Differentially-Private Control-Flow Node Coverage for Software Usage Analysis

Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev
The Ohio State University

Abstract

There are significant privacy concerns about the collection of usage data from deployed software. We propose a novel privacy-preserving solution for a problem of central importance to software usage analysis: control-flow graph coverage analysis over many deployed software instances. Our solution employs the machinery of differential privacy and its generalizations, and develops the following technical contributions: (1) a new notion of privacy guarantees based on a neighbor relation between control-flow graphs that prevents causality-based inference, (2) a new differentially-private algorithm design based on a novel definition of sensitivity with respect to differences between neighbors, (3) an efficient implementation of the algorithm using dominator trees derived from control-flow graphs, (4) a pruning approach to reduce the noise level by tightening the sensitivity bound using restricted sensitivity, and (5) 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.

1 Introduction

Usage data generated by deployed software provides extensive information about users' interactions with this software. Such data can be utilized for user behavior analytics, targeted advertisement, and business decision making [72], as well as to facilitate testing [7, 59], bug isolation [43], failure reproduction [13, 35] and profiling [21, 51, 66]. A prominent recent example of such data collection is the widespread use of web and mobile app analytics infrastructures provided by Google, Facebook, and Yahoo. For instance, a study of 65K popular Android apps showed that Google Firebase code is present in 45% of them [23].

There are significant privacy concerns about the collection and use of such data. While data analysis results can be used for enhancement of app design and marketing purposes, individuals' usage data becomes transparent to software developers (and analysts working with them), as well as to the analytics service providers such as Google and Facebook. 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.

Each user's usage of an app can be characterized by the run-time *control flow* with respect to the execution of software components within the app. Control-flow privacy may be needed to protect sensitive 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. As further elaborated in Section 2, many analytics platforms, including Facebook [24], Firebase [28], and Flurry [56], 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 1 illustrates this process. Each directed graph on the left represents the control-flow behavior of one user's copy of the software. A node represents a software component 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 focus of our work is a fundamental usage analysis for deployed software: the collection of *control-flow graph node coverage* information over many instances of a software application (detailed in Section 3). This abstracted problem 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

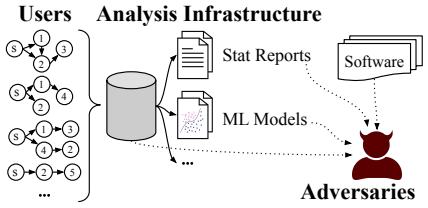


Figure 1: User interactions are reported for data analysis.

individual code statement. We aim to develop a *privacy-preserving* version of this analysis. Specifically, our goal is to introduce a privacy mechanism that, in a principled and quantifiable manner, hides the presence/absence of any particular graph node in a user's coverage information. We develop a privacy-preserving solution for this problem using the machinery of *differential privacy* (DP). To the best of our knowledge, this is the first work to develop a DP control-flow node coverage analysis. 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 our work is a novel privacy definition that accounts for the causal relationships between graph nodes, based on the structure of the control-flow graph.

1.1 Motivation and Problem Overview

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.

Privacy-preserving software analysis. Privacy-preserving data analysis is designed, from the ground up, with guarantees about the loss of privacy and the accuracy of analysis results. The last decade has witnessed the rise of a foundational theory to deal with this challenge, centered around the robust mathematical notion of differential privacy [19] and its extensions and generalizations [6, 49]. DP has recently found several adoptions in industry and government—for example, in the Chrome browser [22, 26], in iOS 10 [2], in Uber [70], and by the U.S. Census Bureau for the 2020 census [15].

Despite significant advances in DP theory, applying such solutions to software analysis is challenging due to the mismatch between traditional DP problem statements/solutions and software analysis needs. We aim to narrow this gap by developing a novel DP solution for the coverage analysis described earlier and illustrated in Figure 1. We target this problem because coverage analysis has various uses (e.g., coverage monitoring [59] and mobile app GUI flow analytics [29]) and also captures essential sub-components of many other analyses of deployed software (e.g., impact analysis, regression testing, failure reproduction, statistical debugging, and performance profiling [7, 21, 32, 43, 51, 58, 66]).

1.2 Challenges and Contributions

The design of our solution demonstrates the key components of a DP software analysis. In Section 4.1, we first define the space of possible data instances—in our case, 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. DP analyses employ the notion of neighbors when defining privacy guarantees: namely, that (in a probabilistic sense) an adversary cannot distinguish between the actual data item and its neighbor data items by observing the analysis results. 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 [9, 39, 40, 45, 74]. In particular, we focus on commonly-occurring correlations due to the *causality* between nodes. In control-flow graphs, 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. For the example shown earlier, the execution of both `a` and `b` should be protected if there are no other calls to `b`. 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 to prevent such causality-based inferences.

We then propose an analysis to achieve these guarantees (Section 4.2) 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. This sensitivity is directly related to both the design and the accuracy of a DP analysis. We define a new notion of sensitivity for graph data and demonstrate how to compute it efficiently using the dominator tree of the dynamic control-flow graph.

Next, we describe a baseline randomizer using the worst-

case upper bound for sensitivity (Section 5.1). This approach achieves the theoretically-optimal worst-case analysis error, but does not provide good accuracy on real data. This is a fundamental limitation that requires further refinements of accuracy/privacy goals and trade-offs. In Section 5.2 we introduce stronger bounds on sensitivity by projecting onto a lower-sensitivity representation, which leads to better accuracy. Our experimental results show 2× error reduction, compared to the baseline approach (Section 6). We also propose to refine the notion of indistinguishability to account for the distance between graph neighbors. As a result, further accuracy gains can be achieved by allowing varying privacy protection across neighbors (Section 5.3). Experiments show that this approach leads to 5.4× error reduction. All experiments are conducted using our randomization layer on top of existing Android analytics libraries.

Summary. We develop the first solution for differentially-private control-flow node coverage analysis. This contribution is important both as a solution to a core software usage analysis and as a step in a broader research agenda to develop privacy-preserving analyses of deployed software.

2 Background

2.1 Threats and Goal

This paper focuses on control-flow data—that is, run-time data generated with respect to some control-flow model of the software. Such a model could come from software design documents or from software analysis. Run-time behaviors relative to such models have been studied extensively by researchers and have been targeted by many techniques for remote analysis of deployed software. One prominent example of such data collection is the widespread adoption of frameworks for web and mobile app analytics [23, 42]. They allow developers to perform data analytics that generates interesting control-flow coverage related statistics, such as event histograms, as well as more sophisticated user behavior models, such as users' routines of using an app. In addition to population-wise behaviors, data analytics also aims at finding specific control-flow patterns by individual users for purposes such as targeted advertisement. In a typical usage scenario for existing analytics frameworks for web/mobile apps [28, 29, 56], each user is assigned a pseudonym (user identifier) that is chosen by the application and developers can make further requests to other services relying on the pseudonym [64]. This mechanism allows developers to track and connect seemingly independent actions and devices by the same user.

We consider each user's local copy of the software and the physical device that it runs on to be secure and not compromised. A broad definition of adversary consists of any party beyond these, including the analysis service providers (e.g., Google and Facebook), the analysts who access the collected data, as well as the network providers. The software and its source code, as well as the information sent to the remote analysis servers, are visible to the adversary. We assume that the adversary also has access to each user's pseudonym as described earlier—that is, she knows the identity of each user by some auxiliary data such as emails and IP addresses.

In our threat model, an adversary tries to identify the control-flow data at each user, more specifically, *to estimate with high probability whether a particular software component has been executed by a user*—for example, an app GUI screen containing sensitive content that can be used to classify users' interests, or a code function to reset a password. Here the execution can be represented by a run-time control-flow graph instance and each component corresponds to a node in the control-flow graph model. The adversary's aim is to discover the presence of a specific node in a given run-time graph instance with high probability. Such information is a key building block in attempts to infer user-specific patterns of behavior. Existing analytics infrastructures provide *no* protection for this scenario, since the raw control-flow data for a user (sent to and stored on remote servers) is also accessible to the adversary, as illustrated in Figure 1.

One natural solution is to perturb the information reported to the remote server in order to hide the execution of software components. Various techniques for perturbation of control-flow information could be considered based on the information available to the adversary. For example, one could imagine a sophisticated adversary who utilizes *a priori* information about the distribution and probabilistic associations of components, and performs statistical inference based on such information [45]. We are not aware of any work that considers such advanced scenarios. Instead, we focus on the more practical scenario where the adversary could exploit the *causal relationships* between nodes in the control-flow graph. An example discussed earlier illustrated such relationships: suppose that the execution of a node n_2 implies that another node n_1 was executed because n_2 can only be caused by n_1. This type of strong correlation, which is typical for software control flow, could potentially be used by an adversary. Our goal is to *prevent adversaries from such causality-based analysis by data randomization, while still allowing analysts to draw useful empirical statistical conclusions of software usage across all users*. Specifically, we define the software usage analysis problem as a *control-flow graph node coverage* problem (in Section 3). We regard the presence/absence of each graph node at users as private information and propose to utilize differential privacy [18, 20] to prevent inference analysis of run-time nodes based on their causal relationships.

2.2 Differential Privacy

Differential privacy (DP) [18, 20] 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 [17, 52, 53], because additional data sources can be combined with the anonymized data to uncover sensitive information. Differential privacy has emerged as a prominent approach for protection against privacy attacks. We will not attempt a detailed description of this rich field of research; extensive overviews are available elsewhere [20, 55].

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 analysis. This model provides privacy guarantees to the software user regardless of the unpredictable actions from the software analysis infrastructure and its clients and adversaries.

More formally, an ε-LDP protocol/algorithm applies an ε-local randomizer $R : \mathcal{D} \to \mathcal{T}$ to each user's item $v \in \mathcal{D}$. The software analysis infrastructure/server collects all $R(v)$ from users for data analysis and provides the results to the client—that is, to the software developer/analyst. The privacy is due to the ε-local randomizer R such that $\forall v, v' \in \mathcal{D}, t \in \mathcal{T}$: $\Pr[R(v) = t] \leq e^{\varepsilon} \Pr[R(v') = t]$. Thus, by observing the output t of the local randomizer (as reported to the remote server), an adversary cannot distinguish with high probability the case when the private data is v from the case when this data is v'. This holds even if the adversary has additional knowledge (beyond R and t) from auxiliary sources. The privacy budget $\varepsilon \geq 0$ defines the strength of privacy protection.

As an example of an ε-local randomizer, consider a single bit element v that is either 1 or 0. *Randomized response* [71] flips the bit with probability q and keeps the bit with probability $1-q$. This simple randomizer satisfies the above definition with $\varepsilon = \ln \frac{q}{1-q}$.

2.3 Privacy for Graphs

Differentially-private analysis of graph data has been considered almost exclusively in the centralized model [37, 38, 54, 61]. Two graph privacy definitions have been proposed. *Node privacy* [61] 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* [37, 54], 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 [61]. We focus on this more challenging problem. As we argue in the next sections, this notion of node privacy cannot be applied directly to analysis of control-flow graphs due to causal relationships between nodes, and novel definitions (and the related new analysis algorithms) need to be developed.

3 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 $G = (N, E, s)$ with node set N and edge set $E \subseteq N \times N$. The start node $s \in N$ 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 $c \subseteq N$ is the set of nodes covered during the program run. *Node coverage analysis* reports the set of nodes c. Alternatively, c could be thought of as a binary coverage vector $c \in \{0,1\}^{|N|}$ which is the indicator vector of the corresponding set of covered nodes. In a minor abuse of notation, we will use c to denote both a set of covered nodes and its corresponding coverage vector.

Information about node coverage plays an important role in the area of remote analysis of mobile and web apps, using analysis infrastructures such as Google/Firebase Analytics [28, 29] and Facebook Analytics [24]. 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 [59] 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 [3, 7, 11, 21, 33, 34, 43, 44, 58, 66, 73].

Differentially-private node coverage analysis. Consider m software users identified by integer ids $i \in \{1, \ldots, m\}$. 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 $c_i \in \{0,1\}^{|N|}$ 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 $n \in N$, the frequency of n's coverage across all users—that is, $f(n) = |\{i \in \{1,\ldots,m\} \mid n \in c_i\}|$. Equivalently, the goal is to obtain an aggregate vector $f \in \mathbb{N}^{|N|}$ such that $f = \sum_i c_i$, where the summation is element-wise for vectors c_i. In a differentially-private setting, instead of f the developer will obtain an estimated aggregate vector $\hat{f}$ where, with high probability, the node frequency estimates $\hat{f}(n)$ are close to the actual node frequencies $f(n)$. 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 Google/Firebase Analytics [28, 29]. 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 [59].

An LDP coverage analysis applies an ε-local randomizer $R : \{0,1\}^{|N|} \to \{0,1\}^{|N|}$ to each user's observed coverage c_i. The resulting $z_i = R(c_i)$ is sent to the server. The server collects all z_i and uses them to compute the estimates $\hat{f}$.

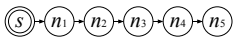
4 Privacy-Preserving Coverage Analysis

In this section, we first define the *neighbor* relation for dynamic graphs G_c and, by extension, the corresponding coverage vectors c. This definition is specifically crafted to eliminate the possibility of causality-based inference. We then propose an LDP analysis based on this definition.

4.1 Defining Neighbors

As described in Section 2.3, one key question for achieving node privacy is the definition of neighbors. Using the traditional notion from DP graph analysis [61], a neighbor graph $G_{c'}$ would be obtained from a given G_c by removing a single node and its adjacent edges. Thus, coverage vector c' would differ from 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 1 (Feasibility)**.** *A dynamic graph G_c with start node s and its coverage vector c are* feasible *if $s \in c$ and every covered node is reachable from s along a path of covered nodes and edges, i.e., for any $n \in c$ there exists a path $\langle s, n_1, \ldots, n_k, n\rangle$ in G_c such that $n_j \in c$ for $1 \le j \le k$.*



feasible $c = [111110]$, infeasible $c' = [110110]$, feasible $\Delta_{n_2}(c) = [110000]$

Figure 2: Feasible and infeasible coverage.

Here $n \in c$ denotes that n is in the set of nodes encoded by c. If G_c and 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 2, where s is the start node. Coverage vector c is feasible, as it represents the covered set $\{s, n_1, n_2, n_3, n_4\}$. However, c' which represents $\{s, n_1, n_3, n_4\}$ is not feasible since n_3 and n_4 cannot be reached from s along a path of covered nodes. No software execution can generate c' as a coverage vector.

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_c and some node $n \in c \setminus \{s\}$, the *neighbor graph* $G_{c'} = \Delta_n(G_c)$ obtained by removing n is defined as follows: (1) $G_{c'}$ is a subgraph of G_c, (2) $n \notin c'$, (3) $G_{c'}$ is feasible, and (4) $G_{c'}$ is maximal (i.e., there does not exist a proper supergraph of $G_{c'}$ with properties 1–3). Intuitively, the last constraint ensures that we do not remove "too many" nodes and edges from G_c.

Graph $\Delta_n(G_c)$ exists and is unique, as shown by Lemma 1. The proof of the lemma is deferred to the appendix. For brevity, we will often use $c' = \Delta_n(c)$ to denote that $G_{c'} = \Delta_n(G_c)$ for a given G_c. For illustration, in Figure 2 the removal of n_2 from c requires the removal of n_2 and n_3 as well, in order to preserve feasibility. Thus, the neighbor $\Delta_{n_2}(c)$ is the covered set $\{s, n_1\}$. If one were to use the traditional definition of neighbors described in Section 2.3, the removal of n_2 would produce the infeasible vector c' shown in the figure.

Lemma 1. *Let G_c be a feasible dynamic graph. For any $n \in c \setminus \{s\}$, there exists a unique feasible subgraph $G_{c'}$ such that $n \notin c'$ and $G_{c'}$ is maximal.*

The set of neighbors for the coverage vector c of a given G_c is defined as follows:

Definition 2 (Neighbors)**.** *Given a feasible coverage vector c, its* neighbors *are the set* $\{\Delta_n(c) \mid n \in c \setminus \{s\}\} \cup \{c' \mid \exists n \in c' \setminus \{s\} : \Delta_n(c') = c\}$.

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

Next, we show that $\Delta_n(c)$ for given G_c and n can be constructed efficiently. In a control-flow graph with a start node s,

a node d *dominates* a node n (denoted d *dom* n) if every path from s to n goes through d [1]. A node trivially dominates itself. Given a feasible G_c, let dom_{G_c} denote its dominator relation. The key observation is that the nodes dominated by n (plus their adjacent edges) are exactly the ones that need to be deleted to obtain the neighbor graph:

Proposition 1. *For any node $n \in c \setminus \{s\}$, we have $\Delta_n(c) = c \setminus \{n' \mid n\ dom_{G_c}\ n'\}$.*

The proof is deferred to the appendix. This property allows us to find efficiently all $\Delta_n(c)$ for a given G_c, which is needed for our randomizer (as described later). Consider the *dominator tree* for G_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_c in Figure 2, the dominator tree is the same as the graph itself (root s dominates all nodes, n_1 dominates all nodes except s, etc.). The dominator tree can be constructed efficiently; we use a classic approach by Lengauer and Tarjan [41] with complexity $O(|E|\log|N|)$. Given $n \in c \setminus \{s\}$, the dominator subtree rooted at n provides all and only nodes that should be removed from c to obtain its neighbor $\Delta_n(c)$.

This property also allows us to take into account the *causal relationships* between nodes. A reversed arrow in the dominator tree represents a deterministic causal effect. For the example in Figure 2, $n_2 \to n_1$ in the reversed dominator tree of G_c indicates that the execution of n_2 is *always* caused by the execution of n_1.[1] Thus, the hiding of n_1 must also involve hiding of n_2. In general, using this observation, we are able to perturb the existence of an entire set of nodes $\{n' \mid n\ dom_{G_c}\ n'\}$ and thus hide their execution simultaneously, in order to avoid causality-based inference.

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

4.2 LDP Analysis

Consider again our problem: for user i, coverage vector $c_i \in \{0,1\}^{|N|}$ describes the behavior of that user's code instance. The same local randomizer $R : \{0,1\}^{|N|} \to \{0,1\}^{|N|}$ is used by all users. Each user reports $R(c_i)$ to the analysis infrastructure. All reports are gathered and post-processed to construct an estimate of $\sum_i c_i$. Such analysis, based on Definition 2, can achieve control-flow graph node privacy as follows:

Definition 3 (ε-Node-LDP)**.** *Randomizer R is* ε-node-LDP *if for any pair of coverage vector neighbors c, c' for G from Definition 2, we have $\Pr[R(c) = t] \le e^{\varepsilon} \Pr[R(c') = t]$, where $\varepsilon \ge 0$ is the privacy budget.*

[1] This does *not* imply "executing n_1 always causes execution of n_2".

Sensitivity. To define analyses that satisfy Definition 3, it is important to consider the concept of *sensitivity*. This notion is employed in various forms by many DP analysis algorithms. In our analyses we will use this idea to capture the properties of the "graph neighbor" relation defined earlier.

Definition 4 (Local Sensitivity)**.** *Consider a feasible graph G_c and its corresponding coverage vector c. The* local sensitivity *of c is $LS(c) = \max_{n \in c \setminus \{s\}} |c| - |\Delta_n(c)|$.*

$LS(c)$ captures how sensitive G_c is to the removal of any of its nodes n.[2] Intuitively, the larger the local sensitivity, the more extensive randomization needs to be added by R in order to satisfy Definition 3, since the randomized output has to "hide" the differences between c and any $\Delta_n(c)$. This increased randomization is manifested by an increased probability of flipping any bit in the coverage vector.

Example 1. In Figure 2, consider $c = \{s, n_1, n_2, n_3, n_4\}$. Here we have $\Delta_{n_4}(c) = \{s, n_1, n_2, n_3\}$, $\Delta_{n_3}(c) = \{s, n_1, n_2\}$, $\Delta_{n_2}(c) = \{s, n_1\}$, and $\Delta_{n_1}(c) = \{s\}$. The local sensitivity is $LS(c) = |c| - |\Delta n_1(c)| = 4$.

Given G_c, computing $LS(c)$ is straightforward. Recall that, with the help of Proposition 1, we can efficiently find all $\Delta_n(c)$ by considering the dominator tree for G_c. Suppose each node n in this tree is annotated with the size $sub_{G_c}(n)$ of the subtree rooted at n. Then $LS(c)$ is the largest value of $sub_{G_c}(n)$ among the nodes n that are children of the start node s in the tree.

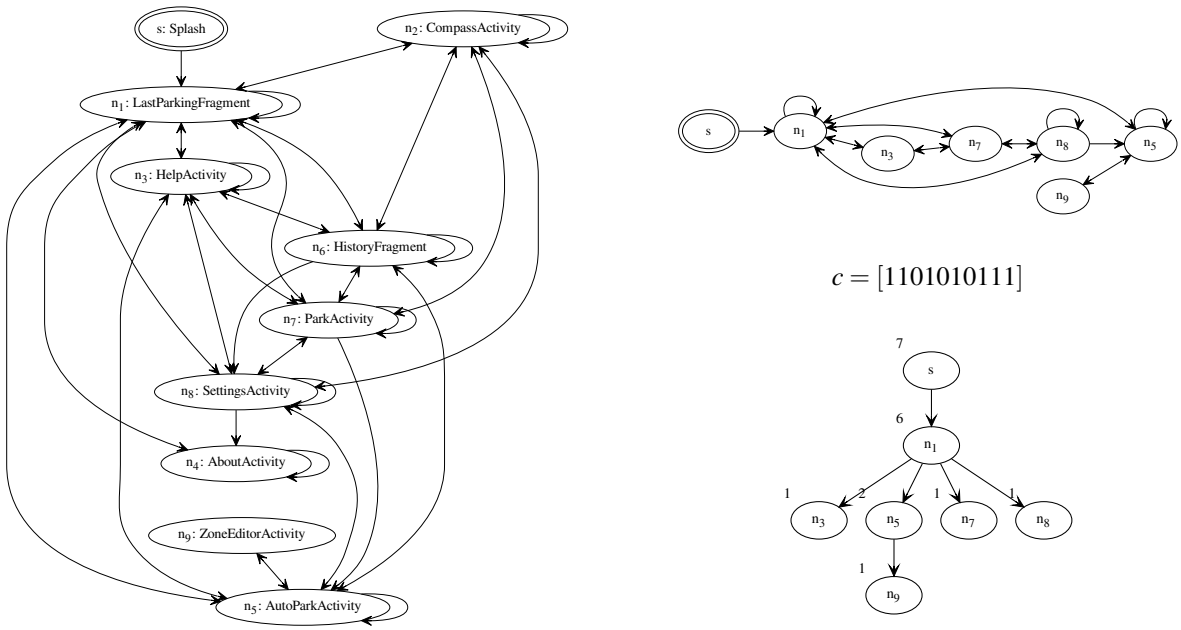
Example 2. Figure 3 shows an example from the `parking` Android app [69]. This app navigates users to parking places, records history of parking locations, and reminds users about parking time. It uses Google Analytics [29] to collect GUI screen view events from users. The developer defines a dictionary of GUI screens to be collected and reported to the Google Analytics remote servers. Figure 3a shows the control-flow model G for this 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_c and its coverage vector $c = [1101010111]$. The graph and its dominator tree are shown in Figure 3b. Each node n in the tree is annotated with $sub_{G_c}(n)$, the size of its corresponding subtree. The local sensitivity for G_c is $LS(c) = sub_{G_c}(n_1) = 6$.

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

Definition 5 (Randomizer)**.** *Given a feasible c, R independently flips each bit in c with probability $p = 1/(1+e^{\frac{\varepsilon}{S}})$.*

Then the following proposition holds:

[2] Since the "neighbor" relation from Definition 3 is symmetric, the sensitivity of adding a node n to G_c will be accounted for by $LS(c')$ for another coverage vector c' such that $c = \Delta_n(c')$.



(a) GUI screen view graph G

(b) G_c, coverage vector c, and dominator tree for G_c

Figure 3: Graphs from the `parking` app.

Proposition 2. *The R from Definition 5 satisfies ε-node-LDP.*

The proof is deferred to the appendix. Each user i applies local randomizer R to add noise to local vector c_i.[3] After the remote software analysis infrastructure collects and reports a histogram $h = \sum_i R(c_i)$ over all users, this noisy data is processed to account for the effects of the randomizers. For any node n, the expected value of the number of occurrences of n in h is $f(n)e^{\frac{\varepsilon}{S}}p + (m - f(n))\,p$ where $f(n)$ is real frequency of n, m is the number of users, and p is probability from Definition 5. If the collected histogram h has a frequency $h(n)$ for n, then the estimate $\hat{f}(n)$ for the real frequency $f(n)$ is

$$\hat{f}(n) = \frac{\left(1 + e^{\frac{\varepsilon}{S}}\right) h(n) - m}{e^{\frac{\varepsilon}{S}} - 1} \tag{1}$$

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

Note that this approach is designed for "one-shot" randomization, i.e., G_c and c are deleted at the user end once $R(c)$ is generated. Any subsequent requests for data will receive the same value of $R(c)$. In contrast, in a framework that allows submission of multiple realizations of $R(c)$, the privacy protection will degrade due to composition [20]. Our approach can prevent such degradation and has practical usage, for example, by Facebook in their ads system [12].

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 Selection of Sensitivity Bound

The choice of probability p in Definition 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 4. 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 [5]. However, as demonstrated by our empirical results (Section 6), the accuracy resulting from the baseline approach is usually modest since the global sensitivity is quite large.

To circumvent this fundamental limitation, we propose al-

[3] 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.

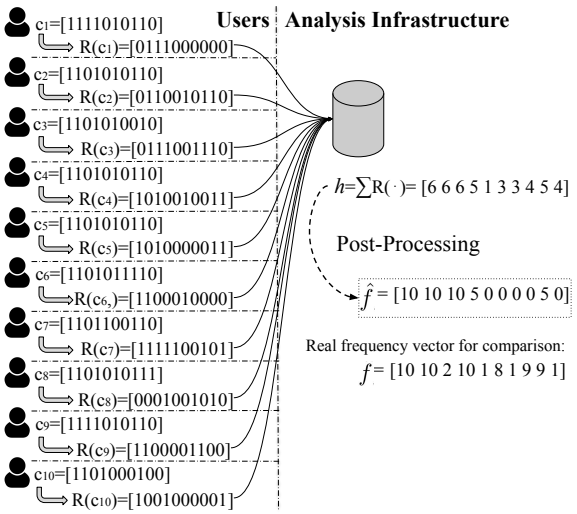


Figure 4: Randomization using global sensitivity S_{gs} and $\varepsilon = 1$ for the `parking` app, with 10 users.

ternative approaches that entail either a relaxation of the utility guarantee (Section 5.2) or a relaxation of the privacy guarantee (Section 5.3). In particular, in Section 5.2, 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 5.3 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 6.

5.1 Baseline: Global Sensitivity

One choice for S in Definition 5 is to consider the worst-case value for $LS(c)$. For our problem, this worst-case value is $S_{gs} = |N| - 1$. Here suffix gs is short for "global sensitivity." For any G and any feasible c for G, $LS(c) \leq S_{gs}$ since, in the worst case, c contains all nodes in N 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 $S = |N| - 1$ to add noise to its local vector. Figure 2 illustrates this case: for $c = \{s, n_1, n_2, n_3, n_4, n_5\}$ and its neighbor $\Delta_{n_1}(c) = \{s\}$, we have $LS(c) = |N| - 1 = 5$.

Example 3. Consider the example of $m = 10$ users for the `parking` app, shown in Figure 4. The sensitivity bound is $S_{gs} = 9$ as there are 10 nodes in the control-flow model from Figure 3a. This bound is *a priori* knowledge to all users. Each user generates her own coverage vector c_i independently and runs R with S_{gs} locally. In this example, ε is set to 1. The analysis infrastructure collects all randomized $R(c_i)$ vectors to get $h = \sum_i R(c_i) = [6\ 6\ 6\ 5\ 1\ 3\ 3\ 4\ 5\ 4]$. Using Equation 1, we then obtain a vector of estimates $\hat{f} = [10\ 10\ 10\ 5\ 0\ 0\ 0\ 0\ 5\ 0]$, with all decimals rounded to the nearest integer.

The real frequency vector is $f = [10\ 10\ 2\ 10\ 1\ 8\ 1\ 9\ 9\ 1]$. 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 $S = |N| - 1 = 100$. 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 $\{0,1\}^{|N|}$. Next, we discuss two techniques that lead to reduction of the noise introduced by the randomization. The potential accuracy improvements were in fact observed in our experiments. As a high-level example, using 15000 dynamic graphs obtained from 15 Android apps, we observed error reduction of 2× and 5.4×, respectively; details are elaborated in Section 6.

5.2 Tighter Bound via Restricted Sensitivity

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 $LS(c) \leq k$ for all $c \in \mathcal{H}_k$. The sensitivity bound is $S = k$ in this case. This technique is similar in spirit to *restricted sensitivity* [6] 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 c is transformed into $\mu(c)$ such that $LS(\mu(c)) \leq k$. Then R is applied to $\mu(c)$. We design μ as follows. For all $c \in \mathcal{H}_k$, we have $\mu(c) = c$. For any other $c \in \mathcal{D}$, we prune G_c according to its dominator relation. Specifically, consider each child node n of the start node s in the dominator tree for which $|c| - |\Delta_n(c)| > k$. (If this condition does not hold, n and its tree descendants do not need to be pruned.) We conduct breadth-first search starting from n and prune the last $sub_{G_c}(n) - k$ traversed nodes from the dominator tree and from G_c. The corresponding bits in c are set to 0.

Example 4. Consider the coverage vector $c = [1101010111]$ and its corresponding dominator tree in Figure 3b. If $k = 5$, as $|c| - |\Delta_{n_1}(c)| = 6 > 5$, by removing the leaf node n_9 at the last level in the subtree, the projection produces $\mu(c) = [1101010110]$. Next, consider an extreme case where $k = 1$. The projection $\mu(c)$ needs to trim from G_c a set of 5 nodes $\{n_3, n_5, n_7, n_8, n_9\}$. The final output of the projection is $\mu(c) = [1100000000]$. Its local sensitivity is $LS(\mu(c)) = 1 \leq k$.

After the projection step, each user reports $R(\mu(c))$ 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 c is lost after the projection. This highlights the trade-offs between privacy and accuracy in any DP analysis. In Section 6, we conduct empirical evaluation on the impact of k and show that practical accuracy can be achieved by properly selecting the value for k.

5.3 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 [8] as a generalization of traditional DP.

We investigate a distance metric d^* based on the difference of each pair of neighboring coverage vectors c and $\Delta_n(c)$. More specifically, $d^*(c,\Delta_n(c)) = \alpha \times |c \setminus \Delta_n(c)| = \alpha \times sub_{G_c}(n)$, where α is a parameter that allows developers to fine-tune the trade-offs between the privacy guarantee and the accuracy of analysis results. We define the privacy budget ε' depending on this metric to achieve (d^*,ε)-privacy [8]: $\varepsilon' = \varepsilon \times d^*(c,\Delta_n(c))$. This can be realized by setting $S = 1/\alpha$ in Definition 5, of which the proof is similar to the one for Proposition 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 3b, we have $\varepsilon' = \varepsilon \times d^*(c,\Delta_{n_1}(c)) = 6\alpha\varepsilon$, while $\varepsilon' = \varepsilon \times d^*(c,\Delta_{n_9}(c)) = \alpha\varepsilon$ 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 6 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 n if its execution is specific for a small group of users, compared to the case where n's execution is deterministic and happens for all users.

As a concrete example, in Figure 3, the "LastParkingFragment" screen (n_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 (n_9) is observed only once. It could be used as a fingerprint for that particular user and thus requires more protection. A vector c and its neighbor $\Delta_{n_9}(c)$ should be indistinguishable after randomization to prevent adversaries from inferring the occurrence of n_9.

This technique is an example of d-privacy [8] 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 [50] 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 [50] 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.

In our experimental evaluation in Section 6.3 and 6.4, we used $\alpha = 0.5$ for comparison with the two techniques introduced earlier. We also obtained data for other values of α and evaluated their effects on accuracy and privacy for this definition d^* of distance metric, as described in Section 6.4.

6 Implementation And Evaluation

6.1 Implementation

We implemented the data collection and randomization as part of a Java library located between the software's application code and any existing analytics libraries. Figure 5 shows an overview of our layer. While this design could be applied to

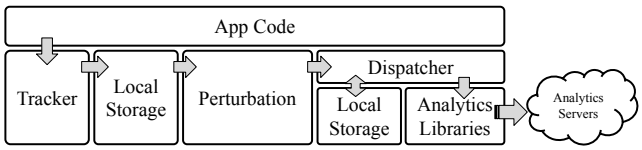


Figure 5: Implementation overview.

other languages and usage scenarios, our specific implementation focuses on the Android platform. The *tracker* component tracks node coverage. Whenever an edge transition occurs, a call to the tracker records the transition and stores it into a local database. The *perturbation* component utilizes Android's `JobService` to regularly query the database to construct coverage vectors and run the randomization. The *dispatcher* component receives the perturbed data and calls corresponding APIs to send it to remote analytics infrastructures. The local storage for the dispatcher is a cache of any unsent data in case the app exits abnormally.

Our implementation supports Google Analytics (GA) [29], a popular framework for collecting data from deployed mobile apps, as its underlying analytics service. A recent study of thousands of Android apps [23] has identified that GA was used by 26% of the analyzed apps. We are in the process of adapting this approach to Firebase and Facebook Analytics, the two other most popular analytics frameworks for Android apps; the implementation details are essentially the same.

We also developed an instrumentation tool that inserts our library into close-source apps using the Soot code rewriting framework [65]. Calls to Google Analytics APIs are redirected to corresponding methods in the tracker component. In our experiments, we use this instrumentation to record screen view events. For call graph data, discussed shortly, we instrumented each method to record any caller-callee relationships on the main thread.

6.2 Data Collection

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 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 [30] to construct a graph $G = (N, E, s)$ 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 1000 executions of the app. To represent the data for each execution, we ran Monkey (independently from any other executions) to obtain $10 \times |N|$ screen view events for that execution. From that trace we determined the coverage vector c and the corresponding subgraph G_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 $10 \times |N|$ method call events. From these traces, call graph coverage vectors c were constructed.

Table 1: Apps and control-flow graph models.

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

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 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 are available at `http://presto-osu.github.io/sec20`.

6.3 Utility Analysis

6.3.1 Metrics

Theoretically, when S is large, the protocol achieves higher privacy (i.e., the probability p in Definition 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 4 and 5, 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 testing and profiling techniques, e.g., residual testing [59]. The answer is the set of nodes that are observed at run time in at least one deployed software instance: $\{n \in N \mid f(n) > 0\}$. Recall that the algorithm from Section 4.2 provides an estimate $\hat{f}$ of the real frequency vector f. Thus, we can estimate the set of nodes by $\{n \in N \mid \hat{f}(n) > 0\}$. 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 profiling, e.g., finding popular app features. We evaluate the accuracy of estimates by computing the errors $|f(n) - \hat{f}(n)|$ and characterizing their distributions, in terms of their minimum, maximum, median, and the first and third quartiles. In addition, we also compute the *mean error* (ME) indicating the average error for each node. We aggregate the errors and then determine the mean of these values across all nodes, i.e., $ME(N, f, \hat{f}) = \sum_n |f(n) - \hat{f}(n)|/|N|$.

6.3.2 GUI Screen Graphs

Answering Q1. We first collected all c_i for $1 \leq i \leq 1000$ to get the ground truth f, as described earlier, and computed $f(n) = \sum_i c_i(n)$ where i ranges over all independent executions that are regarded as individual users. To compute estimates $\hat{f}$, for the same range of i we randomized each c_i independently according to Definition 5, computed $h = \sum_i R(c_i)$, and post-processed h using Equation 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 [4]. 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 trials are due to the randomness when perturbing c_i (since c_i for user i is the same in each trial). Figure 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 a similar figure in the following section.

When applying randomization, we first set the sensitivity bound S to the global sensitivity $S_{gs} = |N| - 1$ (in Section 5.1) to obtain a worst-case baseline. As shown by bars "baseline" in Figure 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

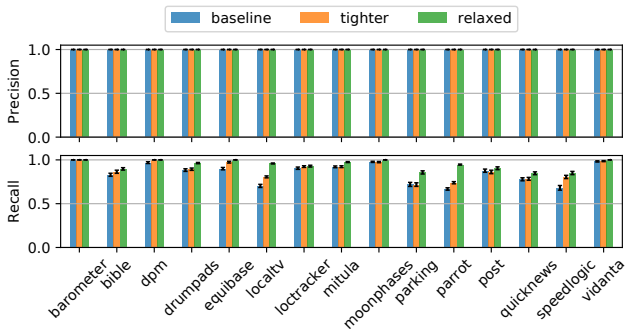


Figure 6: Precision and recall for GUI screen graphs.

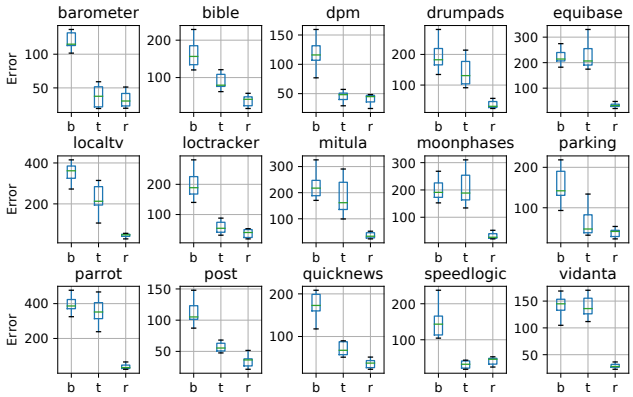


Figure 7: Distribution of errors $|f(n) - \hat{f}(n)|$ for GUI screen graphs. The "b", "t" and "r" on the x-axes are short for "baseline", "tighter" and "relaxed", respectively.

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 tighter bound and relaxed indistinguishability techniques.

To select an appropriate k for the tighter bound, we used $k = \lfloor t \times S_{gs} \rfloor$ where $t = \{0.95, 0.9, \ldots, 0.05\}$ and computed the largest difference between true and estimated frequencies, i.e., $\max_{n \in N} |f(n) - \hat{f}(n)|$. 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× 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 bound and relaxed indistinguishability level for neighbors, the recall is significantly improved. With the tighter bound, the recall is below 0.8 for 3 apps. 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 first collected the error $|f(n) - \hat{f}(n)|$ of each node for every experimental subject in

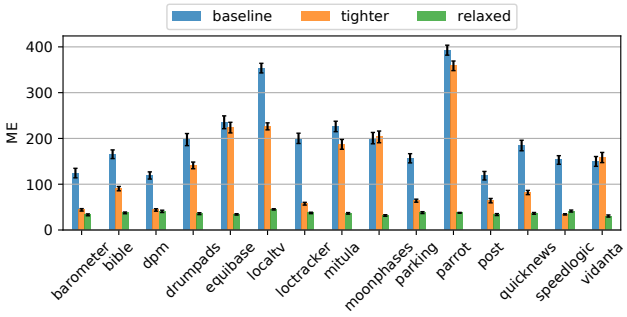


Figure 8: Mean error for GUI screen graphs.

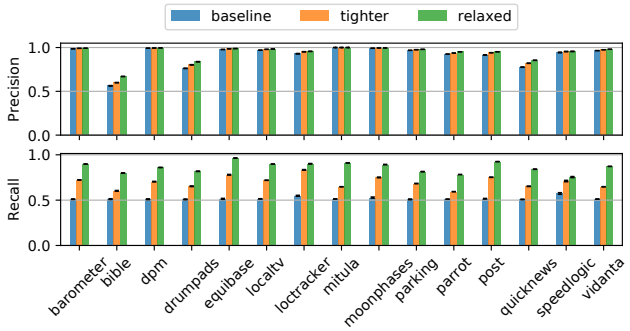


Figure 9: Precision and recall for call graphs.

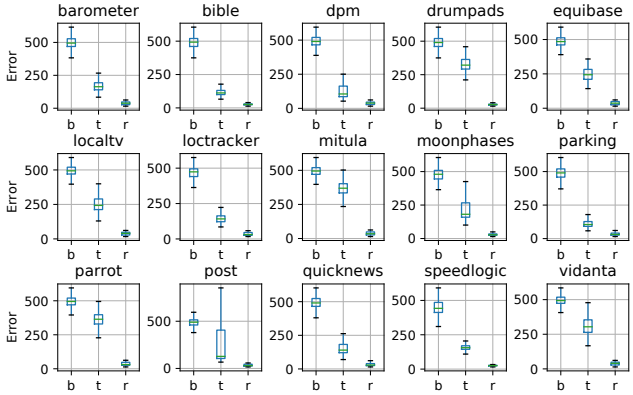


Figure 10: Distribution of errors $|f(n) - \hat{f}(n)|$ for call graphs.

each of the 100 independent trials and computed the mean of the errors. Figure 7 presents the box plots showing the distribution of average errors for all nodes in the screen graphs of the 15 apps. There is significant reduction in error for most apps. For the example of the `parking` app, the average error per node is 2.4× smaller when using tighter bound and 4.1× smaller when using relaxed indistinguishability, comparing to the baseline approach. We can also see that in a few apps, e.g., `equibase`, `moonphases` and `vidanta`, the improvement by using tighter bounds is negligible and sometimes the baseline approach performs even better. The reason is that the local sensitivities of run-time graph instances for these apps are very close to S_{gs}. When we apply tighter bounds, the loss of user data by the projection overwhelms the gain from the reduced noise by the tighter bound for randomization.

We then computed the ME for each app. Recall that ME indicates the amount of error on average each node will encounter when applying the three approaches for selecting sensitivity bound. Figure 8 shows the mean values of ME across 100 trials and the 95% confidence intervals. Compared with the baseline analysis, on average across the apps, ME is 2× and 5.4× smaller when using tighter bound and relaxed indistinguishability, respectively.

6.3.3 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 manner as for the GUI screen graphs. We ran 100 independent trials for each experiment. Figure 9 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 N. 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. Note that frequently-executed methods typically have more accurate estimates, and thus can be discovered successfully. For example, we considered the subset of methods with ground-truth frequency exceeding 25% of the frequency of the most frequent method. Averaged across all apps, the ME for this subset is 2.9× (global sensitivity), 2.2× (tighter sensitivity) and 11.6× (relaxed indistinguishability) smaller than the ME for all nodes.

Using the tighter sensitivity bound, there is only a little improvement on the recall, i.e., 1.3× increase averaged across all apps. This implies that, at least for these specific runs 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. The recall is much higher if we only consider frequently-executed methods. For example, for the subset of high-frequency methods described earlier, the approach achieves perfect recall for all apps when using relaxed indistinguishability.

Answering Q2. Figure 10 shows the distribution of errors of nodes in call graphs. In the baseline approach, the probability p from Definition 5 is very close to 0.5 due to the large $|N|$, causing the flipping of any bit in a vector to be almost random. Accordingly, the error for all apps is 500 on average, which is 50% of users. In the figure, we can see significant decrease

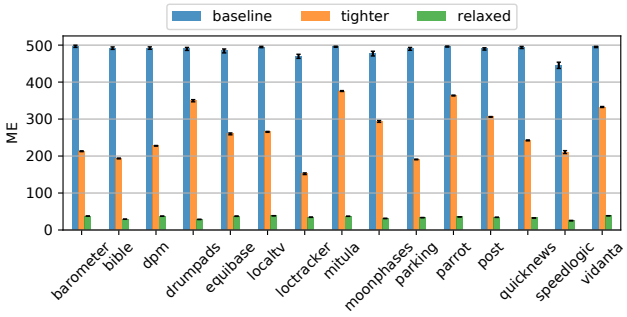


Figure 11: Mean error for call graphs.

of error when applying the other two approaches. As for screen graphs, we used $k = \lfloor t \times S_{gs} \rfloor$ for tighter bounds and found that $t \leq 0.1$ produced the best worst-case accuracy, i.e., $\max_n |f(n) - \hat{f}(n)|$ was minimized. This means that we could reduce the sensitivity bound by at least 90% to improve utility of results while keeping the same privacy guarantee.

Figure 11 shows ME values. The metric is 2× smaller on average when using tighter bound comparing to the baseline approach. The best accuracy is achieved, with 14.5× smaller ME, 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.

Processing cost. The cost of computing $LS(c_i)$ is rather small. For example, call graphs G_{c_i} for app `mitula` are the largest ones in our data, with 2011 nodes per graph on average. The average time to compute $LS(c_i)$ for one of these graphs is about 15 ms. Given the sensitivity bound S, the average time to compute $R(c_i)$ for these graphs is about 30 ms. Clearly, for the graphs considered in our experiments, the cost of data processing is negligible.

6.4 Parameter Exploration

Deciding the appropriateness of a given ε with respect to utility requires understanding the purpose and constraints of the analysis. For example, in Section 6.3, we observed that for some usage scenarios of node coverage analysis, conclusions about infrequently executed nodes cannot be reached with high accuracy. However, such loss of information might not be important if the analysis goal is to discover "hot" nodes. In such cases, a small value of ε can achieve practical utility.

Recall from Section 2.1 that the goal of the adversary is to correctly estimate the run-time existence of specific nodes at users with high probability. Thus, besides utility, another important and interesting question is *how the proposed approach performs in practice under various* ε *in terms of preventing the inference of node coverage information*. In the rest of this section, we demonstrate a causality-based attack utilizing the feasibility property. Although such an attack is under idealized conditions and is unlikely to occur in practice, is provides useful guidance for selecting ε and for comparing the intrinsic privacy trade-offs of the three proposed techniques.

We assume a strong adversary who knows G and the details of the randomizer R, including the values of parameters such as ε. The adversary also knows all users' run-time realizations of the control-flow graph, except for one graph c^* of the target user. Let C_{prior} denote the set of known graph realizations. For the target user, the adversary knows the true values of all bits in c^* except for the bit for one node n^*. The goal of the adversary is, by seeing the output $R(c^*)$, to determine whether n^* is executed an run time, i.e., whether $n^* \in c^*$.

Next, we describe the details of two attacks that can be conducted *before* and *after* the adversary receiving $R(c^*)$.

- **Attack based on feasibility and likelihood.** The attack includes two steps. First, before receiving $R(c^*)$, the adversary estimates the existence of n^* in c^* utilizing the feasibility property from Definition 1. In particular, she sets the bit of n^* in c^* to 0 to generate a new coverage vector c_0^*. Then she checks its feasibility by performing reachability analysis from the start node. If c_0^* is infeasible, i.e., removing n^* results in an illegitimate run-time graph, the adversary concludes that $n^* \in c^*$.
  If by the first step the adversary cannot infer the existence of n^* in c^*, she performs the second step as follows. The adversary learns a probability q of the occurrence of n^* in the coverage vectors in C_{prior}, i.e., the percentage of users who have executed n^* at run time. She then uses this as *a priori* knowledge that allows bias towards the answer with higher probability. If most users ($q > 50\%$) in C_{prior} executed n^* at run time, it is more likely that the target user also did so, i.e., $n^* \in c^*$. Otherwise, the adversary concludes that $n^* \notin c^*$.
- **Attack based on randomization result.** This attack is based on the previous attack but with the following additional statistical analysis after the adversary observing a realization of $R(c^*)$. If $n^* \in R(c^*)$ is observed, she computes and compares the conditional probabilities for $n^* \in R(c^*)$ given conditions $n^* \in c^*$ and $n^* \notin c^*$, i.e., $q \times \Pr[n^* \in R(c^*) \mid n^* \in c^*] = q(1-p)$ and $(1-q) \times \Pr[n^* \in R(c^*) \mid n^* \notin c^*] = (1-q)p$ where q is the probability learned in a similar way as in the attack above and p is from Definition 5. These two conditional probabilities correspond to the scenarios where 1) n^* is in c^* and R keeps the bit of n^* and 2) n^* is not in c^* but R flips the bit, respectively. The adversary draws the conclusion that $n^* \in c^*$ if the former is larger than the latter, and $n^* \notin c^*$ otherwise. The case when $n^* \notin R(c^*)$ is observed is handled similarly.

As discussed earlier, we assume that the adversary is interested in nodes that reveal user-specific usage patterns. In our illustrative case study, instead of picking a node that appears in (almost) all graphs, we choose n^* to be the "SettingsActivity" screen (n_8) in G of the `parking` app in Figure 3. This node is in 460 out of the 1000 executions. In the experiments, we repeat the two attacks independently for 100 trials. In each trial, we randomly select a testing set of 500 graphs for the

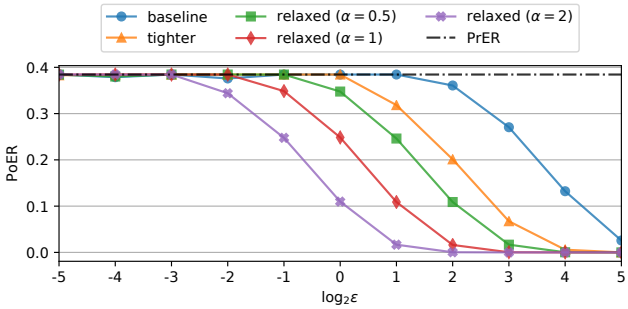


Figure 12: Comparison of privacy loss.

target user. Set C_{prior} consists of the remaining 500 graphs. We record the number of incorrect adversary guesses (out of 500), from which we compute error rates.

The error rates of the two attacks are denoted by *prior error rate* (PrER) and *posterior error rate* (PoER), respectively. PrER indicates the probability that the adversary falsely estimates the execution of n^* at the target user based on known bits in c^* and her knowledge of C_{prior}. PoER indicates the probability that the adversary makes an incorrect guess even after observing $R(c^*)$. The difference between PrER and PoER shows the *loss of privacy* due to revealing $R(c^*)$, and thus implies the effectiveness of R in hiding the occurrence of n^*. If the difference is 0, the adversary's observation of $R(n^*)$ does not reveal any information about n^*. Larger difference implies more privacy loss and thus less protection.

Figure 12 shows the average of PrER and PoER for 100 independent trials. We use $\varepsilon = \{2^{-5}, 2^{-4}, \ldots, 2^5\}$ for each technique in Section 5, denoted as "baseline", "tighter" and "relaxed ($\alpha = 0.5$)", respectively. We also alter the value of α to evaluate its effects on privacy and utility, which will be discussed shortly. We can see that using global sensitivity provides the best protection due to its largest amount of noise added. Applying relaxed indistinguishability performs the worst since it has the lowest bound. The choice of ε plays a critical role. As shown in the figure, all approaches (with $\alpha = 0.5$ for relaxed indistinguishability) are effective for $\varepsilon \leq 1$ with difference < 0.04, i.e., releasing $R(n^*)$ causes the success rate of the adversary's guesses to increase by less than 4%. When $\varepsilon \geq 2^3$, the noise added are not sufficient to deceive the adversary for all three approaches with difference > 0.1. Especially, the protection by using tighter bound and relaxed indistinguishability with $\alpha = 0.5$ is negligible, producing PoER below 10%.

Trade-offs by tuning α. Recall from Section 5.3 that the distance metric d^* is defined based on the size of sub-dominator trees and parameter α. The results discussed so far are for $\alpha = 0.5$. However, the choice of the value of α, and in turn the distance metric, does provide freedom to obtain trade-offs between utility and privacy. We explore the effects of these trade-offs by altering α. Specifically, we conduct additional experiments, compute the precision, recall and ME using $\alpha = \{1, 2\}$ under $\varepsilon = 1$. For screen view graphs, the recall is improved for all apps, being ≥ 0.9 when $\alpha = 1$ and ≥ 0.95 when $\alpha = 2$. We have also observed 10.4× and 23.2× reduction in ME for $\alpha = 1$ and $\alpha = 2$, respectively, relative to the baseline technique. For call graphs, the improvement on precision comparing to the results by setting $\alpha = 0.5$ is barely noticeable. The recall are improved by a few percentages on average across all apps: 0.02 for $\alpha = 1$ and 0.05 for $\alpha = 2$. However, there exists a significant reduction in ME for call graphs. It is 19.2× and 61.4× smaller comparing to the baseline technique, for $\alpha = 1$ and $\alpha = 2$, respectively.

As mentioned earlier, we have also used $\alpha = \{1, 2\}$ to calculate PrER and PoER for evaluating the effects of R on privacy protection. The results are shown in Figure 12, labelled with "relaxed ($\alpha = 1$)" and "relaxed ($\alpha = 2$)". We can see that for the extremely powerful adversary considered in the experiments, these two settings of α cannot achieve the same level of privacy protection provided by other techniques.

Note that the hypothetical adversary we considered is very knowledgeable. In practical circumstances, the adversary is very unlikely to know all but one bits in a coverage vector if privacy protection is applied in the first place. Thus the limits on ε and α can be relaxed in practice. Still, these results allow us to compare the three techniques against each other, and provide a general characterization of their privacy protection properties.

Summary. 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 4 and Section 5, we experimentally identify practical trade-offs that could be used for future developments of privacy-preserving software analyses.

7 Related Work

Differential privacy. Several examples of prior work on differential privacy were already discussed. There exists a large body of work on protection of correlated data [9,39,40,45,74] and graphs with DP [16, 36–38, 61, 62, 67]. In particular, Liu et al. [45] demonstrate an attack based on probabilistic dependence and propose dependent differential privacy that accounts for the dependence in a centralized settings. Our work is focused on the local model and provides a concrete algorithm based on dominator trees to discover deterministic correlation of graph nodes without any other probabilistic assumptions on graph nodes. Liu and Mittal [46] propose LinkMirage to mediate privacy-preserving access to social networks by obfuscation of links. Qin et al. [60] 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 achieves the more challenging node-DP for control-flow graphs considering the causal relationships

between nodes.

User behavior privacy. There is a significant body of work on understanding and defending collection of user data in browsers. Specifically, Lerner et al. [42] find that third-party tracking on the web has increased in prevalence and complexity. Olejnik et al. [57] report that browsing histories can be used for user identification. These works motivate our study on software usage analysis. Fan et al. [25] apply DP on time series of pageviews. Their techniques could potentially be improved by considering structure/hierarchy of websites, similar to control flows that are used in this paper. Reznichenko and Francis [63] collect advertisement data with DP and propose to use *privacy deficit* to measure privacy loss across multiple queries on the data. While our approach permits only one-shot data collection, it may be possible to extend it with similar concepts to allow continuous data collection.

Privacy leakage in mobile apps has also been studied extensively. Liu et al. [47] propose Alde for static and dynamic analysis of the data collection by analytics libraries. Chen et al. [10] discuss attacks that manipulate user profiles to control ads delivering. Meng et al. [48] point out that ads in free mobile apps could potentially leak sensitive user information. Seneviratne et al. [68] find that more than half of the paid apps contain at least one tracker. LinkDroid [27] tracks app-level linkability of usage behaviors of the same user across different apps. Han et al. [31] employ dynamic information flow tracking to monitor sending of sensitive information. These works highlight the privacy issues in mobile app analytics and motivate our work. They consider leakage of personal information such as devices IDs, ignoring control-flow data being collected. Our work focuses specifically on privacy-preserving collection of control-flow data.

General control-flow data collection. Privacy has also been considered in general software analysis. Elbaum and Hardojo [21] marshal and label data with the encrypted sender's name for anonymization at the deployed site. Clause and Orso [14] 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 [17, 52, 53]. Our approach, which is based on DP and its generalizations, is 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 distinguish with high probability the presence/absence of any graph node in the user's private data.

Many works have considered collection of various forms of control-flow data from deployed software, and could be interesting targets for developing DP analyses. Liblit et al. [43] minimize per-user overhead during information gathering by using sampling of program executions. Nagpurkar et al. [51] propose an instruction-based profiling approach for deployed software. DiCE [7] explores system behaviors to check whether the system deviates from its desired behavior. Saha et al. [66] collect execution information across software instances by running the program multiple times with the same input. The control-flow data of interest in these approaches includes execution traces, coverage and counts of statements and functions, and throw/catch counts. Our work presents a promising building block for the development of principled privacy-preserving versions of these approaches.

8 Conclusion

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. We believe that applying similar techniques based on differential privacy to other software analysis problems will be a fruitful direction for future work.

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] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

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

[3] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *ISSTA*, pages 5–15, 2007.

[4] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Thakurta. Practical locally private heavy hitters. In *NIPS*, pages 2285–2293, 2017.

[5] Raef Bassily and Adam Smith. Local, private, efficient protocols for succinct histograms. In *STOC*, pages 127–135, 2015.

[6] Jeremiah Blocki, Avrim Blum, Anupam Datta, and Or Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *ITCS*, pages 87–96, 2013.

[7] Marco Canini, Vojin Jovanović, Daniele Venzano, Boris Spasojević, Olivier Crameri, and Dejan Kostić. Toward online testing of federated and heterogeneous distributed systems. In *USENIX ATC*, pages 20–20, 2011.

[8] Konstantinos Chatzikokolakis, Miguel E Andrés, Nicolás Emilio Bordenabe, and Catuscia Palamidessi. Broadening the scope of differential privacy using metrics. In *PETS*, pages 82–102, 2013.

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

[10] Terence Chen, Imdad Ullah, Mohamed Ali Kaafar, and Roksana Boreli. Information leakage through mobile analytics services. In *HotMobile*, pages 15:1–15:6. ACM, 2014.

[11] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, pages 34–44, 2009.

[12] Andrew Chin and Anne Klinefelter. Differential privacy as a response to the reidentification threat: The Facebook advertiser case study. *NCL Rev.*, 90:1417, 2011.

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

[14] James Clause and Alessandro Orso. Camouflage: Automated anonymization of field data. In *ICSE*, pages 21–30, 2011.

[15] Aref N. Dajani, Amy D. Lauger, Phyllis E. Singer, Daniel Kifer, Jerome P. Reiter, Ashwin Machanavajjhala, Simson L. Garfinkel, Scot A. Dahl, Matthew Graham, Vishesh Karwa, Hang Kim, Philip Leclerc, Ian M. Schmutte, William N. Sexton, Lars Vilhuber, and John M. 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.

[16] Wei-Yen Day, Ninghui Li, and Min Lyu. Publishing graph degree distribution with node differential privacy. In *SIGMOD*, pages 123–138, 2016.

[17] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *PODS*, pages 202–210, 2003.

[18] Cynthia Dwork. Differential privacy. In *ICALP*, pages 1–12, July 2006.

[19] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

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

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

[22] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067, 2014.

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

[24] Facebook. Facebook analytics. https://analytics.facebook.com.

[25] Liyue Fan, Luca Bonomi, Li Xiong, and Vaidy Sunderam. Monitoring web browsing behavior with differential privacy. In *WWW*, pages 177–188, 2014.

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

[27] Huan Feng, Kassem Fawaz, and Kang G Shin. LinkDroid: Reducing unregulated aggregation of app usage behaviors. In *USENIX Security*, pages 769–783, 2015.

[28] Google. Firebase. https://firebase.google.com.

[29] Google. Google Analytics. https://analytics.google.com.

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

[31] Seungyeop Han, Jaeyeon Jung, and David Wetherall. A study of third-party tracking by mobile apps in the wild. *Univ. Washington, Tech. Rep. UW-CSE-12-03-01*, 2012.

[32] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/FSE*, pages 146–155, 2005.

[33] Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.

[34] Lingxiao Jiang and Zhendong Su. Profile-guided program simplification for effective testing and analysis. In *ESEC/FSE*, pages 48–58, 2008.

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

[36] Zach Jorgensen, Ting Yu, and Graham Cormode. Publishing attributed social graphs with formal privacy guarantees. In *SIGMOD*, pages 107–122, 2016.

[37] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. Private analysis of graph structure. In *VLDB*, pages 1146–1157, 2011.

[38] Shiva Prasad Kasiviswanathan, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Analyzing graphs with node differential privacy. In *TCC*, pages 457–476, 2013.

[39] Daniel Kifer and Ashwin Machanavajjhala. No free lunch in data privacy. In *SIGMOD*, pages 193–204. ACM, 2011.

[40] Daniel Kifer and Ashwin Machanavajjhala. A rigorous and customizable framework for privacy. In *PODS*, pages 77–88, 2012.

[41] Thomas Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. *TOPLAS*, 1(1):121–141, January 1979.

[42] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet Jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *USENIX Security*, 2016.

[43] Ben Liblit, Alex Aiken, Alice Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[44] Ben Liblit, Mayur Naik, Alice Zheng, Alex Aiken, and Michael Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.

[45] Changchang Liu, Supriyo Chakraborty, and Prateek Mittal. Dependence makes you vulnberable: Differential privacy under dependent tuples. In *NDSS*, 2016.

[46] Changchang Liu and Prateek Mittal. LinkMirage: Enabling privacy-preserving analytics on social relationships. In *NDSS*, 2016.

[47] Xing Liu, Sencun Zhu, Wei Wang, and Jiqiang Liu. Alde: Privacy risk analysis of analytics libraries in the Android ecosystem. In *SecureComm*, pages 655–672, 2016.

[48] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. The price of free: Privacy leakage in personalized mobile in-apps ads. In *NDSS*, 2016.

[49] Ilya Mironov. Rényi differential privacy. In *CSF*, pages 263–275, 2017.

[50] Takao Murakami and Yusuke Kawamoto. Utility-optimized local differential privacy mechanisms for distribution estimation. In *USENIX Security*, pages 1877–1894, 2019.

[51] Priya Nagpurkar, Hussam Mousa, Chandra Krintz, and Timothy Sherwood. Efficient remote profiling for resource-constrained devices. *TACO*, 3(1):35–66, March 2006.

[52] Arvind Narayanan and Vitaly Shmatikov. Robust deanonymization of large sparse datasets. In *S&P*, pages 111–125, 2008.

[53] Arvind Narayanan and Vitaly Shmatikov. Deanonymizing social networks. In *S&P*, pages 173–187, 2009.

[54] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *STOC*, pages 75–84, 2007.

[55] Kobbi Nissim, Thomas Steinke, Alexandra Wood, Micah Altman, Aaron Bembenek, Mark Bun, Marco Gaboardi, David O'Brien, and Salil Vadhan. Differential privacy: A primer for a non-technical audience (preliminary version). *Vanderbilt Journal of Entertainment and Technology Law*, 2018.

[56] Oath. Flurry. http://flurry.com.

[57] Lukasz Olejnik, Claude Castelluccia, and Artur Janc. Why Johnny can't browse in peace: On the uniqueness of web browsing history patterns. In *HotPETs*, 2012.

[58] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE*, pages 128–137, 2003.

[59] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *ICSE*, pages 277–284, 1999.

[60] Zhan Qin, Ting Yu, Yin Yang, Issa Khalil, Xiaokui Xiao, and Kui Ren. Generating synthetic decentralized social graphs with local differential privacy. In *CCS*, pages 425–438, 2017.

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

[62] Sofya Raskhodnikova and Adam Smith. Lipschitz extensions for node-private graph statistics and the generalized exponential mechanism. In *FOCS*, pages 495–504, 2016.

[63] Alexey Reznichenko and Paul Francis. Private-by-design advertising meets the real world. In *CCS*, pages 116–128, 2014.

[64] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *NSDI*, pages 12–12, 2012.

[65] Sable. Soot analysis framework. https://soot-oss.github.io/soot.

[66] Diptikalyan Saha, Pankaj Dhoolia, and Gaurab Paul. Distributed program tracing. In *ESEC/FSE*, pages 180–190, 2013.

[67] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. Sharing graphs using differentially private graph models. In *IMC*, pages 81–98, 2011.

[68] Suranga Seneviratne, Harini Kolamunna, and Aruna Seneviratne. A measurement study of tracking in paid mobile applications. In *WiSec*, 2015.

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

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

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

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

[73] Alice Zheng, Michael Jordan, Ben Liblit, and Alex Aiken. Statistical debugging of sampled programs. In *NIPS*, pages 603–610, 2004.

[74] Tianqing Zhu, Ping Xiong, Gang Li, and Wanlei Zhou. Correlated differential privacy: Hiding information in non-iid data set. *IEEE Transactions on Information Forensics and Security*, 10(2):229–242, 2014.

A Proofs

Lemma 1. Consider the set of all feasible subgraphs $G_{c'}$ for the given G_c such that $n \notin c'$. 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. □

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

Proposition 2. Let c be a feasible coverage vector. For any $n \in c$ and $t \in \{0,1\}^{|N|}$, consider the ratio between $\Pr[R(c) = t]$ and $\Pr[R(\Delta_n(c)) = t]$. This ratio is bounded from above by the product of x terms $e^{\frac{\varepsilon}{S}}$, where x is the number of bits in c that were changed to obtain $\Delta_n(c)$. 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 c, c', either $c' = \Delta_n(c)$ or $c = \Delta_n(c')$, therefore $\Pr[R(c) = t]/\Pr[R(c') = t] \leq e^{\varepsilon}$. Thus, R satisfies Definition 3. □