Differentially-Private Remote Software Profiling

Dissertation

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

By

Hailong Zhang

Graduate Program in Computer Science and Engineering

The Ohio State University

2020

Abstract

Remote profiling of deployed software has been studied in many contexts. In remote profiling, data is collected locally and then sent to a remote server where it is analyzed by the developers of the software and analysts working with them. There are significant privacy concerns about the collection and use of such data. For example, this sensitive data could potentially be misused due to rogue employees, legal proceedings, unethical business practices, or security breaches. The goal of this dissertation is to introduce principled privacy protection guarantees in the data collection process, after the profiling data has been collected locally but before sending it to the remote server. To provide a privacy-by-design solution with well-defined privacy properties, we employ differential privacy (DP), a powerful technique that allows meaningful statistics to be collected for a population without revealing "too much" information about any individual member of the population.

The first contribution of this dissertation is a parameterized randomization approach for run-time event frequency profiling that achieves DP with respect to event traces. This approach introduces random noise to the local profile at each user in a way that prevents any adversary from inferring the actual event trace during data gathering. To compute useful statistics, software developers post-process the aggregated profile to account for the randomization. Using program analysis techniques, we extract *a priori* knowledge about relationships between events in the run-time profile and incorporate these relationships in the post-processing step. This produces frequency estimates that are consistent with the structure of the true event frequencies and reduces the error of the resulting estimates. We perform a study of method call traces from Android apps and show that well-designed solutions can achieve both high accuracy and principled privacy-by-design for the fundamental problem of event frequency profiling.

As another setting for software frequency profiling, the profile at each user could be a frequency vector instead of an event trace. We propose an approach that reports the user-level frequency information with the addition of random noise drawn from the Laplace distribution. This approach achieves significantly higher accuracy of profiling results without reducing privacy protections in any substantial way, compared to our first approach introduced earlier. In addition, we propose a novel linear programming formulation to compute the magnitude of random noise that should be added to achieve meaningful privacy protections under certain linear constraints. These constraints are due to the intrinsic static structure of the underlying software and are commonly observed in software systems. To the best of our knowledge, no prior work has incorporated such domain-derived data constraints in the design of a DP analysis. Our experimental analysis shows that the privacy protection can be significantly weakened if the DP design does not take into account these constraints.

The third focus for this dissertation is a profiling problem related to control-flow node coverage, where the constraints are with respect to control-flow graph (CFG) nodes. In such data collection, events are represented as nodes in a CFG and the edges in the CFG represent the transitions between events. Every execution of the software corresponds to a subgraph of the CFG that is covered at run time. We propose a novel definition of graph neighbors for a particular run-time covered subgraph, in order to account for the strong correlations between CFG nodes. We demonstrate that such correlations are captured by the notion of dominators, which is traditionally used in compiler optimizations. We use this insight to define the privacy guarantees that need to be achieved by any DP solution for control-flow node coverage analysis, and then propose an analysis to achieve these guarantees by randomizing the coverage information. Our experimental results demonstrate that the proposed analysis can achieve practical accuracy while providing principled DP guarantees.

Overall, this dissertation presents several approaches targeting various profiling tasks with the goal of providing principled privacy guarantees for individual user's profile data, while still allowing developers to learn useful statistical results across the whole user population. These approaches are promising advances in the larger landscape of privacy-preserving software analysis. We believe that applying similar techniques based on DP to other software analysis problems will be a fruitful direction for future work.

To my family

Acknowledgments

I would like to express my sincere gratitude to my advisor, Atanas Rountev, for his generous help with both my research and life. His dedication in training and guiding me through the doctorate studies has helped me become an independent researcher. I am especially thankful for his offer of opportunities for me to attend conferences and workshops to build connections with other researchers, and his numerous valuable suggestions for my career development. They strengthened my determination to pursue a career in academia.

I am also grateful to the many professors that I have interacted and collaborated with at the Ohio State University. Especially, I would like to thank Raef Bassily, Michael Bond, Feng Qin, and Neelam Soundarajan for serving on my dissertation and candidacy committees. I have had many fruitful discussions with Raef. He provided a considerable amount of insightful advice on various projects and paper drafts. Raef, Mike, and Feng have always been supportive during my job search. I thank them for taking the time preparing letters of recommendation and giving feedback on my application materials. I also appreciate the feedback from Yinqian Zhang on paper revisions. His comments and suggestions were instrumental in publishing the work on analysis of control-flow graphs at the 2020 USENIX Security Symposium.

My research would not proceed so smoothly without the support and contributions from members of the PRESTO research group, including Shengqian Yang, Yan Wang, Haowei Wu, Sufian Latif, and Yu Hao. I am thankful for their timely help and constructive discussions regarding my research.

Lastly, I want to thank my wife Jingqiu Liao and my parents for their understanding and unconditional love. Jingqiu is my biggest support during my Ph.D. studies. Her company and encouragement make me feel not alone in my academic journey. My parents always believe in me and are supportive to my decisions. I dedicate this dissertation to them.

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

March 2014	M.E., Computer Technology, Beijing University of Posts and Telecommunications, China.
June 2011	B.E., Network Engineering, Beijing University of Posts and Telecommunications, China.

Publications

Research Publications

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

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

Haowei Wu, Hailong Zhang, Yan Wang, and Atanas Rountev. Sentinel: Generating GUI Tests For Sensor Leaks in Android and Android Wear Apps. In *Software Quality Journal*, December 2019. (The two lead authors contributed equally to this work.)

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

Hailong Zhang, Haowei Wu, and Atanas Rountev. Detection of Energy Inefficiencies in Android Wear Watch Faces. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2018.

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

Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static Window Transition Graphs for Android. In *Automated Software Engineering Journal*, June 2018.

Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. Orlis: Obfuscation-Resilient Library Detection for Android. In *International Conference on Mobile Software Engineering and Systems*, May 2018.

Hailong Zhang and Atanas Rountev. Analysis and Testing of Notifications in Android Wear Applications. In *International Conference on Software Engineering*, May 2017.

Yan Wang, Hailong Zhang, and Atanas Rountev. On the Unsoundness of Static Analysis for Android GUIs. In *International Workshop on the State Of the Art in Program Analysis*, June 2016.

Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated Test Generation for Detection of Leaks in Android Applications. In *International Workshop on Automation of Software Test*, May 2016.

Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. Static Window Transition Graphs for Android. In *International Conference on Automated Software Engineering*, November 2015.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Programming Language and Software Engineering	Prof. Atanas Rountev
Software Dependability and Security	Prof. Feng Qin
High Performance Computing	Prof. P. Sadayappan

Table of Contents

Page

Abstract ii

Dedication v

Acknowledgments vi

Vita viii

List of Tables xiii

List of Figures xiv

1. Introduction 1

    1.1 Overview and Outline 3
    1.2 Contributions and Impact 6

2. Background and Terminology 8

    2.1 Differential Privacy 8
    2.2 Frequency Oracle 9
    2.3 Differential Privacy Techniques 11

3. Event Frequency Profiling with Differential Privacy 13

    3.1 Problem and Motivation 14
    3.2 The Differential Privacy Guarantee 16
        3.2.1 Example 17
        3.2.2 Privacy for Presence/Absence in Trace 19
    3.3 Differentially-Private Event Frequency Profiling 20
        3.3.1 Efficient Randomization 20

3.3.2 Server-Side Computation of Estimates 22
3.3.3 Static Analysis of Call Frequencies 24
3.4 Evaluation 27
3.4.1 Data Collection 27
3.4.2 Implementation 28
3.4.3 Accuracy of Estimates 29
3.4.4 Estimates for Hot Methods 32
3.4.5 Presence/Absence of Infrequent Methods 34
3.4.6 Importance of Consistency Constraints 35
3.5 Discussion and Limitations 37
3.6 Summary 38

4. Differentially-Private Frequency Profiling Under Linear Constraints 39

4.1 Problem and Motivation 40
4.1.1 Frequency Profiling 40
4.1.2 Privacy-Preserving Profiling 41
4.2 Feasible Frequency Vectors Under Linear Constraints 43
4.3 The Differential Privacy Guarantee 44
4.3.1 Indistinguishability 45
4.3.2 Defining Neighbors 46
4.3.3 Randomization Based on Laplace Mechanism 46
4.3.4 Randomization Based on Randomized Response 48
4.3.5 Calibration of Estimates 52
4.4 Hiding The Presence or Hotness of Individual Events 52
4.4.1 Difficulty of Hiding An Event 54
4.4.2 Computing Difficulty Using Linear Programming 54
4.4.3 Computing Difficulty Using Constraint Graph Analysis 55
4.4.4 Hiding The Hotness of An Event 57
4.4.5 Importance of Constraints in Randomizer Design 58
4.5 Overall Design of Data Collection 59
4.6 Evaluation 60
4.6.1 Comparison Between Two Randomizers 62
4.6.2 Hiding The Presence of Events 64
4.6.3 Hiding The Hotness of Events 68
4.6.4 Implications of Enforcing Linear Constraints 70
4.7 Summary 71

5. Differentially-Private Control-Flow Node Coverage Analysis 73

5.1 Problem and Motivation 74
5.1.1 Problem Statement 76

5.1.2 Differentially-Private Node Coverage Analysis 77
5.2 Feasibility and Neighbors 78
5.3 LDP Analysis 82
5.3.1 Difficulty of Hiding Graph Nodes 83
5.3.2 Randomizer Definition 86
5.4 Selection of Sensitivity Bound 88
5.4.1 Baseline: Global Sensitivity 89
5.4.2 Tighter Bound Based on Local Sensitivity 91
5.4.3 Tighter Bound via Restricted Sensitivity 92
5.4.4 Relaxed Indistinguishability of Neighbors 94
5.5 Evaluation 96
5.5.1 Metrics 98
5.5.2 GUI Screen Graphs 99
5.5.3 Call Graphs 103
5.6 Summary 107

6. Related Work 108

7. Conclusions 112

Bibliography 114

List of Tables

Table **Page**

3.1 Benchmarks. 27

4.1 Experimental subjects. 61

4.2 Average RE across all apps for hiding event presence. 65

4.3 Average RE and HMC for hiding the presence of frequently-executed events given $\ell = 0.25$. 67

4.4 Average RE across all apps for hiding event hotness. 68

4.5 Average RE and HMC for hiding the hotness of frequently-executed events given $\ell = 0.25$. 70

5.1 Apps and control-flow graph models. 97

5.2 Average RE across all apps for GUI screen graphs. 102

5.3 Average RE and HNC for hot nodes in GUI screen graphs given $\ell = 0.25$. . 102

5.4 Average RE across all apps for call graphs. 105

5.5 Average RE and HNC for hot nodes in call graphs given $\ell = 0.25$. 107

List of Figures

Figure **Page**

3.1 Relative error of estimates. 30

3.2 Ground-truth $\boldsymbol{F}$ and estimates $\bar{\boldsymbol{F}}$ and $\hat{\boldsymbol{F}}$ of 20 most frequently-executed methods in the speedlogic app with $\varepsilon = \ln 9$. 32

3.3 Comparison of RE for all methods (i.e., the full domain $\mathcal{V}$) and for hot methods with $\ell = 0.25$ and $\varepsilon = \ln 9$. 33

3.4 HMC for $\ell = 0.25$. 35

3.5 Full-domain RE of $\bar{\boldsymbol{F}}$ and $\hat{\boldsymbol{F}}$ with $\varepsilon = \ln 9$. 36

4.1 Code derived from the mitula app. 49

4.2 Comparison between randomization based on randomized response and Laplace mechanism, by simulating 100 users using $\mathcal{V} = \{e_1, e_2, e_3\}$ from Figure 4.1, $k = 3$, $\varepsilon = 1$, and $\tau = 1$. 51

4.3 Comparison of accuracy using randomization based on randomized response and on Laplace mechanism with $\varepsilon = 1$. 63

4.4 Error for hiding event presence with varying h and ε. 66

4.5 Error for hiding event hotness with varying h and ε. 69

5.1 User interactions are reported for data analysis. 75

5.2 Feasible and infeasible coverage. 80

5.3 GUI screen view graph G from the parking app. 85

5.4 $G_{\boldsymbol{c}}$, coverage vector $\boldsymbol{c}$, and dominator tree for $G_{\boldsymbol{c}}$, from the parking app. . . 86

5.5 Randomization using global sensitivity S_{gs} and $\varepsilon = 1$ for the parking app, with 10 users. 90

5.6 Precision and recall for GUI screen graphs. 100

5.7 Relative error for GUI screen graphs. 103

5.8 Precision and recall for call graphs. 104

5.9 Relative error for call graphs. 106

Chapter 1: Introduction

Remote profiling of deployed software has been studied in many contexts. For example, performance profiling of a user's execution behavior can be used to guide program optimizations [2, 43, 63, 66, 87, 90] by considering selective optimization and feedback-directed code generation [4]. Other related uses of such remote analysis include debugging [48, 57, 75], failure reproduction [20, 47], facilitating testing [14, 78], and bug isolation [57].

In such scenarios, profiling data is collected locally and then sent to a remote server where it is analyzed by the developers of the software and analysts working with them. There are significant privacy concerns about the collection and use of such data. While data analysis results can be used for enhancement of the software, individuals' usage data becomes transparent to software developers, as well as to the analysis service providers such as Google [39] and Facebook [34]. This sensitive data could potentially be misused due to rogue employees, legal proceedings, unethical business practices, or security breaches. These concerns are amplified by growing legislative efforts and societal demands for increased transparency and well-defined compromises between the utility of personal data gathering and the corresponding loss of privacy.

The goal of this dissertation is to introduce privacy protection guarantees in the data collection process, after the profiling data has been collected locally but before sending it to the remote server. Other researchers have tried to anonymize such software profiling data during collection [21, 31]. However, data anonymization by itself is not enough to provide strong privacy guarantees, as even anonymized data could be combined with external sources of information to carry out a number of privacy attacks (e.g., person re-identification; linking of related data from independent sources) [67, 68]. To provide a privacy-by-design solution with well-defined privacy properties, we employ differential privacy.

Differential privacy (DP) [28, 30] allows meaningful statistics to be collected for a population without revealing "too much" information about any individual member of the population. In the area of software profiling, this machinery allows profiling data from many users of a deployed software system to be collected and analyzed in a privacy-preserving manner. Such a solution is appealing to many stakeholders, including software users, software developers, providers of software analysis infrastructures, as well as government agencies that enforce consumer protections. This is because DP provides well-defined, quantifiable privacy guarantees of individual user's data, even in the presence of unknown auxiliary data about the user (e.g., from other sources such as public databases), and regardless of any unanticipated privacy attacks that may be applied to the data (e.g., inference and linkage attacks [29, 67, 68]).

DP is currently considered as the "gold standard" for privacy-preserving analysis. Both industry [3, 25, 32, 35, 95] and government [22] have deployed DP solutions. There is a rich body of work in this area [29, 100] but the use of DP for software profiling has not been explored sufficiently. The goal of this dissertation is to study the problem of remote software profiling with differential privacy and to provide feasible solutions for practical use. In the following sections, we introduce the overview and outline of this dissertation, followed by its contributions.

1.1 Overview and Outline

This dissertation targets both theoretical and practical challenges of designing and implementing differentially-private software profiling mechanisms. We design various DP algorithms in this context and conduct extensive experiments to evaluate the proposed techniques.

Background on DP. Chapter 2 first introduces the background of DP and the two major models of DP, i.e., centralized model and local model. We argue that the latter is more suitable for the problem of software profiling, as user profiles are collected and stored on local machines. We then discuss the frequency oracle problem, a simpler and easier-to-understand problem than software profiling. This serves as a concrete example to show the basic concepts and principles of DP. Next, two existing DP mechanisms are described. Our solutions in the following chapters are built on top of them.

Event Frequency Profiling. Chapter 3 presents a parameterized randomization approach for traces of run-time events. While prior work considers randomization of a single event [9, 32, 96], we target quantifiable privacy guarantees for entire event traces that occur in software run-time event frequency profiling. In addition, using program analysis techniques, we extract *a priori* knowledge about relationships between elements of the run-time profile and incorporate these relationship in the post-processing step of the DP profiling analysis. In particular, we consider domain-specific constraints of the form "the frequency of x will always be $\leq$ the frequency of y". This produces estimates that are consistent with the structure of the true frequencies and reduces the error of the resulting estimates.

To understand the actual performance of these techniques and to provide guidelines for their deployment in realistic scenarios, we perform a study of method call traces from Android apps and present the results in Chapter 3. Our conclusion from the analysis of experimental results is that well-designed solutions can achieve both high accuracy and principled privacy-by-design for the fundamental problem of event frequency profiling.

Frequency Profiling Under Linear Constraints. Chapter 4 is focused on a different setting of software frequency profiling where the profile at each user is a frequency vector instead of an event trace. We propose an approach which reports the user-level frequency information with the addition of random noise drawn from the Laplace distribution. Our experimental results show that significantly higher accuracy of profiling results can be achieved without reducing privacy protections in any substantial way, compared to the approach from Chapter 3.

In addition to the new randomization, we provide a novel re-definition of the privacy guarantees and the corresponding privacy mechanisms to account for correlations of the frequencies of different run-time execution events. Such correlations are due to the intrinsic static structure of the underlying software system and are commonly observed in software systems. Specifically, we consider those that can be expressed as linear inequalities, e.g., the frequencies of several events will always be greater than or equal to the frequency of another event. We argue that differential privacy protections must account for such relationships; otherwise, a seemingly-strong privacy guarantee is actually weaker than it appears. In particular, we propose a novel linear programming formulation to compute the magnitude of random noise that should be added to achieve meaningful privacy protections under such linear constraints.

Next, we develop an instance of this general machinery for a special but important subclass of constraints of the form “the frequency of x is greater than or equal to the frequency of y”, similar to the ones from Chapter 3. We demonstrate that in this case the magnitude of random noise can be determined efficiently by employing a reachability analysis of a constraint graph that encodes such constraints. To the best of our knowledge, no prior work has incorporated such domain-derived data constraints in the design of a differentially-private analysis. Our experimental analysis shows that the privacy protection can be significantly weakened if the design does not take into account these constraints.

Control-Flow Node Coverage Analysis. Chapter 5 discusses a profiling problem is related to control-flow node coverage, where the type of constraints are with respect to control-flow graph (CFG) nodes. In such data collection, events are represented as graph nodes in CFGs and the edges correspond to the transitions between events. In CFGs, it is often the case that the execution of a node n_2 is caused by the execution of another node n_1 and, furthermore, n_2 is executed *only if* n_1 is executed. When such correlations are in place, hiding the coverage of n_1 independently from any other graph nodes is not enough, because an adversary could infer information about n_1’s coverage from observations about the coverage of n_2.

We first define the space of possible data instances by considering what constitutes run-time graph coverage observed at any particular deployed copy of the software. Next, we define the critical notion of neighbors for a particular covered subgraph. We show that the traditional notion of graph neighbors used in prior DP analyses for graph data is meaningless for control-flow graphs because nodes in such graphs have strong correlations driven by the underlying graph structure [16, 53, 54, 60, 106]. We demonstrate that such relationships are captured by the notion of dominators, which is traditionally used in compiler optimizations. Based on this insight, we propose a new notion of "graph neighbor" and use it to define the privacy guarantees that need to be achieved by any DP solution for this analysis.

We then propose an analysis to achieve these guarantees by randomizing the coverage information. Our randomizer is based on the DP notion of sensitivity, which, intuitively, captures the difference in the output of the analysis performed on two neighboring graphs. We define a new notion of sensitivity for graph data and demonstrate how to compute it efficiently using the dominator tree of the dynamic CFG. Based on these techniques, we introduce four randomizers that achieve DP. First, we describe a baseline randomizer using the worst-case upper bound for sensitivity that achieves the theoretically-optimal worst-case analysis error, but does not provide good accuracy on real data. Next, we introduce stronger bounds on sensitivity by aggregating the local sensitivities from a set of opt-in users. However, some users may suffer from weakened privacy protection during the actual subsequent coverage collection. Therefore, we propose to lower the bound by projecting the covered subgraph of CFG onto a lower-sensitivity representation. Then, we propose to refine the notion of indistinguishability to account for the distance between graph neighbors. Our experimental results show significant error reduction for the latter three approaches compared to the baseline approach.

1.2 Contributions and Impact

This dissertation demonstrates several novel approaches targeting various profiling tasks that provide principled DP guarantees for individual user's profile data, while still allowing developers to learn useful statistical results across the whole user population. We identify the key challenges for designing effective and practical randomization mechanisms, and propose solutions that advance both the theoretical state of art and provide guidelines for future development of privacy-preserving software profiling infrastructures.

Specifically, in Chapter 3, we propose the first DP solution that protects run-time event traces at the user end and allows developers to fine-tune the trade-offs between privacy and accuracy. This dissertation is also the first to incorporate domain-derived constraints on event frequencies in the design of a differentially-private analysis. In Chapter 4, we propose a linear programming formulation to compute the magnitude of noise that should be added under such constraints. Our experimental results have significant implications for the design of any infrastructure for differentially-private remote profiling of deployed software, suggesting that these constraints must be taken into account in the analysis design to provide expected privacy guarantee. Our DP solution for control-flow graph node coverage analysis in Chapter 5 is an example showing that existing privacy-preserving techniques may be unsuited for software analysis problems. It introduces domain-specific knowledge about correlations between CFG nodes to the solution design and implementation.

In summary, these efforts present novel advances in a broad research agenda for privacy-preserving software profiling. The need for such analysis is driven by the fundamental tension between the privacy of software users and the needs of software developers and researchers. Differential privacy is a rigorous tool for exploring such trade-offs. However, we are not aware of any prior efforts to apply this tool in the analysis of deployed software. Ultimately, these and similar efforts contribute to the important area of software user privacy, in a world where privacy protections are becoming essential.

Chapter 2: Background and Terminology

2.1 Differential Privacy

Differential privacy (DP) [28, 29] is a general approach for protection against a wide range of privacy attacks. In such scenarios, there is release of some data and an adversary attempts to learn private individual information from the data. Anonymizing or removing personally-identifiable information cannot guarantee privacy, as demonstrated in prior work [26, 67, 68, 91], because additional data sources can be combined with the anonymized data to uncover sensitive information. A prominent example is work [67] that identified individual records from Netflix's collection of anonymized viewing histories by linking with another movie database. Practical applications have also been demonstrated. DP solutions have been deployed by several companies (e.g., Google [32, 35], Apple [3, 94], and Uber [95]). The U.S. Census Bureau will use DP to protect the results from the 2020 census [22]. Given the rapid emergence of large-scale data analysis and its detrimental effects on privacy, the importance and urgency of such privacy solutions (including DP) will continue to increase in the foreseeable future.

The DP protection is of the following form: by observing the results of a DP analysis, an adversarial entity will not be able to distinguish between the real data that was included in the analysis input, and any "neighboring" data in the universe of possible input data. This is achieved by introducing random noise, and thus the indistinguishability is probabilistic. DP is attractive because it provides a comprehensive and quantifiable notion of privacy. Furthermore, DP analyses may be able to guarantee that they are in compliance with legal requirements for privacy protection [100]. It is important to note that the indistinguishability protection holds even when a privacy adversary has access to additional information outside the scope of the analysis being considered. Intuitively, regardless of how much an adversary can learn from other sources of information, she still cannot determine (with high confidence) the specific private data that was used as input to the DP analysis.

There are two major models for defining DP problems: *centralized model* and *local model*. In the centralized model, the data curator (also referred to as "server") is trusted for collection of data. In the local model, the server is not trusted: raw data that reaches it can be observed by an adversary. For such *locally-differentially-private* (LDP) problems, each user performs local data perturbation via a local randomizer before releasing any information to the server. The LDP model is particularly well suited for remote software profiling. This model provides privacy guarantees to the software user regardless of the unpredictable actions from the software analysis infrastructure and its clients and adversaries. The server maintainers themselves are protected: even if there are malicious employees, security attacks, or subpoenas by law enforcement, the data on the server cannot be used to reliably infer the private data of the software user.

2.2 Frequency Oracle

For a concrete example of a classic DP analysis, we describe the *frequency oracle* problem [9, 32]. The problems considered in this dissertation in the following chapters are generalizations of this exemplar DP analysis.

Consider a finite domain of data items $\mathcal{V}$. In our context this domain contains some program code entities—for example, $\mathcal{V}$ is the set of all methods/functions defined in the program's source code. Suppose there are n individuals participating in the data collection. For convenience of notation, these individuals are identified via integers $i \in \{1,\ldots,n\}$. Each individual has a single data item $v_i \in \mathcal{V}$. The goal of the data collection is to determine, for each element v of the data domain, the frequency of v—that is, the number of individuals i with $v_i = v$. A frequency oracle is an algorithm that provides an estimate of the frequency of v for any $v \in \mathcal{V}$.

A *locally-differentially-private* (LDP) frequency oracle employs a randomization algorithm $R : \mathcal{V} \rightarrow \mathcal{Z}$, often referred to as the *local randomizer*. The role of R is to introduce random noise to the local information of individual i. Specifically, instead of reporting v_i to the analysis server, the DP frequency oracle algorithm reports $R(v_i)$. The server collects all such randomized reports over the n individuals and uses them to compute the frequency estimates for all v.

Randomizer R must ensure the indistinguishability property, parameterized by the so-called privacy loss parameter $\varepsilon \geq 0$. Higher values of ε imply higher risk to an individual's privacy. Consider any value $z \in \mathcal{Z}$ that could be produced by R. For any $v \in \mathcal{V}$ and $v' \in \mathcal{V}$, from the observation of z it should be impossible to determine, with high confidence, whether the input of R was v or v'. Specifically, $\Pr[R(v) = z]$ and $\Pr[R(v') = z]$ should not differ by more than a factor of e^{ε} where $\Pr[\ldots]$ denotes the probability of an event. We formally define this essential property as follows:

Definition 2.1 (ε-indistinguishability)**.** *Randomizer* $R : \mathcal{V} \rightarrow \mathcal{Z}$ *achieves* ε-indistinguishability *if for any* $v, v' \in \mathcal{V}$ *and for any* $z \in \mathcal{Z}$*, we have*

$$\frac{\Pr[R(v) = z]}{\Pr[R(v') = z]} \leq e^{\varepsilon}$$

This property should be interpreted as follows: even if an outside entity knows the complete details of how R is defined, by observing the output z of R this entity is not able to conclude, with high probability, that the real data of individual i was a particular v_i as opposed to any other element of $\mathcal{V}$. The strength of this protection depends on ε: small values result in strong protection, but also necessitate the introduction of more noise, which affects the accuracy of the frequency estimates.

More generally, in various LDP analyses, the indistinguishability is between “neighbors” in the data space. In this dissertation we apply this notion to event traces (Chapter 3), frequency vectors (Chapter 4), and control-flow graphs (Chapter 5).

2.3 Differential Privacy Techniques

There exist many designs and implementations of the local randomizer that achieves differential privacy under various situations [3, 8, 9, 17, 27, 29, 30, 32, 81, 96]. Here we introduce two well-studied techniques, upon which the approaches proposed in the following chapters are built.

Randomized Response. Randomized response is a classic technique that has been used in social sciences to eliminate evasive answer bias and to gather sensitive data [99] (e.g., about illegal or embarrassing behaviors). In its simplest manifestation, for a single bit input, the local randomizer at each user reports the true value with some probability q (derived from ε) and the flip of the true value with probability $1 - q$. After the collection over all individuals is completed, a post-processing step scales back the population-wide frequencies to account for the randomization effects. This simple randomizer satisfies local differential privacy with $\varepsilon = \ln \frac{q}{1-q}$.

The Laplace Mechanism. The Laplace mechanism [29, 30] is designed to protect numeric values by applying noise drawn from the Laplace distribution to the output of numeric functions. The Laplace probability distribution $\mathrm{Lap}(b)$, parameterized by a scale parameter $b > 0$ and centered at 0, is defined by the probability density function $p(y|b) = \frac{1}{2b}\exp(-\frac{|y|}{b})$. Consider any numeric query function Q that maps its input to m real numbers. A randomizer R for Q based on the Laplace mechanism draws independently random noise $Y_j \sim \mathrm{Lap}(\frac{\Delta Q}{\varepsilon})$ for each $j \in \{1,\ldots,m\}$ and adds $\begin{pmatrix} Y_1 & \ldots & Y_m \end{pmatrix}$ to the output of Q. Here $\Delta Q = \max_{x,y:\|x-y\|_1 \leq 1} \|Q(x) - Q(y)\|_1$ is the sensitivity of Q, where $\|\ldots\|_1$ denotes the L_1 norm. This definition of R achieves ε-differential privacy.

Chapter 3: Event Frequency Profiling with Differential Privacy

In this chapter, we utilize differential privacy to realize privacy-preserving event frequency profiling. In particular, we define a parameterized randomization approach for *traces* of run-time events. First, we define a notion of distance between event traces and the corresponding distance-based privacy properties, which enables tunable trade-offs between privacy and accuracy. To achieve such properties, we propose new randomization which, unlike prior expensive event-by-event randomization [32, 103], applies efficient randomization on the total trace frequencies. We also incorporate additional consistency constraints that reflect domain-specific considerations. For example, the (normalized) estimated frequencies are non-negative and add up to 1. In addition, we consider constraints of the form "the frequency of x will always be $\leq$ the frequency of y in any run-time event trace". Such *run-time* constraints often exist due to the *static* properties of program code. We embed both categories of constraints in a quadratic programming optimization problem, which we then use to produce more accurate frequency estimates.

To provide insights into such trade-offs, we perform a study of method call traces from Android apps. Our results clearly quantify the inherent tension between privacy and accuracy. Specifically, they point out that privacy protections for traces that are "far apart" come at the expense of significantly reduced accuracy. However, a more detailed analysis of these results reveals that for high-frequency events—in our studies, for "hot" methods—the accuracy of frequency estimates is actually quite good. Overall, the results suggest that well-designed solutions can achieve both high accuracy and principled privacy-by-design for the fundamental problem of event frequency profiling.

3.1 Problem and Motivation

Consider a software system deployed locally on the machines of n users. We will use $i \in \{1,\ldots,n\}$ to denote these users. Suppose that software developers are interested in the execution frequency of certain run-time events—for example, events of the form "v was executed", where v is a method/function in the software code. Let $\mathcal{V}$ denote the set of all such events. In our setup $\mathcal{V}$ is decided by the software developers before the software is deployed, and some run-time mechanism (e.g., instrumentation) is used to observe occurrences of such events while the deployed software is running. For each software user i, the execution of that user's deployed software instance produces a trace of events $t_i = v^i_1,\ldots,v^i_k$ that are observed and recorded by the analysis infrastructure. Denote the frequency of each $v \in \mathcal{V}$ in this trace as $f_i(v)$. Without DP, these local frequencies are simply reported to the remote analysis server, which computes and reports to the software developers a global frequency $F(v) = \sum_i f_i(v)$ for each v.

This general problem statement captures a wide range of classic profiling problems, for example, node/edge profiling at various levels of granularity. However, the collection and reporting of this "raw" data raises concerns about the privacy of software users. First, the events themselves may convey sensitive information: for example, the frequency of calls to functions to log into a remote server, to connect to a VPN, or to change a password. Second, such information can be used to classify user's interest and habits, which could later be (mis)used for behavior analytics or targeted advertisement [101]. Finally, and very importantly, the rapidly increasing power of data mining and machine learning, together with the dramatic increase of user-specific data available from various sources, makes it possible to make increasingly-powerful inferences about an individual from the various data streams she produces in her daily life. Even if certain categories of data gathering appear to be harmless on their own, it is hard to predict how they would interact with future unanticipated additional data sources and analyses. Not surprisingly, both society in general and legislative bodies in particular are paying close attention to these privacy issues. From the technical perspective, designing privacy-preserving analyses and "future-proofing" them against unpredictable privacy attacks is an important and challenging problem.

As discussed earlier, *differential privacy* (DP) is a principled framework to address such privacy concerns and to provide privacy guarantees against both known and unknown (i.e., future) data analyses. For our problem, rather than reporting local frequencies $f_i(v)$ to the remote server, the analysis reports a randomized version of it, derived in a way that ensures DP properties. While per-user information is now noisy, the global frequency estimates inferred by the analysis server are accurate estimates of the true global frequencies $F(v)$.

Problem Statement. Consider a software user i and her trace of events $t_i = v_1^i, \dots, v_k^i$. Without loss of generality, assume that k is decided before software deployment and is the same for all i. A way to represent the local frequency information is as vector $\boldsymbol{f}_i$ of $|\mathcal{V}|$ integers—that is, as a histogram with $|\mathcal{V}|$ bins, where each bin is the frequency of some v, and the sum of bin values is k. Given the local trace t_i and its frequency vector $\boldsymbol{f}_i$, a non-private solution reports $\boldsymbol{f}_i$ to the server, where a vector $\boldsymbol{F} = \sum_i \boldsymbol{f}_i$ of true global frequencies is reported to the developers/analysts.

A DP solution applies a randomized algorithm R to the local trace t_i, as described in Section 3.3.1. We will use $R(\boldsymbol{f}_i)$ as shorthand for the frequency vector of this randomized trace. $R(\boldsymbol{f}_i)$ is a vector of $|\mathcal{V}|$ integers, but they do not have to add up to k. This noisy vector is reported to the server and used, together with similar vectors from all other software users, to compute a vector $\hat{\boldsymbol{F}}$ of global frequency estimates (described in Section 3.3.2). The randomizer R is the same for all software users and is fully designed by the developer before software deployment. It is assumed that the details of R are known by any potentially-adversarial entities. Broadly, such entities include anyone who could observe the vector $R(\boldsymbol{f}_i)$ reported to the server (and, in the extreme, the server is also considered to be potentially adversarial).

3.2 The Differential Privacy Guarantee

The privacy guarantees we define are based on the notion of indistinguishability outlined earlier in Chapter 2. Specifically, given some privacy loss parameter ε, consider a pair of traces t and t' and their frequency vectors $\boldsymbol{f}$ and $\boldsymbol{f}'$. Then for any vector $\boldsymbol{z}$, we want to ensure that the probabilities $\Pr[R(\boldsymbol{f}) = \boldsymbol{z}]$ and $\Pr[R(\boldsymbol{f}') = \boldsymbol{z}]$ do not differ by more than a factor of e^{ε}. As we demonstrate in our experiments, it is essential to decide for which pairs of t and t' such protection should be achieved. In particular, we show that if indistinguishability is desired for *all possible* pairs of traces, too much noise needs to be added and the resulting accuracy of estimates is very low.

To capture this essential trade-off between privacy and accuracy, we define restricted indistinguishability which applies to pairs of traces that are "close" to each other. This technique is motivated by existing work on the theoretical properties of distance-based indistinguishability [15]. Consider two traces t and t', each containing k events. Our definition is based on a threshold τ of the difference between the traces: specifically, the number of trace positions $1 \leq j \leq k$ such that event $t[j]$ is different from event $t'[j]$. We define a distance between traces $d(t,t')$ as the number of such j.

Definition 3.1 (ε-τ-differentially private). *Randomizer R is* ε-τ-differentially private *if $\forall t,t',\boldsymbol{z}$, it is true that $d(t,t') \leq \tau$ implies*

$$\frac{\Pr[R(\boldsymbol{f}) = \boldsymbol{z}]}{\Pr[R(\boldsymbol{f}') = \boldsymbol{z}]} \leq e^{\varepsilon}$$

If a pair's difference exceeds the threshold τ, the randomization still provides privacy protection, but with a weakened (i.e., larger) value of ε, scaled by the ratio between the trace distance and τ. When $\tau = k$, indistinguishability holds for all possible pairs of traces. By varying the value of τ, we can explore trade-offs between privacy and accuracy. For real-world deployment of DP solutions in remote software profiling, such trade-offs are essential. Later we also discuss the practical considerations for choosing the threshold τ.

Note that the above definition also implies a form of indistinguishability for individual events in a trace: for any v, if the real frequency is $f(v)$, an observer cannot distinguish with high probability $f(v)$ from $f(v) - \tau$ and $f(v) + \tau$. Though she can still draw conclusions about v, the strength of these conclusions will be weakened based on the threshold τ. Still, an adversary can make various inferences from the randomized data: e.g., "with high probability, event v was more frequent than event v'". In future work, it would be interesting to consider other notions of distance and indistinguishability that provide protection against such inferences.

3.2.1 Example

To illustrate the meaning behind this definition, we use a simple example. Suppose $\mathcal{V} = \{a,b\}$ and $k = 5$. There are $2^5 = 32$ possible traces and six unique frequency vectors:

$\begin{pmatrix}5 & 0\end{pmatrix}, \begin{pmatrix}4 & 1\end{pmatrix}, \ldots, \begin{pmatrix}0 & 5\end{pmatrix}$ where the first element is the frequency of a and the second one is the frequency of b.

Next, we outline a possible definition of the randomizer; a detailed description will be provided later in Section 3.3.1. Suppose we choose $\varepsilon = \ln 9$, as done in prior work [32], and $\tau = 1$. The randomizer uses a probability $p = e^{\frac{\varepsilon}{2\tau}}/(1+e^{\frac{\varepsilon}{2\tau}})$ to randomize each event in the trace. In this example, $e^{\frac{\varepsilon}{2\tau}} = 3$ and thus $p = 0.75$. For this R, when event a is observed, the following two rules are applied. First, with probability $p = 0.75$, a's count is incremented (and thus, with probability 0.25 this observation of a does not modify the count of a). In addition, for this observation of a, b's count is incremented with probability $1 - p = 0.25$ (and, with probability 0.75, this observation of a does not modify the count for b). Similar processing would be applied when event b is observed. As a result, the final noisy histogram could contain anywhere between 0 and $2 \times k = 10$ counts.

Suppose that $\begin{pmatrix}4 & 2\end{pmatrix}$ is produced by R and is observed by a potentially-adversarial entity. What information can be inferred from this observation, assuming that this entity knows the details of R (including ε and τ)? The table below summarizes the probabilities for the 6 possible frequency vectors $\boldsymbol{f}$, for $\tau = 1$ as well as for $\tau = 2$. Note that each value of $\boldsymbol{f}$ could be produced by several different traces t; the shown probabilities for that $\boldsymbol{f}$ apply for each such trace.

$\boldsymbol{f}$	$\Pr[R(\boldsymbol{f}) = \begin{pmatrix}4 & 2\end{pmatrix}]$	
	$\tau = 1$	$\tau = 2$
$\begin{pmatrix}5 & 0\end{pmatrix}$	0.1043	0.1009
$\begin{pmatrix}4 & 1\end{pmatrix}$	0.1265	0.0848
$\begin{pmatrix}3 & 2\end{pmatrix}$	0.0746	0.0606
$\begin{pmatrix}2 & 3\end{pmatrix}$	0.0247	0.0378
$\begin{pmatrix}1 & 4\end{pmatrix}$	0.0061	0.0214
$\begin{pmatrix}0 & 5\end{pmatrix}$	0.0013	0.0112

A way to carry out the calculation of these probabilities is the following: if there were x real occurrences of a, the probability that they contributed exactly y increases to the count for a is $\binom{x}{y}p^y(1-p)^{x-y}$, since this is a binomial experiment with x independent trials, each with success probability p. The probabilities in the table are determined by considering all possible values for x and y, as well as the possible contributions of the $k-x$ events where b was observed.

When $\tau = 1$, for any traces t and t' with $d(t,t') \leq 1$, the corresponding frequency vectors $\boldsymbol{f}$ and $\boldsymbol{f}'$ can differ by at most one count—e.g., they could be $\begin{pmatrix}5 & 0\end{pmatrix}$ and $\begin{pmatrix}4 & 1\end{pmatrix}$. For any such pair, the ratio of the corresponding probabilities is bounded by e^{ε}. In this sense, differential privacy makes it difficult to distinguish between these possible traces for anyone who has observed output $\begin{pmatrix}4 & 2\end{pmatrix}$. However, this does not hold for all possible pairs of inputs. For example, the ratio between the highest and the lowest probability shown in the table for $\tau = 1$ is 98.28 (while for $\tau = 2$ this ratio is 9, as discussed below).

When $\tau = 2$, the privacy protection is stronger: for any pair of traces with $d(t,t') \leq 2$, the ratio of the corresponding probabilities is bounded by e^{ε}. There are benefits even for traces that are "further apart" than τ—e.g., traces with frequency vectors $\begin{pmatrix}5 & 0\end{pmatrix}$ and $\begin{pmatrix}0 & 5\end{pmatrix}$. In our example, the largest ratio of probabilities shown in the table for $\tau = 2$ is 9.

3.2.2 Privacy for Presence/Absence in Trace

One important implication of Definition 3.1 is the following. Suppose that for some v we have $\leq \tau$ occurrences in a trace t. There are many possible traces t' in which v does not occur at all and $d(t,t') \leq \tau$. If we employ an ε-τ-DP scheme, an adversary will not be able to distinguish between t and t'. In other words, she will not be able to conclude that v occurred at all, since it will not be possible to distinguish, with high probability, the case when v occurred $f(v)$ times from the case when v occurred 0 times. Such privacy protection may be important for infrequently-executed but sensitive software components: e.g., code to change a password. In general, the mere presence/absence of any v with $f(v) \leq \tau$ in the run-time trace is obfuscated, in a probabilistic sense as defined by the ratio bound e^{ε}. For the example from above, when $\tau = 2$, the presence/absence of any a events is obfuscated when the actual trace has frequencies $\begin{pmatrix}0 & 5\end{pmatrix}$, $\begin{pmatrix}1 & 4\end{pmatrix}$, or $\begin{pmatrix}2 & 3\end{pmatrix}$ regardless of what is the output of the randomizer.

One extreme case of such obfuscation is to zero out the frequency when $f(v) \leq \tau$. However, by removing this local information, aggregate information about v is also discarded. Instead, an ε-τ-DP scheme preserves v's frequency distribution across the entire population, in addition to providing strong protection for its presence/absence.

3.3 Differentially-Private Event Frequency Profiling

3.3.1 Efficient Randomization

To design an ε-τ-DP randomizer, we use an approach that is a generalization of existing techniques for the frequency oracle problem introduced in Chapter 2. Specifically, we define a probability

$$p = \frac{e^{\frac{\varepsilon}{2\tau}}}{1 + e^{\frac{\varepsilon}{2\tau}}} \tag{3.1}$$

For each event v^i_j in the trace $v^i_1, \ldots, v^i_k$ of user i, the randomizer will increment the count for v^i_j with probability p (and will keep it the same with probability $1 - p$). In addition, when v^i_j is observed, each $v' \in \mathcal{V} \setminus \{v^i_j\}$ is subjected to the following processing: the randomizer increments the count for v' with probability $1 - p$ and keeps it the same with probability p. To achieve such randomization, an instrumentation layer in the software can observe each run-time occurrence of an event v and immediately generate the corresponding contributions to the collected profile. It can be shown that the cumulative result of these contributions indeed satisfies the property from Definition 3.1. The proof is presented as follows:

Proof. Given any two traces t and t' such that $d(t,t') \leq \tau$ and their corresponding frequency vectors $\boldsymbol{f}$ and $\boldsymbol{f}'$. The ratio between $\Pr[R(\boldsymbol{f}) = \boldsymbol{z}]$ and $\Pr[R(\boldsymbol{f}') = \boldsymbol{z}]$ is bounded from above by

$$\left[\left(\frac{e^{\frac{\varepsilon}{2\tau}}}{1+e^{\frac{\varepsilon}{2\tau}}}\right)\left(\frac{e^{\frac{\varepsilon}{2\tau}}}{1+e^{\frac{\varepsilon}{2\tau}}}\right)^{|\mathcal{V}|-1}\right]^{\tau} \Bigg/ \left[\left(\frac{1}{1+e^{\frac{\varepsilon}{2\tau}}}\right)^{2}\left(\frac{e^{\frac{\varepsilon}{2\tau}}}{1+e^{\frac{\varepsilon}{2\tau}}}\right)^{|\mathcal{V}|-2}\right]^{\tau} = e^{\varepsilon}$$

Similarly, this ratio is bounded by $e^{-\varepsilon}$ from below. Thus R satisfies ε-τ-DP. □

The randomization outlined above has a significant limitation: the cost of applying the randomizer could be high. For each of the k events in the trace, each element of $\mathcal{V}$ has to be randomized independently. In practical scenarios, k could contain many thousands of events, and $\mathcal{V}$ could also contains many thousands of elements. For example, in our experiments $\mathcal{V}$ contained all methods in the code of a given Android app, and its size was typically several thousand methods. In fact, several of our experiments with this naive randomizer could not complete within a reasonable time period. To address this limitation, we redefine the randomizer as operating over the entire local frequency vector rather than on individual events in the trace. This allows us to reduce the cost of R from $O(k|\mathcal{V}|)$ to $O(|\mathcal{V}|)$.

This efficient approach works as follows: during run-time execution, the true frequency vector $\boldsymbol{f}$ is constructed but randomization is not applied. After the counts for all k events are accumulated, the resulting vector is randomized independently for each v to obtain a new vector $R(\boldsymbol{f})$. Consider some v and the number of its occurrences $f(v)$. Each of those occurrences would have contributed to v's count in $R(\boldsymbol{f})$ with probability p. The number of such contributions is a random variable with binomial distribution. Recall that binomial distribution gives the probability of getting exactly m successes in n independent trials, where each trial succeeds with probability p. The probability mass function is $q(n,m,p) = \binom{n}{m} p^m (1-p)^{n-m}$. Given $n = f(v)$ and p, we can draw a random value m_1

based on this distribution. We also need to account for contributions to v's count in $R(\boldsymbol{f})$ that are due to the $k - f(v)$ events in which v was not observed. We can draw another random value m_2 from the binomial distribution $q(k - f(v), m, 1-p)$. Then the frequency of v in $R(\boldsymbol{f})$ is set to be $m_1 + m_2$.

For efficiency, instead of using the (discrete) binomial distribution, we use the (continuous) normal distribution. It is well known that the binomial distribution can be approximated using the normal distribution. To draw a random value from the binomial distribution for a given n and p, we draw a random value from the normal distribution with mean np and variance $np(1-p)$. The resulting real number is then rounded to the nearest integer in the range $[0, n]$.

3.3.2 Server-Side Computation of Estimates

Given the reported local randomized frequencies $R(\boldsymbol{f}_i)$ from each user i, the remote software analysis server first computes a vector $\hat{\boldsymbol{H}} = \sum_i R(\boldsymbol{f}_i)$. Due to the randomization, the value $\hat{H}(v)$ cannot directly be used as an estimate of the true global frequency $F(v) = \sum_i f_i(v)$. To compute such an estimate $\hat{F}(v)$, one can consider the expected value of $\hat{H}(v)$. This expected value has two components: (1) each of the $F(v)$ instances of v across all users have been included in $\hat{H}(v)$ with probability p; (2) each of the $nk - F(v)$ instances of other events have contributed to $\hat{H}(v)$ with probability $1-p$. Given this observation, one can define the estimate

$$\hat{F}(v) = \frac{(e^{\frac{\varepsilon}{2\tau}} + 1)\hat{H}(v) - nk}{e^{\frac{\varepsilon}{2\tau}} - 1} \tag{3.2}$$

The expected value of $\hat{F}(v)$ is $F(v)$. After this computation, $\hat{F}(v)$ are normalized by the total number of events nk.

After this processing, we have an estimate $\hat{F}(v)$ for each v. However, these estimates do not satisfy two categories of *consistency constraints*. First, there is no guarantee that $\hat{F}(v) \geq 0$ and $\sum_v \hat{F}(v) = 1$. Second, it is often the case that the structure of the software imposes additional constraints on *any* run-time set of frequencies. One extremely simplified example is the following: suppose that the body of a method m contains only a single `if` statement, inside which there is call to another method m', and, further, this is the only call to m' in the entire program. We can assert that for the true global frequencies, $F(m') \leq F(m)$. However, it is not necessarily the case that in the computed estimates we have $\hat{F}(m') \leq \hat{F}(m)$. More generally, we would like to consider static code structures that imply inequality constraints of the form $F(v) \leq F(v')$ for some pairs of events v and v', and to make the final reported estimates consistent with such constraints. The next subsection provides details on the particular code properties we consider and on the static program analysis used to infer them.

We would like to compute estimates that satisfy these two categories of consistency constraints. Some prior work [97] has also considered the consistency constraint that estimates are non-negative and add up to 1. Unlike this prior work, we also target consistency constraints derived via static analysis, and employ a novel quadratic programming formulation. Our goal is to minimize the squares of the differences between $\hat{F}(v)$ and (unknown) estimates $x(v)$ that satisfy the consistency constraints. The specific optimization problem we define has the following form:

$$\begin{aligned} \min_{x(v)\in\mathbb{R}} \quad & \textstyle\sum_v \left(x(v) - \hat{F}(v)\right)^2 \\ \text{subject to} \quad & x(v) \geq 0 \\ & \textstyle\sum_v x(v) = 1 \\ & x(v) \leq x(v') \end{aligned}$$

The last component represents a set of constraints that are based on the relationships inferred by the static analysis described in the next subsection. This is an instance of a linearly constrained quadratic optimization problem. A variety of solvers are available for such problems; our implementation uses the solver available in MATLAB. Let $\bar{F}(v)$ denote the value for $x(v)$ computed by the solver. This value $\bar{F}(v)$ is reported by the server as the final estimate of the (normalized) global frequency of event v.

One relevant observation is that such constraints are public knowledge since they can be extracted from the app code via static analysis. Thus, an adversary could observe the results of applying a local randomizer to some user's data, and then utilize similar post-processing based on quadratic programming to enforce the constraints on these observations. However, since DP is immune to post-processing [29], the DP guarantee still holds for the resulting estimates.

3.3.3 Static Analysis of Call Frequencies

Frequency vectors have a certain structure that imposes constraints on the relationships between vector elements. Below we illustrate such constraints for the frequency of method calls in Android apps. However, similar machinery could be easily designed for other use cases—for example, profiling of function calls in C programs and method calls in

Java/C++/C# programs, or general node/edge profiling in control-flow graphs [7]. The constraints are of the form $f(m) \leq f(m')$ where m and m' are methods in the app code.

A method m in an Android app could be called in two manners. Inside the app code, there could be a call site that invokes m. A second possibility is that m is invoked by the Android platform code. This is the case, for example, for methods that provide event handlers for GUI events (e.g., `onClick` callbacks for click events) or for window lifecycle events (e.g., `onCreate` callbacks for window creation events). As described below, in some cases constraints can be inferred only for methods that *cannot* be invoked by unknown code from the Android platform. We ensure this by only considering methods that do not override, directly or transitively, any method declared in an Android class or interface. Note that similar considerations would apply in general for object-oriented languages such as Java, C++, and C#, where application methods override library methods, and thus unknown library code invokes application methods. Callbacks could also occur in C code: a typical example is the `qsort` library function, which takes as input a function pointer to a comparator function, and therefore static constraints on the number of comparator invocations cannot be established.

Algorithm 3.1 describes at a high level our static analysis for inferring that the call frequency of a method is always not greater than the call frequency of another method. At line 3, if *cs* is a virtual call site, we determine all possible target methods by considering the class hierarchy of the app code and the Android platform code. If m' is the only possible target, we need to determine that m' will be executed at least once. This is done via dominator analysis of the control-flow graph (CFG) of the caller m. If the call site dominates all exit nodes of the CFG (i.e., all `return` and uncaught `throw` statements), it is guaranteed that the execution of m triggered at least one invocation of m'.

Algorithm 3.1: Find $f(m) \leq f(m')$

```
1 foreach $m \in \mathcal{V}$ do
2     foreach call site cs in m do
3         foreach target $m'$ of cs do
4             if $m'$ is the only target of cs and cs dominates the exits of m then
5                 Record $f(m) \leq f(m')$
6             if $m'$ does not override any framework method and there are no other
              calls to $m'$ in app and cs is not in loops then
7                 Record $f(m') \leq f(m)$
```

A second case implying an inequality constraint is as follows (lines 6–7). Suppose m' is one of several possible target methods at a call site, and this is the only call site in the entire app that invokes m'. Further, suppose that m' cannot be called from the Android platform code, as discussed earlier. Then any invocation of m' must occur as part of an invocation of m. If, in addition, we can establish that m' is not located inside any loops in the CFG of m, this is enough to conclude that $f(m') \leq f(m)$.

To implement this static analysis for Android apps, we use the Soot analysis toolkit [86] to create an intermediate representation of the app's bytecode. For each app method we consider its CFG and the call sites inside it. We record all call sites and their corresponding dispatch targets utilizing class hierarchy analysis. To determine whether a call site dominates the exits of a method m, we perform reachability analysis in m's CFG, starting from the entry node and stopping the traversal at the call site. At the end of the traversal, we determine whether any of m's exits is reached. To decide whether a call site is in loops, we find all natural loops in the CFG using depth-first search to identify back edges.

App	Stmts	$\lvert\mathcal{V}\rvert$	$\leq$ Pairs	Time (s)
barometer	660776	2237	2053	20.23
bible	832654	5340	3819	30.47
dpm	1505454	1362	1127	55.05
drumpads	979900	1903	1672	16.10
equibase	671692	1975	1720	28.64
localtv	1128876	3055	3178	41.53
loctracker	646698	837	540	27.85
mitula	783383	7172	7856	36.78
moonphases	478113	716	584	20.12
parking	482388	1649	1342	21.01
parrot	629429	7433	8000	48.43
post	832654	5340	3819	30.06
quicknews	832654	5340	3819	30.50
speedlogic	308102	265	239	14.27
vidanta	779294	9242	6824	33.37

Table 3.1: Benchmarks.

3.4 Evaluation

3.4.1 Data Collection

To empirically evaluate the proposed techniques, we conducted method frequency profiling for Android apps. The events in this case are method calls. We used 15 Android applications that have been used in other studies [103]. We then applied the Soot analysis toolkit [86] to determine the set $\mathcal{V}$ of methods in each app. Table 3.1 describes the characteristics of these benchmarks. Column "Stmts" lists the numbers of statements in Soot's Jimple IR. The size of $\mathcal{V}$ for each app is shown in column "$\lvert\mathcal{V}\rvert$". We excluded several third-party libraries from this count (e.g., `butterknife` and `okhttp`).

Next, we utilized the Monkey tool for random GUI testing [40] to send GUI events to the apps in order to simulate user interactions. For each benchmark we simulated 1000 independent executions by running Monkey with 1000 different random seeds for the GUI event sequence generation. Before each execution, we created a fresh Android emulator to avoid unintended configurations from previous runs. We recorded every method call during each execution, using instrumentation at the entry of the corresponding method, until $5 \times |\mathcal{V}|$ method invocations were observed. As a result, we obtained 1000 traces each of which contained $k = 5 \times |\mathcal{V}|$ method call events. From these traces, local frequency vectors $\boldsymbol{f}_i$ for $1 \leq i \leq 1000$ were constructed for each app. All frequency vectors, as well as the code for static analysis, randomization and post-processing, are available at `https://presto-osu.github.io/cc20`.

3.4.2 Implementation

3.4.2.1 Static Analysis

Our implementation of the static analysis of inequality constraints for method frequencies was outlined in Section 3.3.3. Column "$\leq$ Pairs" in Table 3.1 shows the number of pairs (m, m') such that $f(m) \leq f(m')$ was inferred by this analysis. The running time of the static analysis is listed in column "Time (s)", for a machine with Xeon E5 2.2GHz and 64GB RAM. The cost of the static analysis is 3.93 seconds per 100K Jimple statements, on average across all apps. This cost is negligible for all practical purposes, since it will be incurred once by the software analysis server.

3.4.2.2 Client Side

Recall from Section 3.3.1 that we use normal distribution to approximate binomial distribution, in order to achieve efficient randomization. We utilize Java's `Random` class to draw random values with normal distribution. We have observed that this implementation yields an accurate approximation of a binomial distribution. Since Java is one of the officially supported languages for Android, it is trivial to adopt this implementation of the randomizer to existing apps. For convenience of experimentation, all randomization in our experiments is performed under an offline setting—that is, each frequency vector is incrementally collected during app execution, but the resulting $\boldsymbol{f}_i$ is then randomized separately from this execution. This enables us to run multiple trials for each experiment, in order to study the reported metrics under many instances of the random values drawn by local randomizers on the same input $\boldsymbol{f}_i$ frequency vectors.

3.4.2.3 Server Side

After receiving the randomized vectors from the n clients, the server first generates an aggregated vector $\hat{\boldsymbol{F}}$ consisting of the estimates of frequencies after post-processing, as discussed in Section 3.3.2. Then, for the quadratic optimization problem, it invokes the quadratic programming solver in MATLAB's Optimization Toolbox [62]. The cost for solving the optimization problem depends on $|\mathcal{V}|$ and the number of inequality constraints. For example, in our experiments, it takes about 30 seconds for the vidanta app, which has the largest $|\mathcal{V}|$ and the third largest number of constraints across all apps. The solver rarely runs for more than 5 seconds for smaller apps such as barometer.

3.4.3 Accuracy of Estimates

We evaluate the proposed techniques by varying the threshold τ for the difference between two traces and the privacy loss parameter ε. In particular, we consider $\varepsilon \in \{\ln 9, \ln 49\}$ and $\tau \in \{10^0, 10^1, 10^2, k\}$. The values of ε are the same as those used in prior work [32].

All ground-truth frequencies $F(v)$ are normalized by nk so that $\sum_v F(v) = 1$. Recall that the estimates $\bar{F}(v)$ produced by the quadratic programming optimization also have a sum

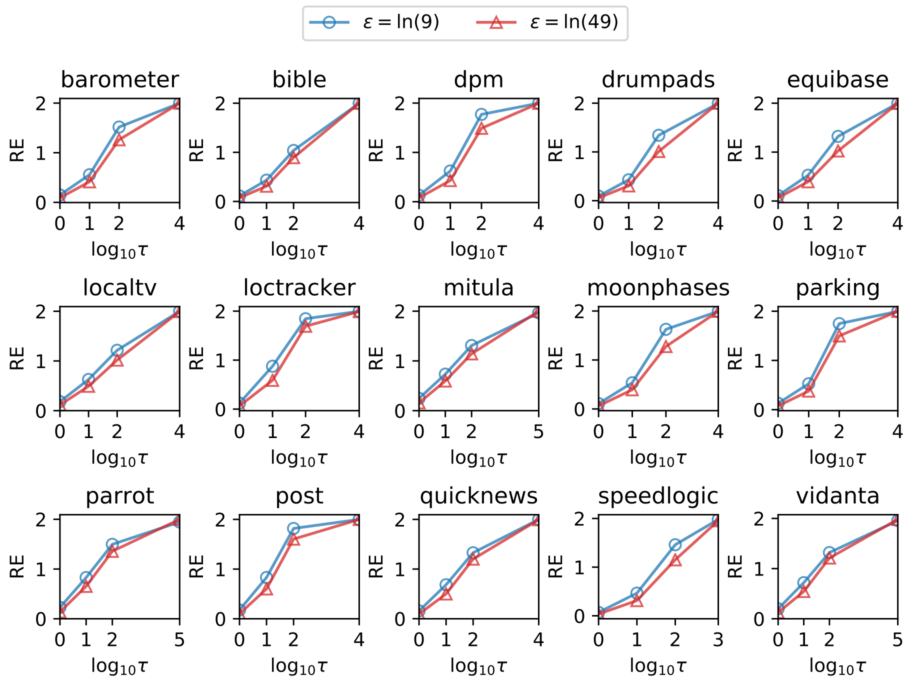


Figure 3.1: Relative error of estimates.

of 1. For each app, we run 100 independent experiments and report the mean under each combination of τ and ε. In all experiments, the resulting standard deviations are typically negligible and thus are not presented.

We use *relative error* (RE) as a metric to evaluate the accuracy of the estimates. This metric measures the overall difference between the estimated frequencies and the actual frequencies. More specifically, given a set $\mathcal{D}$ of methods, for each $v \in \mathcal{D}$ we compute the estimated and ground-truth frequency, calculate and sum their differences, and normalize by the sum by the ground-truth total frequency:

$$\mathrm{RE} = \frac{\sum_{v \in \mathcal{D}} |F(v) - \bar{F}(v)|}{\sum_{v \in \mathcal{D}} F(v)} \tag{3.3}$$

In the case when $\mathcal{D} = \mathcal{V}$ (i.e., the metric is computed for the entire set of methods), the denominator is 1. Later we discuss additional results where $\mathcal{D} \subset \mathcal{V}$. Smaller values for RE means higher accuracy.

Figure 3.1 shows RE values of each app. Recall from Chapter 2 that higher values of ε indicate weaker privacy guarantees. Consider, for example, the speedlogic app. For each value of τ, we can observe that $\varepsilon = \ln 49$ yields less RE compared to $\varepsilon = \ln 9$, and hence provides better accuracy. For instance, when $\tau = 1$, the relative error $\mathrm{RE}_{\ln 9} = 0.067$ is nearly twice as large as $\mathrm{RE}_{\ln 49} = 0.036$. We also tested other values for ε and had similar observations. This conclusion also holds for the other apps, as shown in Figure 3.1.

Next, consider the effects of choosing τ. From Figure 3.1, one can observe that higher values of τ generate less accurate estimates for all apps as they produce more RE. Intuitively, the randomizer needs to introduce more noise (and thus more error) when τ grows in order to "hide" the τ different events between two traces. When $\tau = k$, which provides the strongest privacy protection that guarantees the indistinguishability of any pair of traces, RE reaches its worst-case value of 2. To further investigate the cause of the inaccuracy, for each app, we examined the difference between the ground-truth frequency $\boldsymbol{F}$ and its estimate $\bar{\boldsymbol{F}}$ per method. We observed that (1) a small number of "hot" methods account for the majority of event occurrences, (2) the frequency estimates for these methods are significantly more accurate than the accuracy presented in Figure 3.1, and (3) the overall RE values are large because of the errors contributed by the large number of infrequently-executed methods.

To illustrate these observations, consider again the speedlogic app for $\varepsilon = \ln 9$. Figure 3.2 shows the values of $\boldsymbol{F}$ and $\bar{\boldsymbol{F}}$, as well as $\hat{\boldsymbol{F}}$ which will be discussed shortly, for the 20 most frequently-executed methods in this app. A small number of methods contribute the majority of events in the traces. For example, about 20% of the method calls are to a callback method

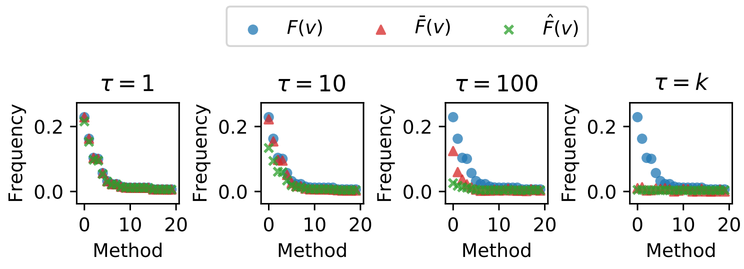


Figure 3.2: Ground-truth $\boldsymbol{F}$ and estimates $\bar{\boldsymbol{F}}$ and $\hat{\boldsymbol{F}}$ of 20 most frequently-executed methods in the speedlogic app with $\varepsilon = \ln 9$.

that is invoked whenever there is new data from the accelerometer sensor. For these hot methods, the estimation errors are small when $\tau \leq 100$. When τ grows to k, the estimates are not useful since the noise introduced by randomization overwhelms the actual frequency. Another observation we made was that the infrequently-executed methods usually have significantly less-accurate estimates, and since the number of such methods is very large, their accumulated error is an essential source of RE.

3.4.4 Estimates for Hot Methods

To quantify the observations described above, we compute RE for hot methods only. Following prior work [107], the set of hot methods is defined based on a threshold $0 \leq \ell \leq 1$. Given a frequency vector $\boldsymbol{f}$, the set of hot methods in $\boldsymbol{f}$ is defined by $\text{hot}(\boldsymbol{f}, \ell) = \{v \mid f(v) \geq \ell \times \max_v f(v)\}$. That is, hot methods are ones with frequencies close to the frequency of the hottest method [107]. Next, we follow the procedure in Section 3.4.3 to compute RE for hot methods, with $\mathcal{D} = \text{hot}(\boldsymbol{f}, \ell)$ in Equation 3.3. Besides ε and τ, we also alter the limit ℓ. Figure 3.3 shows RE values when $\varepsilon = \ln 9$. We omit $\varepsilon = \ln 49$ since its effects are similar to the ones outlined in Figure 3.1. In the experiments, we use $\ell \in \{0.25, 0.50, 0.75\}$ and observe similar results, with the accuracy increasing when ℓ increases. Thus, we only show

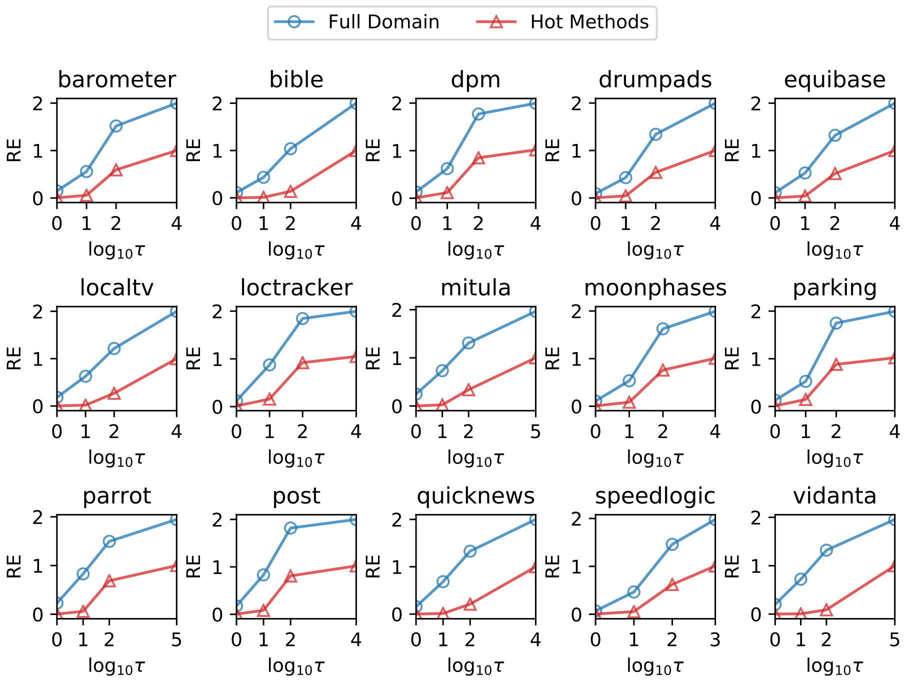


Figure 3.3: Comparison of RE for all methods (i.e., the full domain $\mathcal{V}$) and for hot methods with $\ell = 0.25$ and $\varepsilon = \ln 9$.

the metrics for $\ell = 0.25$. The corresponding RE values from Figure 3.1 are also included in the figure for comparison. We can see that the randomization generates much less RE for hot methods, and thus the frequency of such methods can be estimated with high accuracy, especially with $10 \leq \tau \leq 100$. As discussed later in Section 3.4.5, even such relatively small values of τ provide significant privacy protections for more than 95% app methods in our experiments; in particular, they allow "plausible deniability" about the presence/absence of such methods in a local trace.

Instead of estimating the frequencies of hot methods, one could ask a simpler question: what is the set of hot methods? Such identification of hot methods can be useful, for example, to focus the efforts for manual or automated performance optimization. To measure the quality of the DP estimates for this question, we use the *hot method coverage* (HMC) metric defined by others [107]:

$$\mathrm{HMC}(\ell) = \frac{|\mathrm{hot}(\boldsymbol{F},\ell) \cap \mathrm{hot}(\bar{\boldsymbol{F}},\ell)|}{|\mathrm{hot}(\boldsymbol{F},\ell)|} \tag{3.4}$$

Intuitively, higher values of HMC indicate that hot methods remain hot even after DP processing is applied.

Figure 3.4 shows the average HMC across 100 independent repetitions of the same experiment for $\varepsilon = \ln 9$ and $\varepsilon = \ln 49$, with different values of τ. For smaller values of τ, the set of hot methods is identified very accurately. For example, when $\tau = 1$, perfect hot method coverage is observed for all apps (i.e., HMC $= 1$). Most apps (13 out of 15) have high HMC (≥ 0.9) when τ grows to 10. This suggests that reasonable accuracy can be achieved with proper values of τ.

3.4.5 Presence/Absence of Infrequent Methods

As discussed in Section 3.2.2, Definition 3.1 implies privacy protection of the absence/presence of methods with local frequencies $f(v) \leq \tau$. In other words, from the reported randomized frequencies it is not possible to decide, with high probability, whether such a method was executed at all. To explore the extent of this protection, we collect the number of methods satisfying this property for each of the 1000 traces for an app. The detailed results are not shown here, but can be summarized as follows. More than 95% of the methods in $\mathcal{V}$ (averaged across all apps) have such protection when $\tau \geq 10$, and around 87% of methods are protected even when $\tau = 1$. Such protection could be especially important for infrequently-executed methods that implement sensitive functionality. One example is a method in the mitula app for changing the user's password. Such a method will not be executed frequently in any trace, yet its actions are highly sensitive and may be used for

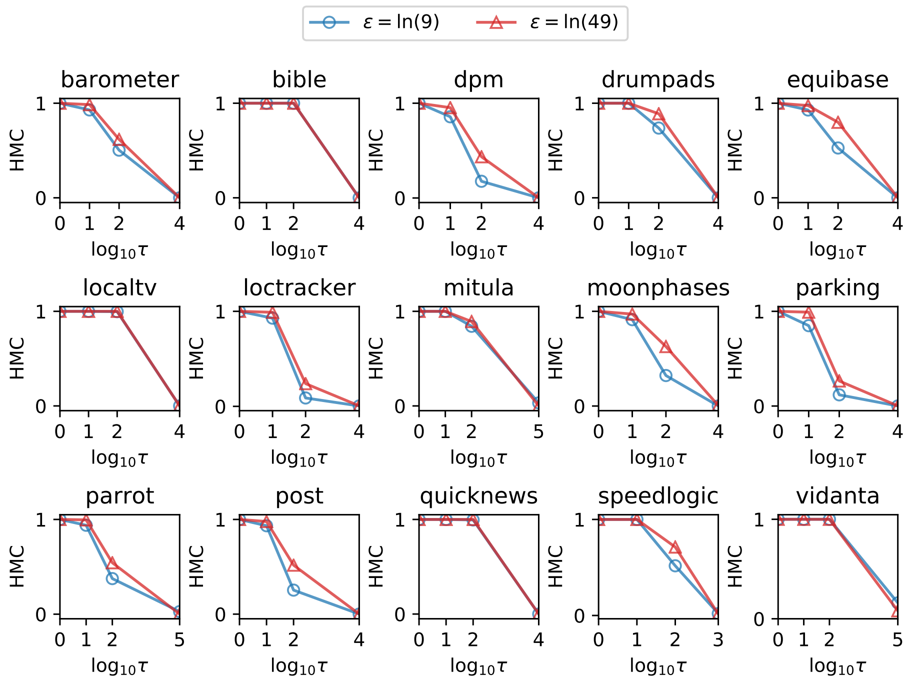


Figure 3.4: HMC for $\ell = 0.25$.

other types of analysis, such as user labelling. Overall, our experimental results clearly show that the approach achieves strong privacy protection for infrequently-executed methods. These results are consistent with the findings from the previous subsections.

3.4.6 Importance of Consistency Constraints

Recall from Section 3.3.2 that we compute estimates $\bar{F}(v)$ based on domain-specific consistency constraints. To evaluate the effects of these constraints, we also compute estimates $\hat{F}(v)$ without the quadratic programming step, as described in Section 3.3.2. For proper comparison with $\bar{F}(v)$, estimates $\hat{F}(v)$ are then normalized by their sum. Figure 3.5 shows the RE values for the two categories of estimates. For all apps, the enforcement of consistency constraints significantly improves the accuracy when $\tau \leq 10$. The RE of $\bar{\boldsymbol{F}}$ is

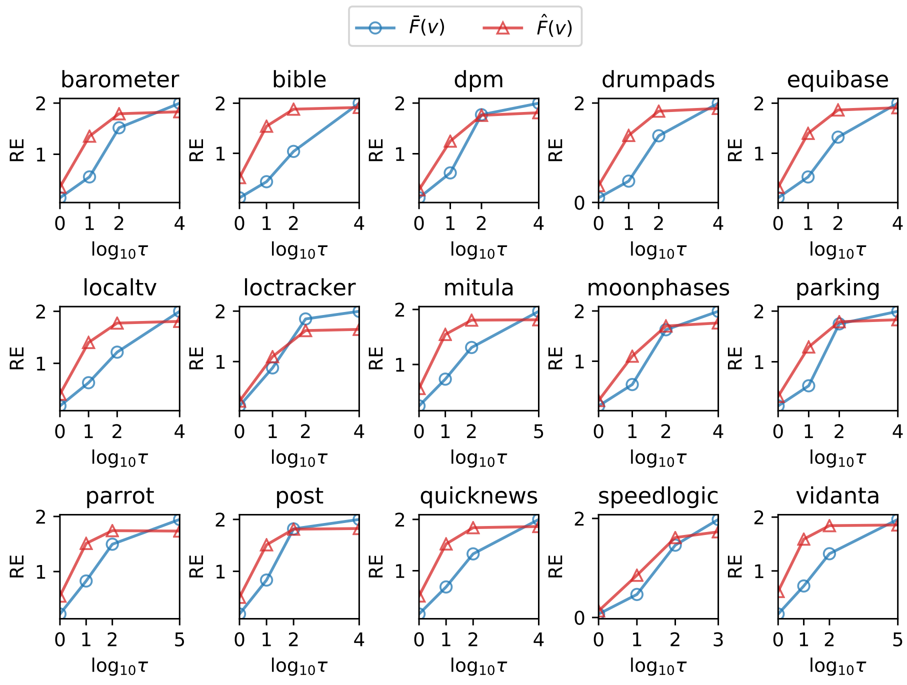


Figure 3.5: Full-domain RE of $\bar{\boldsymbol{F}}$ and $\hat{\boldsymbol{F}}$ with $\varepsilon = \ln 9$.

2.5× smaller than the RE of $\hat{\boldsymbol{F}}$ when $\tau = 1$ and 2.2× smaller for $\tau = 10$, averaged across all apps. For most apps, there are also accuracy benefits when $\tau = 100$.

We also collect similar measurements for hot methods and find that the application of quadratic programming provides improvements for both hot-method RE and HMC. To further illustrate these observations, Figure 3.2 shows $\boldsymbol{F}$, $\bar{\boldsymbol{F}}$, and $\hat{\boldsymbol{F}}$ for the 20 most frequent methods in the speedlogic app. The results indicate reduced accuracy of estimates for hot methods if the consistency constraints are not incorporated in the analysis. We have observed similar effects for other apps. Our conclusion is that the extra step of enforcing consistency constraints is essential for error reduction.

3.5 Discussion and Limitations

The experiments demonstrate that there is no “free lunch”: increased privacy comes with decreased accuracy. However, practical compromises are possible to achieve: for hot methods, one can obtain accurate estimates with some degree of privacy protection, while for infrequently-executed methods methods strong privacy guarantees can be provided at the expense of inaccurate estimates. In many scenarios, the identification and analysis of hot methods (and, more generally, hot statements, edges, paths, etc.) are of primary importance and DP solutions can likely be successfully deployed. In all such cases, DP analysis would have to be tuned to achieve the desired trade-offs, based on the parameterization we propose. Software developers can conduct pre-deployment testing (e.g., using automated testing tools) to obtain profiling information and then analyze it using experiments similar to ours, in order to guide the selection of parameters given some desired privacy guarantees.

Limitations. The proposed approach is designed for event frequency profiling. Other forms of profiling (e.g., for execution time or memory usage) present different challenges and are important targets for future work. In addition, although the approach can effectively hide the presence/absence of infrequent events, it does not perform well for frequency estimation of such events and thus may be unsuitable for some profiling tasks. Another concern is that the randomization requires developers to decide the privacy parameters prior to the actual profiling. Our experimental setup provides a blueprint of how such decisions could be made before deployment, but additional work is needed to improve the automation of this process. Last but not least, there are potential optimizations that the proposed approach does not consider. For example, the communication cost can be reduced by data compression and dimensionality reduction. We leave these enhancements for future work.

3.6 Summary

There is strong interest in privacy-preserving data analysis, driven by legal and societal demands. We study the foundational problem of software event frequency profiling and propose a novel tunable approach for achieving differential privacy. Our techniques are efficient and easy to deploy. Using domain-specific constraints, the approach significantly improves the quality of the frequency estimates. Our experiments indicate that, despite the tension between accuracy and privacy, practical trade-offs can be achieved. Future work on other categories of profiling techniques should continue to grow the body of work in the increasingly-important area of privacy-preserving remote software analysis.

Chapter 4: Differentially-Private Frequency Profiling Under Linear Constraints

In Chapter 3, we propose a randomized-response-based approach to add noise to run-time event traces to ensure their indistinguishability, and optimize the approach by applying efficient randomization on the total trace frequencies. This solution provides tunable trade-offs between privacy and accuracy. In this chapter, we discuss a different setting of software frequency profiling where the order of events is typically ignored and event frequencies are of developers' core interest. Specifically, we propose an approach for differentially-private collection of *frequency vectors* from software executions. Frequency information is reported with the addition of random noise drawn from the Laplace distribution.

A key observation behind the design of our scheme is that event frequencies are closely *correlated* due to the static code structure. Differential privacy protections must account for such relationships; otherwise, a seemingly-strong privacy guarantee is actually weaker than it appears. Motivated by this observation, we propose a novel and general differentially-private profiling scheme when correlations between frequencies can be expressed through *linear inequalities*. Unlike the approach in Chapter 3, which uses simple constraints only in post-processing, here we consider general linear constraints in defining the privacy guarantee. Using a linear programming (LP) formulation, we show how to determine the magnitude of random noise that should be added to achieve meaningful privacy protections under such linear constraints. Next, we develop an efficient instance of this general machinery for an important subclass of constraints. Instead of LP, our solution uses a reachability analysis of a constraint graph. As an exemplar, we employ this approach to implement differentially-private method frequency profiling for Android apps.

Any differentially-private scheme has to balance two competing aspects: privacy and accuracy. Through an experimental study to characterize these trade-offs, we (1) show that our proposed randomization achieves much higher accuracy compared to related prior work, (2) demonstrate that high accuracy and high privacy protection can be achieved simultaneously, and (3) highlight the importance of linear constraints in the design of the randomization. These promising results provide evidence that our approach is a good candidate for privacy-preserving frequency profiling of deployed software.

4.1 Problem and Motivation

4.1.1 Frequency Profiling

Recall from Chapter 3 that we consider a software system that is deployed on the remote machines of n users. A set of events $\mathcal{V}$ is defined by software developers before software deployment. The execution of the software instance at each user $i \in \{1,\ldots,n\}$ triggers at run time a trace of k *event instances* that are recorded by the analysis infrastructure and sent to the remote server for further analysis. Without loss of generality, assume that k is decided by developers ahead of time and is publicly known. Thus, we consider "one-shot" data collection that uses a window of observation of k event instances, and does not collect any other data. Generalizing to a scenario with continuous and unrestricted data collection and reporting requires more advanced privacy-preserving techniques [29, Section 12.3] and is left for future work.

Each of the k event instances is an instance of some event $v \in \mathcal{V}$. For example, v could be "the run-time execution entered method m" and there could be several instances of v among the k event instances. We will use $f_i(v)$ to denote the number of instances of event v in the run-time trace of user i. For convenience, we will shorten "event instance" to "event" when this does not create any ambiguities. The local information for user i can be thought of as a vector of local frequencies $\boldsymbol{f}_i \in \mathbb{N}^{|\mathcal{V}|}$, with each entry corresponding to some event v. The profiling problem we consider is to obtain estimates of population-wide frequency information—that is, to estimate $F(v) = \sum_i f_i(v)$ for every $v \in \mathcal{V}$, or equivalently, to estimate the vector $\boldsymbol{F} = \sum_i \boldsymbol{f}_i$.

As a concrete example, consider analytics frameworks for Android apps. In such apps, each run-time event is logged by calling certain APIs with an identifier of the event. For example, in Firebase Analytics [39], which appears in close to half of popular apps [33], method `logEvent` is used for recording events. This method takes as parameters a string and a map that uniquely identifies the event. When the method is invoked, the underlying analysis infrastructure issues an HTTP request to record the logged event to the remote analytics server.

4.1.2 Privacy-Preserving Profiling

There is a large body of work on various forms of profiling, starting from basic information such as the frequencies of nodes and edges in control-flow models [7] and extending to much more sophisticated run-time properties. When a user releases such information to other parties—e.g., software developers or analysis infrastructure companies—there is a fundamental question about privacy. The profiling information can provide details about execution of sensitive software functionality (e.g., how often the user changes her login credentials). The collected data can be used for characterizing user habits and interests, which can then be combined with other sources of information for the purposes of behavior profiling. There is a growing trend of aggregating data from various sources to create rich knowledge about users—e.g., by cross-linking records from various data providers and by applying sophisticated data mining. Data anonymization is not sufficient to address this problem [67, 68]. The user has no control over unethical business practices (e.g., selling the data to third parties) or unexpected uses of the data due to silent changes in end-user privacy agreements, subpoenas by law enforcement, or security breaches in which user data is stolen and then shared with malicious actors.

As multiple sources of data about a person can be combined in ways that cannot be anticipated at the time when a profiling technique is deployed, it becomes increasingly important to deploy profiling techniques that are designed with theoretical guarantees against unknown privacy threats. Differential privacy is such a theoretical framework. A key property of this approach is that it ensures privacy protections in the extreme setting where an adversarial entity has access to large amounts of auxiliary data about the individual, and employs such data in ways that are unknown to the designers of the differentially-private data gathering. These strong properties make differential privacy an appealing target for the designers of privacy-preserving software analyses.

Next, we outline the differentially-private version of our frequency profiling analysis. Without privacy, a user reports to the remote analysis server her local frequency information, encoded as a vector of local frequencies $\boldsymbol{f}_i$. The server then simply computes and reports the global frequency vector $\boldsymbol{F} = \sum_i \boldsymbol{f}_i$. With a differentially-private schema, the local frequencies are modified with the help of some randomization mechanism R. We consider randomizers $R : \mathbb{N}^{|\mathcal{V}|} \to \mathbb{R}^{|\mathcal{V}|}$, where the frequency $f_i(v) \in \mathbb{N}$ of event $v \in \mathcal{V}$ for user i is transformed into a “noisy” frequency $R(f_i(v)) \in \mathbb{R}$. As discussed shortly, the noise being added is drawn from a non-discrete probability distribution, which is why the randomized data is not in the domain of integers.

The randomizer R is the same for all software users and is designed and embedded in the software implementation before the software is deployed. It is important to note that differential privacy assumes that a privacy adversary knows the exact design of R. For example, since the randomization is implemented in the code of the software that resides on a user’s machine, reverse engineering of this code could reveal the exact algorithm for R. Nevertheless, the randomizer should still provide the differential privacy guarantee (described shortly) even in this adversarial setting. Intuitively, by observing $R(\boldsymbol{f}_i)$ an external entity should *not* be able to produce high-confidence estimates of $\boldsymbol{f}_i$.

Each user i reports $R(\boldsymbol{f}_i)$ to the analysis server. The server computes a vector $\hat{\boldsymbol{F}} \in \mathbb{R}^{|\mathcal{V}|}$ which estimates the vector of true global frequencies $\boldsymbol{F}$. The computation of $\hat{\boldsymbol{F}}$ depends on the randomization mechanism R. For example, the approach in Chapter 3 requires a post-processing step to account for the randomization of R to calculate $\hat{\boldsymbol{F}}$. In the approach we propose in this chapter, we can simply compute $\hat{\boldsymbol{F}} = \sum_i R(\boldsymbol{f}_i)$.

4.2 Feasible Frequency Vectors Under Linear Constraints

In many program profiling problems, there are constraints on the run-time frequencies that are imposed by the static structure of the code. For example, consider the following code:

```
void m1() { if (...) { m3(); m3(); } }
void m2() { if (...) m3(); }
```

Further, assume that there are no other calls to m3 in the entire program. From this code structure, we can conclude that $2f(\texttt{m1}) + f(\texttt{m2}) \geq f(\texttt{m3})$ for any run-time execution of this code. Here $f(\mathtt{m})$ denotes the frequency of method m in the method-execution-frequency vector $\boldsymbol{f}$.

Linear Constraints on Run-time Frequencies. In this chapter we focus on a class of commonly-occurring constraints that can be expressed as *linear inequalities* over the elements of the frequency vector $\boldsymbol{f}$. These inequalities are of the form $\boldsymbol{Af} \geq \boldsymbol{b}$, where $\boldsymbol{A}_{m\times|\mathcal{V}|}$ is an integer matrix encoding m linear functions of $|\mathcal{V}|$ variables and $\boldsymbol{b}_{m\times 1}$ is an integer vector. Here we assume that $\boldsymbol{f}$ is an integer vector of dimensionality $|\mathcal{V}| \times 1$. For the example from above, $\boldsymbol{A} = \begin{pmatrix} 2 & 1 & -1 \end{pmatrix}$ and $\boldsymbol{b} = \begin{pmatrix} 0 \end{pmatrix}$.

Feasible Frequency Vectors. For our problem statement, not all frequency vectors are *feasible* at run time. Infeasible frequency vectors do not represent any run-time user behavior and will never be observed during software execution. For the code example shown above, there is no execution that can produce a vector $\boldsymbol{f}$ with $2f(\mathtt{m1}) + f(\mathtt{m2}) < f(\mathtt{m3})$. We define this property as follows:

Definition 4.1 (Feasibility)**.** *A frequency vector* $\boldsymbol{f}$ *is* feasible *if* $\boldsymbol{f} \geq \mathbf{0}$, $\boldsymbol{Af} \geq \boldsymbol{b}$, *and* $\|\boldsymbol{f}\|_1 = k$.

Here $\|\boldsymbol{f}\|_1$ denotes the L_1 norm of a vector $\boldsymbol{f}$, i.e., $\sum_{v\in\mathcal{V}} |f(v)|$. As discussed later, the design of a differentially-private scheme must consider this notion of feasibility, and in particular the linear constraints that feasible vectors satisfy. If the designers of the analysis do not account for such constraints, the scheme is fundamentally flawed.

4.3 The Differential Privacy Guarantee

Informally, the differential privacy guarantee is of the following form: for any possible output of a differentially-private data analysis, the probability this output was produced from real data a and the probability this output was produced from some "neighbor" data b are close to each other. Thus, just by observing the analysis output, an adversary cannot distinguish with high probability the case when the input was a from the case when the input was some neighbor b of a. This probabilistic *indistinguishability* guarantee is the essence of differential privacy.

4.3.1 Indistinguishability

In the context of our problem, this property applies to any two neighbor feasible frequency vectors $\boldsymbol{f}$ and $\boldsymbol{f}'$ that could represent the local data of one software user. The "neighbor" relation will be defined shortly. Following the standard approach from the differential privacy literature [29], we define *ε-indistinguishability* as follows:

Definition 4.2 (ε-indistinguishability). *Randomizer R achieves* ε-indistinguishability *if for any two neighboring feasible frequency vectors $\boldsymbol{f}, \boldsymbol{f}' \in \mathbb{N}^{|\mathcal{V}|}$ and for any set $\mathcal{O} \subseteq \mathbb{R}^{|\mathcal{V}|}$ of possible outputs of the randomizer, the following holds:*

$$\frac{\Pr[R(\boldsymbol{f}) \in \mathcal{O}]}{\Pr[R(\boldsymbol{f}') \in \mathcal{O}]} \leq e^{\varepsilon}$$

This definition should be interpreted as follows: for any randomizer output, the likelihood that the real data was $\boldsymbol{f}$ is close to the likelihood that the real data was $\boldsymbol{f}'$. Thus, $\boldsymbol{f}$ and $\boldsymbol{f}'$ are indistinguishable in a probabilistic sense from the point of view of an adversary who observes the randomizer output. Parameter ε defines the strength of this protection. Smaller values of ε provide stronger protection but necessitate more "noisy" randomization which leads to less accurate population-wide estimates. In practical applications, values of ε such as $\ln(9)$ have been used [32].

4.3.2 Defining Neighbors

Recall that in our problem statement, for any feasible frequency vector $\boldsymbol{f}$ we have $\|\boldsymbol{f}\|_1 = k$. The L_1 distance between two such frequency vectors $\boldsymbol{f}$ and $\boldsymbol{f}'$ is $\left\|\boldsymbol{f} - \boldsymbol{f}'\right\|_1$ and this distance is in the set $\{0, 2, 4, \ldots, 2k\}$. To normalize, we define the distance between two feasible frequency vectors as

$$d(\boldsymbol{f}, \boldsymbol{f}') = \frac{1}{2}\left\|\boldsymbol{f} - \boldsymbol{f}'\right\|_1 = \frac{1}{2}\sum_{v \in \mathcal{V}} |f(v) - f'(v)|$$

We can then define the *neighbors* of a vector $\boldsymbol{f}$ as

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

Here threshold τ is used to define the extent of this neighborhood. Note the different role of parameters τ and ε. Using τ, we define for which pairs of vectors we aim to ensure indistinguishability. The next section discusses the implications of this choice. Once the notion of neighbors is defined, ε determines the desired strength of the indistinguishability between any such pair of neighbors.

4.3.3 Randomization Based on Laplace Mechanism

We next show how to design a randomizer that achieves ε-indistinguishability. Our approach is a direct application of a classic technique from differential privacy: it draws random noise from the Laplace distribution and adds it to the "raw" frequency vector. The Laplace probability distribution $\text{Lap}(b)$, parameterized by a scale parameter $b > 0$ and centered at 0, is defined by the probability density function $p(y|b) = \frac{1}{2b}\exp(-\frac{|y|}{b})$. This is a symmetric version of the exponential distribution. We define the following randomizer:

$$R(\boldsymbol{f}) = \boldsymbol{f} + \begin{pmatrix} Y_1 \\ \vdots \\ Y_{|\mathcal{V}|} \end{pmatrix}$$

where each $Y_j \sim \mathrm{Lap}(b)$ is drawn independently from the Laplace distribution with some scale parameter b. As a direct corollary from standard results in differential privacy, this R achieves ε-indistinguishability as long as $b \geq \frac{2\tau}{\varepsilon}$. Below we provide an outline of the proof for completeness.

Proof. Consider two feasible vectors $\boldsymbol{f}$ and $\boldsymbol{f}'$ such that $d(\boldsymbol{f}, \boldsymbol{f}') \leq \tau$. Let $g : \mathbb{R}^{|\mathcal{V}|} \to [0,1]$ be the probability density function (PDF) of $R(\boldsymbol{f})$ and $g' : \mathbb{R}^{|\mathcal{V}|} \to [0,1]$ be the PDF for $R(\boldsymbol{f}')$. Let $p : \mathbb{R} \to [0,1]$ denote the PDF for $\mathrm{Lap}(b)$. Given some $\boldsymbol{z} = \begin{pmatrix} z_1 & \dots & z_{|\mathcal{V}|} \end{pmatrix} \in \mathcal{O} \subseteq \mathbb{R}^{|\mathcal{V}|}$, we have $g(\boldsymbol{z}) = \prod_j p(z_j - f_j)$ where f_j is the value of the j-th element in $\boldsymbol{f}$. Similarly, $g'(\boldsymbol{z}) = \prod_j p(z_j - f'_j)$. The ratio of $g(\boldsymbol{z})$ and $g'(\boldsymbol{z})$ is a product of ratios of $p(z_j - f_j)$ and $p(z_j - f'_j)$:

$$
\begin{aligned}
\frac{g(\boldsymbol{z})}{g'(\boldsymbol{z})} &= \prod_j \frac{p(z_j - f_j)}{p(z_j - f'_j)} \\
&= \prod_j \left(e^{-\frac{|z_j - f_j|}{b}} \Big/ e^{-\frac{|z_j - f'_j|}{b}} \right) \\
&\leq \prod_j e^{\frac{|f_j - f'_j|}{b}} \\
&= e^{\frac{\|f - f'\|_1}{b}} \\
&\leq e^{\frac{2\tau}{b}} \\
&\leq e^{\varepsilon}
\end{aligned}
$$

where the first inequality follows from the triangle inequality, and the second follows from the constraint $d(\boldsymbol{f}, \boldsymbol{f}') \leq \tau$ which implies $\left\| \boldsymbol{f} - \boldsymbol{f}' \right\|_1 \leq 2\tau$. The last inequality is due to

$b \geq \frac{2\tau}{\varepsilon}$. Then we can conclude

$$\begin{aligned}\frac{\Pr[R(\boldsymbol{f}) \in \mathcal{O}]}{\Pr[R(\boldsymbol{f}') \in \mathcal{O}]} &= \frac{\int_{\mathcal{O}} g(\boldsymbol{z})d\boldsymbol{z}}{\int_{\mathcal{O}} g'(\boldsymbol{z})d\boldsymbol{z}} \\ &\leq \max_{\boldsymbol{z}\in\mathcal{O}} \frac{g(\boldsymbol{z})}{g'(\boldsymbol{z})} \\ &\leq e^{\varepsilon}\end{aligned}$$

Thus R achieves ε-indistinguishability. □

4.3.4 Randomization Based on Randomized Response

Several differentially-private data analyses (e.g., [9, 32, 96]) have employed as a basic building block in their construction the technique of *randomized response* [99]. This techniques was initially proposed in social sciences to survey information that could be sensitive. The essence of randomized response is that a user's data is represented as a binary vector, with each bit corresponding to an event of interest, and then bits are inverted with some probability. The resulting randomized vectors from many users are aggregated and calibrated to obtain population-wide estimates.

This machinery has been employed in Chapter 3 to gather event frequency information from software executions. Each run-time event is considered as a binary vector of size $|\mathcal{V}|$ with a single bit equal to 1, corresponding to the observed event. Each such vector is then randomized. The privacy guarantee for this approach ensures indistinguishability between the real event trace and all other traces that differ from it by at most τ events.

Such an approach may be suitable for scenarios where event-by-event reports are sent to the server—for example, for current mobile app analytics frameworks where the randomization has to be conducted immediately after an event is triggered. However, it is unnecessarily "noisy" for cases when only the frequency vector of the trace needs to be shared. Rather than randomizing each event (via randomized response) and then accumulating the results in a

```
class SignInActivity {
  User user;
  boolean validPasswords(String p1, String p2) {
    Analytics.logEvent("SignInActivity.validPasswords"); // e1
    return !Utils.isEmpty(p1) && isPasswordValid(p1)
        && !Utils.isEmpty(p2) && p1.equals(p2);
  }
  void registration() {
    Analytics.logEvent("SignInActivity.registration"); // e2
    EditText v1 = ...;
    EditText v2 = ...;
    String password = v1.getText();
    String repeatPassword = v2.getText();
    if (validPasswords(password, repeatPassword)) {
      user.setPassword(password);
    }
  }
}
class User {
  String password;
  void setPassword(String password) {
    Analytics.logEvent("User.setPassword"); // e3
    this.password = password;
  }
}
```

Figure 4.1: Code derived from the mitula app.

frequency vector, we propose to accumulate the actual frequencies first and then randomize the resulting vector using the Laplace mechanism. Theoretically, the second approach provably achieves better accuracy by roughly a factor of $\sqrt{k}$; this observation is a consequence of well-known results in differential privacy [29, Section 12.1]. Our experimental results in Section 4.6 empirically confirm that significantly higher accuracy is indeed observed in practice.

Example 4.1. As a concrete example, consider the collection the frequencies of method invocations in mobile apps. Figure 4.1 shows a snippet of code derived from the mitula app, which has more than 1 million downloads according to Google Play store [64]. Class `SignInActivity` allows the user to register a new account or log in using an existing account. After signing in to the app, the user's password, along with other critical information such as email address and authentication token, is stored in an instance of the `User` class. As introduced in Section 4.1.1, method invocations can be recorded by calling APIs in analytics frameworks. Here, we use Firebase Analytics [39] for demonstration. As highlighted in the figure, the app can be instrumented by inserting a call to the `logEvent` API at the entry of every method with the method's signature being a parameter, so that each method call at run time is recorded and sent to the analytics server. For simplicity, we drop other parameters of `logEvent` in this example. At run time, the frequency of e_1 is guaranteed to be greater than or equal to the frequency of e_2, since each invocation of `registration` involves a call to `validPasswords` in class `SignInActivity`. There are no additional constraints for the three events as there are other call sites in the app of which the target is `SignInActivity.validPasswords` or `User.setPassword`.

Using the randomized-response-based approach, each call to `logEvent` is intercepted and the corresponding method-invocation event is perturbed. Figure 4.2 illustrates this process. It includes event traces from 100 simulated app users, derived from the actual data used in our experimental evaluation presented later in Section 4.6. Each trace records $k = 3$ run-time occurrences of the three events $\mathcal{V} = \{e_1, e_2, e_3\}$ in Figure 4.1. The actual aggregated frequency vector in this concrete example is $\boldsymbol{F} = \begin{pmatrix} 105 & 88 & 107 \end{pmatrix}$. We set $\varepsilon = 1$ and $\tau = 1$ (number of differing events in neighbor traces), which defines the functionality of randomizer R. At the server, the randomization results for each event at each user are

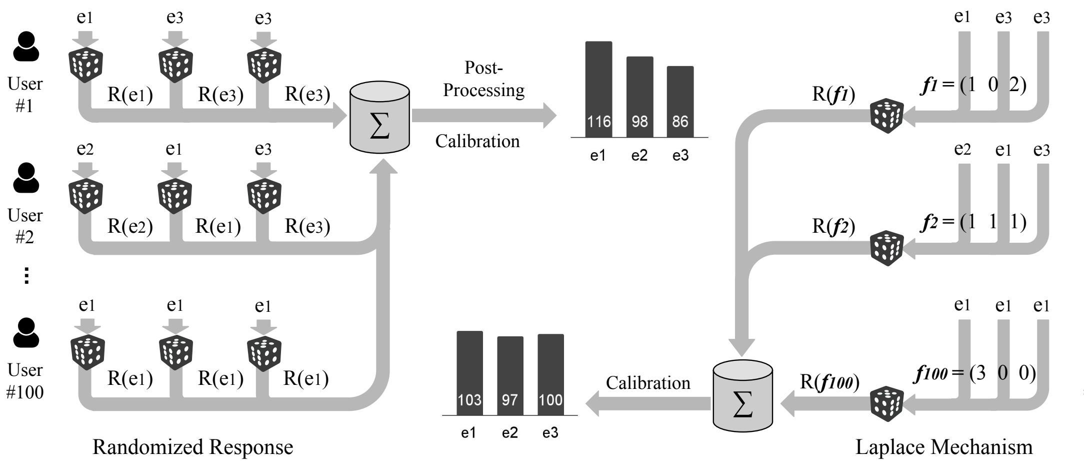


Figure 4.2: Comparison between randomization based on randomized response and Laplace mechanism, by simulating 100 users using $\mathcal{V} = \{e_1, e_2, e_3\}$ from Figure 4.1, $k = 3$, $\varepsilon = 1$, and $\tau = 1$.

accumulated and post-processed to account for the added random noise. The aggregation result is then calibrated to produce the final frequency estimates. Details of this calibration step will be introduced shortly. As shown in the figure, the resulting vector of frequency estimates is $\begin{pmatrix}116 & 98 & 86\end{pmatrix}$.

The randomization based on Laplace mechanism is applied to local frequency vectors instead of individual events. When an event is triggered at a user i, it is accumulated locally in a frequency vector $\boldsymbol{f}_i$. The randomizer R directly adds noise drawn from the Laplace distribution to $\boldsymbol{f}_i$, as introduced in Section 4.3.3. To compare with the randomization based on randomized response, we use the same parameters for R, i.e., $\varepsilon = 1$ and $\tau = 1$. As a result, the aggregated frequency vector in this concrete example is $\begin{pmatrix}103 & 97 & 100\end{pmatrix}$ after calibration. Its distance to the actual frequency vector $\boldsymbol{F}$ is 9, much smaller than the distance for the approach based on randomized response which is 21. Thus, the resulting estimates is "closer" to the true frequencies. This indicates that, at least for the events and settings in this example, the randomization based on Laplace mechanism produces more accurate analysis results for frequency profiling. In Section 4.6, we conduct extensive experiments on a set of apps and show the benefits of the proposed approach.

4.3.5 Calibration of Estimates

One problem of the random frequency perturbation performed by R is that it leads to *inconsistent* aggregated frequency vectors. As introduced in Section 4.2, every run-time frequency vector $\boldsymbol{f}$ satisfies the feasibility property, i.e., $\boldsymbol{f} \geq \mathbf{0}, \boldsymbol{A}\boldsymbol{f} \geq \boldsymbol{b}$, and $\|\boldsymbol{f}\|_1 = k$. It is desirable that the aggregated frequency estimate $\hat{\boldsymbol{F}} = \sum_i R(\boldsymbol{f}_i)$ satisfies the constraints that would have been satisfied by the true aggregated frequency $\boldsymbol{F} = \sum_i \boldsymbol{f}_i$. It is easy to see that those constraints are $\hat{\boldsymbol{F}} \geq \mathbf{0}, \boldsymbol{A}\hat{\boldsymbol{F}} \geq n\boldsymbol{b}$, and $\left\|\hat{\boldsymbol{F}}\right\|_1 = nk$. However, in general $\hat{\boldsymbol{F}}$ will violate the constraints. A standard approach to enforce such constraints is to calibrate the estimates in $\hat{\boldsymbol{F}}$. Following similar techniques from Chapter 3, we compute a calibrated frequency estimate vector $\bar{\boldsymbol{F}} \in \mathbb{R}^{|\mathcal{V}|}$ which satisfies $\bar{\boldsymbol{F}} \geq \mathbf{0}, \boldsymbol{A}\bar{\boldsymbol{F}} \geq n\boldsymbol{b}$, and $\left\|\bar{\boldsymbol{F}}\right\|_1 = nk$ and minimizes its squared Euclidean distance to $\hat{\boldsymbol{F}}$. This calibrated estimate is the final output of the analysis.

4.4 Hiding The Presence or Hotness of Individual Events

One key question for the proposed randomization is how to select the value for τ. The notion of "protected distance" between frequency vectors ultimately has to be derived from some desired higher-level privacy properties. In this chapter we consider two examples of such properties. First, we discuss a scenario where the analysis designer aims to ensure that the presence of a certain run-time event is hidden from a privacy adversary. More precisely, for some event $v \in \mathcal{V}$ that occurred at least once at run time, we would like to ensure that the adversary cannot distinguish, in the differential privacy sense, the actual run-time behavior from another possible behavior in which v was not observed at all. Equivalently, for any

$\boldsymbol{f}$ with $f(v) > 0$, we would like to ensure that there exists at least one feasible $\boldsymbol{f}'$ such that $f'(v) = 0$ and $d(\boldsymbol{f}, \boldsymbol{f}') \leq \tau$.

A second scenario is where the analysis designer aims to hide from a privacy adversary the "hotness" of some event, by ensuring indistinguishability with vectors in which this event is "cold". In this scenario, an event whose frequency in $\boldsymbol{f}$ exceeds a certain threshold is considered frequently-executed (i.e., hot) in $\boldsymbol{f}$. Here for any $\boldsymbol{f}$ in which v is hot, we should ensure that there exists at least one feasible $\boldsymbol{f}'$ in which v is cold and $d(\boldsymbol{f}, \boldsymbol{f}') \leq \tau$. We next discuss the first scenario; the second one is presented later in this section.

We formulate the following problem: given a set $\mathcal{U} \subseteq \mathcal{V}$ of events whose presence in the run-time execution needs to be hidden, select the smallest distance threshold τ such that for any particular $v \in \mathcal{U}$, the presence of v is hidden in the sense discussed above. (A more precise formulation will be presented shortly in Section 4.4.1.) The choice of $\mathcal{U}$ depends on the usage scenario. For example, there may be some sensitive code functionality, e.g., changing a password, whose presence at run time should be hidden. Then $\mathcal{U}$ will contain any event related to these parts of the code. As another example, we could select a criterion of the form "at least $h\%$ of events are protected" based on some threshold h. In this chapter, we explore the second use case, but the underlying techniques are directly applicable to any other choices for $\mathcal{U}$.

Next we discuss the following key subproblem: given $\mathcal{U}$ and some feasible vector $\boldsymbol{f}$, determine the smallest distance threshold τ such that the presence of any particular $v \in \mathcal{U}$ in $\boldsymbol{f}$ is hidden by the randomization. Section 4.5 describes the general problem: how to select the value of τ to achieve privacy for the entire approach, over many software users i with different local vectors $\boldsymbol{f}_i$.

4.4.1 Difficulty of Hiding An Event

Consider any $\boldsymbol{f}$ with $f(v) > 0$. The presence of v in $\boldsymbol{f}$ can be hidden if $\boldsymbol{f}$ is indistinguishable from some neighbor $\boldsymbol{f}'$ with $f'(v) = 0$. We define the *difficulty of hiding v in $\boldsymbol{f}$* as the smallest distance at which such an $\boldsymbol{f}'$ exists:

$$D_v(\boldsymbol{f}) = \min_{\boldsymbol{f}':f'(v)=0} d(\boldsymbol{f},\boldsymbol{f}')$$

Here both $\boldsymbol{f}$ and $\boldsymbol{f}'$ must be feasible. If no feasible $\boldsymbol{f}'$ exists such that $f'(v) = 0$, it means that v always occurs at least once in all possible executions. In this case, it is impossible to provide protection to hide the presence of this v and we have $D_v(\boldsymbol{f}) = \infty$.

If we have a randomizer that uses a value of $\tau \geq D_v(\boldsymbol{f})$, the presence of v in $\boldsymbol{f}$ is hidden by the randomization. To achieve our goal of hiding any particular event from $\mathcal{U}$ in a given $\boldsymbol{f}$, this property has to hold for each $v \in \mathcal{U}$. In order to find the smallest such τ, we need to be able to compute $D_v(\boldsymbol{f})$ for any given v and $\boldsymbol{f}$. In the following subsection, for the general case of arbitrary linear constraints, we describe this problem as an integer linear programming problem. We then demonstrate that for the class of constraints of the form $f(v) \geq f(v')$, a more efficient alternative is to define a constraint graph and then perform analysis of this graph.

4.4.2 Computing Difficulty Using Linear Programming

We can compute $D_v(\boldsymbol{f})$ for any given $\boldsymbol{f}$ and v by solving the following optimization problem: minimize $\frac{1}{2}\|\boldsymbol{x}-\boldsymbol{f}\|_1$ for $\boldsymbol{x} \in \mathbb{N}^{|\mathcal{V}|}$, $\boldsymbol{x} \geq \boldsymbol{0}$, $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$, $\|\boldsymbol{x}\|_1 = k$, and $x(v) = 0$. Here $\boldsymbol{x}$ is the unknown variable representing $\boldsymbol{f}'$. All constraints except for the last one ensure that $\boldsymbol{x}$ is a feasible frequency vector. This problem can be transformed into an *integer linear* *programming* problem:

$$\begin{aligned}
\min_{\boldsymbol{x},\boldsymbol{s}\in\mathbb{N}^{|\mathcal{V}|}} \quad & \tfrac{1}{2}^{\mathsf{T}}\boldsymbol{s} \\
\text{subject to} \quad & \boldsymbol{x} \geq \boldsymbol{0} \\
& \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \\
& \boldsymbol{1}^{\mathsf{T}}\boldsymbol{x} = k \\
& x(v) = 0 \\
& \boldsymbol{s} \geq \boldsymbol{x} - \boldsymbol{f} \\
& \boldsymbol{s} \geq \boldsymbol{f} - \boldsymbol{x}
\end{aligned}$$

The optimal value for the objective function gives us the desired value of $D_v(\boldsymbol{f})$.

4.4.3 Computing Difficulty Using Constraint Graph Analysis

Next, we focus on the special case of linear constraints of the form $f(v) \geq f(v')$. Such constraints can naturally occur when the static structure of the code enforces properties such as "the only way event v' could occur is if it is caused by event v".

4.4.3.1 Linear Constraints

To illustrate such constraints, consider method frequency profiling in a language such as Java. In this case $\mathcal{V}$ contains all methods m in the program code (excluding library/framework methods). Let $f(m)$ denote the number of times method m was executed at run time. We borrow the following exemplar static analysis from Chapter 3. First, if a call site in a caller method m' has only one possible target m, and this call site dominates the exit of m', then the frequency of m' is no greater than the frequency of m. Second, if there is a single call site in the program that may invoke a method m, this call site is not in a control-flow-graph loop, and m does not override any method from standard libraries or relevant frameworks (e.g., the Android framework), the frequency of m' is greater than or equal to the frequency of m. For example, consider the following code:

```
void m1() { m2(); do { m3(); } while (...); }
void m2() { m4(); if (...) m5(); }
```

Suppose there are no other calls to `m5` in the entire program, and `m5` does not override library/framework methods. We can conclude that the following constraints hold: $f(\texttt{m2}) \geq f(\texttt{m1})$, $f(\texttt{m3}) \geq f(\texttt{m1})$, $f(\texttt{m4}) \geq f(\texttt{m2})$, and $f(\texttt{m2}) \geq f(\texttt{m5})$. For the example in Figure 4.1, we can conclude that $f(e_1) \geq f(e_2)$.

4.4.3.2 Constraint Graph Analysis

Each constraint $f(v) \geq f(v')$ can be encoded as an edge $v \rightarrow v'$ in a directed graph $G = (\mathcal{V}, \mathcal{E})$. Given this *constraint graph*, the computation of difficulty $D_v(\boldsymbol{f})$ for any given v and $\boldsymbol{f}$ with $f(v) > 0$ is as follows. Let R_v be the set of nodes reachable from v in G, including v. In any $\boldsymbol{f}'$ with $f'(v) = 0$, all nodes in R_v must have frequency of 0. To compute $D_v(\boldsymbol{f})$ we need to find such an $\boldsymbol{f}'$ that minimizes $d(\boldsymbol{f}, \boldsymbol{f}')$. A lower bound on $d(\boldsymbol{f}, \boldsymbol{f}')$ is the sum of frequencies in $\boldsymbol{f}$ for the nodes in R_v (since all of them will become 0 in $\boldsymbol{f}'$). If there exists $w \notin R_v$ such that w does not have any predecessors in graph, this lower bound is tight. In this case, to construct $\boldsymbol{f}'$, we can increase the frequency of w by $\sum_{v' \in R_v} f(v')$, without affecting any other nodes. Theoretically, such a w does not necessarily exist in general cyclic graphs. However, in the examples we have seen from real code, there is always a non-trivial number of events whose frequency is not constrained from above (i.e., they do not have incoming edges), and therefore this assumption is always satisfied. Based on these observations, we compute $D_v(\boldsymbol{f}) = \sum_{v' \in R_v} f(v')$. While the same result could be achieved with the more general linear programming formulation presented earlier, this graph analysis is more efficient and easy to implement.

Example 4.2. For the earlier example, $\mathcal{E} = \{\texttt{m4} \to \texttt{m2}, \texttt{m2} \to \texttt{m1}, \texttt{m2} \to \texttt{m5}, \texttt{m3} \to \texttt{m1}\}$. Further, suppose that $k = 16$ and we are given a vector $\boldsymbol{f} = \begin{pmatrix} 2 & 3 & 4 & 5 & 2 \end{pmatrix}$, where the first element is the frequency of `m1`, the second one the frequency of `m2`, etc. Suppose we are interested in $D_{\texttt{m2}}(\boldsymbol{f})$. Since $R_{\texttt{m2}} = \{\texttt{m1}, \texttt{m2}, \texttt{m5}\}$, $D_{\texttt{m2}}(\boldsymbol{f}) = 7$ and the corresponding neighbor vector $\boldsymbol{f}' = \begin{pmatrix} 0 & 0 & 4 & 12 & 0 \end{pmatrix}$ or $\boldsymbol{f}' = \begin{pmatrix} 0 & 0 & 11 & 5 & 0 \end{pmatrix}$. As another example, $D_{\texttt{m4}}(\boldsymbol{f}) = 12$ with $\boldsymbol{f}' = \begin{pmatrix} 0 & 0 & 16 & 0 & 0 \end{pmatrix}$.

4.4.4 Hiding The Hotness of An Event

The approach presented earlier can be adapted to provide a different form of privacy protection. Suppose we have defined two categories of events: “hot” and “cold”. An event v is hot in $\boldsymbol{f}$ if $f(v) > \eta$ where η is a pre-defined threshold, possibly dependent on k and $|\mathcal{V}|$. Given a hot event v in some $\boldsymbol{f}$, we can ensure that an adversary cannot distinguish this situation from another possible situation in which v was cold in $\boldsymbol{f}$.

Suppose we are given some v and $\boldsymbol{f}$ with $f(v) > \eta$. The hotness of v in $\boldsymbol{f}$ can be hidden if $\boldsymbol{f}$ is indistinguishable from some neighbor $\boldsymbol{f}'$ with $f'(v) \le \eta$. In this case, the *difficulty of hiding the hotness of v in $\boldsymbol{f}$* is the smallest distance at which such an $\boldsymbol{f}'$ exists:

$$D_v(\boldsymbol{f}) = \min_{\boldsymbol{f}' : f'(v) \le \eta} d(\boldsymbol{f}, \boldsymbol{f}')$$

Here both $\boldsymbol{f}$ and $\boldsymbol{f}'$ must be feasible. If no feasible $\boldsymbol{f}'$ with $f'(v) \le \eta$ exists, this means that v is always hot in all possible executions. The computation of this difficulty can be expressed as an integer linear programming problem similar to the one in Section 4.4.2, except that constraint $x(v) = 0$ is replaced by $x(v) \le \eta$. For the class of constraints described in Section 4.4.3, the constraint graph analysis considers all hot nodes v' reachable from v and computes $D_v(\boldsymbol{f}) = \sum_{v'} (f(v') - \eta)$.

4.4.5 Importance of Constraints in Randomizer Design

In designing our randomization approach, it is critical to account for the linear constraints $\boldsymbol{A}\boldsymbol{f} \geq \boldsymbol{b}$ that must be satisfied by any feasible $\boldsymbol{f}$. To illustrate this point, suppose that for a given $\boldsymbol{f}$ with $f(v) > 0$, we are interested in hiding the presence of v.

Example 4.3. Consider the example presented earlier: constraints $f(\mathtt{m2}) \geq f(\mathtt{m1})$, $f(\mathtt{m3}) \geq f(\mathtt{m1})$, $f(\mathtt{m4}) \geq f(\mathtt{m2})$, and $f(\mathtt{m2}) \geq f(\mathtt{m5})$, $k = 16$, and $\boldsymbol{f} = \begin{pmatrix} 2 & 3 & 4 & 5 & 2 \end{pmatrix}$. As discussed, $D_{\mathtt{m2}}(\boldsymbol{f}) = 7$. Now suppose that instead of considering $\boldsymbol{f}'$ for which $\boldsymbol{f}' \geq \boldsymbol{0}$, $\boldsymbol{A}\boldsymbol{f}' \geq \boldsymbol{b}$, $\left\|\boldsymbol{f}'\right\|_1 = k$, and $f'(v) = 0$, the designer of the randomizer ignored the linear constraints (or was not aware of them) and instead considered $\boldsymbol{f}'$ with $\boldsymbol{f}' \geq \boldsymbol{0}$, $\left\|\boldsymbol{f}'\right\|_1 = k$, and $f'(v) = 0$. It is easy to see that in that case the conclusion would be that the difficulty of hiding `m2` is $f(\mathtt{m2}) = 3$. For example, vector $\boldsymbol{f}' = \begin{pmatrix} 2 & 0 & 7 & 5 & 2 \end{pmatrix}$ would be considered one legitimate neighbor, and the conclusion would be that $D_{\mathtt{m2}}(\boldsymbol{f}) = 3$.

What are the implications of this outcome? Suppose that $\mathcal{U} = \{\mathtt{m2}\}$; that is, the only event we want to hide is `m2`. To achieve this protection for `m2`, the appropriate selection is $\tau = 7$. However, if instead we chose $\tau = 3$ because we ignored the linear constraints, we added insufficient noise because we used $\text{Lap}(\frac{6}{\varepsilon})$ to draw the random noise when in reality we should have used $\text{Lap}(\frac{14}{\varepsilon})$. Equivalently, the "effective" value of ε was increased by a factor of $\frac{7}{3}$. Recall from Definition 4.2 that the role of ε is to bound the ratio of probabilities between possible neighbors, which determines the strength of the indistinguishability guarantee. In effect, this ratio is now increased by a factor of $e^{\frac{7}{3}} \approx 10.3$. For example, if the designer of the randomizer was intending for the ratio of "probability that the real data was $\boldsymbol{f}$" and "probability that the real data was $\boldsymbol{f}'$" to be bounded by 3, in reality it is only bounded by 31. This could make one of these alternatives much more likely than the other one, which significantly weakens the privacy guarantee of indistinguishability.

In the general case, if the linear constraints are ignored, the effective value of ε is increased by a factor of $\frac{D_v(\boldsymbol{f})}{f(v)}$ for an event v. In Section 4.6 we further quantify these effects and show that this increase could be substantial. Thus, to ensure that the differential privacy guarantee is indeed provided with the expected strength defined by ε, it is imperative to design the randomizer to account for the linear constraints among frequencies, which in turn are the result of inherent correlations among software elements.

4.5 Overall Design of Data Collection

Given a set of events $\mathcal{U}$ to be protected and a frequency vector $\boldsymbol{f}$, one can select $\tau = \max_{v \in \mathcal{U}} D_v(\boldsymbol{f})$ to ensure privacy for each element of this set. However, in a differentially-private scheme the value of τ must be selected ahead of time, before the randomizer is embedded in the software and distributed to software users. As a general property, if any parameter of the randomizer (e.g., ε or τ) depends on the local data $\boldsymbol{f}_i$ of an user i, the analysis is not differentially-private for that user. To address this problem, we employ a model with two groups of users: *opt-in users* and *regular users*. Different privacy protections are provided for the two groups. Others have also used similar partitioning of users [6] and this concept has been adopted by widely-used analytics libraries such as Firebase Analytics [39].

Opt-in Users. Each opt-in user i sends to the server the set of pairs $\langle v, D_v(\boldsymbol{f}_i)\rangle$ for all v that are observed at run time. This does leak some information to the server, and in general does not provide differential privacy for that user. However, we consider this a reasonable compromise since the raw profile $\boldsymbol{f}_i$ is not shared with anyone; only its coarse-grain characterization via this set of pairs becomes known to the server. Given this information, for each v reported by users the server computes $\tau(v) = \max_i D_v(\boldsymbol{f}_i)$. The set of $\tau(v)$ is sorted in ascending order and then used to select a set of events $\mathcal{U}$ covering the first $h\%$ of this sorted list. The largest $\tau(v)$ in this set is selected as the final value of τ that will be used later by regular users. The intent behind this approach is to calibrate the privacy/accuracy trade-offs by selecting the value of h and then determining which $h\%$ events can be protected with the smallest possible τ (i.e., the highest possible accuracy).

Regular Users. After the server computes τ from the opt-in users, this value is embedded in the randomizer which is then distributed to the regular users. For any regular user i, $R(\boldsymbol{f}_i)$ is sent to the server and used to compute the calibrated frequencies $\bar{\boldsymbol{F}}$ described in Section 4.3.5. One detail is that it is possible for some $D_v(\boldsymbol{f}_i)$ to exceed the bound τ. In this case, the user's data for v is obtained with weakened privacy protection. Formally, the randomizer achieves ε'-indistinguishability, where $\varepsilon' = \frac{D_v(\boldsymbol{f}_i)}{\tau}\varepsilon$. An alternative would be to exclude the data for v for such users from the data collection.

4.6 Evaluation

To empirically evaluate the proposed approach and its performance for privacy protection of event presence and hotness, we use the same set of 15 Android applications that were also used in Chapter 3. As in Section 3.4, we simulate 1000 users interacting with each app using the Monkey tool [40]. During this process, we collect method execution frequency information as follows. We instrument the entry of every method in each app to record the run-time execution of every method and accumulate its frequency in a local frequency vector at each user. We stop a simulation when the total frequency in the vector reaches $k = 5 \times |\mathcal{V}|$. As a result, we get 1000 run-time profiles of method frequencies for each experimental subject. Table 4.1 shows the details of the subjects used in our experiments. Column "Classes" and "Methods" list the number of application classes and methods, excluding

App	Classes	Methods	*k*
barometer	378	2237	11185
bible	1087	5340	26700
dpm	271	1362	6810
drumpads	446	1903	9515
equibase	251	1975	9875
localtv	714	3055	15275
loctracker	197	837	4185
mitula	966	7172	35860
moonphases	165	716	3580
parking	370	1649	8245
parrot	1235	7433	37165
post	1087	5340	26700
quicknews	1087	5340	26700
speedlogic	77	265	1325
vidanta	1568	9242	46210

Table 4.1: Experimental subjects.

several well-known third-party Android libraries, e.g., `dagger` and `okio`. The threshold *k* for each app is shown in column "*k*" in the table.

The instrumentation is based on Soot [86]. The profile-gathering mechanism utilizes Android Debug Bridge [38] to communicate with and manage multiple instances of emulators for user simulation. Given the collected profiles, we run all randomization on a local machine separately from the executions that gather run-time frequencies. This allows us to conduct each experiment for multiple trials to report rigorous statistical results that account for the randomness introduced by local randomizers [36].

The benchmarks and code for the evaluation in the rest of this section are released to the public at `https://presto-osu.github.io/dp-freq-prof`.

4.6.1 Comparison Between Two Randomizers

To compare the accuracy of the two approaches based on randomized response and the Laplace mechanism discussed in Section 4.3, we fix the privacy parameter $\varepsilon = 1$, which has been used in evaluation of other differentially-private analyses [9], and alter the value of threshold τ for defining neighbors. We consider $\tau \in \{10^0, 10^1, 10^2, k\}$ and compute the vectors containing calibrated frequency estimates for the two approaches. We use the same metric as in Chapter 3, *relative error* (RE, "error" for short) with respect to the ground-truth frequencies $\boldsymbol{F}$, to measure the accuracy of the estimated frequency vectors. Formally, given a frequency vector $\boldsymbol{x}$, its RE is defined as

$$\mathrm{RE} = \frac{\sum_{v \in \mathcal{D}} |F(v) - x(v)|}{\sum_{v \in \mathcal{D}} F(v)}$$

where $\mathcal{D} = \mathcal{V}$ and the denominator in this case is k. Note that the value of RE is $\frac{2}{k} \times d(\boldsymbol{F}, \boldsymbol{x})$. The worst-case value for the error is 2, e.g., when $\boldsymbol{F} = \begin{pmatrix} k & 0 \end{pmatrix}$ and $\boldsymbol{x} = \begin{pmatrix} 0 & k \end{pmatrix}$ for a $\mathcal{V}$ with two events, where the distance for the two vectors is $d(\boldsymbol{F}, \boldsymbol{x}) = \frac{1}{2}(|0-k| + |k-0|) = k$.

Figure 4.3 shows the measurements of the error for the randomized-response-based and Laplace-mechanism-based randomization for different values of τ. Unless otherwise specified, each experiment in this and the following sections is repeated 30 times with the mean and 95% confidence interval of the results reported. The confidence intervals are typically very small, and barely noticeable in the figures.

Summary of Results. As can be seen from these experiments, *randomization based on the Laplace mechanism outperforms the randomized-response-based approach in all cases*, especially for small values of τ. In particular, the error for the Laplace mechanism is 39.8$\times$ smaller than the error for the randomization based on randomized response when $\tau = 10^0$, 16.6$\times$ smaller for $\tau = 10^1$, and 5.9$\times$ smaller for $\tau = 10^2$, averaged across all apps. With the

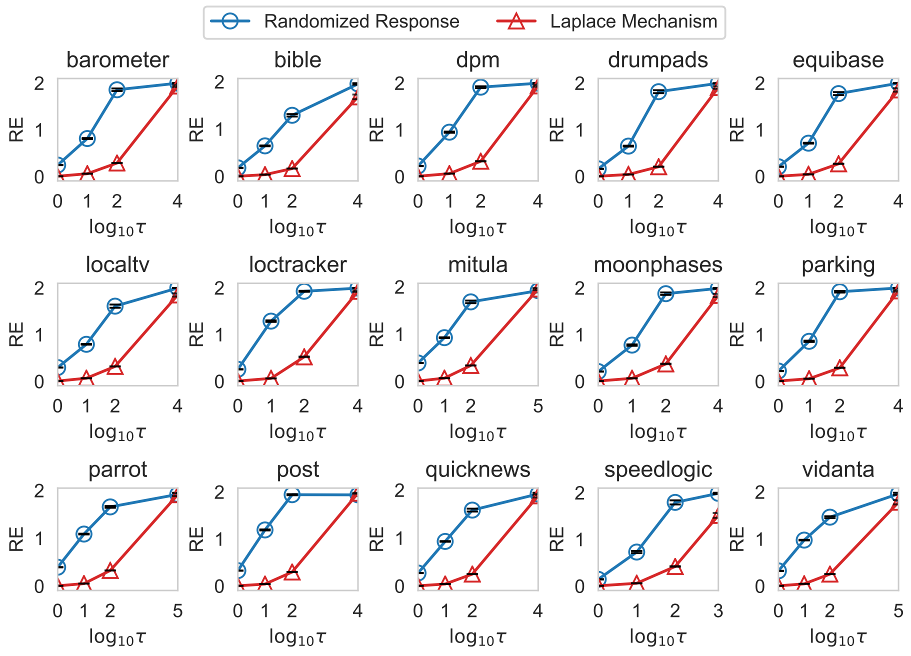


Figure 4.3: Comparison of accuracy using randomization based on randomized response and on Laplace mechanism with $\varepsilon = 1$.

increase of τ, both randomizers require more extensive randomization in order to achieve differential privacy, and thus result in lower accuracy, which is to be expected.

Our conclusions is that instead of aiming for privacy protection for event traces, as done in Chapter 3, a design for privacy at the level of frequency vectors via the Laplace mechanism achieves significant accuracy benefits. This observation is important for profiling frameworks for deployed software. It also raises interesting questions about potential redesign of widely-used software analytics frameworks such as Google Firebase Analytics [39] and Facebook Analytics [34]. The current design of these frameworks provides logging mechanisms at the level of individual events, as illustrated by the code example in Section 4.3.4. It may be desirable to provide alternatives at a higher level of abstraction—e.g., a frequency vector that accumulates the effects of several events—since stronger privacy protections are possible at this case, and such protections could be implemented as optional functionality by the framework. Given the widespread use of these tracking frameworks in many thousands of popular mobile apps [33], this is a worthwhile direction of future investigations.

4.6.2 Hiding The Presence of Events

In this and the following subsection, we evaluate the accuracy of data collection and analysis introduced in Section 4.5 that allows the protection of each event from a set $\mathcal{U} \subseteq \mathcal{V}$, where $\mathcal{U}$ is selected depending on a threshold h and the difficulties reported by opt-in users. We first consider the goal of hiding the presence of individual events. Recall from Section 4.4 that each opt-in user i computes her difficulty $D_v(\boldsymbol{f}_i)$ for each event v such that $f_i(v) > 0$ using an analysis of the constraint graph. We run this computation on a PC with Xeon E5 2.2GHz CPU and 64GB RAM. The average cost of running the algorithm is 6.6 ms per user to calculate $D_v(\boldsymbol{f_i})$ for all qualified events v.

After collecting the difficulties from opt-in users, for each reported event v we compute $\tau(v) = \max_i D_v(\boldsymbol{f}_i)$ and set the final value of τ based on a threshold h such that *the presence of at least h% events is hidden*, i.e., τ covers the first h% of the sorted list of all $\tau(v)$ values. This τ is then used for the actual collection of frequency data from regular users. In the experiments, we consider threshold values $h \in \{25, 50, 75, 100\}$ to evaluate the accuracy of the proposed profiling under different protection goals. Higher threshold values provide protection for more events, lead to higher values for τ, and are expected to produce more error. To select opt-in users, we have used various settings and observed that using 10% of all users as the opt-in user group provides a reasonable τ and only a few users from the remaining 90% regular users have local difficulty exceeding this τ. Specifically, on average

h	RE		
	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
25	0.054	0.024	0.014
50	0.138	0.078	0.042
75	0.296	0.194	0.130
100	1.834	1.586	1.292

Table 4.2: Average RE across all apps for hiding event presence.

for each event, 10.5% users have difficulty $> \tau$ given $h = 25$. The percentage is 6.6% for $h = 50$, 2.9% for $h = 75$, and, of course, 0% for $h = 100$.

Table 4.2 shows the average error of the proposed approach over all apps for different values of ε and h. We have used various values for ε in the experimental evaluation. Here we only show the results for $\varepsilon \in \{0.5, 1, 2\}$ since we observe similar trends for other values. We can see that *despite the inherent trade-offs between accuracy and privacy, high accuracy can be achieved simultaneously with a high protection goal (large h) and strong indistinguishability (small ε).* In particular, the error is below 0.15 if we want to protect the presence of at least a quarter or a half of all events, i.e., $h = 25$ and $h = 50$, even with very strong indistinguishability, e.g., $\varepsilon = 0.5$. When the protection goal is raised to hide the presence of 75% of events, we still have relatively low error (≤ 0.3) for small values of ε. For example, the error is 0.194 when $\varepsilon = 1$. However, when $h = 100$, i.e., the protection goal is to protect the presence of all events, the error is above 1 for all choices of ε. This is to be expected, since some events have high frequencies across all users and large amounts of noise are needed to hide their presence.

Figure 4.4 shows more detailed measurements from these experiments. The analysis produces more accurate results for each app with larger ε since the indistinguishability

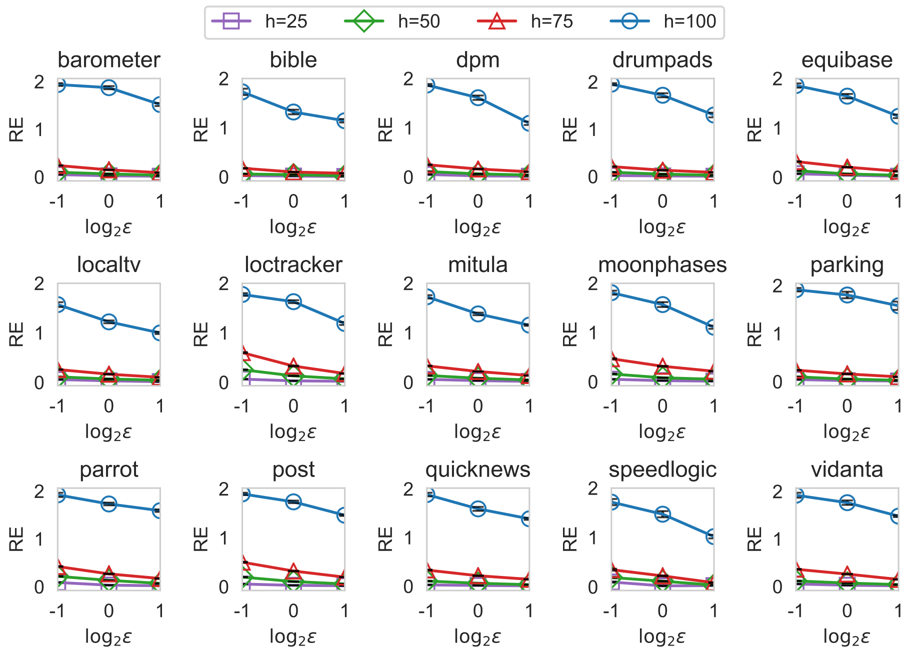


Figure 4.4: Error for hiding event presence with varying h and ε.

strength in Definition 4.2 is reduced, leading to less noise. This conforms with the theoretical guarantee provided by the definition. As discussed before, when we enhance the protection to hide more events by increasing the value of h, there is more error in the resulting frequency estimate vector. For example, compared to $h = 25$, the error is $3.3\times$ larger for $h = 50$ and $8.9\times$ larger when $h = 75$, averaged across all apps for $\varepsilon = 1$. If we want to protect the presence of all events, i.e., $h = 100$, the result is essentially unusable as the error is close to the worst-case value 2.

Accuracy for Frequently-Executed Events. Recall that for the experiments from Chapter 3 the inaccuracy was mainly caused by a large number of infrequently-executed methods

h	RE			HMC		
	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
25	0.0012	0.0005	0.0003	1	1	1
50	0.0043	0.0020	0.0009	0.9989	1	1
75	0.0148	0.0069	0.0034	0.9938	0.9989	1
100	0.8104	0.5602	0.2804	0.1895	0.5270	0.7420

Table 4.3: Average RE and HMC for hiding the presence of frequently-executed events given $\ell = 0.25$.

with low frequencies. Here we also expect that the frequency estimates for the frequently-executed methods are significantly more accurate than those for infrequently-executed ones. To quantify these observations, we compute the RE for only frequent methods and the *hot method coverage* (HMC, defined by Equation 3.4) under different values of parameters, similarly to the experimental evaluation in Chapter 3. Table 4.3 shows the average values of RE and HMC across all apps using $\ell = 0.25$. We can see that the error is orders-of-magnitude less for frequent events, compared to the corresponding values in Table 4.2. For example, when $h = 75$ and $\varepsilon = 1$, the RE is 28.1$\times$ smaller. The coverage of these methods are also very high, approaching the optimal value 1, for relatively large h, e.g., $h = 75$, even under small values of ε such as $\varepsilon = 0.5$. In the experiments, we have also observed similar trends for other values of ℓ.

Summary of Results. These experimental results are promising. Despite the fundamental tension between privacy (which requires higher magnitude of noise) and accuracy (which is reduced by this noise), we see evidence of a "sweet spot". In particular, our measurements demonstrate that high-accuracy estimates can be achieved together with privacy protection of

h	RE		
	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
25	0.084	0.044	0.024
50	0.198	0.118	0.066
75	0.442	0.286	0.178
100	1.838	1.596	1.292

Table 4.4: Average RE across all apps for hiding event hotness.

the presence of most methods (i.e., high h) and strong differential privacy indistinguishability (i.e., low ε).

4.6.3 Hiding The Hotness of Events

In this section we evaluate the effectiveness of the proposed approach for hiding the "hotness" of events. Recall from Section 4.4.4 that an event v is hot in a frequency vector $\boldsymbol{f}$ if $f(v) > \eta$ where η is a pre-defined threshold. Here we set $\eta = \frac{k}{|\mathcal{V}|}$. We have used other values of η and see similar results. As in the experiments from the previous subsection, we consider four definitions of set $\mathcal{U}$, based on a threshold h. For each reported event v we compute the largest value of the hotness-hiding difficulty $D_v(\boldsymbol{f}_i)$ across all opt-in users. We then set the final τ value to hide the hotness of at least h% of these events for $h \in \{25, 50, 75, 100\}$. A summary of the average error over all apps is shown in Table 4.4. The detailed results for each app are shown in Figure 4.5.

We can see that the error is low (< 0.2) for all values of ε used in the experiment when the protection goal is low, e.g., when we provide protection for half of the hot methods. When the value of h is increased to 75, the analysis can provide accurate results with somewhat larger values of ε. For example, we have error < 0.3 using $\varepsilon = 1$, which is a

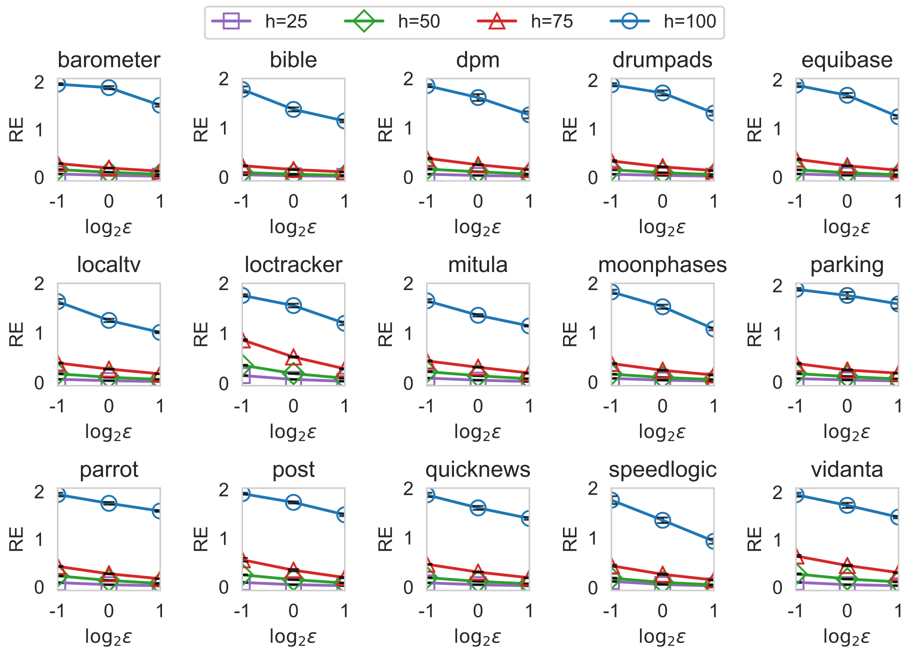


Figure 4.5: Error for hiding event hotness with varying h and ε.

typical value for this parameter in practice. At this protection level, the error is on average 2.4× and 6.5× larger than the error for $h = 50$ and $h = 25$, respectively. We reach the highest error when $h = 100$, similarly to the results from Section 4.6.2. This is due to the large frequencies of some extremely hot methods. One example of such a method is `DownloadProgress.onProgressUpdate` in the `dpm` app, which will be invoked multiple times to draw the progress bar when the user is downloading preset sound tracks. The randomizer has to introduce considerable amount of noise to hide the hotness of such methods.

The average RE and HMC for frequently-executed methods are listed in Table 4.5. The results show similar trends as described in the previous subsection. The error caused

h	RE			HMC		
	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
25	0.0021	0.0010	0.0005	1	1	1
50	0.0061	0.0029	0.0015	0.9978	1	1
75	0.0223	0.0106	0.0051	0.9845	0.9956	1
100	0.8329	0.5599	0.2876	0.1625	0.5318	0.7292

Table 4.5: Average RE and HMC for hiding the hotness of frequently-executed events given $\ell = 0.25$.

by frequent events is significantly smaller than the overall error in Table 4.4—that is, the estimates are much more accurate for these events.

Summary of Results. The conclusions from these experimental results are similar to the ones from Section 4.6.2. For many (but not all) hot methods, our technique can effectively hide their hotness while at the same time providing high-accuracy estimates. Together with the earlier results for hiding event presence, this evaluation provides evidence of the promise of differentially-private analyses for remote profiling of deployed software.

4.6.4 Implications of Enforcing Linear Constraints

Section 4.4.5 discussed the implications of including or excluding the linear constraints $\boldsymbol{Af} \geq \boldsymbol{b}$ in the randomizer design. We empirically evaluated the effects on the privacy guarantee for these two scenarios and show the implicit degradation when the randomizer does not consider the constraints. For each of the two scenarios we computed the value of τ needed to hide the presence of individual events given $h = 25$, as was done for the previous experiments. (We have computed similar results for other values of h and observed even larger degradation.) We denote the two values by τ_{true} and τ_{false}, where the subscript indicates whether or not the constraints are taken into account during the computation of difficulties. We then calculate the degradation of the privacy guarantee as follows:

$$\text{Degradation}(\tau_{true}, \tau_{false}) = e^{\frac{\tau_{true}}{\tau_{false}}}$$

This definition shows the extent to which the bound on the ratio of the two probabilities in Definition 4.2 implicitly increases if the randomizer simply ignores the constraints. The mean value of this degradation for 30 independent repetitions of the same experiment is 26.5, averaged across all apps. This means that the probability bound is increased by a factor of 26.5 on average, by which the indistinguishability between neighboring frequency vectors is significantly weakened.

4.7 Summary

Differential privacy is a desirable theoretical framework for designing privacy-preserving software analysis, due to its inherently strong and rigorous privacy guarantees. When such machinery is applied to software profiling problems, or more generally to dynamic program analysis, it is important to consider domain-derived insights into the analyzed data. For the profiling problem we consider such data relationships are represented via linear constraints on frequencies. As we discuss theoretically and demonstrate experimentally, such constraints must be accounted for when designing a randomization scheme, in order to achieve the expected privacy protections.

For the specific problem of frequency estimation, we show that randomization of frequency vectors is more suitable than event trace randomization. This observation has implications for the design of mobile app analytics frameworks such as Google Firebase Analytics and Facebook Analytics, as well we for any infrastructure for remote profiling of deployed software. Even with this design choice, it is still a challenge to select the parameters of the randomization in order to achieve desired trade-offs between accuracy and privacy. We show how to address this problem by defining and computing the difficulty of protecting certain events. Our results demonstrate that with careful application of this approach, it is possible to achieve both high privacy and high accuracy for the target goals of this work.

Chapter 5: Differentially-Private Control-Flow Node Coverage Analysis

The approach introduced in Chapter 4 takes into account the linear constraints of frequency vectors in the design and configuration of the randomization. As we argued in that chapter, it is important to account for such intrinsic constraints. Otherwise, a seemingly-strong privacy guarantee is actually weaker than it appears. In this chapter, we focus on a new type of problem which is related to *control-flow node coverage*, where the type of constraints are with respect to control-flow graph nodes. Although the constraints are different from the ones in Chapter 4, which are related to the frequency of events, they are both based on the structural properties of application code. We carefully define the differential privacy protection of graph nodes and algorithms to achieve it, and conduct extensive experiments to empirically evaluate the effectiveness of the proposed algorithms.

Specifically, we propose a new notion of privacy guarantees based on a the neighbor relation between control-flow graphs and a new differentially-private algorithm design based on a novel definition of sensitivity with respect to differences between neighbors. We provide an efficient implementation of the algorithm using *dominator trees* derived from control-flow graphs. To improve the utility of the analysis results, we introduce a hybrid approach similar to the one described in Section 4.5 to aggregate a bound of sensitivity, a pruning approach to reduce the noise level by tightening the sensitivity bound using restricted sensitivity, and a refined notion of relaxed indistinguishability based on distances between neighbors. Our evaluation demonstrates that these techniques can achieve practical accuracy while providing principled privacy-by-design guarantees.

An alternative version of this work [104] will appear at the USENIX Security Symposium 2020. There are some minor differences in theoretical machinery and experimental evaluation between that work and what is described in this chapter. However, the core ideas, major algorithm design decisions, and conclusions from the experimental evaluation are the same.

5.1 Problem and Motivation

For the purposes of this chapter, each relevant event triggered at run time corresponds to the execution of a specific component in the software. Thus, the triggering of events can be characterized by the run-time *control flow* with respect to the execution of software components. Privacy is needed to protect sensitive control-flow data. Consider the following example:

```
if (sensitive condition) a();

void a() { b(); }
```

If the program execution reveals that function `a` was invoked at run time, an adversary can infer that the sensitive condition were true. Furthermore, this inference could be indirect: for example, even if the invocation of `a` were obfuscated, revealing that function `b` was executed could also be used to infer the condition. Many analytics platforms, including Facebook [34], Firebase [39], and Flurry [73], allow developers to gather raw control-flow data by collecting remotely users' interactions for data analysis and more complex tasks such as machine learning. Figure 5.1 illustrates this process. Each directed graph on the

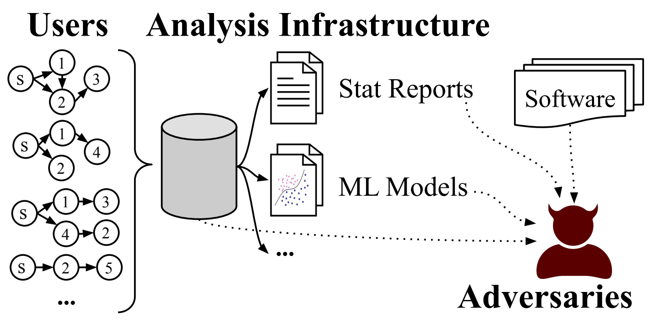


Figure 5.1: User interactions are reported for data analysis.

left represents the control-flow behavior of a user's copy of the software. A node represents a software component that corresponds to an event, and an edge represents control flow between components. When a user interacts with her copy, her actions trigger a particular control-flow graph instance which is cached locally and eventually sent to remote servers for data analysis.

The above problem can be abstracted as the problem of collecting *control-flow graph node coverage* information over many instances of a software application. It could be instantiated at various levels of granularity: a graph node could represent a coarse-grained software component, a GUI element, a function in the application code, or an individual code statement. We aim to develop a *privacy-preserving* solution for this problem. Specifically, our goal is to introduce a *differential privacy* (DP) mechanism that, in a principled and quantifiable manner, hides the presence/absence of any particular graph node in a user's coverage information. In essence, our solution helps a software user to hide from others whether any component of the software, represented by a graph node, was executed by this user. One of the key technical contributions of this chapter is a novel privacy definition that accounts for the intrinsic constraints between graph nodes, based on the structure of the control-flow graph.

The motivation for this privacy-preserving analysis stems from two factors. The coverage information itself may reveal sensitive conditions, for example, whether the user has executed security-related functionality such as changing a password or connecting to a VPN. Furthermore, user habits can be mined from such data for the purposes of behavior analytics. The power of such data mining continues to increase, by combining user data from multiple sources to draw even-more-powerful inferences. Neither software users nor software developers can anticipate all future uses of such information for mining of many seemingly-unrelated data streams generated by the same user. Proactive protection against unknown future uses (and misuses) is a desirable high-level goal that benefits not only the users of the software but also its developers, who can claim with confidence that they provide proactive, principled, and quantifiable privacy protections.

5.1.1 Problem Statement

In many software analysis problems, a control-flow model is instantiated at run time when the software is executed. Examples of such models include statement-level control-flow graphs, call graphs, calling context trees, and GUI screen transition models. Generally, such a model is a directed graph $G = (\mathcal{V}, \mathcal{E}, s)$ with node set $\mathcal{V}$ and edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The start node $s \in \mathcal{V}$ represents the start of any run-time execution and the root of G.

When the software is executed, run-time events correspond to dynamic instances of graph nodes and edges. For example, if G is the program's call graph, run-time event "function m_i calls function m_j" corresponds to a dynamic instance of graph edge $m_i \to m_j$. We use G_c to denote the subgraph of G defined by these run-time-covered nodes and edges.

Here $\boldsymbol{c} \in \{0,1\}^{|\mathcal{V}|}$ is the indicator vector of the corresponding set of covered nodes during the program run. In a minor abuse of notation, we will use $\boldsymbol{c}$ to denote both a set of covered nodes and its corresponding coverage vector. *Node coverage analysis* reports the set of nodes in $\boldsymbol{c}$.

Information about node coverage plays an important role in the area of mobile and web apps, using analysis infrastructures such as Google Analytics [37] and Facebook Analytics [34]. For example, Google Analytics presents to developers reports of histograms of events about the population of users who have executed them. Such information is also essential for various software monitoring tasks. For instance, residual coverage monitoring [78] cumulatively collects and calculates the basic block coverage in the control-flow graph of a program. In general, many analyses of deployed software depend on some form of control-flow coverage information [5, 14, 18, 20, 31, 45–48, 57, 58, 75, 77, 87, 105].

5.1.2 Differentially-Private Node Coverage Analysis

Consider n software users identified by integer ids $i \in \{1,\dots,n\}$. All users run the same software, which has some publicly known control-flow model G. This model would typically be constructed by the software developers for their own analytics needs. The deployed software would contain instrumentation to record and report events related to run-time coverage of G. We consider G to be publicly known, as an adversary could reverse engineer this model from the code of the deployed software using a wide range of existing techniques.

The node coverage $\boldsymbol{c}_i \in \{0,1\}^{|\mathcal{V}|}$ of user i describes the run-time behavior of that user's instance of the software. In node coverage analysis, the software developer's goal is to determine, for each node $v \in \mathcal{V}$, the frequency of v's coverage across all users—that is,

$F(v) = |\{i \in \{1,\ldots,n\} \mid v \in \boldsymbol{c}_i\}|$. Equivalently, the goal is to obtain an aggregate vector $\boldsymbol{F} \in \mathbb{N}^{|\mathcal{V}|}$ such that $\boldsymbol{F} = \sum_i \boldsymbol{c}_i$, where the summation is element-wise for vectors $\boldsymbol{c}_i$. In a differentially-private setting, instead of $\boldsymbol{F}$ the developer will obtain an estimated aggregate vector $\hat{\boldsymbol{F}}$ where, with high probability, the node frequency estimates $\hat{F}(v)$ are close to the actual node frequencies $F(v)$. This analysis provides information about how users of the deployed software interact with it—for example, how many users have accessed a particular screen in an app's GUI, which is a typical concern in mobile app analysis via infrastructures such as Firebase Analytics [39]. As another example, gathering data about which code regions are executed by software users provides rich feedback to software developers and helps them validate and refine assumptions they have used in pre-deployment testing and validation [78].

An LDP coverage analysis applies an ε-local randomizer $R : \{0,1\}^{|\mathcal{V}|} \rightarrow \{0,1\}^{|\mathcal{V}|}$ to each user's observed coverage $\boldsymbol{c}_i$. The resulting $\boldsymbol{z}_i = R(\boldsymbol{c}_i)$ is sent to the server. The server collects all $\boldsymbol{z}_i$ and uses them to compute the estimates $\hat{\boldsymbol{F}}$.

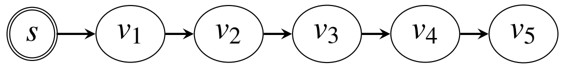
5.2 Feasibility and Neighbors

Differentially-private analysis of graph data has been considered almost exclusively in the centralized model of DP [50, 52, 72, 83]. Two graph privacy definitions have been proposed. *Node privacy* [83] considers the indistinguishability of two undirected neighbor graphs G and G', where G' can be obtained from G by deleting one node and all its adjacent edges. A node-private analysis provides plausible deniability about the presence of any particular node in the graph. More precisely, for any graph G, if an adversary observes the randomized output $R(G)$, the probability that the input to the randomizer R was G is very close (by a factor of e^{ε}) to the probability that the input to R was any neighbor of G in which one node of G was removed (together with its adjacent edges). Thus, the adversary cannot conclude with high probability that any graph node was actually present in the protected private graph. An alternative weaker notion of privacy is *edge privacy* [50, 72], which obfuscates the presence of any graph edge. Node privacy provides stronger protection, but achieving high accuracy for node-private analyses is inherently more difficult than for edge-private ones [83]. For our problem of collecting control-flow node coverage, we will focus on the more challenging node privacy.

One key question for achieving such node privacy is the definition of neighbors. Using the traditional notion from DP graph analysis [83], a neighbor graph $G_{\boldsymbol{c}'}$ would be obtained from a given $G_{\boldsymbol{c}}$ by removing a single node and its adjacent edges. Thus, coverage vector $\boldsymbol{c}'$ would differ from $\boldsymbol{c}$ by a single bit. However, this notion is meaningless for control-flow graphs and their coverage vectors, since not all vectors represent *feasible* run-time behaviors—that is, we will never observe them during execution. We define this key property of feasibility as follows:

Definition 5.1 (Feasibility)**.** *A dynamic graph $G_{\boldsymbol{c}}$ and its coverage vector $\boldsymbol{c}$ are* feasible *if $s \in \boldsymbol{c}$ and every covered node is reachable from s along a path of covered nodes and edges, i.e., for any $v \in \boldsymbol{c}$, there exists a path $\langle s, v_1, \ldots, v_k, v \rangle$ in $G_{\boldsymbol{c}}$ such that $v_j \in \boldsymbol{c}$ for $1 \leq j \leq k$.*

Here $v \in \boldsymbol{c}$ denotes that v is in the set of nodes encoded by $\boldsymbol{c}$. If $G_{\boldsymbol{c}}$ and $\boldsymbol{c}$ do *not* satisfy these properties, there does not exist a run-time execution that could have produced them. To illustrate this point, consider the graph G in Figure 5.2, where s is the start node. Coverage vector $\boldsymbol{c}$ is feasible, as it represents the covered set $\{s, v_1, v_2, v_3, v_4\}$. However, $\boldsymbol{c}'$ which represents $\{s, v_1, v_3, v_4\}$ is not feasible since v_3 and v_4 cannot be reached from s along a path of covered nodes. No software execution can generate $\boldsymbol{c}$ as a coverage vector.



feasible $\boldsymbol{c} = [111110]$
infeasible $\boldsymbol{c}' = [110110]$
feasible $\Delta_{v_2}(\boldsymbol{c}) = [110000]$

Figure 5.2: Feasible and infeasible coverage.

As with traditional DP graph analyses, we consider the removal of a graph node in order to define the notion of a neighbor graph. However, our definition takes into account the feasibility constraint. Given a feasible dynamic $G_{\boldsymbol{c}}$ and some node $v \in \boldsymbol{c} \setminus \{s\}$, the *neighbor graph* $G_{\boldsymbol{c}'} = \Delta_v(G_{\boldsymbol{c}})$ obtained by removing v is defined as follows: (1) $G_{\boldsymbol{c}'}$ is a subgraph of $G_{\boldsymbol{c}}$, (2) $v \notin \boldsymbol{c}'$, (3) $G_{\boldsymbol{c}'}$ is feasible, and (4) $G_{\boldsymbol{c}'}$ is maximal (i.e., there does not exist a proper supergraph of $G_{\boldsymbol{c}'}$ with properties 1–3). Intuitively, the last constraint ensures that we do not remove "too many" nodes and edges from $G_{\boldsymbol{c}}$.

Graph $\Delta_v(G_{\boldsymbol{c}})$ exists and is unique, as shown by Lemma 5.1, followed by its proof.

Lemma 5.1. *Let $G_{\boldsymbol{c}}$ be a feasible dynamic graph. For any $v \in \boldsymbol{c} \setminus \{s\}$, there exists a unique feasible subgraph $G_{\boldsymbol{c}'}$ such that $v \notin \boldsymbol{c}'$ and $G_{\boldsymbol{c}'}$ is maximal.*

Proof. Consider the set of all feasible subgraphs $G_{\boldsymbol{c}'}$ for the given $G_{\boldsymbol{c}}$ such that $v \notin \boldsymbol{c}'$. This set is not empty because it contains, at the very least, the trivial graph containing only the starting node s. Since the set is finite, at least one of its elements is maximal. To show uniqueness, suppose that two different graphs from the set are both maximal. It is easy to see that the graph containing the union of their nodes and edges also belongs to the set, which means that neither of the original two graphs could have been maximal. $\square$

For brevity, we will often use $\boldsymbol{c}' = \Delta_v(\boldsymbol{c})$ to denote that $G_{\boldsymbol{c}'} = \Delta_v(G_{\boldsymbol{c}})$ for a given $G_{\boldsymbol{c}}$. For illustration, in Figure 5.2 the removal of v_2 from $\boldsymbol{c}$ requires the removal of v_2 and v_3 as well, in order to preserve feasibility. Thus, the neighbor $\Delta_{v_2}(\boldsymbol{c})$ is the covered set $\{s, v_1\}$. If one were to use the traditional definition of neighbors described earlier, the removal of v_2 would produce the infeasible vector $\boldsymbol{c}'$ shown in the figure.

The set of neighbors for the coverage vector $\boldsymbol{c}$ of a given $G_{\boldsymbol{c}}$ is defined as follows:

Definition 5.2 (Neighbors)**.** *Given a feasible coverage vector $\boldsymbol{c}$, its* neighbors *are the set* $\{\Delta_v(\boldsymbol{c}) \mid v \in \boldsymbol{c} \setminus \{s\}\} \cup \{\boldsymbol{c}' \mid \exists v \in \boldsymbol{c}' \setminus \{s\} : \Delta_v(\boldsymbol{c}') = \boldsymbol{c}\}$.

This definition considers both the removal of a node v from $\boldsymbol{c}$ (the first term in the formula) and the addition of a node v to $\boldsymbol{c}$ (the second term in the formula) as means of obtaining a neighbor vector. Thus, the neighbor relation is symmetric.

Next, we show that $\Delta_v(\boldsymbol{c})$ for given $G_{\boldsymbol{c}}$ and v can be constructed efficiently. In a control-flow graph with a start node s, a node d *dominates* a node v (denoted d *dom* v) if every path from s to v goes through d [1]. A node trivially dominates itself. Given a feasible $G_{\boldsymbol{c}}$, let $dom_{G_{\boldsymbol{c}}}$ denote its dominator relation. The key observation is that the nodes dominated by v (plus their adjacent edges) are exactly the ones that need to be deleted to obtain the neighbor graph:

Proposition 5.1. *For any node $v \in \boldsymbol{c} \setminus \{s\}$, we have* $\Delta_v(\boldsymbol{c}) = \boldsymbol{c} \setminus \{v' \mid v\ dom_{G_{\boldsymbol{c}}}\ v'\}$.

The proof of the proposition is shown below:

Proof. Consider $G_{\boldsymbol{c}}$ and its subgraph $G_{\boldsymbol{c}'}$ obtained by removing all nodes in $\{v' : v\ dom_{G_{\boldsymbol{c}}}\ v'\}$ and their adjacent edges. We need to show that (1) $v \notin \boldsymbol{c}'$; (2) $G_{\boldsymbol{c}'}$ is feasible; and (3) $G_{\boldsymbol{c}'}$ is maximal. (1) trivially follows from $v\ dom_{G_{\boldsymbol{c}}}\ v$. For (2) we need to establish that in $G_{\boldsymbol{c}'}$, all nodes are reachable from the start node s. Suppose this is not true for some $k \in \boldsymbol{c}'$. Clearly, k is reachable from s in $G_{\boldsymbol{c}}$. Graph $G_{\boldsymbol{c}'}$ is obtained from $G_{\boldsymbol{c}}$ by removing all v' such that $v\ dom_{G_{\boldsymbol{c}}}\ v'$. Thus, every path from s to k in $G_{\boldsymbol{c}}$ contains at least one such v'. Since $v\ dom_{G_{\boldsymbol{c}}}\ v'$, each such path also must contain v. This means that $v \; dom_{G_{\boldsymbol{c}}} \; k$, which contradicts $k \in \boldsymbol{c}'$. Finally, (3) requires that $G_{\boldsymbol{c}'}$ be maximal. Consider some proper supergraph $G_{\boldsymbol{c}''}$ of $G_{\boldsymbol{c}'}$ that is a feasible subgraph of $G_{\boldsymbol{c}}$ and has $v \notin \boldsymbol{c}''$. It is easy to see that $\boldsymbol{c}''$ contains at least one node k such that $v \; dom_{G_{\boldsymbol{c}}} \; k$. Since $\boldsymbol{c}''$ is feasible, there is at least one path from s to k in $G_{\boldsymbol{c}''}$. This path also exists in $G_{\boldsymbol{c}}$, and thus v belongs to it because it dominates k in $G_{\boldsymbol{c}}$. This contradicts $v \notin \boldsymbol{c}''$. □

This property allows us to find efficiently all $\Delta_v(\boldsymbol{c})$ for a given $G_{\boldsymbol{c}}$, which is needed for our randomizer (as described later). Consider the *dominator tree* for $G_{\boldsymbol{c}}$, which is a standard representation of the dominator relation. For any node, the set of its ancestors in the tree is exactly the set of its dominators. For the simple $G_{\boldsymbol{c}}$ in Figure 5.2, the dominator tree is the same as the graph itself (root s dominates all nodes, v_1 dominates all nodes except s, etc.). The dominator tree can be constructed efficiently; we use a classic approach by Lengauer and Tarjan [55] with complexity $O(|\mathcal{E}| \log |\mathcal{V}|)$. Given $v \in \boldsymbol{c} \setminus \{s\}$, the dominator subtree rooted at v provides all and only nodes that should be removed from $\boldsymbol{c}$ to obtain its neighbor $\Delta_v(\boldsymbol{c})$.

As discussed shortly, our randomizer only needs to consider the size of set $\Delta_v(\boldsymbol{c})$ rather than the actual nodes in it. A linear-time bottom-up traversal of the dominator tree for $G_{\boldsymbol{c}}$ can annotate each node v with the size $sub_{G_{\boldsymbol{c}}}(v)$ of the subtree rooted at that node. Thus, given any v, we can easily obtain $|\Delta_v(\boldsymbol{c})|$ as $|\boldsymbol{c}| - sub_{G_{\boldsymbol{c}}}(v)$.

5.3 LDP Analysis

Consider again our problem: for user i, coverage vector $\boldsymbol{c}_i \in \{0,1\}^{|\mathcal{V}|}$ describes the behavior of that user's code instance. The same local randomizer $R : \{0,1\}^{|\mathcal{V}|} \to \{0,1\}^{|\mathcal{V}|}$ is used by all users. Each user reports $R(\boldsymbol{c}_i)$ to the analysis infrastructure. All reports are gathered and post-processed to construct an estimate of $\boldsymbol{F} = \sum_i \boldsymbol{c}_i$. Such analysis, based on Definition 5.2, can achieve control-flow graph node privacy as follows:

Definition 5.3 (ε-Node-LDP)**.** *Randomizer R is* ε-node-LDP *if for any pair of coverage vector neighbors $\boldsymbol{c}, \boldsymbol{c}'$ for G from Definition 5.2, we have*

$$\frac{\Pr[R(\boldsymbol{c}) = \boldsymbol{z}]}{\Pr[R(\boldsymbol{c}') = \boldsymbol{z}]} \leq e^{\varepsilon}$$

5.3.1 Difficulty of Hiding Graph Nodes

To define analyses that satisfy Definition 5.3, it is important to consider the distances between a given feasible graph/vector and all its neighbors. The goal of the randomizer is to "obfuscate" such distances to ensure their indistinguishability. The notion of distance here also implies the *difficulty* of the obfuscation by the randomizer. If the distances are large, they are hard to obfuscate and substantial obfuscation has to be applied. Given a feasible coverage vector $\boldsymbol{c}$ and a node v, the difficulty of hiding v in $\boldsymbol{c}$ is $D_v(\boldsymbol{c}) = |\boldsymbol{c}| - |\Delta_v(\boldsymbol{c})|$, i.e., the distance between $\boldsymbol{c}$ and its nearest feasible neighbor vector $\Delta_v(\boldsymbol{c})$ which has 0 for v.

When we consider a set of nodes to be protected, one has to consider the largest difficulty for nodes in the set. For the rest of this chapter, we would like to protect all nodes in G except for the start node (since the start node is always present at run time). It is straightforward to extend our solution to protect a subset $\mathcal{U} \subseteq \mathcal{V}$, as was done in Chapter 4. The difficulty in this case is $\max_{v \in \boldsymbol{c} \setminus \{s\}} D_v(\boldsymbol{c})$. It turns out that this is exactly the same as the concept of *local sensitivity*, as defined below, which is employed in various forms by many DP analysis algorithms. In our analyses, we will use it to capture the properties of the "graph neighbor" relation defined earlier.

Definition 5.4 (Local Sensitivity)**.** *Consider a feasible graph $G_{\boldsymbol{c}}$ and its corresponding coverage vector $\boldsymbol{c}$. The* local sensitivity *of $\boldsymbol{c}$ is*

$$LS(\boldsymbol{c}) = \max_{v \in \boldsymbol{c} \setminus \{s\}} D_v(\boldsymbol{c}) = \max_{v \in \boldsymbol{c} \setminus \{s\}} |\boldsymbol{c}| - |\Delta_v(\boldsymbol{c})|$$

$LS(\boldsymbol{c})$ captures how sensitive $G_{\boldsymbol{c}}$ is to the removal of any of its nodes v. Since the "neighbor" relation from Definition 5.3 is symmetric, the sensitivity of adding a node v to $G_{\boldsymbol{c}}$ will be accounted for by $LS(\boldsymbol{c}')$ for another coverage vector $\boldsymbol{c}'$ such that $\boldsymbol{c} = \Delta_v(\boldsymbol{c}')$.

Intuitively, the larger the local sensitivity, the more extensive randomization needs to be added by R in order to satisfy Definition 5.3, since the randomized output has to "hide" the differences between $\boldsymbol{c}$ and any $\Delta_v(\boldsymbol{c})$. This increased randomization is manifested by an increased probability of flipping any bit in the coverage vector.

Example 5.1. In Figure 5.2, consider $c = \{s, v_1, v_2, v_3, v_4\}$. We have $\Delta_{v_4}(\boldsymbol{c}) = \{s, v_1, v_2, v_3\}$, $\Delta_{v_3}(\boldsymbol{c}) = \{s, v_1, v_2\}$, $\Delta_{v_2}(\boldsymbol{c}) = \{s, v_1\}$, and $\Delta_{v_1}(\boldsymbol{c}) = \{s\}$. The local sensitivity is $LS(\boldsymbol{c}) = |\boldsymbol{c}| - |\Delta v_1(\boldsymbol{c})| = 4$.

Given $G_{\boldsymbol{c}}$, computing $LS(\boldsymbol{c})$ is straightforward. Recall that, with the help of Proposition 5.1, we can efficiently find all $\Delta_v(\boldsymbol{c})$ by considering the dominator tree for $G_{\boldsymbol{c}}$. Suppose each node v in this tree is annotated with the size $sub_{G_{\boldsymbol{c}}}(v)$ of the subtree rooted at v. Then $LS(\boldsymbol{c})$ is the largest value of $sub_{G_{\boldsymbol{c}}}(v)$ among the nodes v that are children of the start node s in the tree.

Example 5.2. Figure 5.3 and 5.4 show an example from the parking Android app [92]. This app navigates users to parking places, records history of parking locations, and reminds users about parking time. It uses Google Analytics [37] to collect GUI screen view events from users. The developer defines a dictionary of GUI screens to be collected and reported to the Google Analytics remote servers. Figure 5.3 shows the control-flow model G for this

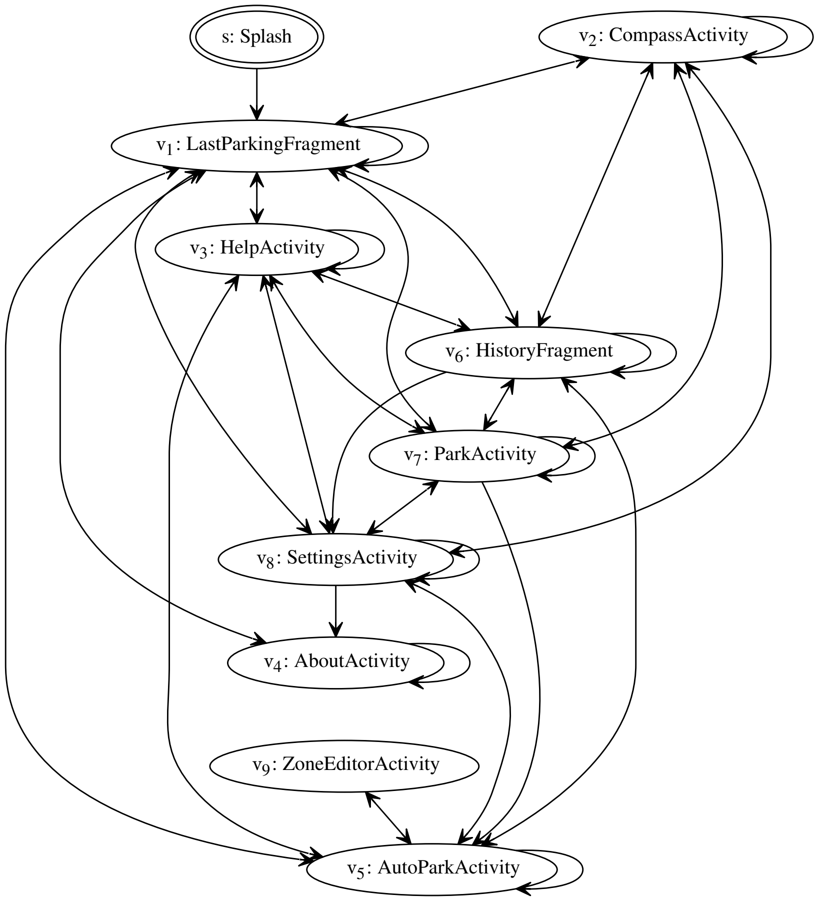
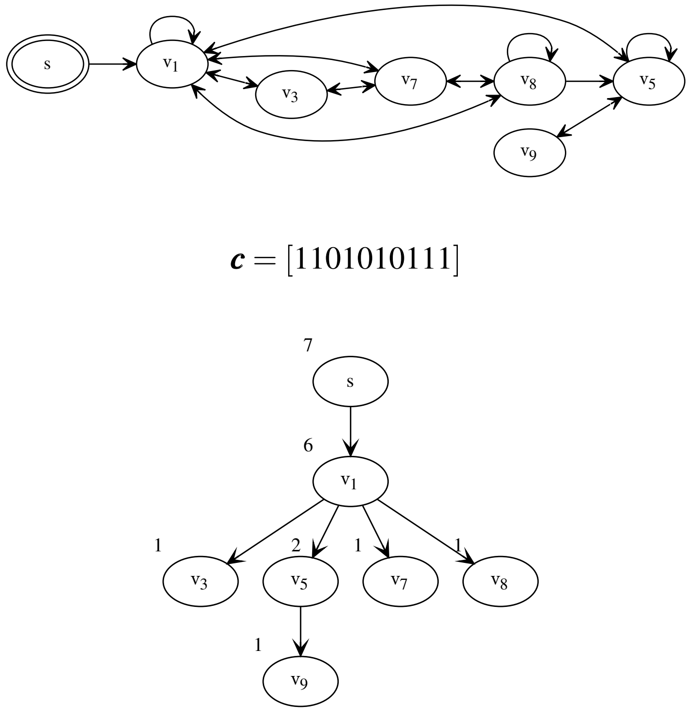


Figure 5.3: GUI screen view graph G from the parking app.

app, with nodes corresponding to different screens and edges showing possible transitions between screens. Consider the run-time behavior of one app user, corresponding to graph $G_{\boldsymbol{c}}$ and its coverage vector $\boldsymbol{c} = [1101010111]$. The graph and its dominator tree are shown in Figure 5.4. Each node v in the tree is annotated with $sub_{G_{\boldsymbol{c}}}(v)$, the size of its corresponding subtree. The local sensitivity for $G_{\boldsymbol{c}}$ is $LS(\boldsymbol{c}) = sub_{G_{\boldsymbol{c}}}(v_1) = 6$.



$$\boldsymbol{c} = [1101010111]$$

Figure 5.4: $G_{\boldsymbol{c}}$, coverage vector $\boldsymbol{c}$, and dominator tree for $G_{\boldsymbol{c}}$, from the parking app.

5.3.2 Randomizer Definition

Suppose we know an *upper bound* S of $LS(\boldsymbol{c})$ for all possible feasible $\boldsymbol{c}$ for a given graph G. The randomizer R can be defined as follows:

Definition 5.5 (Randomizer)**.** *Given a feasible* $\boldsymbol{c}$*, R independently flips each bit in* $\boldsymbol{c}$ *with probability*

$$p = \frac{1}{1+e^{\frac{\varepsilon}{S}}}$$

Then the following proposition holds:

Proposition 5.2. *The randomizer R from Definition 5.5 satisfies ε-node-LDP.*

The proof of the proposition is as follows:

Proof. Let $\boldsymbol{c}$ be a feasible coverage vector. For any $v \in \boldsymbol{c}$ and $\boldsymbol{t} \in \{0,1\}^{|\mathcal{V}|}$, consider the ratio between $\Pr[R(\boldsymbol{c}) = \boldsymbol{t}]$ and $\Pr[R(\Delta_v(\boldsymbol{c})) = \boldsymbol{t}]$. This ratio is bounded from above by the product of x terms $e^{\frac{\varepsilon}{S}}$, where x is the number of bits in $\boldsymbol{c}$ that were changed to obtain $\Delta_v(\boldsymbol{c})$. Each of the x terms is contributed by one of the flipped bits. Since $x \leq S$, this ratio is bounded from above by e^{ε}. Similarly, the ratio is bounded from below by $e^{-\varepsilon}$. Given any neighbors $\boldsymbol{c}, \boldsymbol{c}'$, either $\boldsymbol{c}' = \Delta_v(\boldsymbol{c})$ or $\boldsymbol{c} = \Delta_v(\boldsymbol{c}')$, therefore $\Pr[R(\boldsymbol{c}) = \boldsymbol{t}] \leq e^{\varepsilon}\Pr[R(\boldsymbol{c}') = \boldsymbol{t}]$. Thus, R satisfies Definition 5.3. □

Each user i applies local randomizer R to add noise to local vector $\boldsymbol{c}_i$. (Since we are interested in (estimates of) total node frequencies across all users, and not for individual users, we design R to produce vectors that are not necessarily feasible.) After the remote software analysis infrastructure collects and reports a histogram $\boldsymbol{H} = \sum_i R(\boldsymbol{c}_i)$ over all users, this noisy data is processed to account for the effects of the randomizers. For any node v, the expected value of the number of occurrences of v in $\boldsymbol{H}$ is $F(v)e^{\frac{\varepsilon}{S}}p + (n - F(v))\,p$ where $F(v)$ is real frequency of v, n is the number of users, and p is probability from Definition 5.5. If the collected histogram $\boldsymbol{H}$ has a frequency $H(v)$ for v, then the estimate $\hat{F}(v)$ for the real frequency $F(v)$ is

$$\hat{F}(v) = \frac{\left(1 + e^{\frac{\varepsilon}{S}}\right) H(v) - n}{e^{\frac{\varepsilon}{S}} - 1} \tag{5.1}$$

It is easy to see that the expected value of estimate $\hat{F}(v)$ is $F(v)$. Thus, $\hat{F}(v)$ is an unbiased estimator of $F(v)$. To improve accuracy, the estimate is reset to zero if it is negative, and is reset to n if it exceeds n.

Note that this approach is designed for "one-shot" randomization, i.e., $G_{\boldsymbol{c}}$ and $\boldsymbol{c}$ are deleted at the user end once $R(\boldsymbol{c})$ is generated. Any subsequent requests for data will receive the same value of $R(\boldsymbol{c})$. In contrast, in a framework that allows submission of multiple realizations of $R(\boldsymbol{c})$, the privacy protection will degrade due to composition [29]. Our approach can prevent such degradation and has practical usage, for example, by Facebook in their ads system [19].

The approach also excludes the consideration of *contexts*, i.e., from which nodes a node is reached at run time. A classic example of a context is the calling context (i.e., the chain of callers) for a call graph node. Solving this problem requires the randomizer to record and obfuscate paths in the control-flow graph model. We leave this challenging problem for future work.

5.4 Selection of Sensitivity Bound

The choice of probability p in Definition 5.5 guarantees that R is ε-node-LDP. Thus, the main question is how to select the sensitivity upper bound S. One obvious choice for S is given by the global sensitivity, which is the maximum value of the local sensitivity taken over all realizations of feasible coverage vectors. In our baseline approach, we instantiate S with the global sensitivity in the technique described in Section 5.3.2. It is important to point out that this baseline approach provably achieves the optimal worst-case estimation error, which scales with the global sensitivity. This follows from a straightforward extension of the known lower bound on the worst-case error associated with LDP frequency estimation [8]. However, as demonstrated by our empirical results (Section 5.5), the accuracy resulting from the baseline approach is usually modest since the global sensitivity is quite large.

To circumvent this fundamental limitation, we propose an alternative approach similar to the data collection in Section 4.5 that splits users into an opt-in user group and a regular user group, computes the bound S using opt-in user data, and aggregates node coverage from regular users. However, this approach may yield weakened protection for regular users, as discussed in Section 5.4.2. Therefore, we propose another two approaches ensuring strict ε-node-LDP for all users that entail either a relaxation of the utility guarantee (Section 5.4.3) or a relaxation of the privacy guarantee (Section 5.4.4). In particular, in Section 5.4.3, the proposed approach offers a conditional utility guarantee, i.e., it achieves good accuracy but only for a sub-collection of well-behaved control-flow graphs. The approach in Section 5.4.4 entails assigning different levels of privacy protection for different nodes in the graph (depending on how "revealing" a node is). These approaches are simple, practical alternatives that provide meaningful privacy guarantees, while significantly improving the accuracy resulting from the baseline approach as demonstrated in the experiments shown in Section 5.5.

5.4.1 Baseline: Global Sensitivity

One choice for S in Definition 5.5 is to consider the worst-case value for $LS(\boldsymbol{c})$. For our problem, this worst-case value is $S_{gs} = |\mathcal{V}| - 1$. Here suffix gs is short for "global sensitivity." For any G and any feasible $\boldsymbol{c}$ for G, $LS(\boldsymbol{c}) \leq S_{gs}$ since, in the worst case, $\boldsymbol{c}$ contains all nodes in $\mathcal{V}$ and its farthest neighbor contains only the start node s. Since G is known to all remote instances of the software, each local randomizer R can use the same value $S = |\mathcal{V}| - 1$ to add noise to its local vector. Figure 5.2 illustrates this case: for $c = \{s, v_1, v_2, v_3, v_4, v_5\}$ and its neighbor $\Delta_{v_1}(\boldsymbol{c}) = \{s\}$, we have $LS(\boldsymbol{c}) = |\mathcal{V}| - 1 = 5$.

Example 5.3. Consider the example of $n = 10$ users for the parking app, shown in Figure 5.5. The sensitivity bound is $S_{gs} = 9$ as there are 10 nodes in the control-flow model from Figure 5.3. This bound is *a priori* knowledge to all users. Each user generates her own coverage vector $\boldsymbol{c}_i$ independently and runs R with S_{gs} locally. In this example, ε is set to

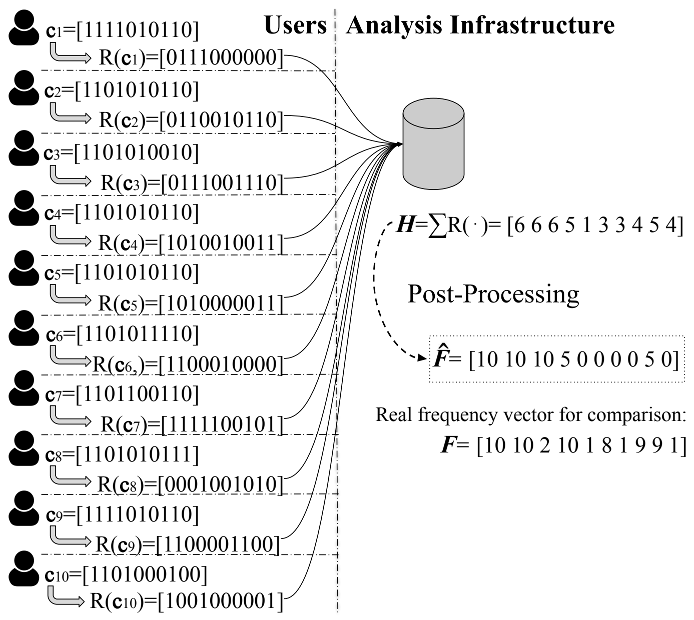


Figure 5.5: Randomization using global sensitivity S_{gs} and $\varepsilon = 1$ for the parking app, with 10 users.

1. The analysis infrastructure collects all randomized $R(\boldsymbol{c}_i)$ vectors to get $\boldsymbol{H} = \sum_i R(\boldsymbol{c}_i) =$ [6 6 6 5 1 3 3 4 5 4]. Using Equation 5.1, we then obtain a vector of estimates $\hat{\boldsymbol{F}} =$ [10 10 10 5 0 0 0 0 5 0], with all decimals rounded to the nearest integer.

The real frequency vector is $\boldsymbol{F} =$ [10 10 2 10 1 8 1 9 9 1]. Clearly, the differentially-private estimates for this example are rather inaccurate. This is due to the small number of users as well as the loose upper bound S_{gs}.

This baseline approach could introduce significant amount of noise. For illustration, consider $\varepsilon = 1$ and $S = |\mathcal{V}| - 1 = 100$. The probability p of flipping any bit is 0.4975, which is very close to the probability 0.5 that would produce uniformly-distributed random vectors drawn from $\{0,1\}^{|\mathcal{V}|}$. Next, we discuss three techniques that lead to reduction of the noise introduced by the randomization.

5.4.2 Tighter Bound Based on Local Sensitivity

One naive attempt to improve the worst-case analysis from above is the following. Each software user i computes the local sensitivity $LS(\boldsymbol{c}_i)$ of its coverage $\boldsymbol{c}_i$ and reports it to the software analysis infrastructure. The largest value of these reports is used as the upper bound $S = \max_i LS(\boldsymbol{c}_i)$. This S is reported back to all users' software instances and used in all randomizers R_i as shown in Definition 5.5. If this bound is lower than the worst-case bound of $|\mathcal{V}| - 1$, the analysis accuracy could be improved.

Unfortunately, publicly releasing the value of $LS(\boldsymbol{c}_i)$ and the randomizer output $R(\boldsymbol{c}_i)$ reveals too much information about the underlying data $\boldsymbol{c}_i$ and the resulting scheme is not differentially private [72]. As in Chapter 4, we propose to address this problem by separating the analysis users in two groups: *opt-in users* and *regular users*. For each opt-in user i, only the local sensitivity $LS(\boldsymbol{c}_i)$ is reported, but not the actual noisy data $R(\boldsymbol{c}_i)$. The local sensitivities for opt-in users are gathered without privacy protections. However, each such user i only reports the size of the largest subtree in the dominator tree of her local graph $G_{\boldsymbol{c}_i}$ induced by $\boldsymbol{c}_i$, but the identity of nodes v with large $sub_{G_{\boldsymbol{c}_i}}(v)$ is never revealed. We consider this technique to be a compromise that offers a reasonable privacy protection while allowing for much better accuracy.

Given the set of $LS(\boldsymbol{c}_i)$ from the opt-in users, the largest of these values is used as the bound S and is reported to all regular users for use in their randomizers (according to Definition 5.5). Each regular user i applies this R to the local coverage $\boldsymbol{c}_i$ and reports $R(\boldsymbol{c}_i)$ to the software analysis infrastructure. All reported noisy coverage vectors from regular users are post-processed to account for the randomization, as described in Equation 5.1.

It is possible that $LS(\boldsymbol{c}_i)$ exceeds this proposed bound S for a regular user i. One option is to exclude the user from the data gathering. However, if the identity of such users is known to the server, this technique will not generally satisfy LDP (in the strict sense of Definition 5.3) for those users. Intuitively, the reason is that two "neighboring" local vectors for such a user may lead to two different scenarios, where the user is participating in one, and not participating in the other. An alternative handling of such "sensitive" users would be to apply the randomizer based on S and to report $R(\boldsymbol{c}_i)$ anyway. This means that the user's data is obtained with weakened privacy protection. Formally, the privacy for a sensitive user i satisfies Definition 5.3 but with a larger value of the privacy loss parameter $\varepsilon' = \frac{LS(\boldsymbol{c}_i)}{S}\varepsilon$. In our experimental studies we have observed that using 10% of the software users as opt-in users provides a reasonable bound S and only a few users from the remaining 90% (i.e., regular users) have local sensitivity exceeding this S and thus their c_i is protected with weakened guarantees.

5.4.3 Tighter Bound via Restricted Sensitivity

The approach discussed earlier cannot guarantee strong privacy protection for regular users whose local sensitivities are above the S computed from the opt-in user data. To deal with this issue, in this and the next subsection we introduce two approaches that yield strict ε-node-LDP for all users.

A tighter sensitivity bound can be achieved with certain hypotheses. A hypothesis $\mathcal{H}$ in our context is a subset of the set $\mathcal{D}$ of all possible feasible coverage vectors. The specific hypotheses we consider are parameterized by a value $k < S_{gs}$ and defined as $\mathcal{H}_k \subseteq \mathcal{D}$ where $LS(\boldsymbol{c}) \leq k$ for all $\boldsymbol{c} \in \mathcal{H}_k$. The sensitivity bound is $S = k$ in this case. This technique is similar in spirit to *restricted sensitivity* [10] that guarantees differential privacy for a restricted class of datasets. The result of the analysis is useful if the hypothesis is correct, which in our case means that all coverage vectors have local sensitivity not exceeding k. The result may be inaccurate if some vectors have local sensitivity greater than k.

To ensure that the hypothesis holds for the input domain of randomizer R, one solution is to define a projection function $\mu : \mathcal{D} \to \mathcal{H}_k$ by which $\boldsymbol{c}$ is transformed into $\mu(\boldsymbol{c})$ such that $LS(\mu(\boldsymbol{c})) \leq k$. Then R is applied to $\mu(\boldsymbol{c})$. We design μ as follows. For all $\boldsymbol{c} \in \mathcal{H}_k$, we have $\mu(\boldsymbol{c}) = \boldsymbol{c}$. For any other $\boldsymbol{c} \in \mathcal{D}$, we prune $G_{\boldsymbol{c}}$ according to its dominator relation. Specifically, consider each child node v of the start node s in the dominator tree for which $|\boldsymbol{c}| - |\Delta_v(\boldsymbol{c})| > k$. (If this condition does not hold, v and its tree descendants do not need to be pruned.) We conduct breadth-first search starting from v and prune the last $sub_{G_{\boldsymbol{c}}}(v) - k$ traversed nodes from the dominator tree and from $G_{\boldsymbol{c}}$. The corresponding bits in $\boldsymbol{c}$ are set to 0.

Example 5.4. Consider the coverage vector $\boldsymbol{c} = [1101010111]$ and its corresponding dominator tree in Figure 5.4. If $k = 5$, as $|\boldsymbol{c}| - |\Delta_{v_1}(\boldsymbol{c})| = 6 > 5$, by removing the leaf node v_9 at the last level in the subtree, the projection produces $\mu(\boldsymbol{c}) = [1101010110]$. Next, consider an extreme case where $k = 1$. The projection $\mu(\boldsymbol{c})$ needs to trim from $G_{\boldsymbol{c}}$ a set of 5 nodes $\{v_3, v_5, v_7, v_8, v_9\}$. The final output of the projection is $\mu(\boldsymbol{c}) = [1100000000]$. Its local sensitivity is $LS(\mu(\boldsymbol{c})) = 1 \leq k$.

After the projection step, each user reports $R(\mu(\boldsymbol{c}))$ to the server for further analysis. Overall accuracy depends not only on R but also on k. When $k \ll S_{gs}$, we have a very tight bound such that the noise introduced by R reduces significantly, while the noise due to the projection μ increases. For the extreme example above, the utility of the analysis result is expected to drop since most of the information of $\boldsymbol{c}$ is lost after the projection. This highlights the trade-offs between privacy and accuracy in any DP analysis. In Section 5.5, we conduct empirical evaluation on the impact of k and show that practical accuracy can be achieved by properly selecting the value for k.

5.4.4 Relaxed Indistinguishability of Neighbors

The above techniques ensure the same level of indistinguishability for all neighbors of a coverage vector. However, in practice, not all neighbors are of the same significance in terms of privacy protection. For instance, consider a news app that records users' reading content. It might be acceptable to reveal that a user is reading sports news instead of business news, but disclosing whether it is about basketball or football may be undesirable as this information can be used for targeted advertisement. As another example, API methods invoked by the Android framework (e.g., activity lifecycle callbacks) are expected to be covered in any non-trivial execution. The weakened hiding of their presence is a reasonable compromise. Thus a *relaxed* indistinguishability level depending on some notion of "distance" between any pair of neighbors would be useful. Intuitively, neighbors with small distance require more extensive randomization. For the above example of the news app, the more specific the news topic is, i.e., news are "closer" to each other, the more privacy concerns a user may have and the more noise is needed. Distance-based indistinguishability has been studied theoretically [15] as a generalization of traditional DP.

We investigate a distance metric d^* based on the difference of each pair of neighboring coverage vectors $\boldsymbol{c}$ and $\Delta_v(\boldsymbol{c})$. More specifically, $d^*(\boldsymbol{c}, \Delta_v(\boldsymbol{c})) = |\boldsymbol{c} \setminus \Delta_v(\boldsymbol{c})| = sub_{G_{\boldsymbol{c}}}(v)$. We define the privacy budget ε' depending on this metric to achieve (d^*, ε)-privacy [15]: $\varepsilon' = \varepsilon \times d^*(\boldsymbol{c}, \Delta_v(\boldsymbol{c}))$. This can be realized by setting $S = 1$ in Definition 5.5, of which the proof is similar to the one for Proposition 5.2. In general, if the distance is large between two neighbors, the privacy budget will also be large and R only introduces a small amount of noise to “hide” their difference. If the distance is 1, we will have the same protection as by the traditional DP techniques introduced earlier. For the example in Figure 5.4, we have $\varepsilon' = \varepsilon \times d^*(\boldsymbol{c}, \Delta_{v_1}(\boldsymbol{c})) = 6\varepsilon$, while $\varepsilon' = \varepsilon \times d^*(\boldsymbol{c}, \Delta_{v_9}(\boldsymbol{c})) = \varepsilon$ which guarantees stronger protection.

The intuition behind this metric is that nodes that are close to the root of the dominator tree are likely to be covered by most run-time executions and thus are less sensitive in terms of privacy. For instance, the analysis of 15000 realizations of screen view graphs for 15 Android apps from Section 5.5 shows that nodes that are in all dynamic graphs for an app (which strongly indicates that their executions are deterministic and the protection of their existence is impossible) have an average dominator tree level of 2, while nodes that appear in less than half of the graphs have an average level of 4. Intuitively, stronger protection is desirable for a node v if its execution is specific for a small group of users, compared to the case where v's execution is deterministic and happens for all users.

As a concrete example, in Figure 5.3, the “LastParkingFragment” screen (v_1) in the parking app is the landing screen after the “Splash” screen and is observed in all run-time executions in our experiments. Such population-wise behaviors likely cannot be used as user-specific usage patterns and may be of less interest to the adversary. Thus we believe that reducing the effort to hide its existence is a reasonable compromise. Meanwhile, among 1000 independent executions of the parking app in the experimental evaluation, the “ZoneEditorActivity” screen (v_9) is observed only once. It could be used as a fingerprint for that particular user and thus requires more protection. A vector $\boldsymbol{c}$ and its neighbor $\Delta_{v_9}(\boldsymbol{c})$ should be indistinguishable after randomization to prevent adversaries from inferring the occurrence of v_9.

This technique is an example of d-privacy [15] which is a generalization of differential privacy. There are other possible choices for techniques to help improve utility. For example, consider a set of non-sensitive nodes that is defined as part of the analysis specification. Metric d^* can set the distance of neighbors with respect to these nodes to a very large value (e.g., $d^* = \infty$), so that the privacy protection for such neighbors are minimized. Utility-optimized LDP [65] can also be used, by providing ε-node-LDP protection for graph instances that include sensitive nodes while relaxing the protection of graphs containing only non-sensitive nodes. However, the original algorithms in [65] fail to consider the correlation between data items and cannot be directly employed here. It would be interesting to investigate the problem of control-flow node coverage with predefined non-sensitive nodes, depending on domain-specific and software-specific considerations.

5.5 Evaluation

To evaluate the proposed techniques, we gathered two kinds of control-flow graphs: *GUI screen graphs* and *call graphs*. Each GUI screen graph was obtained by analyzing the sequence of Google Analytics (GA) GUI screen view events. A GA GUI screen view event indicates that a particular screen in the app's GUI was displayed. Each screen has a unique string name that is used as an identifier. With the help of app code instrumentation, in our experiments we intercepted and recorded such events to a local database (by the *tracker* component). The transitions from one screen to the next define a GUI screen graph, in which nodes are screens and edges are transitions between screens. We first ran extensive experiments with the Monkey tool for GUI testing [40] to construct a graph $G = (\mathcal{V}, \mathcal{E}, s)$ that captures possible screen transitions. Alternatively, app developers could have GUI design information that provides such a graph G directly. Given this G, we simulated

App	Screen Graph		Call Graph	
	#Nodes	#Edges	#Nodes	#Edges
barometer	9	69	1066	1683
bible	11	75	832	1412
dpm	8	36	623	1016
drumpads	14	108	613	868
equibase	18	297	340	826
localtv	28	366	1741	3102
loctracker	14	151	199	335
mitula	16	169	3700	6879
moonphases	15	126	254	454
parking	10	58	712	1223
parrot	51	1239	3748	9804
post	9	54	791	1635
quicknews	14	120	970	1861
speedlogic	10	75	124	186
vidanta	12	112	2290	4089

Table 5.1: Apps and control-flow graph models.

1000 executions of the app. To represent the data for each execution, we ran Monkey (independently from any other executions) to obtain $10 \times |\mathcal{V}|$ screen view events for that execution. From that trace we determined the coverage vector $\boldsymbol{c}$ and the corresponding subgraph $G_{\boldsymbol{c}}$ of G. The call graph models G were obtained in a similar manner; here nodes represent methods in the app code and edges represent calling relationships, with an artificial start node s representing the Android framework code. Using separate Monkey runs and code instrumentation, we created 1000 traces each with $10 \times |\mathcal{V}|$ method call events. From these traces, call graph coverage vectors $\boldsymbol{c}$ were constructed.

To obtain apps that use Google Analytics, we analyzed popular apps in each category in the Google Play store and identified apps that include GA API calls. The apps and their control-flow models G are described in Table 5.1. As can be seen from these measurements, a call graph is typically one to two orders of magnitude larger than the GUI screen graph for the same app (as can be expected). We chose to study data for both GUI screen graphs and call graphs in order to observe the effects of graph size on the accuracy of the analysis. All graphs G and the 1000 run-time realizations of G_c for each app are available at `https://presto-osu.github.io/sec20`.

5.5.1 Metrics

Theoretically, when S is large, the protocol achieves higher privacy (i.e., the probability p in Definition 5.5 is large) at a cost of lower utility. Such trade-offs between privacy and utility are inherent in DP analyses and need to be explored carefully in order to design practical solutions. In Sections 5.3.2 and 5.4, we propose techniques based on ε-node-LDP and d-privacy that utilize different bounds to achieve high utility of analysis results. To evaluate the effectiveness of these techniques, we consider two practical usage scenarios and questions:

- **Q1: Which control-flow graph nodes are executed by at least one user?** This question is the core to many debugging and testing techniques, e.g., residual testing [78]. The answer is the set of nodes that are observed at run time in at least one deployed software instance: $\{v \in \mathcal{V} \mid F(v) > 0\}$. Recall that the algorithm from Section 5.3 provides an estimate vector $\hat{\boldsymbol{F}}$ of the real frequency vector $\boldsymbol{F}$. Thus, we can estimate the set of nodes by $\{v \in \mathcal{V} \mid \hat{F}(v) > 0\}$. We use precision and recall to measure the utility of the estimation.

- **Q2: Given a node, what is the number of users who have executed it?** This information is useful for tasks such as finding popular app features. We evaluate the accuracy of estimates by computing the *relative error* (RE), which is also used in the experimental evaluation in Chapter 3 and 4, defined as follows:

$$\text{RE} = \frac{\sum_{v\in\mathcal{D}} |F(v) - \bar{F}(v)|}{\sum_{v\in\mathcal{D}} F(v)}$$

where $\bar{\boldsymbol{F}}$ is a calibrated estimate vector computed by applying the same quadratic programming technique used in previous chapters. We minimize the squared Euclidean distance from $\bar{\boldsymbol{F}}$ to $\hat{\boldsymbol{F}}$ under the constraints that $\bar{\boldsymbol{F}} \geq \mathbf{0}$ and the sum of all elements in $\bar{\boldsymbol{F}}$ is $\sum_{v\in\mathcal{V}} F(v)$. This guarantees that the worst-case value of RE is 2, which is consistent with the metrics in earlier chapters. In the experiments, we calculate both the overall RE for all nodes, i.e., $\mathcal{D} = \mathcal{V}$, and the RE for frequently-executed nodes across all users, i.e., $\mathcal{D} = \text{hot}(\boldsymbol{F}, \ell)$ as introduced in Section 3.4.4. We also compute the *hot node coverage* (HNC), defined similarly to the *hot method coverage* metric in Section 3.4.4: $\text{HNC}(\ell) = |\text{hot}(\boldsymbol{F}, \ell) \cap \text{hot}(\bar{\boldsymbol{F}}, \ell)| / |\text{hot}(\boldsymbol{F}, \ell)|$. In this section, we only show the results for $\ell = 0.25$ as other threshold values show similar trends.

5.5.2 GUI Screen Graphs

Answering Q1. We first collected all $\boldsymbol{c}_i$ for $1 \leq i \leq 1000$ to get the ground truth $\boldsymbol{F}$, as described earlier, and computed $F(v) = \sum_i \boldsymbol{c}_i(v)$ where i ranges over all independent executions that are regarded as individual users. To compute estimates $\hat{\boldsymbol{F}}$, for the same range of i we randomized each $\boldsymbol{c}_i$ independently according to Definition 5.5, computed $\boldsymbol{H} = \sum_i R(\boldsymbol{c}_i)$, and post-processed $\boldsymbol{H}$ using Equation 5.1. To empirically compare the accuracy of the proposed techniques, we used $\varepsilon = 1$ for the randomization; this choice was motivated by a popular DP analysis [9]. During post-processing, the estimate was set to 0 if it was negative, and to the number of analyzed users if it exceeded that number. Then each estimate was rounded to the nearest integer. We repeated this process for 100 independent trials and collected the precision and recall for each trial. The variations among the 100

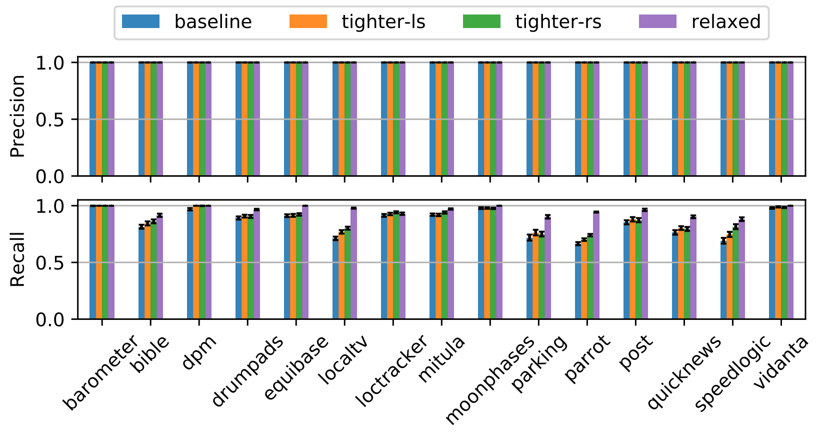


Figure 5.6: Precision and recall for GUI screen graphs.

trials are due to the randomness when perturbing $\boldsymbol{c}_i$ (since $\boldsymbol{c}_i$ for user i is the same in each trial). Figure 5.6 reports the mean values and the 95% confidence intervals for the 100 trials, using the GUI screen graphs. The confidence intervals are typically very small, and barely noticeable in this figure and other figures in the this section.

When applying randomization, we first set the sensitivity bound S to the global sensitivity $S_{gs} = |N| - 1$ (in Section 5.4.1) to obtain a worst-case baseline. As shown by bars "baseline" in Figure 5.6, using global sensitivity yields perfect precision but relatively low recall. The precision is perfect due to the fact that in our run-time traces every node in graph G has been executed by at least one user, i.e., there are no false positives. We include this redundant precision data only for completeness and for uniformity with the data for call graphs presented later (where false positives are present). We have recall below 0.8 in 5 out of the 15 apps. Practically, this means that more than 20% of nodes are lost after randomization and post-processing. The figure also shows similar measurements for the remaining three techniques.

To obtain tighter bounds based on local sensitivities from opt-in users as introduced in Section 5.4.2, we randomly selected 100 (from the 1000) users to form the opt-in user group. For each selected user i, its local sensitivity $LS(\boldsymbol{c}_i)$ was calculated using the dominator tree induced by $\boldsymbol{c}_i$. The maximum of all local sensitivities of the sampling group was set to be the bound S used for randomization. The remaining 900 users perturbed their coverage vectors accordingly. The resulting precision and recall for each app is shown by bars “tighter-ls” in Figure 5.6.

To select a proper k for tighter bound via restricted sensitivity, we used $k = \lfloor t \times S_{gs} \rfloor$ where $t = \{0.95, 0.9, \ldots, 0.05\}$ and computed the largest difference between the estimated and the true frequencies, i.e., $\max_{v \in \mathcal{V}} |F(v) - \hat{F}(v)|$. We chose the k that minimized the largest difference for each app. For example, we set $k = \lfloor 0.35 \times S_{gs} \rfloor = 3$ for the parking app leading the bound to be $3\times$ smaller. The impact of k varies from app to app. This is mainly due to variance of the structure of graphs and dominator trees of each app.

By using the tighter bounds and relaxed indistinguishability level for neighbors, the recall is significantly improved. With the tighter bound, the recall is below 0.8 for 4 apps if based on local sensitivity, and for only 3 apps if using restricted sensitivity. With relaxed indistinguishability level, the recall is ≥ 0.85 for all apps and is perfect for 5 apps. This means that the LDP algorithm successfully preserves the presence of all nodes that were actually observed at run time.

Answering Q2. To evaluate the ability of the proposed techniques to recover frequencies, we collected the RE for every experimental subject in each of the 100 independent trials and computed the mean of the errors. Table 5.2 lists the average RE across all apps for different values of ε. Figure 5.7 shows the mean values of RE across 100 trials and the 95% confidence

	RE		
	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
baseline	0.490	0.321	0.190
tighter-ls	0.392	0.261	0.147
tighter-rs	0.313	0.200	0.120
relaxed	0.064	0.032	0.015

Table 5.2: Average RE across all apps for GUI screen graphs.

	RE			HNC		
	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
baseline	0.312	0.193	0.110	0.640	0.754	0.875
tighter-ls	0.247	0.157	0.086	0.706	0.804	0.909
tighter-rs	0.201	0.122	0.073	0.749	0.842	0.917
relaxed	0.034	0.017	0.008	0.992	0.996	0.997

Table 5.3: Average RE and HNC for hot nodes in GUI screen graphs given $\ell = 0.25$.

intervals for each app under various values of ε. As expected, the baseline approach provides the least accurate estimates. The two techniques, which yield tighter sensitivity bounds based on local sensitivity ("tighter-ls" in the table and figure) and restricted sensitivity ("tighter-rs" in the table and figure), achieve comparable improvements in accuracy. For example, they produce 1.2$\times$ and 1.6$\times$ less RE compared to the baseline approach when $\varepsilon = 1$, respectively. We can observe significant improvement when applying relaxed indistinguishability. The average RE in this case is 9.9$\times$ smaller than the baseline RE under $\varepsilon = 1$, and is 12.6$\times$ smaller when ε grows to 2.

If we only focus on the frequency estimates for frequently-executed nodes, the accuracy is higher. As shown in Table 5.3, the average RE for hot nodes in GUI screen graphs across

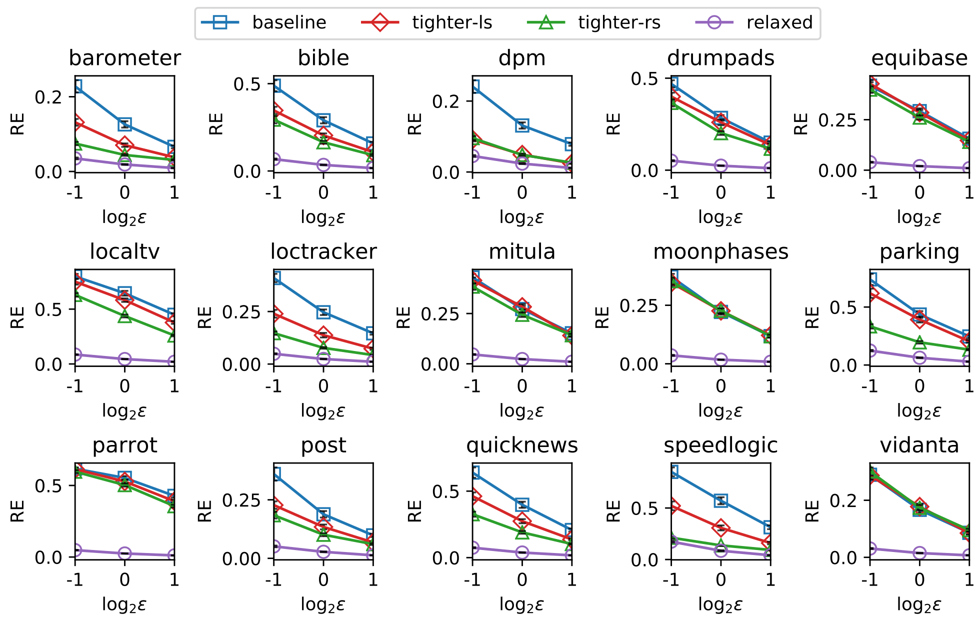


Figure 5.7: Relative error for GUI screen graphs.

all 15 experimental subjects is much smaller than the RE for all nodes. For example, for the baseline approach, the RE is 1.6×, 1.7× and 1.7× smaller when ε is set to 0.5, 1, and 2. As another example, the relaxed-indistinguishability-based approach produces 1.9× less RE on average for all values of ε. The HNC is reasonable (≥ 0.8) only when appropriate ε is chosen. For example, the two approaches that tighten the bound only provide acceptable HNC when $\varepsilon \geq 1$.

5.5.3 Call Graphs

Answering Q1. GUI screen graphs for Android applications are typically small, since the GUI structure of an app is highly unlikely to contain hundreds of screens. To evaluate the performance of the proposed techniques on larger graphs, we obtained call graph data as described earlier. The coverage measurements for this data were computed in the same

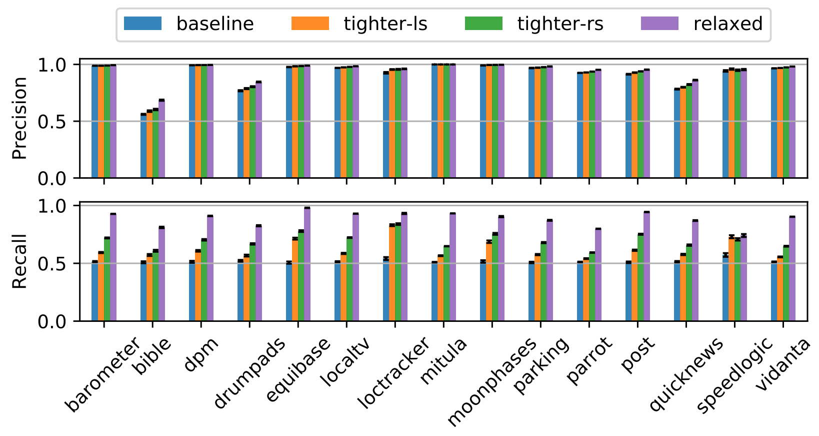


Figure 5.8: Precision and recall for call graphs.

manner as for the GUI screen graphs. We ran 100 independent trials for each experiment. Figure 5.8 reports the means and 95% confidence intervals of precision and recall over the 100 trials. As expected, using the global sensitivity as the sensitivity bound introduces the largest amount of noise due to the large size of $\mathcal{V}$. Though only 3 out of the 15 apps have precision below 0.9 due to the correct discovery of frequently executed methods, the recall is under 0.6 for all apps. By manual investigation, we found that for many *infrequently executed methods* the frequency estimates were negative and thus were zeroed out after post-processing. In these cases, the noise overwhelmed the small frequency counts and the nodes were not correctly discovered.

Using tighter sensitivity bound based on local sensitivities, there is only a little improvement on the recall, i.e., 1.1$\times$ increase averaged across all apps for $\varepsilon = 0.5$, 1.2$\times$ increase for $\varepsilon = 1$ and 1.3$\times$ increase for $\varepsilon = 2$. Using tighter bound based on restricted sensitivities, the improvement is also not obvious. The recall is 1.2$\times$, 1.3$\times$ and 1.4$\times$ larger for $\varepsilon = 0.5$, $\varepsilon = 1$ and $\varepsilon = 2$, respectively. This implies that, at least for these specific runs

	RE		
	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
baseline	1.080	1.068	1.048
tighter-ls	0.973	0.867	0.709
tighter-rs	0.807	0.656	0.527
relaxed	0.082	0.042	0.019

Table 5.4: Average RE across all apps for call graphs.

in the experiment, the balance between the accuracy gain and loss by the projection is hard to achieve and strong privacy guarantees cannot be achieved without sacrificing accuracy. The recall has an observable jump when the relaxed indistinguishability is used and 13 out of 15 apps have recall ≥ 0.8 when ε is small (0.5 and 1).

Answering Q2. Table 5.4 shows the average RE for all apps. Figure 5.9 shows detailed results for each app. For most apps, there is observable less RE for the techniques that tighten the sensitivity bound. Comparing to the baseline approach, the one based on local sensitivity achieves $1.1\times$, $1.2\times$ and $1.5\times$ less RE on average for $\varepsilon = 0.5$, 1 and 2, respectively. The one based on restricted sensitivity performs a little better, producing estimates with $1.3\times$, $1.6\times$ and $2\times$ less RE for the three different ε values. The best accuracy is achieved when using relaxed indistinguishability by providing less protection for neighbors that are "far away" from each other, which in the case of call graphs means that two neighboring executions share only a small set of common methods. For example, when $\varepsilon = 2$, this approach guarantees $53.8\times$ smaller RE, averaged across all apps.

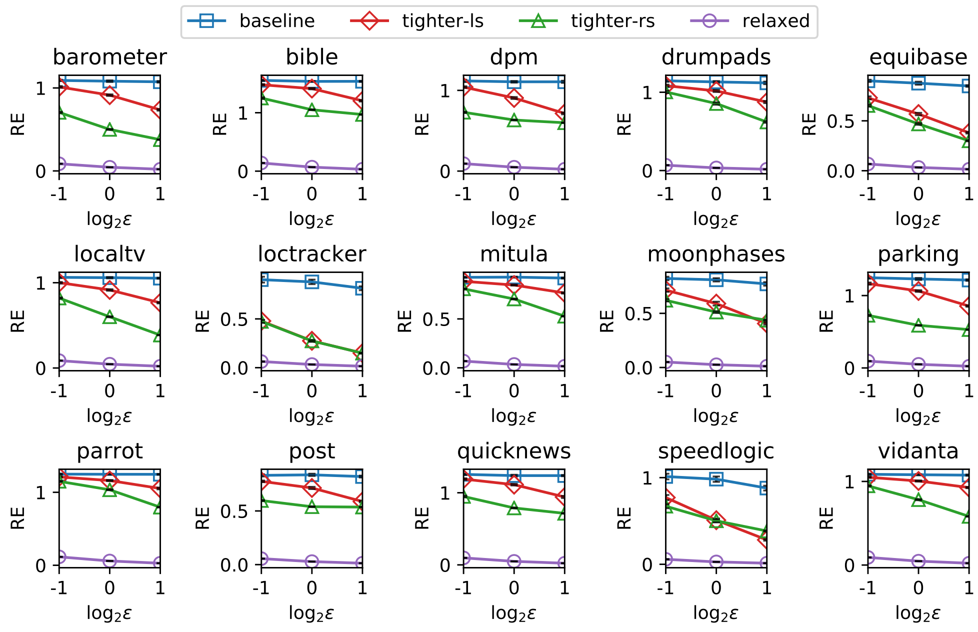


Figure 5.9: Relative error for call graphs.

Table 5.5 shows the average RE and HNC for hot nodes in call graphs. We will not elaborate the details as we can draw similar conclusion as in the previous subsection, i.e., the accuracy is much better if only estimates for frequently-executed nodes are of interest.

To summarize, our results demonstrate that node coverage analysis of control-flow graphs could be achieved with both privacy and practical accuracy. At the same time, the results clearly show that there is a fundamental trade-off between the degree of privacy protection and the utility of the analysis estimates. In addition to the theoretical exploration of this space presented in Section 5.3 and Section 5.4, we experimentally identify practical trade-offs that could be used for future developments of privacy-preserving software analyses.

Processing cost. The cost of computing $LS(\boldsymbol{c}_i)$ is rather small. For example, call graphs $G_{\boldsymbol{c}_i}$ for app mitula are the largest ones in our data, with 2011 nodes per graph on average. The

	RE			HNC		
	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
baseline	0.576	0.570	0.558	0.491	0.499	0.506
tighter-ls	0.512	0.450	0.360	0.493	0.538	0.622
tighter-rs	0.433	0.365	0.312	0.530	0.563	0.622
relaxed	0.032	0.016	0.007	0.990	0.996	0.998

Table 5.5: Average RE and HNC for hot nodes in call graphs given $\ell = 0.25$.

average time to compute $LS(\boldsymbol{c}_i)$ for one of these graphs is about 15 ms. Given the sensitivity bound S, the average time to compute $R(\boldsymbol{c}_i)$ for these graphs is about 30 ms. Clearly, for the graphs considered in our experiments, the cost of data processing is negligible.

5.6 Summary

Over the last decade, pervasive data gathering has become the norm. Combined with rapid advances in large-scale data analytics and machine learning, this presents fundamental challenges to privacy. Exploring the trade-offs between privacy protections and the utility of data gathering/analysis is a critical scientific challenge. To study such trade-offs in the analysis of deployed software, we explore the use of differential privacy. With the help of this rigorous technique, we develop a novel node coverage analysis of control-flow graphs. By carefully defining feasibility constraints and neighbor relations for such graph, our study highlights the key trade-offs in algorithm design and presents effective choices for these trade-offs. Our evaluation demonstrates that, with these choices, both privacy and accuracy can be achieved for this control-flow analysis. This work is a promising step in the larger landscape of privacy-preserving software analysis and analytics.

Chapter 6: Related Work

Differential privacy. Several examples of prior work on differential privacy were already discussed in Chapter 2. Many other analytics problems have also been considered: for example, heavy hitters [9, 13], distribution estimation [27], clustering [71], learning [51], and convex optimization [89]. These theoretical models have not yet been applied to software profiling and present a rich source of powerful privacy-preserving techniques.

Several applications of LDP have been realized in practice [3, 25, 32, 35, 70, 94, 95]. Google's RAPPOR combines randomized responses and Bloom filters to encode and identify popular URLs in the Chrome browser [32, 35]. Apple applies DP for gathering analytics data for emoji and quick type suggestions [3, 94]. Samsung proposed the Harmony LDP system to collect data from smart devices [70]. Microsoft uses LDP to collect telemetry data across millions of devices [25]. Several of these existing approaches are focused on collection of a single data item per user, while this dissertation aims at profiling where accurate results are needed for multidimensional local frequency profiles.

There also exists a large body of work on protection of correlated data [16, 53, 54, 60, 106] and graphs with DP [23, 49, 50, 52, 83, 84, 88]. A few examples are outlined below. Karwa et al. [50] consider subgraph counting queries and present solutions with edge differential privacy. Liu and Mittal [59] propose LinkMirage to mediate privacy-preserving access to social networks by obfuscation of links. Qin et al. [82] aim at providing LDP of social networks where each user holds an adjacency list of her friends. While these studies provide edge-DP, our solution in Chapter 5 achieves the more challenging node-DP for control-flow graphs considering the causal relationships between nodes.

Profiling and its privacy. Many works have considered collection of various forms of data from deployed software for profiling and analysis purposes, and could be interesting targets for developing DP analyses. Residual coverage monitoring [78] collects statement coverage cumulatively. Liblit et al. [57] minimize per-user overhead during information gathering by using sampling of program executions. Bond and McKinley [11] propose a hybrid instrumentation and sampling approach for continuous path and edge profiling. Ricci et al. [85] track garbage collection events to help the development of new garbage collection algorithms. The GAMMA system [76] tracks data from many users by assigning monitoring subtasks to different software instances and integrating results from probes to provide information about the original task. Orso et al. [77] leverage GAMMA for impact analysis and regression testing using profiling and coverage data at block and method levels.

Haran et al. [44] use random forest to classify failing executions based on execution data. Nagpurkar et al. [66] propose an instruction-based profiling approach for deployed software. DiCE [14] explores system behaviors to check whether the system deviates from its desired behavior. BugRedux [47] collects execution data and synthesizes in-house executions the reproduce field failures. Their follow-up work [48] enhances BugRedux with debugging capabilities by synthesizing a set of program executions that mimic a field failure based on call sequences. Saha et al. [87] collect execution information across software instances by running the program multiple times with the same input. Diep et al. [24] propose a probe distribution algorithm to collect profile events. Clause and Orso [20] record program execution information, including interactions between software and environment and the input data for each interaction, for failure reproduction and in-house debugging. Ohmann et al. [74] propose algorithms based on dominator trees to minimize the coverage probes required for monitoring. A related effort [75] introduces a new language for answering control-flow queries based on incomplete data from post-deployment failure reports.

Privacy has also been considered. Elbaum and Hardojo [31] marshal and label data with the encrypted sender's name for anonymization at the deployed site. Clause and Orso [21] anonymize inputs that cause failures in deployed software. There are no theoretical guarantees about the privacy protection these techniques provide. In general, such approaches may suffer from carefully tuned attacks such as linkage attacks, in which data is gathered from several sources to reveal personally-identifiable information [26, 67, 68]. The approaches introduced in this dissertation, which are based on DP and its generalizations, are designed from the ground up with strong and well-defined privacy guarantees: despite any additional information an adversary may obtain from other sources, she cannot determine with high probability what is a user's private data.

Privacy in programming languages and software engineering. There is a growing number of efforts on differential privacy in the programming languages community. Zhang and Kifer [102] propose LightDP, a relational type system, to verify sophisticated differential privacy algorithms requiring less annotations from programmers than customized logics based proofs. Inspired by LightDP, Wang et al. [98] further propose a flow-sensitive type system, ShadowDP, that uses shadow execution to verify more differential privacy algorithms and reduce the complexity of the verification process. Near et al. [69] propose the Duet language for verifying differential privacy of general-purpose higher-order programs.

Some emphasis has also been put on privacy in various fields of software engineering [42]. Two typical areas of interest are testing [12, 41, 61, 93] and defect prediction [56, 79, 80]. Budi et al. [12] propose to use *kb*-anonymity for testing and debugging data. MORPH [79] preserves data privacy of software defects in a cross-company scenario, by perturbing instance values. The CLIFF+MORPH approach [80] uses perturbation to preserve data privacy of software defect information. With the exception of the work presented in this dissertation, we are not aware of any efforts to create DP versions of existing program analyses and profiling, which we believe is a fruitful target for future work.

Chapter 7: Conclusions

Over the last decade, pervasive data gathering has become the norm. Combined with rapid advances in large-scale data analytics and machine learning, this presents fundamental challenges to privacy. Exploring the trade-offs between privacy protections and the utility of data gathering/analysis is a critical scientific challenge. To study such trade-offs in the analysis and profiling of deployed software, we explore the use of differential privacy. With the help of this rigorous technique, this dissertation proposes several solutions targeting various profiling tasks to protect user profile data, as follows:

- **Chapter 3** presents an approach that allows tunable trade-offs between accuracy and privacy with respect to traces of run-time events, and enforces consistency constraints of event frequencies to improve estimate accuracy.
- **Chapter 4** presents a Laplace-mechanism-based approach for randomization of event frequency vectors, a novel linear programming formulation to compute the magnitude of random noise that should be added to achieve meaningful privacy protections, and the design of a hybrid approach to collect user profiles in a differentially-private manner that utilizes raw data from opt-in users.
- **Chapter 5** presents a differentially-private approach that ensures the privacy protection of control-flow graph nodes during node coverage profiling. The privacy

guarantees depend on the intrinsic correlations of nodes that can be captured by the dominator relationships between nodes, and can be effectively achieved by the proposed approach while providing practical accuracy.

The proposed approaches advance both the theoretical state of art and provide guidelines for future development of privacy-preserving software profiling and analysis infrastructures. They are the first to introduce differential privacy into remote software frequency and coverage profiling in order to provide principled privacy guarantees for user profiles. Our experimental evaluation shows that, with careful application, it is possible to achieve both high privacy and high accuracy for the target profiling tasks of these approaches.

In conclusion, the techniques described in this dissertation present promising novel advances in a broader research agenda to develop privacy-preserving analyses of deployed software. We anticipate that many interesting technical challenges will arise in attempts to apply differential privacy to other forms of dynamic program analysis of remote software. This dissertation provides several potentially-useful building blocks for such future attempts.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[2] G. Ammons, J. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *European Conference on Object-Oriented Programming*, pages 172–196, 2004.

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

[4] 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.

[5] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *International Symposium on Software Testing and Analysis*, pages 5–15, 2007.

[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 Symposium*, pages 747–764, 2017.

[7] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[8] R. Bassily and A. Smith. Local, private, efficient protocols for succinct histograms. In *ACM Symposium on Theory of Computing*, pages 127–135, 2015.

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

[10] J. Blocki, A. Blum, A. Datta, and O. Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *Innovations in Theoretical Computer Science Conference*, pages 87–96, 2013.

[11] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *IEEE/ACM International Symposium on Microarchitecture*, pages 130–140, 2005.

[12] A. Budi, D. Lo, L. Jiang, and Lucia. kb-anonymity: A model for anonymized behaviour-preserving test and debugging data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 447–457, 2011.

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

[14] M. Canini, V. Jovanović, D. Venzano, B. Spasojević, O. Crameri, and D. Kostić. Toward online testing of federated and heterogeneous distributed systems. In *USENIX Annual Technical Conference*, 2011.

[15] K. Chatzikokolakis, M. Andrés, N. Bordenabe, and C. Palamidessi. Broadening the scope of differential privacy using metrics. In *Proceedings on Privacy Enhancing Technologies*, pages 82–102, 2013.

[16] R. Chen, B. C. Fung, P. S. Yu, and B. C. Desai. Correlated network data publication via differential privacy. *The International Journal on Very Large Data Bases*, 23(4): 653–676, 2014.

[17] R. Chen, H. Li, A. K. Qin, S. P. Kasiviswanathan, and H. Jin. Private spatial data aggregation in the local setting. In *International Conference on Data Engineering*, pages 289–300, 2016.

[18] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *International Conference on Software Engineering*, pages 34–44, 2009.

[19] A. Chin and A. Klinefelter. Differential privacy as a response to the reidentification threat: The Facebook advertiser case study. *North Carolina Law Review*, 90:1417, 2011.

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

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

[22] 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.

[23] W. Y. Day, N. Li, and M. Lyu. Publishing graph degree distribution with node differential privacy. In *International Conference on Management of Data*, pages 123–138, 2016.

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

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

[26] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 202–210, 2003.

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

[28] C. Dwork. Differential privacy. In *International Colloquium on Automata, Languages and Programming*, pages 1–12, July 2006.

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

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

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

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

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

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

[35] G. Fanti, V. Pihur, and Ú. Erlingsson. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. *Proceedings on Privacy Enhancing Technologies*, 2016(3):41–61, 2016.

[36] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 57–76, 2007.

[37] Google. Google Analytics. `https://analytics.google.com`, 2019.

[38] Google. Android Debug Bridge. `https://developer.android.com/studio/command-line/adb`, 2020.

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

[40] Google. Monkey: UI/Application exerciser for Android. `http://developer.android.com/tools/help/monkey.html`, 2020.

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

[42] 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.

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

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

[45] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, 2007.

[46] L. Jiang and Z. Su. Profile-guided program simplification for effective testing and analysis. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 48–58, 2008.

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

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

[49] Z. Jorgensen, T. Yu, and G. Cormode. Publishing attributed social graphs with formal privacy guarantees. In *International Conference on Management of Data*, pages 107–122, 2016.

[50] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. In *International Conference on Very Large Data Bases*, pages 1146–1157, 2011.

[51] S. P. Kasiviswanathan, H. Lee, K. Nissim, S. Raskhodnikova, and A. Smith. What can we learn privately? *SIAM Journal on Computing*, 40(3):793–826, 2011.

[52] S. P. Kasiviswanathan, K. Nissim, S. Raskhodnikova, and A. Smith. Analyzing graphs with node differential privacy. In *Theory of Cryptography Conference*, pages 457–476, 2013.

[53] D. Kifer and A. Machanavajjhala. No free lunch in data privacy. In *International Conference on Management of Data*, pages 193–204. ACM, 2011.

[54] D. Kifer and A. Machanavajjhala. A rigorous and customizable framework for privacy. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 77–88, 2012.

[55] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, Jan. 1979.

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

[57] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[58] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.

[59] C. Liu and P. Mittal. LinkMirage: Enabling privacy-preserving analytics on social relationships. In *Network and Distributed System Security Symposium*, 2016.

[60] C. Liu, S. Chakraborty, and P. Mittal. Dependence makes you vulnberable: Differential privacy under dependent tuples. In *Network and Distributed System Security Symposium*, 2016.

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

[62] MathWorks. Optimization toolbox. `https://www.mathworks.com/help/optim`, 2019.

[63] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[64] Mitula. Mitula Homes. `https://play.google.com/store/apps/details?id=com.mitula.homes`, 2020.

[65] T. Murakami and Y. Kawamoto. Utility-optimized local differential privacy mechanisms for distribution estimation. In *USENIX Security Symposium*, pages 1877–1894, 2019.

[66] P. Nagpurkar, H. Mousa, C. Krintz, and T. Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Transactions on Architecture and Code Optimization*, 3(1):35–66, Mar. 2006.

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

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

[69] 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):1–30, 2019.

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

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

[72] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *ACM Symposium on Theory of Computing*, pages 75–84, 2007.

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

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

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

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

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

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

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

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

[81] Z. Qin, Y. Yang, T. Yu, I. Khalil, X. Xiao, and K. Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 192–203, 2016.

[82] Z. Qin, T. Yu, Y. Yang, I. Khalil, X. Xiao, and K. Ren. Generating synthetic decentralized social graphs with local differential privacy. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 425–438, 2017.

[83] S. Raskhodnikova and A. Smith. Private analysis of graph data. In *Encyclopedia of Algorithms*, pages 1–6. Springer Berlin Heidelberg, 2014.

[84] S. Raskhodnikova and A. Smith. Lipschitz extensions for node-private graph statistics and the generalized exponential mechanism. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 495–504, 2016.

[85] N. Ricci, S. Guyer, and J. Moss. Elephant tracks: Portable production of complete and precise GC traces. In *ACM SIGPLAN International Symposium on Memory Management*, pages 109–118, 2013.

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

[87] D. Saha, P. Dhoolia, and G. Paul. Distributed program tracing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 180–190, 2013.

[88] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Y. Zhao. Sharing graphs using differentially private graph models. In *Internet Measurement Conference*, pages 81–98, 2011.

[89] A. Smith, A. Thakurta, and J. Upadhyay. Is interaction necessary for distributed private learning? In *IEEE Symposium on Security and Privacy*, pages 58–77, 2017.

[90] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 180–195, 2001.

[91] L. Sweeney. Weaving technology and policy together to maintain confidentiality. *The Journal of Law, Medicine & Ethics*, 25(2-3):98–110, 1997.

[92] TalentApps. ParKing: Where is my car? Find my car - Automatic. `https://play.google.com/store/apps/details?id=il.talent.parking`, 2020.

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

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

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

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

[97] T. Wang, M. Lopuhaä-Zwakenberg, Z. Li, B. Skoric, and N. Li. Consistent and accurate frequency oracles under local differential privacy. In *Network and Distributed System Security Symposium*, 2020.

[98] Y. Wang, Z. Ding, G. Wang, D. Kifer, and D. Zhang. Proving differential privacy with shadow execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 655–669, 2019.

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

[100] 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.

[101] Yale Privacy Lab. App trackers for Android. `https://privacylab.yale.edu/trackers.html`, 2017.

[102] D. Zhang and D. Kifer. LightDP: Towards automating differential privacy proofs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 888–901, 2017.

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

[104] H. Zhang, S. Latif, R. Bassily, and A. Rountev. Differentially-private control-flow node coverage for software usage analysis. In *USENIX Security Symposium*, Aug. 2020. To appear.

[105] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems*, pages 603–610, 2004.

[106] T. Zhu, P. Xiong, G. Li, and W. Zhou. Correlated differential privacy: Hiding information in non-iid data set. *IEEE Transactions on Information Forensics and Security*, 10(2):229–242, 2014.

[107] X. Zhuang, M. Serrano, H. W. Cain, and J. D. Choi. Accurate, efficient, and adaptive calling context profiling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–271, 2006.