A Study of Event Frequency Profiling with Differential Privacy

Hailong Zhang
Ohio State University
Columbus, OH, USA
zhang.4858@osu.edu

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

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

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

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

Abstract

Program profiling is widely used to measure run-time execution properties—for example, the frequency of method and statement execution. Such profiling could be applied to deployed software to gain performance insights about the behavior of many instances of the analyzed software. However, such data gathering raises privacy concerns: for example, it reveals whether (and how often) a software user accesses a particular software functionality. There is growing interest in adding privacy protections for many categories of data analyses, but such techniques have not been studied sufficiently for program event profiling.

We propose the design of *privacy-preserving event frequency profiling* for deployed software. Each instance of the targeted software gathers its own event frequency profile and then randomizes it. The resulting noisy data has well-defined privacy properties, characterized via the powerful machinery of differential privacy. After gathering this data from many software instances, the profiling infrastructure computes estimates of population-wide frequencies while adjusting for the effects of the randomization. The approach employs static analysis to determine constraints that must hold in all valid run-time profiles, and uses quadratic programming to reduce the error of the estimates under these constraints. Our experiments study different choices for randomization and the resulting effects on the accuracy of frequency estimates. Our conclusion is that well-designed solutions can achieve both high accuracy and principled privacy-by-design for the fundamental problem of event frequency profiling.

CCS Concepts • **Software and its engineering → Dynamic analysis**; • **Security and privacy → Privacy-preserving protocols**.

Keywords dynamic analysis, differential privacy, profiling

ACM Reference Format:
Hailong Zhang, Yu Hao, Sufian Latif, Raef Bassily, and Atanas Rountev. 2020. A Study of Event Frequency Profiling with Differential Privacy. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377555.3377887

CC '20, February 22–23, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7120-9/20/02…$15.00
https://doi.org/10.1145/3377555.3377887

1 Introduction

The remote analysis 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 [1, 23, 28, 29, 38, 40] by considering selective optimization and feedback-directed code generation [3]. Other related uses of such remote analysis include debugging [26, 34] and reproduction of field failures [11, 24]. 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. Prior work in this area has focused primarily on the efficiency of data gathering (e.g., reducing overhead via sampling). However, the focus of our work is orthogonal: we aim to introduce privacy guarantees in the data collection process, after the profiling data has been collected locally but before sending it to the remote server. 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) [30, 31]. To provide a privacy-by-design solution with well-defined privacy properties, we employ differential privacy.

Differential privacy (DP) [15, 16] is a foundational approach for providing quantifiable privacy guarantees in data analysis, and is currently considered as the "gold standard" for privacy-preserving analysis. Both industry [2, 20, 21, 42] and government [12] have deployed DP solutions. There is a rich body of work in this area [17, 47] but the use of DP for software analysis has not been explored sufficiently. Our goal is to study the problem of event frequency profiling with differential privacy. At a basic level, one can use event randomization to achieve such privacy. For example, several existing DP techniques for data analysis are based on the following general idea. Consider a set $\mathcal{V}$ of possible events and suppose that one event $v \in \mathcal{V}$ is observed. Rather than simply reporting v, a DP analysis will use some probability p and will perform the following randomization: (1) with probability p, event v is reported; (2) for any other $v' \in \mathcal{V} \setminus \{v\}$, event v' is reported with probability $1 - p$. Based on the selection of p, one can make quantifiable claims about the level of privacy protection achieved by such randomization.

Challenges. Despite the large body of existing work on differential privacy for general data analysis, attempts to apply DP techniques to software event frequency profiling face several unexplored challenges. The major such challenges are as follows. **C1**: How can existing techniques for single-event DP data analysis [5, 20, 43] be generalized to the types of event traces that occur in software run-time frequency event profiling? This challenge requires a solution that allows tunable trade-offs between privacy and profiling accuracy, as well as low-cost randomization techniques. **C2**: How can domain-specific constraints on the frequency profiles be inferred and embedded in the DP analysis? This challenge requires program analysis techniques to extract *a priori* knowledge about relationships between elements of the run-time profile, as well as machinery to incorporate these relationship in the DP profiling analysis. **C3**: What practical accuracy/privacy trade-offs can be achieved for real-world software? While theoretical bounds provide some indication of the inherent properties of DP techniques, it is important to understand the actual performance of these techniques and to provide guidelines for their deployment in realistic scenarios. Such understanding cannot be obtained from existing work.

Contributions. To answer these questions, we developed an approach for DP software event frequency profiling.

(C1) Tunable and efficient DP trace analysis: We define a parameterized randomization approach for traces of run-time events. While prior work considers randomization of a single event [5, 20, 43], we target quantifiable privacy guarantees for entire event traces. In particular, we define a notion of distance between 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 [20, 50], applies efficient randomization on the total trace frequencies.

(C2) Domain-specific consistency constraints: While DP techniques have been designed for general data analysis, we incorporate additional consistency constraints that reflect domain-specific considerations. First, we ensure that 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 infer the constraints for the use case of method call frequency profiling, we define a static analysis of call graphs and control-flow graphs. This novel approach has two advantages: (1) it produces estimates that are consistent with the structure of the true frequencies, and (2) it reduces the error of the estimates by minimizing a suitable objective function, subject to the consistency constraints.

(C3) Achievable privacy/accuracy trade-offs: 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. Furthermore, our experiments also indicate that (1) the set of hot methods can be identified accurately while providing significant privacy guarantees, (2) the "hiding" of method presence/absence can be successfully accomplished for a very large number of infrequently-executed methods, and (3) the domain-specific consistency constraints significantly improve the accuracy of the estimates. These results are the first to shed light on the achievable accuracy of DP solutions for software event frequency profiling.

2 Background

2.1 Differential Privacy

DP can be applied to the collection and analysis of information from individuals, in a manner that protects the privacy of the individuals participating in this collection. This 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 (as discussed shortly). 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 [47]. 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.

DP machinery has been studied in two scenarios: the *curator model* and the *local model*. The first scenario assumes a trusted centralized data curator, while the second one does not rely on such a trusted entity. In our work we consider the local model: the private data collected by a local profiling analysis on the user's instance of the targeted software is *not* released directly to the remote analysis server. Rather, the private data is randomized before being sent to the server. This approach protects the user not only from malicious actors that intercept the communication with the server, but also from the server itself (which is controlled by the developers of the software or by a third party working on their behalf). 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 [5, 20]. 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. The problem we consider in our work is a generalization of this exemplar DP analysis, as described in Section 3.

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 $R : \mathcal{V} \to \mathcal{Z}$, 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 $\epsilon \geq 0$. Higher values of ϵ imply a 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}. Here $Pr[A]$ is the probability of event A.

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.

Several LDP approaches for this problem [5, 20, 43] employ *randomized response*, a survey technique first used in the social sciences to eliminate evasive answer bias [46]. In its simplest manifestation, for given input v the randomizer will report v with some probability (derived from ϵ) and, further, will report each $v' \neq v$ with another probability (also derived from ϵ). After such collection over all individuals is completed, a post-processing step scales back the population-wide frequencies to account for the randomization effects. Our work explores several generalizations for this approach. In this exploration, the key question is this: *how can the strong privacy guarantees of DP be achieved by a software frequency profiling analysis, with high accuracy and low cost?*

3 Problem Statement for Our Work

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 $v_1^i, \ldots, v_k^i$ that are observed and recorded by the analysis infrastructure. The frequency of each $v \in \mathcal{V}$ in this trace is recorded locally 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 $G(v) = \sum_i f_i(v)$ 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 [48]. 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.

Differential privacy 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 randomized versions of these frequencies, derived in a way that ensures DP properties. While per-user information is now noisy, the global frequency estimates $\hat{G}(v)$ inferred by the analysis server are accurate estimates of the true global frequencies $G(v)$. This section describes the details of this problem.

3.1 Problem Statement

Consider a software user i and her trace of events $T_i = v_1^i, \ldots, v_k^i$. 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 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 vector F_i based on trace T_i, a non-private solution reports F_i to the server, where a vector $G = \sum_i F_i$ of true global frequencies is reported.

A DP solution applies a randomized algorithm R to the local trace T_i, as described in Section 4.1. We will use $R(F_i)$ as shorthand for the frequency vector of this randomized trace. $R(F_i)$ 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 $\hat{G}$ of global frequency estimates (described in Section 4.2). 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 $R(F_i)$ reported to the server (and, in the extreme, the server is also considered to be potentially adversarial).

3.2 Privacy Guarantee

The privacy guarantees we define are based on the notion of indistinguishability outlined earlier. Specifically, given some privacy loss parameter ϵ, consider a pair of traces T and T' and their frequency vectors F and F'. Then for any vector Z, we want to ensure that the probabilities $Pr[R(F) = Z]$ and $Pr[R(F') = Z]$ 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 [10]. Consider two traces T and T', each containing k events. Our definition is based on a threshold t of the difference between the traces: specifically, the number of trace positions $1 \le j \le k$ 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. R is *ϵ-t-differentially private* if $\forall T, T', Z$, it is true that $d(T, T') \le t$ implies $\frac{Pr[R(F)=Z]}{Pr[R(F')=Z]} \le e^\epsilon$.

If a pair's difference exceeds the threshold t, 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 t. When $t = k$, indistinguishability holds for all possible pairs of traces. By varying the value of t, 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 t.

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 $F(v) - t$ and $F(v) + t$. Though she can still draw conclusions about v, the strength of these conclusions will be weakened based on the threshold t. Still, an adversary can still 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.

3.3 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:

$(5 \quad 0), (4 \quad 1), \ldots, (0 \quad 5)$ 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 4.1. Suppose we choose $\epsilon = \ln 9$, as done in prior work [20], and $t = 1$. The randomizer uses a probability $p = e^{\frac{\epsilon}{2t}}/(1 + e^{\frac{\epsilon}{2t}})$ to randomize each event in the trace. In this example, $e^{\frac{\epsilon}{2t}} = 3$ 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 $(4 \quad 2)$ 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 t)? The table below summarizes the probabilities for the 6 possible frequency vectors F, for $t = 1$ as well as for $t = 2$. Note that each value of F could be produced by several different traces T; the shown probabilities for that F apply for each such trace.

F	$Pr[R(F) = (4 \quad 2)]$	
	$t = 1$	$t = 2$
$(5 \quad 0)$	0.1043	0.1009
$(4 \quad 1)$	0.1265	0.0848
$(3 \quad 2)$	0.0746	0.0606
$(2 \quad 3)$	0.0247	0.0378
$(1 \quad 4)$	0.0061	0.0214
$(0 \quad 5)$	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 $t = 1$, for any traces T and T' with $d(T, T') \leq 1$, the corresponding frequency vectors F and F' can differ by at most one count—e.g., they could be $(5 \quad 0)$ and $(4 \quad 1)$. 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 $(4 \quad 2)$. 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 $t = 1$ is 98.28 (while for $t = 2$ this ratio is 9, as discussed below).

When $t = 2$, 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 t—e.g., traces with frequency vectors $(5 \quad 0)$ and $(0 \quad 5)$. In our example, the largest ratio of probabilities shown in the table for $t = 2$ is 9.

3.4 Privacy for Presence/Absence in the Trace

One important implication of Definition 3.1 is the following. Suppose that for some v we have $\leq t$ occurrences in a trace T. There are many possible traces T' in which v does not occur at all and $d(T, T') \leq t$. If we employ an ϵ-t-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 $F(v) \leq t$ 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 $t = 2$, the presence/absence of any a events is obfuscated when the actual trace has frequencies $(0 \quad 5)$, $(1 \quad 4)$, or $(2 \quad 3)$ regardless of what is the output of the randomizer.

One extreme case of such obfuscation is to zero out the frequency when $F(v) \leq t$. However, by removing this local information, aggregate information about v is also discarded. Instead, an ϵ-t-DP scheme preserves v's frequency distribution across the entire population, in addition to providing strong protection for its presence/absence.

4 Differentially Private Profiling

4.1 Efficient Randomization

To design an ϵ-t-DP randomizer, we use an approach that is a generalization of existing techniques for randomizing a single item per user [5, 20, 43]. Specifically, we define a probability

$$p = e^{\frac{\epsilon}{2t}}/(1 + e^{\frac{\epsilon}{2t}}) \tag{1}$$

For each event v_j^i in the trace $v_1^i, \ldots, v_k^i$ of user i, the randomizer will increment the count for v_j^i with probability p (and will keep it the same with probability $1 - p$). In addition, when v_j^i is observed, each $v' \in \mathcal{V} \setminus \{v_j^i\}$ 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 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 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 $R(F)$. Consider some v and the number of its occurrences $F(v)$. Each of those occurrences would have contributed to v's count in $R(F)$ 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 $f(n, m, p) = \binom{n}{m} p^m (1-p)^{n-m}$. 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 $R(F)$ 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 $f(k - F(v), m, 1-p)$. Then the frequency of v in $R(F)$ 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 $[0, n]$.

4.2 Server-Side Computation of Estimates

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

$$\hat{G}(v) = ((e^{\frac{\epsilon}{2t}} + 1)\hat{F}(v) - nk)/(e^{\frac{\epsilon}{2t}} - 1) \quad (2)$$

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

After this processing, we have an estimate $\hat{G}(v)$ for each v. However, these estimates do not satisfy two categories of *consistency constraints*. First, there is no guarantee that $\hat{G}(v) \geq 0$ and $\sum_v \hat{G}(v) = 1$. 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, $G(m') \leq G(m)$. However, it is not necessarily the case that in the computed estimates we have $\hat{G}(m') \leq \hat{G}(m)$. More generally, we would like to consider static code structures that imply inequality constraints of the form $G(v) \leq G(v')$ 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 [45] 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 $\hat{G}(v)$ and (unknown) estimates $x(v)$ that satisfy the consistency constraints. The specific optimization problem we define has the following form:

$$\begin{aligned} \min_{x(v)\in\mathbb{R}} \quad & \textstyle\sum_v \left(x(v) - \hat{G}(v)\right)^2 \\ \text{s.t.} \quad & x(v) \geq 0 \\ & \textstyle\sum_v x(v) = 1 \\ & x(v) \leq x(v') \end{aligned}$$

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 $G^*(v)$ denote the value for $x(v)$ computed by the solver. This value $G^*(v)$ 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.

Algorithm 1: Find $F(m) \leq F(m')$

```
1 foreach m ∈ 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) ≤ 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′) ≤ F(m)
```

However, since DP is immune to post-processing [17], the DP guarantee still holds for the resulting estimates.

4.3 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 [4]. 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 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'.

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 [37] 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.

5 Implementation and Evaluation

5.1 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 [50]. We then applied the Soot analysis toolkit [37] to determine the set $\mathcal{V}$ of methods in each app. Table 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 "$|\mathcal{V}|$". We excluded several third-party libraries from this count (e.g., `butterknife` and `okhttp`).

Next, we utilized the Monkey tool for random GUI testing [22] 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 F_i for $1 \leq i \leq 1000$ were constructed for each app.

Table 1. Benchmarks.

App	Stmts	$\|\mathcal{V}\|$	≤ 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

5.2 Implementation

Static analysis. Our implementation of the static analysis of inequality constraints for method frequencies was outlined in Section 4.3. Column "≤ Pairs" in Table 1 shows the number of pairs (m, m') such that $G(m) \leq G(m')$ 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.

Client side. Recall from Section 4.1 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 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 F_i frequency vectors.

Server side. After receiving the randomized vectors from the n clients, the server first generates an aggregated vector $\hat{G}$ consisting of the estimates of frequencies after post-processing, as discussed in Section 4.2. Then, for the quadratic optimization problem, it invokes the quadratic programming solver in MATLAB's Optimization Toolbox [27].

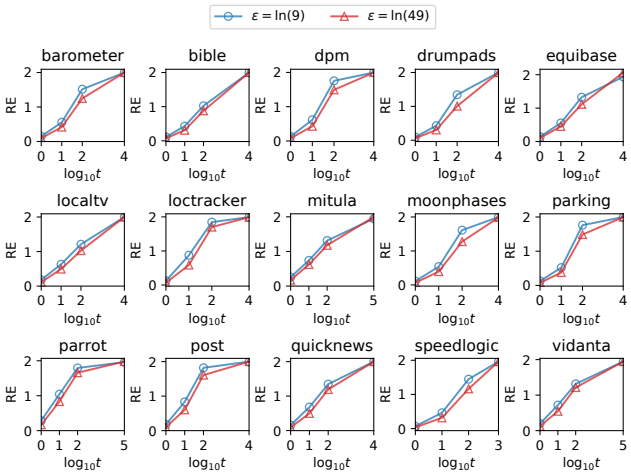

Figure 1. Relative error of estimates.

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.

5.3 Accuracy of Estimates

We evaluate the proposed techniques by varying the threshold t for the difference between two traces and the privacy loss parameter ϵ. In particular, we consider $\epsilon \in \{\ln 9, \ln 49\}$ and $t \in \{10^0, 10^1, 10^2, k\}$. The values of ϵ are the same as those used in prior work [20]. We also collected results for $\epsilon \in \{\ln 25, \ln 81, \ln 121\}$. Due to space constraints, this data is not included here but is available at http://web.cse.ohio-state.edu/presto. The conclusions presented in this section also apply to this additional data.

All ground-truth frequencies $G(v)$ are normalized by nk so that $\sum_v G(v) = 1$. Recall that the estimates $G^*(v)$ produced by the quadratic programming optimization also have a sum of 1. For each app, we run 100 independent experiments and report the mean under each combination of t 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:

$$\text{RE} = \frac{\sum_{v \in \mathcal{D}} |G(v) - G^*(v)|}{\sum_{v \in \mathcal{D}} G(v)} \tag{3}$$

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

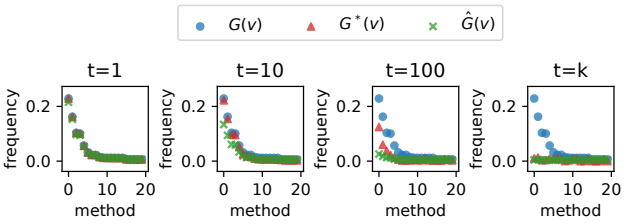

Figure 2. Ground-truth G and estimates G^* and $\hat{G}$ of 20 most frequently-executed methods in speedlogic with $\epsilon = \ln 9$.

discuss additional results where $\mathcal{D} \subset \mathcal{V}$. Smaller values for RE means higher accuracy.

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

Next, consider the effects of choosing t. From Figure 1, one can observe that higher values of t 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 t grows in order to "hide" the t different events between two traces. When $t = k$, 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 G and its estimate G^* 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 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 $\epsilon = \ln 9$. Figure 2 shows the values of G and G^*, as well as $\hat{G}$ 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 that is invoked whenever there is new data from the accelerometer sensor. For these hot methods, the estimation errors are small when $t \leq 100$. When t 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,

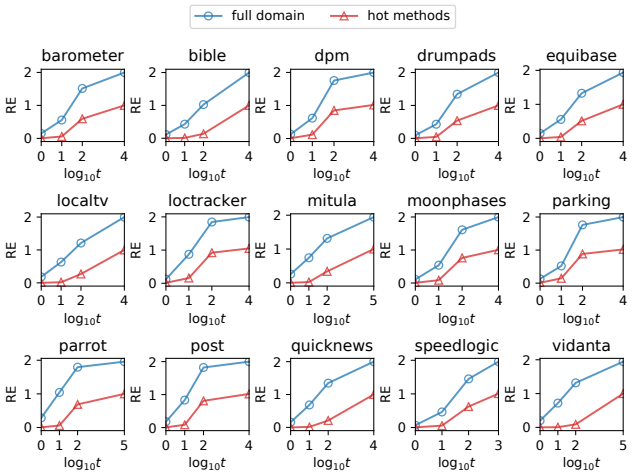

Figure 3. Comparison of RE for all methods (i.e., the full domain $\mathcal{V}$) and for hot methods with $\ell = 0.25$ and $\epsilon = \ln 9$.

and since the number of such methods is very large, their accumulated error is an essential source of RE.

5.4 Estimates for Hot Methods

To quantify the observations described above, we compute RE for hot methods only. Following prior work [51], the set of hot methods is defined based on a threshold $0 \leq \ell \leq 1$. Given a frequency vector F, the set of hot methods in F is defined by $\text{hot}(F, \ell) = \{v \mid F(v) \geq \ell \times \max_v F(v)\}$. That is, hot methods are ones with frequencies close to the frequency of the hottest method [51]. Next, we follow the procedure in Section 5.3 to compute RE for hot methods, with $\mathcal{D} = \text{hot}(F, \ell)$ in Equation 3. Besides ϵ and t, we also alter the limit ℓ. Figure 3 shows RE values when $\epsilon = \ln 9$. We omit $\epsilon = \ln 49$ since its effects are similar to the ones outlined in Figure 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 the metrics for $\ell = 0.25$. The corresponding RE values from Figure 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 $10 \leq t \leq 100$. As discussed later in Section 5.5, even such relatively small values of t 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

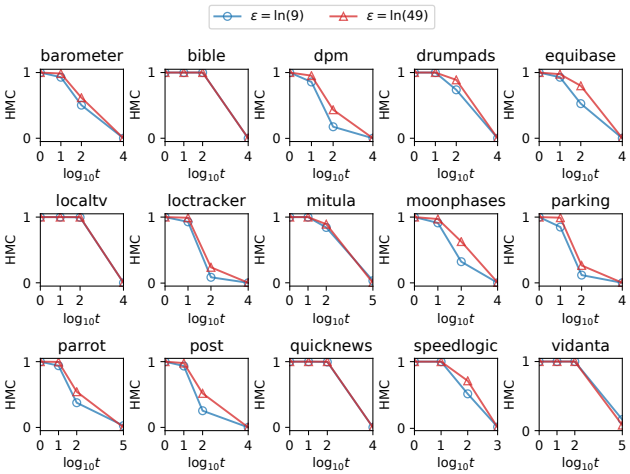

Figure 4. HMC for $\ell = 0.25$.

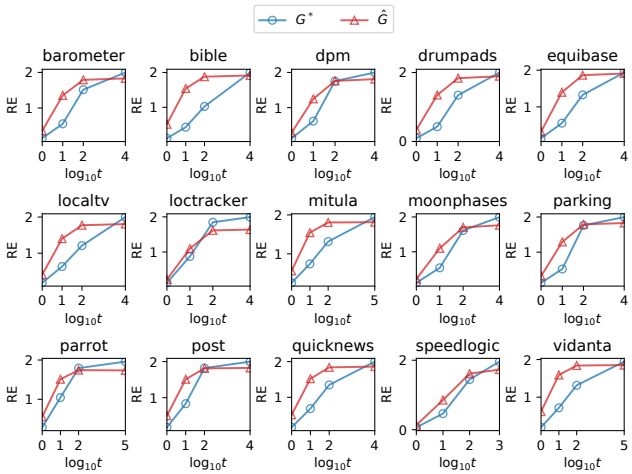

Figure 5. Full-domain RE of G^* and $\hat{G}$ with $\epsilon = \ln 9$.

defined by others [51]:

$$\text{HMC}(\ell) = \frac{|\text{hot}(G, \ell) \cap \text{hot}(G^*, \ell)|}{|\text{hot}(G, \ell)|} \tag{4}$$

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

Figure 4 shows the average HMC across 100 independent repetitions of the same experiment for $\epsilon = \ln 9$ and $\epsilon = \ln 49$, with different values of t. For smaller values of t, the set of hot methods is identified very accurately. For example, when $t = 1$, 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 t grows to 10. This suggests that reasonable accuracy can be achieved with proper values of t.

5.5 Presence/Absence of Infrequent Methods

As discussed in Section 3.4, Definition 3.1 implies privacy protection of the absence/presence of methods with local frequencies $F(v) \leq t$. 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 $t \geq 10$, and around 87% of methods are protected even when $t = 1$. 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 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.

5.6 Importance of Consistency Constraints

Recall from Section 4.2 that we compute estimates $G^*(v)$ based on domain-specific consistency constraints. To evaluate the effects of these constraints, we also compute estimates $\hat{G}(v)$ without the quadratic programming step, as described in Section 4.2. For proper comparison with $G^*(v)$, estimates $\hat{G}(v)$ are then normalized by their sum. Figure **??** shows the RE values for the two categories of estimates. For all apps, the enforcement of consistency constraints significantly improves the accuracy when $t \leq 10$. The RE of G^* is 2.5× smaller than the RE of $\hat{G}$ when $t = 1$ and 2.2× smaller for $t = 10$, averaged across all apps. For most apps, there are also accuracy benefits when $t = 100$.

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 2 shows G, G^*, and $\hat{G}$ 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.

5.7 Summary

Our 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.

6 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.

7 Related Work

Differential privacy. There exists a large body of work on differential privacy in the context of distribution estimation [14], clustering [33], heavy hitters [5, 9], learning [25], and convex optimization [39]. Several applications of LDP have been realized in practice [2, 13, 20, 32, 41, 42]. Many of these studies focus on the single-item scenario where each user holds one data item. Google deploys RAPPOR [20] in Chrome to learn the distribution of users' homepage URLs by applying randomized responses and Bloom filters. Bassily and Smith [6] consider one single categorical event per user and propose an asymptotically optimal solution for building succinct histograms under LDP. Set-valued data have also been studied. Apple adopts LDP to collect words and emojis typed by its users [2]. LDPMiner [35] uses a two-phase strategy to find frequent items. Wang et al. [44] discover not only frequent items but also frequent itemsets. Both approaches assume that each user holds a set of items, while our work focuses on the more challenging problem where the local data is a trace that contains duplicates. Wang et al. [45] employ consistency constraints in the post-processing of estimates. Our constraints are more general as they consider relationships inferred via static analysis.

Our previous work [50] is focused on frequency profiling of Google Analytics events in mobile apps. This prior effort employs a different definition of privacy protection. Specifically, it considers indistinguishability of entire traces and thus provides DP guarantees defined by $k\epsilon$. In contrast, here we define a stricter ϵ-t-DP approach. The previous approach provides much weaker privacy guarantees, especially considering that k could be many thousands. In addition, here we propose efficient randomization based on binomial distribution, which reduces the overhead of randomization by a factor of $O(k)$, since in the prior work each event is perturbed as soon as it is observed. Further, our current work adopts static analysis to construct consistency constraints and utilizes quadratic programming to enforce these constraints and thus achieve better accuracy.

Profiling and its privacy. Profiling of remote software executions has been widely adopted for testing and debugging [7, 18, 24, 34, 49] and optimization [1, 23, 28, 29, 38, 40]. Liblit et al. [26] propose a low-overhead approach to gather run-time events such as assertions for bug isolation via sampling. Bond and McKinley [8] propose a hybrid instrumentation and sampling approach for continuous path and edge profiling. Nagpurkar et al. [29] propose an instruction-based profiling approach for deployed software. Ricci et al. [36] track garbage collection events to help the development of new garbage collection algorithms. Privacy has also been considered. Elbaum and Hardojo [19] marshal and label data with the encrypted sender's name for anonymization at the deployed site. However, anonymization is not enough to ensure strong privacy [30, 31]. We develop the first general approach for DP software event frequency profiling, which provides well-defined privacy guarantees by design and could be adopted in the development of principled privacy-preserving versions of the aforementioned techniques.

8 Conclusions

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.

Acknowledgments

We thank the reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1907715.

References

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

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

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

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

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

[6] R. Bassily and A. Smith. 2015. Local, private, efficient protocols for succinct histograms. In *STOC*. 127–135.

[7] M. D. Bond, G. Z. Baker, and S. Z. Guyer. 2010. Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses. In *PLDI*. 13–24.

[8] M. D. Bond and K. S. McKinley. 2005. Continuous path and edge profiling. In *MICRO*. 130–140.

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

[10] K. Chatzikokolakis, M. Andrés, N. Bordenabe, and C. Palamidessi. 2013. Broadening the scope of differential privacy using metrics. In *PETS*. 82–102.

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

[12] 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. 2017. 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.

[13] B. Ding, J. Kulkarni, and S. Yekhanin. 2017. Collecting telemetry data privately. In *NIPS*. 3571–3580.

[14] J. Duchi, M. Jordan, and M. Wainwright. 2013. Local privacy and statistical minimax rates. In *FOCS*. 429–438.

[15] C. Dwork. 2006. Differential privacy. In *ICALP*. 1–12.

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

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

[18] S. Elbaum and M. Diep. 2005. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering* 31, 4 (2005), 312–327.

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

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

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

[22] Google. 2019. Monkey: UI/Application exerciser for Android. http://developer.android.com/tools/help/monkey.html.

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

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

[25] S. P. Kasiviswanathan, H. Lee, K. Nissim, S. Raskhodnikova, and A. Smith. 2011. What can we learn privately? *SICOMP* 40, 3 (2011), 793–826.

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

[27] MathWorks. 2019. Optimization Toolbox. https://www.mathworks.com/help/optim.

[28] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai. 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *TPDS* 24, 6 (2013), 1245–1255.

[29] P. Nagpurkar, H. Mousa, C. Krintz, and T. Sherwood. 2006. Efficient remote profiling for resource-constrained devices. *TACO* 3, 1 (March 2006), 35–66.

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

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

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

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

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

[35] Z. Qin, Y. Yang, T. Yu, I. Khalil, X. Xiao, and K. Ren. 2016. Heavy hitter estimation over set-valued data with local differential privacy. In *CCS*. 192–203.

[36] N. Ricci, S. Guyer, and J. Moss. 2013. Elephant tracks: Portable production of complete and precise GC traces. In *ISMM*. 109–118.

[37] Sable. 2019. Soot analysis framework. http://www.sable.mcgill.ca/soot.

[38] D. Saha, P. Dhoolia, and G. Paul. 2013. Distributed program tracing. In *ESEC/FSE*. 180–190.

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

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

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

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

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

[44] T. Wang, N. Li, and S. Jha. 2018. Locally differentially private frequent itemset mining. In *S&P*. 127–143.

[45] T. Wang, M. Lopuhaä-Zwakenberg, Z. Li, B. Skoric, and N. Li. 2019. Consistent and accurate frequency oracles under local differential privacy. *arXiv:1905.08320* (2019).

[46] S. Warner. 1965. Randomized response: A survey technique for eliminating evasive answer bias. *J. Amer. Statist. Assoc.* 309, 60 (1965), 63–69.

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

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

[49] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang, and W. Ma. 2006. Automated known problem diagnosis with event traces. In *EuroSys*. 375–388.

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

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