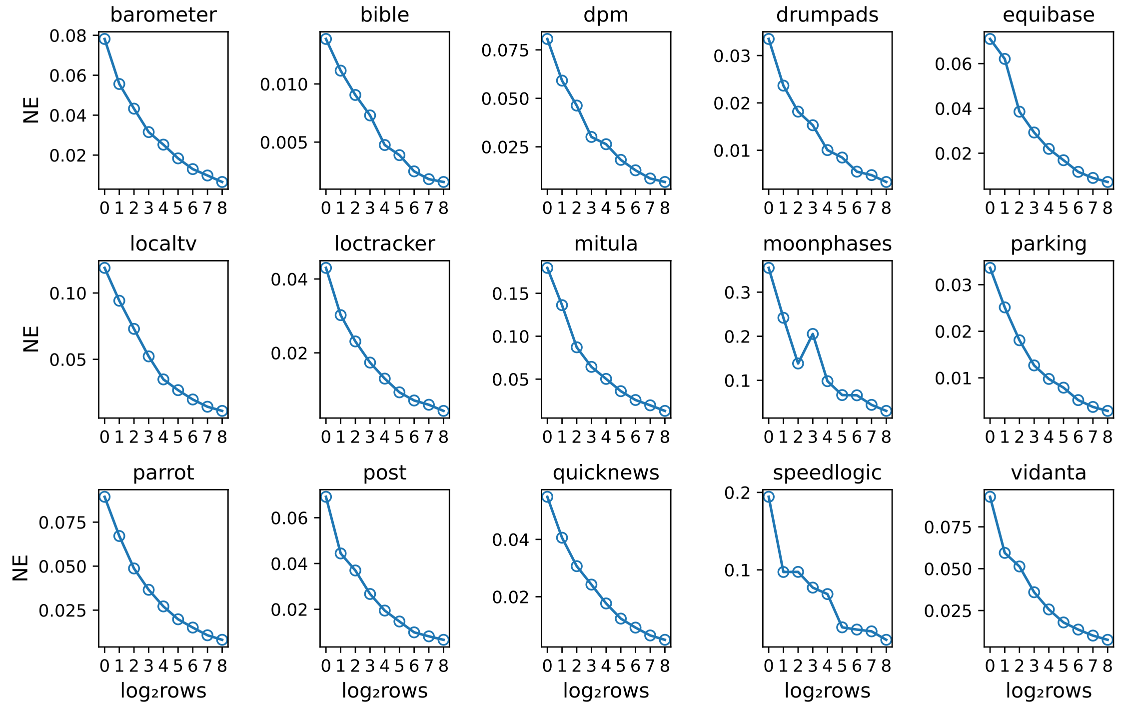


Figure 5.2: The normalized error of the estimates of all enter/exit traces with variant number of sketch rows in frequency analysis.

process is the same as the original approach in Section 4.2. Then a series of experiments are conducted on the test users using various configuration for the size of the count sketch. In each experiment, the local randomizer is run on each user and local sketches are constructed. The global sketch is constructed by aggregating all the local sketches. By inquiring the global sketch, the developers obtain estimates of frequency for all the software traces covered by at least one test user. The normalized error of the estimates (as used in the evaluation in Section 4.3) is calculated as the comparing metric for different sketch size configurations.

The only difference between the experiments is the size of count sketch. The base line configuration is $256(rows) \times m(columns)$ where m is the smallest power of 2 that is greater than or equal to the size of the set of traces covered by the test users. Other configurations

are $1 \times 256m$, $2 \times 128m$, $4 \times 64m$, ..., $128 \times 2m$. The number of rows is always a power of 2 and the total size is always $256m$. Each configuration is repeated 5 times and the average error is taken. The one with the smallest number of rows that achieves the target error, which is specified by the developers, will be used in the final deployment of the software. Because the error decreases almost monotonically as the number of rows grows, as we learned from the characterization study, a binary search strategy can be used instead of trying all of the configurations. An alternative simple strategy is to search from the smallest number of rows and stop until the target error is reached. This strategy takes less time than the binary search when the smaller numbers of rows are enough to achieve the target error. For example, if one single row is enough to achieve the target error, the binary search strategy requires 3 trails (4 rows, 2 rows, 1 row) while the simple approach takes just one trial.

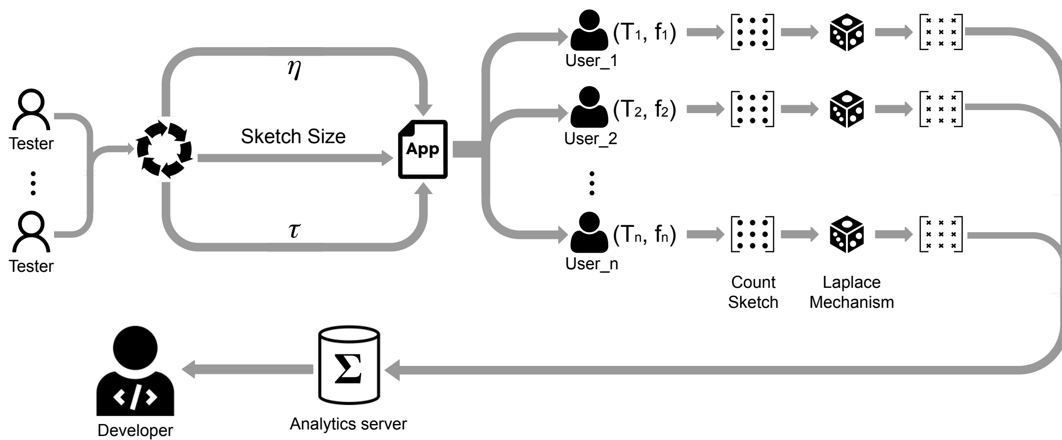


Figure 5.3: Data collection scheme with configuration of number of sketch rows for frequency analysis.

5.1.3 Evaluation

We evaluated the amended approach on the same 1000 simulated users for 15 Android apps as used in the evaluation for the original approach. More details of the process of obtaining the simulated data is in Section 4.3. However, in order to get the result for a larger population of users (which is the realistic scenario where LDP data collection is useful), we replicated each of the 1000 users by ten times. Then, 10% of the 10000 total resulting users were randomly selected as test users. Note that for each repetition of the experiments, the test users are selected independently. We set the target error to be 0.1.

Reduced Privacy Budget The most important goal of amending the data collection approach is to reduce the number of rows in count sketch and therefore to reduce the privacy budget. Table 5.1 shows the final count sketch size determined by the in-house characterization stage for the frequency analysis of both call chains and enter/exit traces, using $\varepsilon = 2$ and protecting 50% traces' presence. For call chains, one single row is enough to achieve the 0.1 target error for all apps. For enter/exit traces, one single row is enough for most apps, and 4 rows are enough for all apps. This greatly reduces the privacy budget from 256×2 to $2-8$ compared to the original approach. The number of columns is increased by $64-256$ times to keep the total sketch size the same as in the original approach which uses 256 rows. So, the volume of data shared between the users and the analytics server is not increased. Note that increasing the number of columns does not affect the privacy budget.

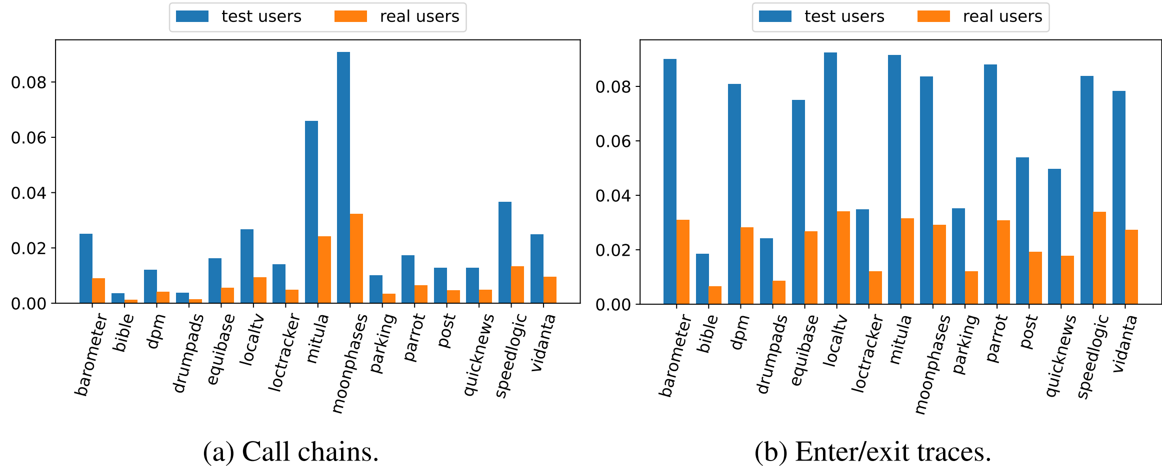
Accuracy of Frequency Estimates An important question that needs to be answered is how good are the estimates of frequencies of traces, using this amended approach. According to the trend in Figure 5.1 and Figure 5.2, reducing the number of rows causes the error of

app	Count Sketch Size in Log Base 2			
	call chains		enter/exit traces	
	$log_2(\#rows)$	$log_2(\#columns)$	$log_2(\#rows)$	$log_2(\#columns)$
barometer	0	20	0	20
bible	0	19	0	20
dpm	0	19	0	20
drumpads	0	18	0	19
equibase	0	18	0	19
localtv	0	21	1	20
loctracker	0	17	0	19
mitula	0	24	2	19
moonphases	0	19	2	17
parking	0	19	0	20
parrot	0	21	0	21
post	0	20	0	20
quicknews	0	20	0	21
speedlogic	0	16	0	17
vidanta	0	22	1	20

Table 5.1: The count sketch size (average of 5 runs) decided by the in-house characterization stage of the amended approach, for frequency analysis using $\varepsilon = 2$ and 50% protection of presence.

estimates go up. However, our goal is to keep the error under the target error while using as few rows as possible. The in-house characterization experiments can only make sure that the error for the test users is below the target error limit. So, this part of the experiments focuses on the error for the real users.

Figure 5.4a and Figure 5.4b compare the normalized error for test users versus real users, for call chains and enter/exit traces respectively. It indicates that the experimental study on test users tends to exaggerate the error of estimates for real users. One likely explanation is that the number of test users is expected to be significanly smaller than the number of real users, and theoretically LDP analysis accuracy increases with the number of users. Given



(a) Call chains.

(b) Enter/exit traces.

Figure 5.4: Normalized error (average of 5 runs) of estimates for test users versus real users in the frequency analysis.

that the error for test users is below the target limit, the final global count sketch obtained from the data of real users is very likely to achieve the target error limit.

5.2 Potential Under-Randomization

Both approaches for trace coverage and frequency analysis employ the local differential privacy model where randomization on the data happens at user end compared to the central model where the randomization happens at server side. The local randomizer at the user end perturbs the user's private raw data to ensure differential privacy. In the approach for frequency analysis, the local randomizer is parameterized by τ which determines the level of noise added to the user's raw data. Recall that in Section 4.2.3, we proposed two scenarios for hiding the information of software traces: hiding the presence of certain traces, and hiding their hotness. We introduced the definition of neighbors of frequency vectors based on their ℓ_1 distance and parameterized by τ which specifies the maximum distance between

neighboring frequency vectors. Consequently, the larger the value of τ is, the more noise is needed by the local randomizer to achieve the intended DP indistinguishability among neighbors.

In the evaluation in Section 4.3, a small portion (10%) of the 1000 emulated users is selected (randomly) as testing users whose raw data is shared with the analytics server. One of the ways of how these testing users are used is to configure the value of τ. From the raw data of the test users, the developers can get the set of traces covered by at least one test user and their frequencies by each individual user. Then the value of τ can be decided based on how many traces (25%, 50%, or 75%) need to be protected. Section 4.3 describes the details of this process. Essentially, the value of τ embedded in the local randomizer is an approximation of what it would have to be for the real users.

However, when the randomization happens at each user's local side, τ could be different from what it must be in order to protect the traces presence (or hotness) for that specific user. This deviation could cause under-randomization for that user when τ is less than what it should be to deliver the claimed privacy guarantee for her data. Take the scenario of hiding the presence of 50% traces as an example. If for some user's frequency vector, τ is only big enough to hide significantly fewer than 50% of her traces, there is a under-randomization and for her, the DP guarantee is weakened.

We conducted a characterization study to obtain further insights on this issue. At the local side, for each user, we compute τ_{local} and compare it with the value of τ used in the actual local randomizer. More specifically, let T_i be the frequency vector of traces by user u_i, and $T_i(t)$ is the frequency of trace t (i.e. the number of times t is covered by user u_i). For each trace t such that $T_i(t) > 0$, we compute the difficulty of hiding its presence (or hotness) in T_i. The definition of difficulty and how it is computed are described in Section 4.2.3.

Then the difficulties are sorted in ascending order and the x percentile is taken as τ_{local}. Recall that x here is the parameter that specifies how many traces are intend to be protected. The DP guarantee claims that $x\%$ of the user's traces are protected. Under-randomization happens when τ_{local} is greater than τ.

Table 5.2 shows the under-randomization rates for each app, using $x = 50\%$ (for hiding presence and hotness respectively) and $\varepsilon = 2$ on both the analysis of call chains and enter/exit traces. On average, 38%/48% (for hiding the presence and hotness respectively for call chains) and 46%/48% (for hiding the presence and hotness respectively for enter/exit traces) users are affected by this issue. For hiding the presence, 3 out of 15 apps (call chains) and 8 out of 15 apps (enter/exit traces) have more than half of users experiencing under-randomization. The number is 8 (call chains) and 7 (enter/exit traces) for hiding the hotness. This shows us that for many users, their DP guarantee is weakened. For example, for `localtv`, the DP guarantee that at least 50% of the covered call chains' presence is hidden fails for 90% users. The next part of this section proposes an approach to tackle this problem.

5.2.1 Mitigating Under-Randomization

The underlying cause of the under-randomization issue is the difference between the distributions of the data of test users and the data of real users. In order to alleviate (if not completely eliminate) under-randomization, software developers have to be aware of the severity of under-randomization and adjust the value of τ accordingly. Our solution is to establish a feedback loop regarding τ between the real users and the developers. Figure 5.5 demonstrates the amended data collection process. Compared to the original scheme in Figure 4.1, two changes are made. First, along with the randomized count sketch, the local τ_i of each user are also shared with the analytics server. Unlike the frequency vectors, τ_i

app	call chains		enter/exit traces	
	presence	hotness	presence	hotness
barometer	22%	82%	83%	100%
bible	31%	83%	53%	74%
dpm	28%	18%	58%	8%
drumpads	21%	92%	96%	7%
equibase	1%	12%	2%	50%
localtv	90%	6%	74%	9%
loctracker	67%	2%	65%	3%
mitula	34%	51%	1%	92%
moonphases	2%	70%	59%	31%
parking	32%	93%	47%	99%
parrot	10%	19%	27%	70%
post	43%	82%	36%	72%
quicknews	46%	12%	68%	5%
speedlogic	40%	36%	17%	78%
vidanta	1%	68%	0%	16%

Table 5.2: Percentage of users with under-randomization for call chain analysis and enter/exit trace analysis with protecting 50% presence and hotness, $\varepsilon = 2.0$. Averaged over 5 runs.

should not be considered as sensitive data that could leak anything meaningful about a user's privacy. The second change is that the server now computes a new parameter value τ' by taking the maximum value of all local τ_i and the old value of τ. The new value τ' is used by all the local randomizers for the next data collection cycle. As a result, the value of τ is gradually incremented until the under-randomization issue is mitigated.

This scheme can be applied in the general setting where there are several separate rounds of data collection. In each round, some group of users from the previous round drops from the collection, some group from the previous round remains, and some new users join the collection. With each data collection round, the value of τ increases (if needed). Such a scheme is likely to decrease the impact of under-randomization over time. The experiments presented below quantify this observation.

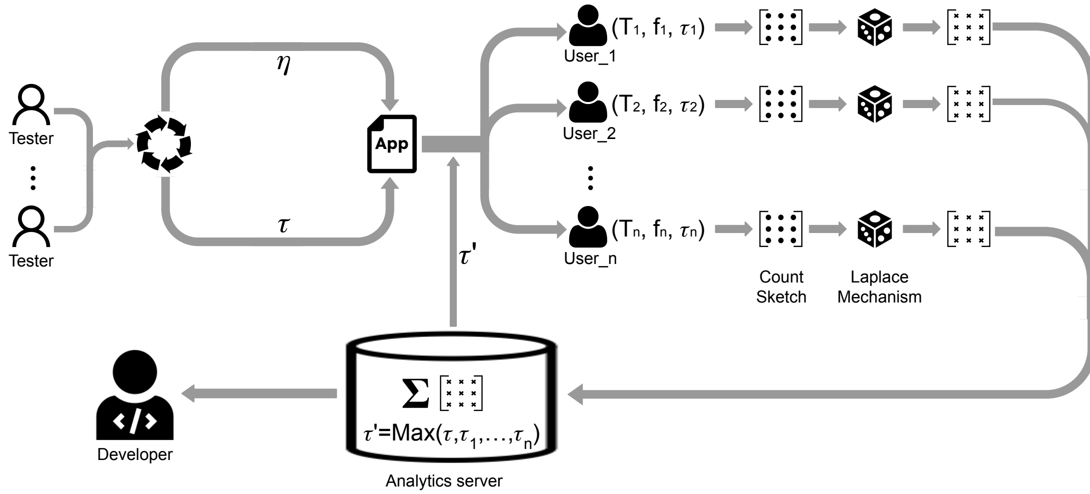


Figure 5.5: Data collection scheme with incremental update of τ for frequency analysis.

5.2.2 Evaluation

To simulate a data collection process with multiple rounds, we randomly divide the simulated 10000 Android users into four groups: 1000 of them are used as test users, and 3000 of the rest for each batch. Each batch is considered as the participants in a data collection round. The source code that implements the two refinements proposed in this chapter, the simulated traces, and instructions to reproduce the experimental results are publicly available in `https://presto-osu.github.io/dp-trace-freq`.

Figure 5.6, Figure 5.7, Figure 5.8, and Figure 5.9 illustrate how the percentage of users experiencing under-randomization in each batch changes as the batch number increases. The experiments in Figure 5.6 and Figure 5.7 (Figure 5.8 and Figure 5.9) are conducted on call chains (enter/exit traces), protecting 50% traces' presence and hotness respectively, and using $\varepsilon = 2.0$. Here the definition of under-randomization rate is the number of users whose

local τ is greater than τ used by the randomizer divided by the total number of users in each batch. Each experiment is repeated 5 times and the average is taken.

The figures show that for all apps, especially those having high under-randomization rate at the first batch, the rate drops rapidly at the second batch. Even though some users may have a weakened DP protection during the first batch, our approach can greatly curb the under-randomization and thus improve privacy beginning at the second batch.

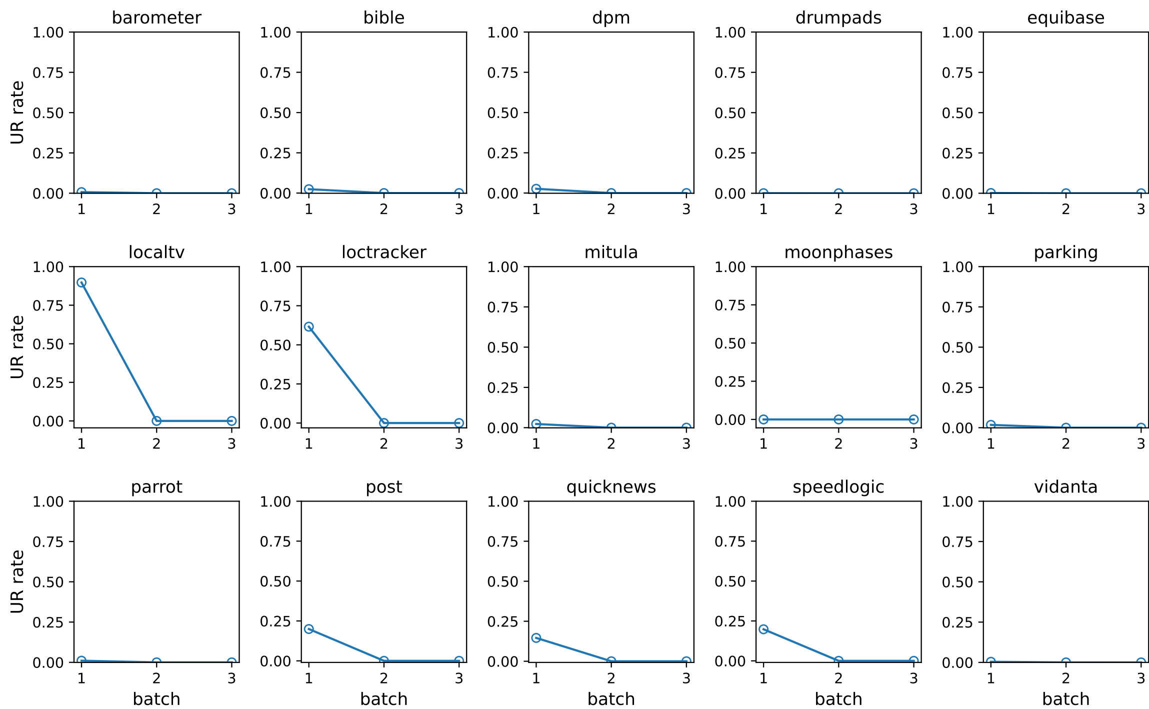


Figure 5.6: Under randomization (UR) rate (average of 5 runs) by batch for hiding the presence of 50% call chains, $\varepsilon = 2.0$.

As the value of τ is gradually increased, there is expected to be some impact on the accuracy of the estimates because greater τ leads to more random noise added to the raw data. Figure 5.10, Figure 5.11, Figure 5.12, and Figure 5.13 show the normalized error

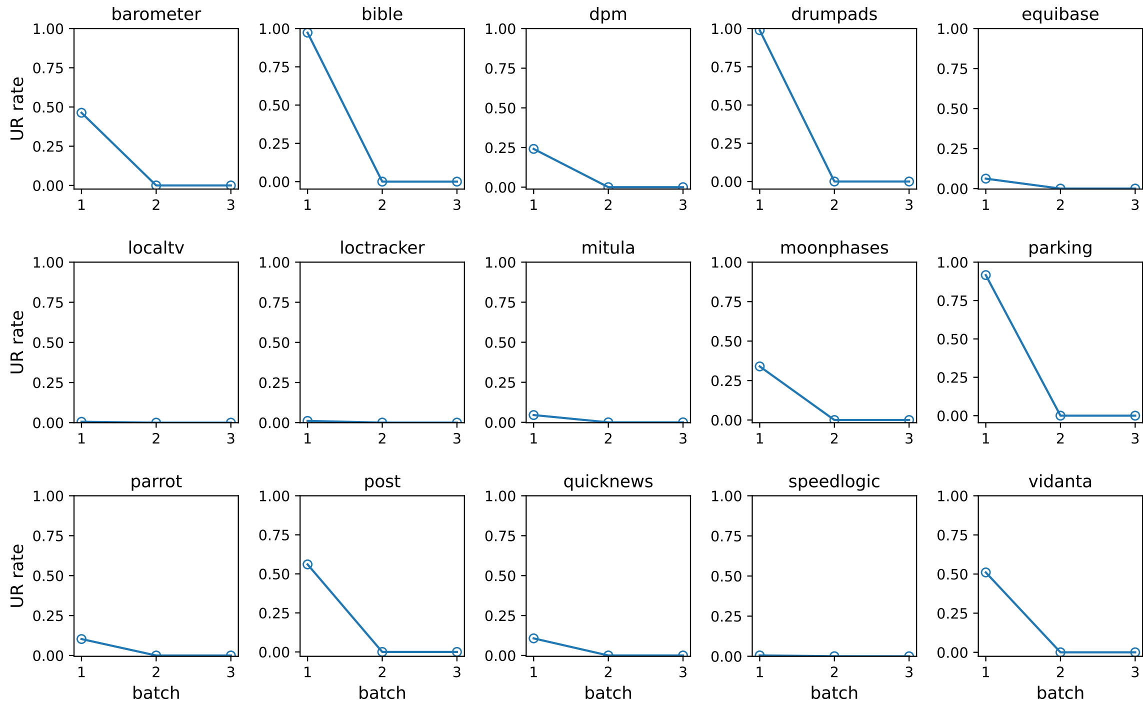


Figure 5.7: Under randomization (UR) rate (average of 5 runs) by batch for hiding the hotness of 50% call chains, $\varepsilon = 2.0$.

of estimates for the experiments corresponding to Figure 5.6, Figure 5.7, Figure 5.8 and Figure 5.9 respectively. On average, the error becomes 1.7 times (in Figure 5.10), 6.3 times (in Figure 5.11), 3.5 times (in Figure 5.12), and 6.6 times (in Figure 5.13) larger for the third round of data collection compared to the error for the first round. However, for most apps, the error is still below around 1% (call chains) and 5% (enter/exit traces).

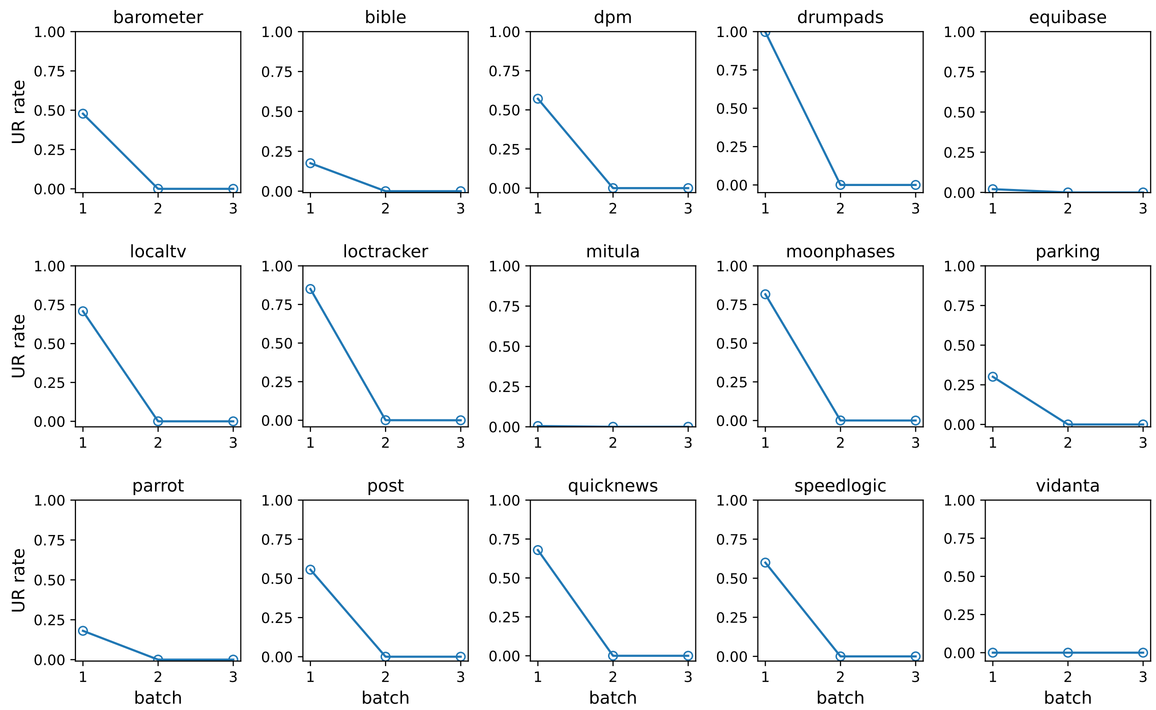


Figure 5.8: Under randomization (UR) rate (average of 5 runs) by batch for hiding the presence of 50% enter/exit traces, $\varepsilon = 2.0$.

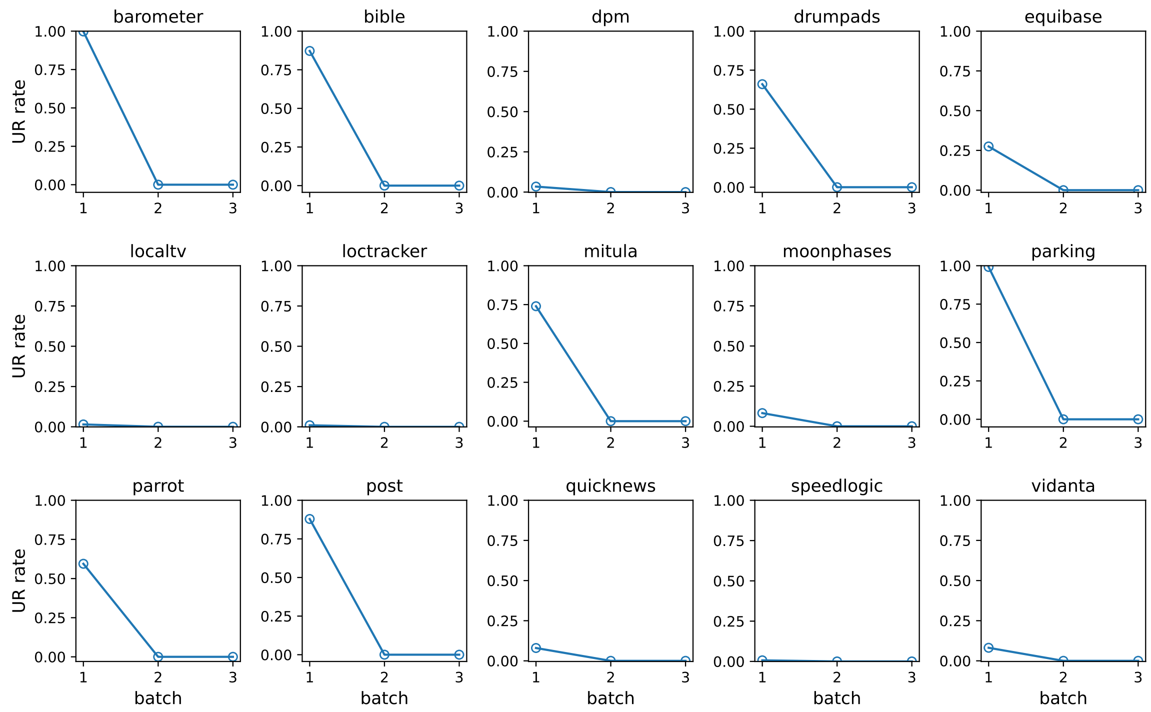


Figure 5.9: Under randomization (UR) rate (average of 5 runs) by batch for hiding the hotness of 50% enter/exit traces, $\varepsilon = 2.0$.

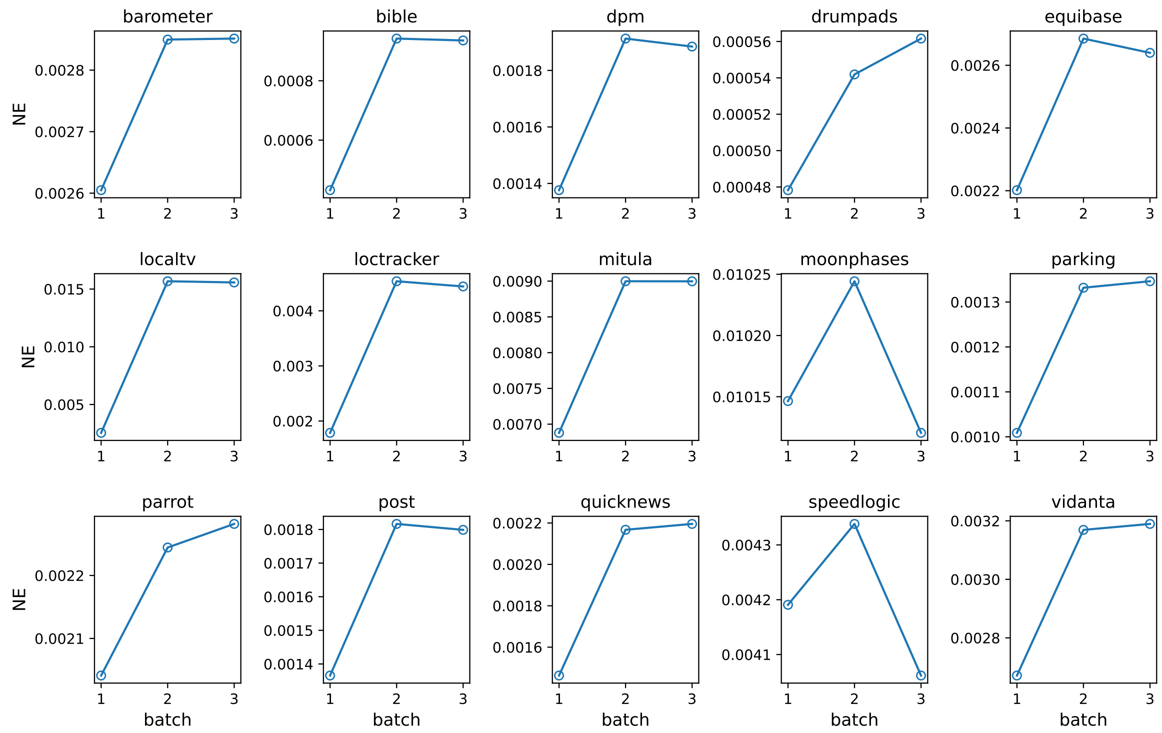


Figure 5.10: Normalized error (average of 5 runs) by batch for hiding the presence of 50% call chains, $\varepsilon = 2.0$.

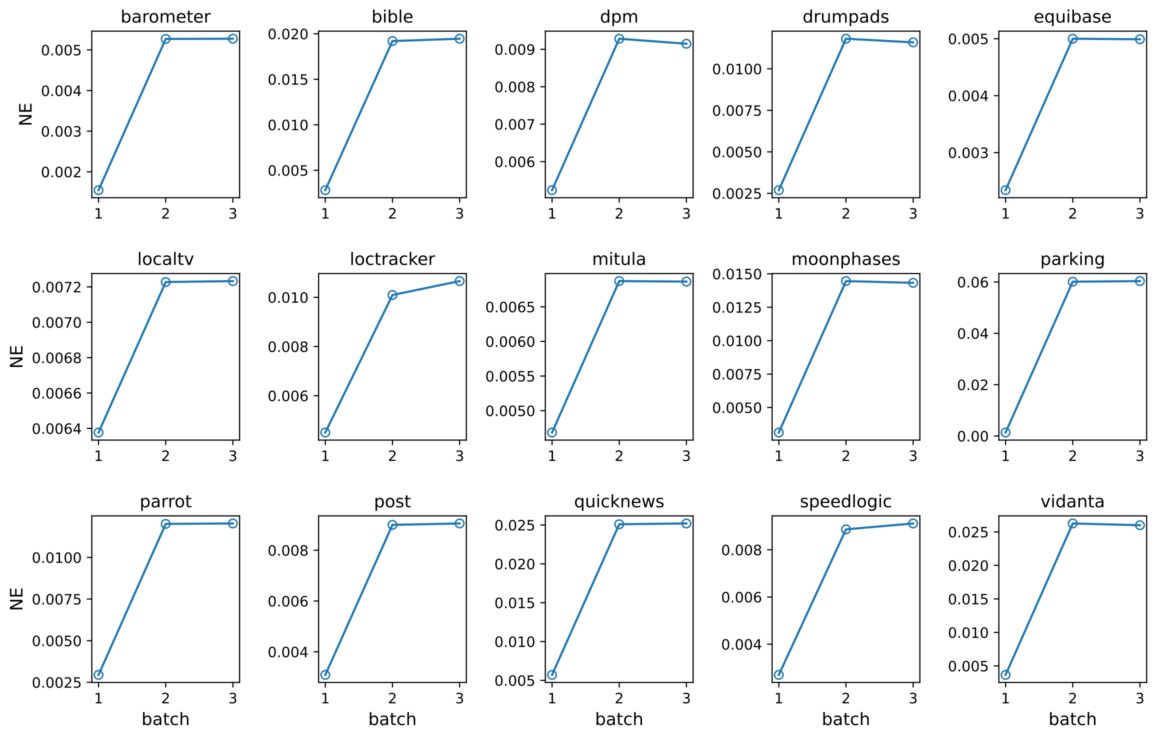


Figure 5.11: Normalized error (average of 5 runs) by batch for hiding the hotness of 50% call chains, $\varepsilon = 2.0$.

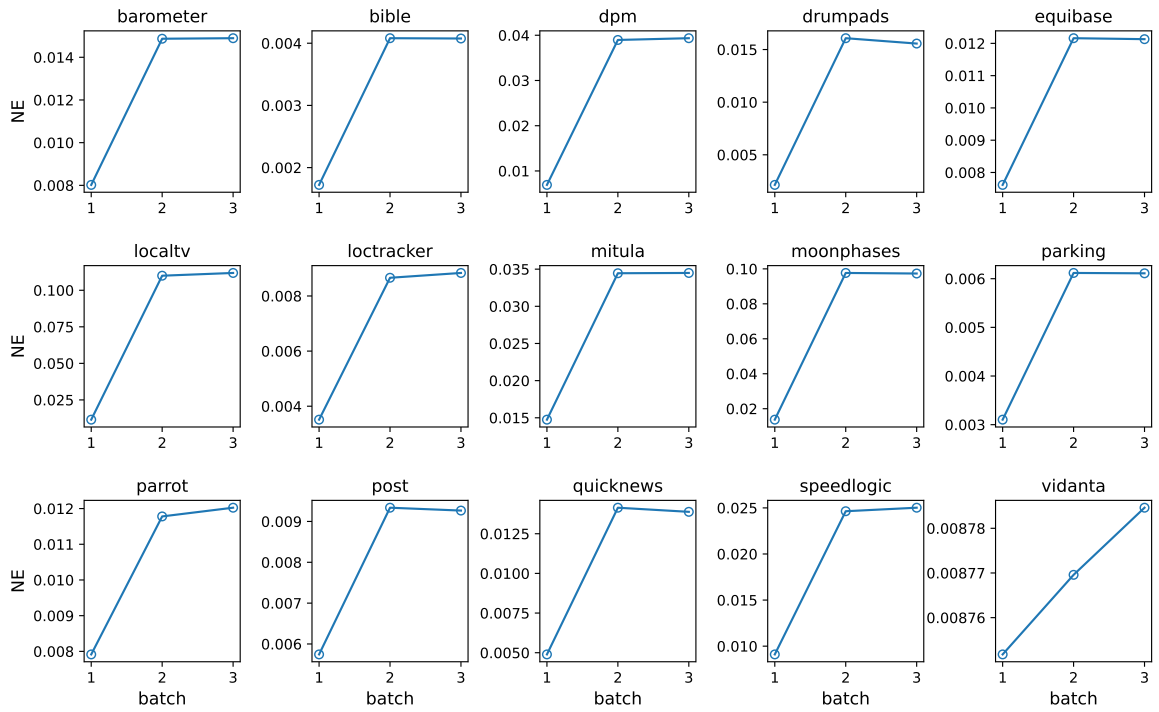


Figure 5.12: Normalized error (average of 5 runs) by batch for hiding the presence of 50% enter/exit traces, $\varepsilon = 2.0$.

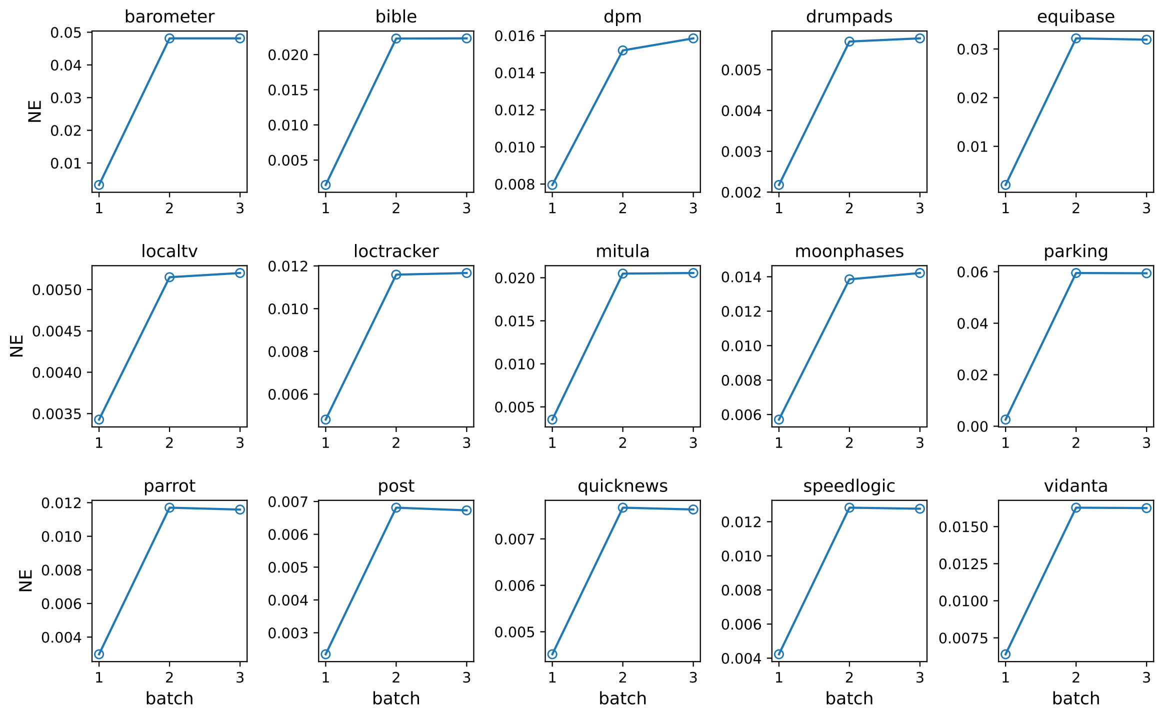


Figure 5.13: Normalized error (average of 5 runs) by batch for hiding the hotness of 50% enter/exit traces, $\varepsilon = 2.0$.

5.3 Summary

This chapter focuses on two main issues regarding the deployment of the proposed LDP frequency analysis: high privacy budget and potential under-randomization.

The first part designs and conducts a characterization study on the effects of the number of rows in the count sketch. The overall privacy budget is $\varepsilon\times$ the number of rows, where ε is the privacy budget for a single row. We found that the accuracy of frequency estimates of traces tends to decrease as fewer rows are used, with the overall size of count sketch fixed. Based on this insight, we propose to configure the sketch size by conducting pre-deployment experiments on a set of test users. The smallest number of rows that achieves the target accuracy will be used in the deployment. Evaluation of this solution shows that for most of the analyzed apps, one row is enough to achieve 1% error.

The experiment in the second part reveals that a large number of users may be affected by under-randomization, which happens when the τ parameter used by the local randomizer is smaller than it needs to be in order to hide the intended percentage of traces covered by the user. The amount of random noise added to each user's raw data is determined by τ and the value of τ is the same for all the local randomizers. To mitigate under-randomization, we propose to increase the value of τ for the next data collection cycle based on feedback from the users' local τ values. Experiments show that the under-randomization rate drops quickly while high accuracy is still achieved.

Chapter 6: Related Work

Differential privacy There is a large body of work on both the theory and practice of differential privacy. As already discussed, several approaches based on randomized response consider a single data item per user [8, 18, 55], while we are interested in a set of data items (i.e., a set of locally-covered traces). Differentially-private analysis of software executions has also been studied in prior work [61–63]. In those efforts the domain of possible items is small, enumerated ahead of time before software deployment, and the randomizer output is straightforward to generate and store. A key distinguishing feature of our work is that the domain of possible traces is either infinite or very large, which requires different randomization techniques. We address this problem by using a count sketch representation. This allows tunable trade-offs between accuracy and representation size, as well as higher accuracy for high-frequency traces. Efficient randomization of simple bitvectors has also been considered [61]. Our efficient randomization (Section 3.2.3) requires more general reasoning. Because of the small number of possible data items, these prior efforts do not need to explore a large domain in order to identify hot items. In contrast, we need to develop effective search in a domain containing billions of possible traces. We demonstrate how to achieve this using considerations of trace prefixes and suffixes, and illustrate the approach

for context-free-language domains by exploring the states of the corresponding automaton (Section 3.3).

Privacy-preserving techniques in programming languages and software engineering The programming languages community has investigated techniques for testing and verification of differentially-private algorithms and implementations [38, 56, 59, 64]. Privacy issues are also important for many areas in software engineering, including design [25], testing [9, 24, 34, 53], and defect prediction [32, 46, 47]. Other than the work described earlier, we are not aware of attempts to employ differential privacy techniques in this area. Given the strong theoretical properties of such techniques, and their increasing adoption in industry and government [4, 14, 18, 35, 54], it is a worthwhile research goal to reconsider a range of software engineering techniques using differential privacy machinery.

Analysis of deployed software Remote analysis of deployed software is an area with a significant body of prior work. As one example, residual coverage monitoring [45] uses coverage information from software users for testing purposes. GAMMA [43] collects data from software users and orchestrates the data collection across program instances. Placement of profiling probes has been considered by several projects [15, 40]. Failure reproduction and debugging are aided by collected data from deployed software [12]. Similarly, researchers have proposed analysis of post-deployment failure reports [41].

Privacy in remote software analysis has been targeted by prior efforts. Anonymization of collected data has taken several forms [13, 17]. As shown by privacy researchers [36, 37], anonymization is not enough to provide strong privacy guarantees. Instead, we consider the principled protection provided by local differential privacy. Remote software analyses from prior work could potentially benefit from developing differentially-private versions for them.

Examples of such analyses include impact analysis and regression testing [44], as well as failure analysis [28, 30, 31].

Chapter 7: Conclusions

There is a large body of prior work on software analysis that could be revisited with increased emphasis on privacy in general, and differential privacy in particular [12, 13, 28, 30, 31, 41, 43, 44]. Such studies will contribute to broader efforts to integrate privacy-preserving techniques in the analysis of deployed software, in response to growing needs for better privacy of data collection. This dissertation studies one particular category of software profiling: software execution traces. We propose novel approaches based on local differential privacy to achieve privacy-preserving remote analysis of software traces.

In particular, we consider two problems: coverage analysis and frequency analysis. The former is about whether a trace is covered, while the latter is about how frequently a trace is executed. Chapter 3 proposes a novel approach for coverage analysis, employing count sketch to handle the exponentially large trace domain problem, as well as efficient randomization. In addition, a technique is proposed for discovering the frequently covered (hot) traces. Chapter 4 and Chapter 5 extend the work to the analysis of frequency of traces, employing Laplacian random noise for achieving differential privacy and providing the flexibility to choose between different protection modes. This work is the first to integrate differential privacy into the profiling of software traces. Extensive experiments using simulated users on Android application are conducted to demonstrate the efficacy of the proposed approaches.

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, page 85–96, 1997.

[3] G. Ammons, J. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, pages 172–196, 2004.

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

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[6] B. Avent, A. Korolova, D. Zeber, T. Hovden, and B. Livshits. BLENDER: Enabling local search with a hybrid differential privacy model. In *USENIX Security*, pages 747–764, 2017.

[7] T. Ball and J. Larus. Optimally profiling and tracing programs. *TOPLAS*, 16(4):1319–1360, July 1994.

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

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

[10] M. Canini, V. Jovanović, D. Venzano, B. Spasojević, O. Crameri, and D. Kostić. Toward online testing of federated and heterogeneous distributed systems. In *USENIX ATC*, pages 20–20, 2011.

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

[12] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE*, pages 261–270, 2007.

[13] J. Clause and A. Orso. Camouflage: Automated anonymization of field data. In *ICSE*, pages 21–30, 2011.

[14] 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.

[15] M. Diep, M. Cohen, and S. Elbaum. Probe distribution techniques to profile events in deployed software. In *ISSRE*, pages 331–342, 2006.

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

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

[18] Ú. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067, 2014.

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

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

[21] Google. Google Analytics. `https://analytics.google.com`.

[22] Google. Firebase Analytics. `https://firebase.google.com`, 2020.

[23] Google. Monkey: UI/Application exerciser for Android. `https://developer.android.com/studio/test/monkey`, 2020.

[24] M. Grechanik, C. Csallner, C. Fu, and Q. Xie. Is data privacy always good for software testing? In *ISSRE*, pages 368–377, 2010.

[25] 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.

[26] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.

[27] Y. Hao, S. Latif, H. Zhang, R. Bassily, and A. Rountev. Differential privacy for coverage analysis of software traces. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 8:1–8:25, 2021.

[28] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/FSE*, pages 146–155, 2005.

[29] 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.

[30] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *ICSE*, pages 474–484, 2012.

[31] W. Jin and A. Orso. F3: Fault localization for field failures. In *ISSTA*, pages 213–223, 2013.

[32] Z. Li, X. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying. On the multiple sources and privacy preservation issues for heterogeneous defect prediction. *IEEE Transaction on Software Engineering*, pages 1–21, 2017.

[33] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

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

[35] 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`, 2020.

[36] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *S&P*, pages 111–125, 2008.

[37] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *S&P*, pages 173–187, 2009.

[38] J. P. Near, D. Darais, C. Abuah, T. Stevens, P. Gaddamadugu, L. Wang, N. Somani, M. Zhang, N. Sharma, A. Shan, and D. Song. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), Oct. 2019.

[39] Oath. Flurry. `http://flurry.com`.

[40] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit. Optimizing customized program coverage. In *ASE*, pages 27–38, 2016.

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

[42] OpenDP. OpenDP. `https://projects.iq.harvard.edu/opendp`, 2020.

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

[44] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE*, pages 128–137, 2003.

[45] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *ICSE*, pages 277–284, 1999.

[46] F. Peters and T. Menzies. Privacy and utility for defect prediction: Experiments with MORPH. In *ICSE*, pages 189–199, 2012.

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

[48] T. Reps. Program analysis via graph reachability. *IST*, 40(11-12):701–726, 1998.

[49] Soot. Soot analysis framework. `https://soot-oss.github.io/soot`, 2020.

[50] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[51] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *OOPSLA*, pages 180–195, 2001.

[52] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. *IEEE Transaction on Software Engineering*, 38(5):1160–1177, 2012.

[53] K. Taneja, M. Grechanik, R. Ghani, and T. Xie. Testing software in age of data privacy: A balancing act. In *ESEC/FSE*, pages 201–211, 2011.

[54] Uber. Uber releases 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*, pages 729–745, 2017.

[56] Y. Wang, Z. Ding, G. Wang, D. Kifer, and D. Zhang. Proving differential privacy with shadow execution. In *PLDI*, pages 655–669, 2019.

[57] S. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 309(60):63–69, 1965.

[58] A. Wood, M. Altman, A. Bembenek, M. Bun, M. Gaboardi, J. Honaker, K. Nissim, D. O'Brien, T. Steinke, and S. Vadhan. Differential privacy: A primer for a non-technical audience. *Vanderbilt Journal of Entertainment and Technology Law*, 21(1): 209–276, 2018.

[59] D. Zhang and D. Kifer. LightDP: Towards automating differential privacy proofs. In *PLDI*, pages 888–901, 2017.

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

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

[62] 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.

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

[64] H. Zhang, E. Roth, A. Haeberlen, B. Pierce, and A. Roth. Testing differential privacy with dual interpreters. *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA), Nov. 2020.

[65] X. Zhuang, M. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, pages 263–271, 2006.