Differential Privacy for Coverage Analysis of Software Traces

Yu Hao
Ohio State University, Columbus, OH, USA

Sufian Latif
Ohio State University, Columbus, OH, USA

Hailong Zhang
Fordham University, New York, NY, USA

Raef Bassily
Ohio State University, Columbus, OH, USA

Atanas Rountev
Ohio State University, Columbus, OH, USA

Abstract

This work considers software execution traces, where a trace is a sequence of run-time events. Each user of a software system collects the set of traces covered by her execution of the software, and reports this set to an analysis server. Our goal is to report the local data of each user in a *privacy-preserving manner* by employing local differential privacy, a powerful theoretical framework for designing privacy-preserving data analysis. A significant advantage of such analysis is that it offers principled "built-in" privacy with clearly-defined and quantifiable privacy protections. In local differential privacy, the data of an individual user is modified using a *local randomizer* before being sent to the untrusted analysis server. Based on the randomized information from all users, the analysis server computes, for each trace, an estimate of how many users have covered it.

Such analysis requires that the domain of possible traces be defined ahead of time. Unlike in prior related work, here the domain is either infinite or, at best, restricted to many billions of elements. Further, the traces in this domain typically have structure defined by the static properties of the software. To capture these novel aspects, we define the trace domain with the help of context-free grammars. We illustrate this approach with two exemplars: a *call chain analysis* in which traces are described through a regular language, and an *enter/exit trace analysis* in which traces are described by a balanced-parentheses context-free language. Randomization over such domains is challenging due to their large size, which makes it impossible to use prior randomization techniques. To solve this problem, we propose to use *count sketch*, a fixed-size hashing data structure for summarizing frequent items. We develop a version of count sketch for trace analysis and demonstrate its suitability for software execution data. In addition, instead of randomizing separately each contribution to the sketch, we develop a much-faster one-shot randomization of the accumulated sketch data.

One important client of the collected information is the identification of high-frequency ("hot") traces. We develop a novel approach to identify hot traces from the collected randomized sketches. A key insight is that the very large domain of possible traces can be efficiently explored for hot traces by using the frequency estimates of a visited trace and its prefixes and suffixes. Our experimental study of both call chain analysis and enter/exit trace analysis indicates that the frequency estimates, as well as the identification of hot traces, achieve high accuracy and high privacy.

2012 ACM Subject Classification Software and its engineering → Dynamic analysis; Security and privacy → Privacy-preserving protocols

Keywords and phrases Trace Profiling, Differential Privacy, Program Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.8

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.7`

Funding This material is based upon work supported by the National Science Foundation under Grant No. CCF-1907715.

Acknowledgements We thank the ECOOP reviewers for their valuable feedback.

1 Introduction

In this work we consider privacy-preserving coverage analysis for software traces. A trace is an event sequence $t \in \mathcal{E}^+$ over some pre-defined set of possible run-time events $\mathcal{E}$. Consider a software system deployed over a set of software users $u_1, \ldots, u_n$. Each user u_i executes her own copy of the software and that execution records a local set T_i of traces that were observed at run time. This data is collected locally and then sent to an analysis server. We consider the following problem: for each $t \in \mathcal{E}^+$, estimate the frequency of t over the population of users, that is, $f(t) = |\{i : t \in T_i\}|$, *while collecting the local data of each user in a privacy-preserving manner and sending it to an untrusted analysis server.*

Trace information has a wide range of uses in software analysis and transformation. For example, high-frequency traces can focus software optimization, testing, and static checking on important user behaviors. Similarly, behavior flow analysis in mobile/web analytics [21, 19, 22, 37] allows developers to optimize the functionality and performance of common paths taken by app users through the app code.

1.1 The Need for Privacy-Preserving Analysis

Our goal is to design an analysis that obtains accurate trace coverage statistics across a population of software users, while controlling carefully the "privacy loss" of each user. This is motivated by the increased importance of reducing the amount of user information collected by business entities. Both societal and legislative pressures have highlighted the need for such reduction. For software-generated event information – for example, collected with the help of popular analysis infrastructures for mobile/web analytics (e.g., provided by Google and Facebook) – typically there are no "built-in" privacy protection mechanisms. The infrastructures themselves collect a wealth of information, including user IP addresses and GUI events. App-specific data collection can provide even more fine-grained knowledge about user's behavior and interaction with the software. For example, trace coverage information can provide details about what paths through the code a user has taken, and what functionality (possibly sensitive) she has executed. This data could potentially be used to infer user-specific habits, interests, and characteristics.

From the point of view of software users, the release of data collected from software executions is often undeclared or obscured. Even if the user is aware of the data collection, they are unlikely to have true appreciation of its implications. What is particularly troubling is that the collected data could be linked with other sources of information about this user (which cannot be prevented even with anonymization [34, 35]) and could be used as part of future larger-scale data mining and machine learning attempts to infer user-specific information. At data collection time, it is impossible to predict what extra data sources will be linked and what future inferences will be possible using that data.

Privacy-preserving data analysis aims to develop systematic mechanisms for addressing this problem. Such analysis benefits two categories of stakeholders. First, the privacy of individual users is protected in a well-defined manner. Further, entities performing data collection (e.g., Google and app developers using Google's analytics infrastructure) benefit as well: they are responsive to privacy expectations and do not have access to raw data that can compromised by unexpected data leaks or unethical business practices. In this work we focus on one particular privacy-preserving mechanism: *local differential privacy*. Our goal is to use local differential privacy to design a privacy-preserving trace coverage analysis.

1.2 Local Differential Privacy

Differential privacy [16] is a powerful theoretical framework for privacy-preserving data analyses. This approach has been recognized by the theoretical computer science community [1] and has been employed by high-profile companies such as Google, Apple, and Microsoft [18, 4, 33, 51] and by the U.S. government [14]. More widespread use of this technology is becoming possible via recent open-source tools [40].

A significant advantage of differentially-private data analyses is that they offer principled "built-in" privacy with clearly-defined and quantifiable privacy protections. Broadly, privacy is achieved by randomizing the collected data in a way that ensures the impossibility of high-confidence inferences from the randomized data. There is increasing interest in using such techniques in the context of software. For example, the programming languages community has considered techniques for verification of differentially-private algorithms [56, 53, 36, 61]. Others have considered uses of differential privacy for event frequency analysis of deployed software [58, 60, 57], with natural applications to mobile app analytics [59].

In local differential privacy, the data of an individual user is modified using a *local randomizer* before being sent to an untrusted analysis server. Thus, the raw local information never leaves the user's environment. In our setting the analysis server computes, for each trace, an estimate of how many users have covered it.

A major advantage of this approach is that it is resilient to other known and unknown categories of knowledge that may become available about this user in the future. Thus, a differentially-private analysis is designed to be robust even under the current trend of fast-growing data collection and linking from various sources, used by businesses and governments to create user behavior profiles and to mine them for user-specific patterns [55, 16]. Such user protection is also important under the threat of unexpected data releases caused by obscure changes to privacy agreements, information requests by law enforcement, or security breaches in which user data is compromised. The quantifiable privacy-preserving machinery of local differential privacy is appealing not only to software users, but also to software developers and analysis infrastructure providers. Both the developers and the infrastructure providers can claim, with confidence, that they deploy privacy-by-design in their data collection. As a result, the data they collect and store is protected, in the statistical differentially-private sense, against data breaches, rogue employees/subcontractors, and scrutiny from government agencies and law enforcement. Section 2 further discusses the assumptions behind this analysis model and the nature of privacy protection achieved under it.

1.3 Challenges and Contributions

Various differentially-private approaches have been considered for similar styles of problems. Some have considered data analysis with single data item per user [18, 52, 8]. Others have studied data collection for a pre-defined small set of items from software executions [58, 57, 60, 59]. However, the analysis of software traces presents unique challenges. Our contributions in solving these challenges are summarized below.

Contribution 1: Analysis for structured large domains

Differentially-private analyses require that the domain of possible data items be defined ahead of time, as part of algorithm design. Unlike prior related work where software executions generate data over a small unstructured domain containing a few thousand elements [58, 57, 60], here the domain is either infinite or, at best, restricted to many billions of elements. Further, the traces in this domain typically have structure defined by the static properties of the software. To capture these novel aspects, we propose to define the trace

domain with the help of context-free grammars. This approach has the key advantage that both the domain definition and the exploration of its elements are formulated using popular programming language machinery. We illustrate this approach with two exemplar analyses: a *call chain analysis* in which traces are described through a regular language, and an *enter/exit trace analysis* in which traces are described by a balanced-parentheses context-free language. Both kinds of structures are widely used in program analysis and are applicable to a range of techniques beyond these two exemplars. This formulation plays a key role in our approach for identification of high-frequency domain elements, as described later.

Contribution 2: Count sketch

The domain of possible traces is very large. For example, in a realistic setting for our benchmarks, this set has billions of elements. One of the key features of a differentially-private approach is that it produces per-user randomized data *that could contain an arbitrarily large subset* of this extremely large domain. This is essential for achieving the differential privacy guarantee, but is clearly impractical for our purposes, in terms of both space and generation time. We address this exponential explosion by using *count sketch* [10], a fixed-size hashing data structure originally designed to collect data about frequent items in data streams. While prior work [8] has considered the theoretical applications of count sketch for a simple single-item differentially-private data analysis, we develop a version of count sketch that is applicable to the more complex analysis we consider, and demonstrate its suitability for real software execution data.

Contribution 3: Efficient randomization

The standard approach for designing the local randomizer is to randomize individual contributions (i.e., observations of covered traces) as they are observed. However, our experience shows that the cost of such randomization is high and not suitable for real-world software analysis. Instead, we develop a technique to accumulate the effects of unrandomized count sketch updates, and only perform the local randomization as a one-shot step on this accumulated data. This reduces the cost of the randomization by orders of magnitude and makes it practical to use for realistic data gathering from deployed software.

Contribution 4: Analysis of hot traces

After the randomized local data is collected by the analysis server, a resulting *global count sketch* captures the population-wide information about observed traces. From this global sketch, a frequency estimate can be obtained for any given trace t from the domain of possible traces. However, this alone is not enough for many forms of data analyses, since the number of possible traces is exponential or even infinite, and obtaining an estimate for each t (and then analyzing all these estimates) is not possible. We focus on one particular data analysis of significant importance: identifying the *hot traces* and estimating their frequencies. A hot trace has a frequency that exceeds some threshold. Knowledge of such traces is useful to identify common user behaviors, leading to focused performance optimization, testing/checking, and application-flow optimization. We develop a novel approach to identify likely-hot traces from the randomized data in the global sketch. The key insight of our approach is that the explosively-large domain of possible traces can be efficiently explored for hot traces by using the frequency estimates of a visited trace and its prefixes and suffixes. We illustrate this approach with the help of the two exemplar analyses mentioned earlier, and demonstrate how the exploration of the domain can be performed by exploring the states of the automaton corresponding to the underlying context-free language.

Contribution 5: Study of privacy/accuracy trade-offs

The trade-offs between accuracy and privacy are central in the design of privacy-preserving algorithms. Our experimental study characterizes these trade-offs in several dimensions for both call chain analysis and enter/exit trace analysis. The conclusions from this study can be summarized as follows: (1) frequency estimates for software traces can be obtained with both high accuracy and high privacy, as long as the data collection includes a sufficiently-large number of software users, (2) the set of hot traces can be discovered with high recall and precision by reasoning about a trace's prefixes and suffixes, (3) the frequency estimates for hot traces are accurate and better than the estimates for the remaining traces, (4) after a certain point in the accuracy/privacy trade-off spectrum, reduced privacy does not result in significantly better accuracy; this point provides a natural choice for selecting such trade-offs.

2 Background and Problem Statement

2.1 Software Traces

We consider software traces, collected over a set of software users u_i for $i \in [1, n]$. Each user u_i executes her own copy of the software. During execution, run-time events are observed and recorded. Let $\mathcal{E}$ be the finite set of possible run-time events. This set is defined before software deployment, as part of the design of the trace analysis. For convenience of definitions, we assume that $\mathcal{E}$ contains an artificial "start" event s denoting the start of a trace. A trace t is a string $t \in \mathcal{E}^+$, starting with s. We will use the notation $t = \langle s, e_1, \ldots, e_k \rangle$ to denote a trace t of length k. (Note that we exclude s when defining trace length.)

Let $\mathcal{T}$ be a domain describing conservatively (i.e., over-approximating) the set of all possible traces that could be observed at run time. We expect this domain to be statically described as part of the design of the trace analysis. In the simplest case, $\mathcal{T} = \mathcal{E}^+$. More precise definitions of $\mathcal{T}$ may be possible via static analysis. Regardless of the means to derive $\mathcal{T}$, we expect it to be very large (e.g., exponential in the static size of the program). In addition, traces typically have structure that is constrained by the static properties of the software. In particular, one important special case we investigate is when $\mathcal{T}$ is defined inductively through a family of "extension" functions ext_k: $\mathcal{E}^k \times \mathcal{E} \to \mathcal{P}(\mathcal{E}^{k+1})$. Here $\mathcal{P}(X)$ denotes the power set of X and k ranges over the natural numbers. For any $t \in \mathcal{T}$ of length k, $\mathsf{ext}_k(t)$ is the set of all traces $t' \in \mathcal{T}$ of length $k+1$ such that t is a prefix of t'. That is, $\mathsf{ext}_k(t)$ shows all ways in which t could be extended with one more event to form a valid trace. For simplicity, we will omit the subscript k in ext_k when it is clear from the context. As discussed later, this definition of $\mathcal{T}$ enables incremental search for "hot" traces. While our definitions of privacy-preserving analysis are conceptually applicable to broader categories of $\mathcal{T}$, the application of this approach for identification of traces that appear frequently in the user population requires such incremental definition of $\mathcal{T}$ (Section 4).

Below we discuss two examples of such trace domains $\mathcal{T}$, both with direct connections to popular categories of analyses. These exemplars illustrate how common properties of such analyses can be mapped to the problem definition and solution described in this work. In particular, we define these two domains via well-understood formal languages – a regular language and a balanced-parentheses context-free language – which provides a natural definition for the domain and its extension function. This choice is also motivated by the fact that such languages are widely used in various existing software analysis techniques. Our approach is directly applicable to other trace analyses where the trace domain has a similar structure. This machinery is likely to be generalizable to broader domains (e.g., ones that correspond to more general context-free languages) but we do not explore these generalizations in this work.

Both domains are based on a set of events corresponding to entering or exiting a software component (e.g., method, module, or GUI window). We simplify the definition by assuming that each component is uniquely identified by an integer id from $[1, c]$. In addition, we introduce an artificial component with id 0 which corresponds to the external environment – e.g., the caller of the main method, or the framework code that invokes Android app entry points. The set of events is then $\mathcal{E} = \mathit{Enter} \cup \mathit{Exit}$ where $\mathit{Enter} = \{\mathsf{enter}(i) : i \in [0, c]\}$ and $\mathit{Exit} = \{\mathsf{exit}(i) : i \in [0, c]\}$. The artificial start event s is $\mathsf{enter}(0)$.

2.1.1 Exemplar 1: Call Chains

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

2.1.2 Exemplar 2: Enter/Exit Traces

Next we define an exemplar analysis in which $\mathcal{T}$ is based on a balanced-parentheses context-free language. This language captures the standard notion of *interprocedurally valid paths* [46] and is defined by the following grammar:

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

where λ is the empty string. Non-terminal *Balanced* defines a sequence of matching enter and exit events. Starting non-terminal *Valid* describes a sequence with some not-yet-matched enter events. Grammars of similar structure have been used extensively in a wide variety of static analyses (e.g., [46, 48]). For our exemplar analysis we consider the domain of enter/exit traces $\mathcal{T}$ to be strings derived from *Valid* and starting with $\mathsf{enter}(0)$. We further restrict the strings to respect a given set of static call edges $i \to j$. This can be easily encoded in the definition of the corresponding pushdown automaton, as follows. We can define a deterministic pushdown automaton with a single state. The input alphabet is $\mathit{Enter} \cup \mathit{Exit}$ and the stack alphabet is *Enter*, with initial stack symbol $\mathsf{enter}(0)$. The transitions upon observing input event $\mathsf{enter}(j)$ when the top of the stack is $\mathsf{enter}(i)$ is defined only if there is a static call edge $i \to j$. This transition pushes $\mathsf{enter}(j)$ onto the stack. If the input symbol is $\mathsf{exit}(i)$, the transition is defined only if the top of the stack is $\mathsf{enter}(i)$, in which case the stack top is popped. The trace extension function ext, which captures all ways in which a given trace is extended with one more event, is easily derivable from this pushdown automaton.

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

possible to generalize such machinery to more general pushdown automata, but the definition of the extension functions (derived from the automata) would be more complex than the simple extension functions described above.

2.2 Trace Coverage Analysis for Deployed Software

When the program is executed by a software user, some subset of $\mathcal{T}$ is actually observed (i.e., covered) at run time. A variety of run-time techniques can be used to determine this coverage (e.g., [7, 3, 62, 49]). We consider such coverage across a large number of software users, each running her copy of the program. Let there be n software users denoted by $u_1, \ldots, u_n$ and let $T_i \subseteq \mathcal{T}$ be the set of traces covered when user u_i executes the program. We consider the following *trace coverage analysis*: for each $t \in \mathcal{T}$, estimate the frequency of t over the population of users, that is, $f(t) = |\{i : t \in T_i\}|$, while collecting the local data of each user with differential privacy.

Trace information has been used extensively to analyze and optimize software performance [7, 3, 5, 62, 26, 2]. The frequency information defined above can be used to focus such efforts on important user behaviors. Similarly, testing and static checking can be focused on traces that are commonly observed in the user population. Another example is behavior flow analysis in mobile and web analytics frameworks [21, 37], which allows developers to see different paths that users take through the app. The paths can be thought of as a form of traces across GUI components, and the analysis annotates each edge with the number of users who have performed the corresponding transition. A similar example is funnel analysis [21, 19, 22, 37], which visualizes the completion rate of a task in terms of a series of specific events and helps developers find optimizations in their software design. Traces collected for funnel analysis may contain sensitive information. For example, events in trace "launch the app, open the news page, navigate to sports news, perform sports merchandise purchase" can be used for targeted advertising. Our approach allows developers to conduct frequency analysis while ensuring worst-case privacy guarantees even when users are unaware of the data being collected and the unexpected/unpredictable future uses of this data.

2.3 Differential Privacy

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

Differential privacy guarantee

Suppose $R(T_i)$ is released publicly by a software user. We aim to design a randomizer R that ensures the following differential privacy property: *for every possible $t \in \mathcal{T}$, an external observer of $R(T_i)$ cannot have high confidence that the hidden raw data contains that t*. In other words, whether t is in T_i cannot be ascertained with high probability based only on the observation of $R(T_i)$. In essence, the presence of t in the private local data is hidden in a probabilistic sense.

More precisely, let $P[R(X) = Z]$ be the probability that given input X, the randomizer produces output Z. For any Z and any two $X \subseteq \mathcal{T}$ and $Y \subseteq \mathcal{T}$ that differ by a single element t, the ratio of $P[R(X) = Z]$ and $P[R(Y) = Z]$ should be bounded by e^{ϵ}. Here X and Y are considered to be "neighbors" in the space of inputs to the randomization algorithm. Because the two probabilities are close to each other, when someone observes any output Z, she cannot have much higher confidence in the statement "the raw data contained t", compared to the confidence she can have in the statement "the raw data did not contain t". Here ϵ is the *privacy loss parameter*, which is used to tune accuracy/privacy trade-offs. A typical value used in related work is $\ln(9)$ [18, 52, 58]; for example, this value is used in the "basic one-time" version of a popular randomization technique [18]. Larger values of ϵ improve the accuracy of analysis results, but weaken the privacy guarantee.

A key assumption is that the adversarial observer of $R(T_i)$ knows fully all details of how randomizer R works, for example, because this observer designed the randomizer in the first place, or because she reverse-engineered it from the program code. As part of this assumption, the observer also knows the value ϵ which was embedded in the randomizer design. Even under such strong assumptions, the differential privacy guarantee makes it impossible to distinguish, in a probabilistic sense, neighbor inputs to the randomizer after the randomizer output is publicly released. Such principled and quantifiable protection is one of the reasons differential privacy has been employed by companies such as Google [18], Microsoft [33], Apple [4], and Uber [51], as well as by the U.S. Census Bureau [14]. More widespread use of such protection has become possible via recent open-source tools for differentially-private analysis [40].

Randomized response

To illustrate this key *indistinguishability property*, we present a classic simplified example. For illustration, suppose that the raw data for user u_i is a single trace $t_i \in \mathcal{T}$. A well-know randomization technique is derived from *randomized response*, an approach used in social sciences to handle evasive answers to sensitive questions [54]. The randomizer $R : \mathcal{T} \to \mathcal{P}(\mathcal{T})$ takes as input a single trace t and produces a set of traces, based on the following rules: (1) the input t is included in the output with some probability p, and (2) for every other $t' \in \mathcal{T}$, t' is included in the output with probability $1 - p$. Thus, the real trace could be missing from the output, and any other trace could be part of the output. Note that this approach is applicable only when $\mathcal{T}$ is finite and, practically, the size of $\mathcal{T}$ is relatively small.

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

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

2.4 Assumptions

Several assumptions need to be explicitly stated before we describe our differentially-private analysis (Section 3). As usual in this type of work, it is assumed that the design and implementation of the approach are fixed before any data collection and are publicly known by all stakeholders, including untrustred parties. Another assumption is that the software code correctly implements the design; in particular, it does implement the randomization as publicly announced, and does not try to circumvent it by sending the raw data (or some version of it) to a malicious party. Although this is a strong assumption, it is no different than what is currently used in remote analysis of deployed software, where the design is typically undocumented and/or obfuscated, and there is no checking of the implementation of the data collection for correctness or presence of malicious code.

If a software developer commits to using the correct design and implementing it as expected, this raises the confidence of software users and watchdog agencies that indeed privacy is protected. Further, several techniques can be used to increase this confidence, including (1) open-source implementations, (2) use of certified and trusted third-party libraries, (3) scrutiny by privacy experts, and (4) code analysis via automated tools. Note that there are no assumptions about the analysis server to which the randomized data is sent. This server could be part of a privacy attack, possibly involving additional external sources of information about the targeted software user. Even with this assumption, the differential privacy guarantee holds [55].

3 Randomized Count Sketch for Software Traces

Even if a user's local information contains a single trace, the approach outlined in the previous section is not possible when $\mathcal{T}$ is infinite, since every elements of $\mathcal{T}$ must be visited when randomization is applied. Even if $\mathcal{T}$ is made finite – for example, by using a pre-defined limit on trace length – the approach is still not practical. For illustration, consider call chains for the `localtv` Android app used in our experiments. The alphabet size $|Enter| = 2974$ in this app is close to the median for our set of benchmarks. Even if we only consider chains of at most three methods and count the strings recognized by the corresponding finite-state automaton (as described in Section 2.1.1), we have $|\mathcal{T}| = 3,272,137$. Increasing this length by one, the size of $\mathcal{T}$ becomes more than 163 million. A further length increase by one results in $|\mathcal{T}|$ of over 8 billion. Our implementation of the finite-state automaton is based on a call graph constructed through class hierarchy analysis. Using a more precise call graph analysis (e.g., based on context-sensitive analysis) may reduce $|\mathcal{T}|$. However, it is likely that $\mathcal{T}$ will still be very large, since conditional behaviors (e.g., calls guarded by conditionals) are common and easily produce exponential growth in the number of statically-possible traces. The cost of the randomizer described earlier is proportional to the size of $\mathcal{T}$, as each element $t \in \mathcal{T}$ must be visited and a random value must be generated for that t (independently of the processing of the remaining elements of $\mathcal{T}$) in order to decide whether t is included in the randomizer output. Further, the randomizer output, which needs to be sent to the analysis server, has size dependent on the exponentially-large size of $\mathcal{T}$. Clearly, these costs are infeasible.

3.1 Count Sketch

To address this problem we employ *count sketch* [10], a data structure originally designed to find frequent items in data streams. Prior work [8] has considered the theoretical analysis of using count sketch for a restricted form of differentially-private data analysis, where

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

Local Sketch

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

Figure 1 Count sketch illustration, with $m = 8$ and $s = 3$.

each user has a single data item. However, there is no clarity on the practical use of this data structure for analysis of real-world software execution data and for the more general problem we consider, where each user has a set of local traces. Using insights from this prior work, we develop a version of count sketch for our trace analysis and demonstrate its effectiveness on data from actual software executions. We first describe count sketch without any privacy-related randomization. The next subsection shows how randomization can be applied to achieve the differential privacy guarantee.

Counts sketch in our setting is based on s pairs of independent hash functions (h_k, g_k), for $1 \leq k \leq s$, such that $h_k : \mathcal{T} \to \{1, \ldots, m\}$ and $g_k : \mathcal{T} \to \{+1, -1\}$. Here parameters s and m are chosen ahead of time; this choice will be discussed later. To perform analysis without differential privacy, each user would create a *local sketch* and then send it to the analysis server, where a *global sketch* is constructed and used to produce frequency estimates. The local sketch for user u_i is a $s \times m$ matrix S_i initialized with 0 elements. For every locally-covered trace $t \in T_i$, and for every $1 \leq k \leq s$, matrix element $S_i[k, h_k(t)]$ is updated by adding to it the value of $g_k(t)$. In essence, for every row k in the matrix, we use hash function h_k to hash t into a value from $\{1, \ldots, m\}$, and then update a counter for that value with $+1$ or -1 depending on hash function g_k. The local sketches S_i for all users are then sent to the analysis server, where a global sketch S_g is constructed by element-wise addition of all S_i. Finally, for any $t \in \mathcal{T}$, a frequency estimate can be obtained by reporting the median value of $S_g[k, h_k(t)] \times g_k(t)$ over all $1 \leq k \leq s$.

Example

Figure 1 illustrates the local sketch for one user, based on data obtained from our implementation on one of our benchmarks (Android app `drumpads`). We use integer method ids to denote app methods. For example, id 473 corresponds to method `MainActivity.initOnboarding` and id 971 corresponds to `OnboardingView.createImageScene`. For brevity, the example uses the method id to signify an `enter` event for the corresponding method; id 0 corresponds to the start event.

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

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

The advantage of using this approach is that a local sketch S_i for user u_i provides a fixed-sized representation of the arbitrary subset T_i of the set $\mathcal{T}$ of possible traces. Further, randomization of the local sketch, as described shortly, can be performed in time proportional to this $s \times m$ sketch size. Thus, instead of recording the raw data T_i and randomizing it with randomized response to achieve the differential privacy guarantee over $\mathcal{T}$, we will record the sketch S_i and randomize it to achieve the differential privacy guarantee over local sketches. Finally, the count sketch technique is theoretically proven to produce accurate estimates for high-frequency items, which aligns well with our goal to produce information about frequently-occurring traces, as discussed further in Section 4.

3.2 Sketch Randomization

Trace-level randomization

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

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

The resulting randomized vector contains only $+1$ and -1 values. We can think of this process as applying a randomizer $R_k : \mathcal{T} \to \{+1, -1\}^m$. It can be proven that this approach achieves indistinguishability between t and any $t' \in \mathcal{T}$. The outline of this proof is as follows. First, consider the case when t and t' are hashed to the same position in count sketch row – that is, $h_k(t) = h_k(t')$. For any $Z \in \{+1, -1\}^m$, it is easy to see that the ratio of $P[R_k(t) = Z]$ and $P[R_k(t') = Z]$ can be bounded by the ratio of $e^\epsilon/(e^\epsilon + 1)$ (i.e., the probability that the sign at the non-zero position is preserved) and $1/(e^\epsilon + 1)$ (i.e., the probability that the sign at the non-zero position is inverted). The second case is when t and t' are hashed to different positions. Then the ratio of $P[R_k(t) = Z]$ and $P[R_k(t') = Z]$ is bounded by the ratio of $\frac{1}{2}e^\epsilon/(e^\epsilon + 1)$ and $\frac{1}{2}/(e^\epsilon + 1)$; here $\frac{1}{2}$ is the probability associated with the randomization of the zero positions. In either case, for any vector Z containing m values $+1/-1$, the probabilities $P[R_k(t) = Z]$ and $P[R_k(t') = Z]$ differ by at most a factor of e^ϵ. By observing Z, a malicious observer cannot conclude with high confidence that the underlying trace was t as opposed to any other $t' \in \mathcal{T}$.

Set-level randomization

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

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

3.3 Efficient Randomization

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

Algorithm 1 describes the details of this approach. Consider a cell $[k, j]$ in the sketch. Let $N_{+1}[k, j]$ be the number of traces that contribute $+1$ to the value in this cell, without randomization. Similarly, let $N_{-1}[k, j]$ be the number of traces that contribute -1 to the cell. Our approach first records these counts (function `add`) without randomization. After data collection is completed, `finalize` computes the randomized sketch. With randomization, each of the $N_{+1}[k, j]$ occurrences of $+1$ contributes $+1$ with probability p and -1 with probability $1 - p$. Binomial distribution gives us the probability of y successes in x independent trials, where each trial succeeds with probability p. Let $binomial(x, p)$ denote a random value drawn from this distribution. The randomization will contribute $binomial(N_{+1}[k, j], p)$ values of $+1$ to the cell value; the remaining $N_{+1}[k, j] - binomial(N_{+1}[k, j], p)$ contributions will be -1. Thus, at line 19 of the algorithm we compute the cumulative contribution of the "raw" $+1$ values as the difference between these two quantities – that is, as $2 \times binomial(N_{+1}[k, j], p) - N_{+1}[k, j]$. A similar computation is performed at line 20 for the -1 values. Finally, we also have to account for the randomization of 0 values, which is done at line 21. The combined effect of these three cases is computed at line 22 as the cell value in the randomized sketch. This approach has cost in the order of $s \times m$, while a naive implementation with separate randomization for each observed trace will have cost in the order of $|T_i| \times s \times m$.

3.4 Server-Side Processing

The randomized local sketches $R(S_i)$ from all users are collected by the analysis server and their element-wise sum is computed. To obtain unbiased estimates, all elements of the sum need to be scaled by $(e^\epsilon + 1)/(e^\epsilon - 1)$. The resulting $s \times m$ matrix S_g is the *global sketch* produced by the analysis. For any $t \in \mathcal{T}$, an estimate $\hat{f}(t)$ of the true frequency $f(t)$ can be obtained as the median value of $S_g[k, h_k(t)] \times g_k(t)$ over all sketch rows k. This processing is described in Algorithm 2. It is important to note that summing up of the local sketches is essential in order for the randomized noises to "cancel out" across the population of users.

3.5 Selecting Sketch Size

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

Algorithm 1 Randomized count sketch.

output : S_i: randomized local sketch

Function `init()`**:**
- $S_i \leftarrow \{0\}^{s \times m}$
- $N_{+1} \leftarrow \{0\}^{s \times m}$
- $N_{-1} \leftarrow \{0\}^{s \times m}$

Function `add(`t`)`**:**
- $T_i \leftarrow T_i \cup \{t\}$
- **for** $k \leftarrow 1$ **to** s **do**
  - **if** $g_k(t) = +1$ **then**
    - $N_{+1}[k, h_k(t)] \leftarrow N_{+1}[k, h_k(t)] + 1$
  - **else**
    - $N_{-1}[k, h_k(t)] \leftarrow N_{-1}[k, h_k(t)] + 1$
  - **end**
- **end**

Function `finalize()`**:**
- $p \leftarrow \frac{e^\epsilon}{1+e^\epsilon}$
- **for** $k \leftarrow 1$ **to** s **do**
  - **for** $j \leftarrow 1$ **to** m **do**
    - $z \leftarrow |T_i| - N_{+1}[k, j] - N_{-1}[k, j]$
    - $n_{+1} \leftarrow 2 \times \mathit{binomial}(N_{+1}[k, j], p) - N_{+1}[k, j]$
    - $n_{-1} \leftarrow 2 \times \mathit{binomial}(N_{-1}[k, j], p) - N_{-1}[k, j]$
    - $n_0 \leftarrow 2 \times \mathit{binomial}(z, \frac{1}{2}) - z$
    - $S_i[k, j] \leftarrow n_{+1} - n_{-1} + n_0$
  - **end**
- **end**

of the union of all local sets T_i. The value of m has to be selected ahead of time, before deployment, so that the randomization machinery is included in the distributed code. To make this selection, we use an approach similar in spirit to existing techniques [6, 57]. First, a group of *opt-in* users is used to obtain detailed information in a non-differentially-private manner. Specifically, the set of local traces T_i from each opt-in user u_i is collected and reported to the analysis server. Then, the union of these sets is determined. The value of m is defined as the smallest power of 2 greater than or equal to the size of this union. This value of m is then used by the *regular* software users, whose copy of the software embeds this m value and only reports the randomized sketch of their local information.

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

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

Algorithm 2 Server-side processing.

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

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

4 Identification of Hot Traces

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

Exploration of estimated hot traces

Our approach takes as input the global sketch S_g, together with the set $\mathcal{E}$ of possible run-time events, the start event $s \in \mathcal{E}$, and the family of extension functions ext. We perform a pruned exploration of the elements of $\mathcal{T}$ defined by $\mathcal{E}$ and ext. The key observation is that if a trace

t is *not* hot, any t' that has t as a prefix cannot be hot, since the number of users that covered t' cannot exceed the number of users that covered t. This leads to the following approach: starting with the length-0 trace $\langle s \rangle$, explore the space of possible trace extensions defined by ext. For each explored trace t, estimate its frequency using sketch S_g and stop the exploration if the frequency estimate $\hat{f}(t)$ is below the hotness threshold h. Otherwise, continue the exploration with all traces in $\mathsf{ext}(t)$.

A key assumption of this approach is that for any given trace t, the set of extended traces $\mathsf{ext}(t)$ can be computed efficiently. We chose the two exemplar analyses presented in Section 2 – call chains and enter/exit traces – to illustrate two common cases where this computation is naturally derived from the definition of the underlying formal language. Such trace structure is not specific to these two examples; other dynamic analyses (e.g., paths in control-flow graphs) have similar properties. For call chains, the traces are strings in a regular language. Thus, the exploration is equivalent to exploring paths in the corresponding finite-state automaton. The extension function is defined by the set of possible transitions from the current state of the automaton. For enter/exit traces, defined by a Dyck context-free language (i.e., a language of balanced parentheses), the corresponding pushdown automaton can be maintained during the exploration of strings, and the extension function is again defined by the possible transitions from the current automaton state. Our implementation of these exemplar analyses uses exactly this approach. In both cases, the transitions are efficient: the cost of computing $\mathsf{ext}(t)$ is linear in the size of this set. Note that this approach is also applicable in the more general case where $\mathcal{T}$ is defined by an arbitrary context-free grammar, as the corresponding pushdown automaton can be maintained during trace exploration and consulted to decide how to extend the current trace.

Relaxed hotness criterion

Our experience indicates that the approach described above has the following disadvantage: sometimes entire groups of hot traces with a common prefix are not discovered because this prefix is misclassified as not being hot due to an inaccuracy of its frequency estimate. As a result, the exploration stops too early. To address this problem, we designed a more robust "relaxed" check for hot traces. If for some explored trace t we have $h/2 \leq \hat{f}(t) < h$, we consider this trace a potentially-misclassified hot trace due to an inaccurate estimate. In such cases, we check whether at least one $t' \in \mathsf{ext}(t)$ has an estimate above the threshold h. If such a t' exists, we take it as strong indication that t itself is hot and treat it as such. The details of the entire approach are presented in Algorithm 3.

For illustration, consider an enter/exit trace derived from actual data for the `equibase` app, which is one of our experimental subjects. The trace is $t = \langle \mathsf{enter}(0), \mathsf{enter}(1685), \mathsf{enter}(1678), \mathsf{enter}(910), \mathsf{enter}(805), \mathsf{enter}(10), \mathsf{exit}(10), \mathsf{exit}(805), \mathsf{exit}(910), \mathsf{enter}(1677) \rangle$. The true frequency is $f(t) = 818$. For the hotness cut-off $h = 810$ which was used in that experiment, the trace is hot. However, because of estimate $\hat{f}(t) = 763$, the exploration will stop at this trace if the relaxed criterion is not employed. As a result, 15 hot traces that have t as a prefix would be missed. Using the relaxed criterion, all 15 traces are correctly discovered by Algorithm 3.

5 Evaluation

For evaluation, we used 15 Android applications that were used by prior related work [59, 58]. We simulated 1000 users interacting with each app using the Monkey tool [23]. Specifically, we performed 1000 independent Monkey runs and considered each Monkey execution as triggered by one simulated user. During this process, for each run, we collected the sequence

Algorithm 3 Identification of hot traces.

```
output : H: set of estimated hot traces
Function hot_traces():
    H ← ∅
    for t ∈ ext(s) do explore(t)
Function explore(t):
    if hot(t) then
        H ← H ∪ {t}
        for t′ ∈ ext(t) do explore(t′)
Function hot(t):
    e ← f̂(t)
    if e ≥ h then return true
    if e < h/2 then return false
    for t′ ∈ ext(s) do
        if f̂(t′) ≥ h then return true
    end
    return false
```

of method enter/exit events until the total number of enter events reaches 10× the number of methods defined in the static app code (excluding libraries). If the app crashed or Monkey triggered events very slowly, we restarted Monkey and continued collecting the trace for this simulated user until the total number of enter events reached this targeted value. From this sequence of enter/exit events we constructed the set of observed call chains for that simulated user u_i – that is, set T_i for call chain analysis. In addition, we also considered the entry methods of the app and collect the subsequences that start at the enter events of those methods; these subsequences form set T_i for enter/exit trace analysis. Thus, for each of the two analyses we gathered sets $T_1, T_2, \ldots, T_{1000}$. We also wanted to study the effects of the number of users, but since execution of a large number of Monkey runs in device emulators takes a very long time, we employed an approach used by others [59]: each of the 1000 sets was replicated 10 times to generate T_i for $n = 10000$.

Our trace collection approach creates a threat to validity: it is well known that the app coverage achieved via tools such as Monkey can be limited [11]. In general, data generated by automated GUI crawling may not be representative of the behavior of real-world app users. One indication of coverage for our experiments is the size of $\cup_i T_i$, shown in columns "Total" in Table 1. For most apps, more than a thousand different traces were observed.

The instrumentation is based on the Soot code rewriting tool [47]. Only application code is instrumented, as this is the most likely focus of interest for app developers. We treat the following methods as app entry methods: methods that implement/override any Android framework methods (e.g., callbacks such as `onClick`); `<clinit>` methods; and `<init>` methods of application subclasses of Android framework classes.

Given the data collected by the instrumentation, we ran all randomization separately from the executions that gather the traces. This allowed us to conduct each experiment 30 times, in order to report rigorous statistical results that account for the randomness introduced by local randomizers [20]. Experiments were performed for several values of ϵ used in prior work [18, 52, 59, 58]. For brevity, most results are presented for $\epsilon = \ln(9)$, but the effects of other values are also discussed. To implement count sketch, we used SHA-256

Table 1 Experimental subjects and analyzed traces.

App	Classes	Call Chains				Enter/Exit Traces			
		Total	Len_{avg}	Time_u	Time_s	Total	Len_{avg}	Time_u	Time_s
`barometer`	379	2765	4.3	0.3	25	2717	10.2	0.4	6.4
`bible`	1107	1604	3.3	0.2	64	2427	8.8	0.2	21
`dpm`	272	1272	4.0	0.1	4.3	2475	10.6	0.2	3.7
`drumpads`	447	926	3.4	0.1	6	1289	8.8	0.1	4.1
`equibase`	252	773	3.0	0.1	3.2	1602	9.1	0.3	4.9
`localtv`	716	4037	4.6	0.3	42	5285	10.4	0.3	12
`loctracker`	198	480	1.8	0.1	0.8	1098	6.2	0.1	8.9
`mitula`	973	24757	7.0	2.8	1784	5614	10.2	0.8	27
`moonphases`	166	1755	6.4	0.2	3.3	947	9.9	0.1	0.6
`parking`	379	1477	3.1	0.1	10	2875	8.8	0.2	4.6
`parrot`	1099	7575	4.7	0.8	427	6499	10.0	0.9	63
`post`	1107	2358	3.8	0.4	92	3564	9.9	0.5	45
`quicknews`	1107	3668	3.4	0.4	51	6062	8.9	0.7	57
`speedlogic`	86	244	3.0	0.0	0.1	304	8.1	0.0	0.3
`vidanta`	1608	7811	4.9	0.8	833	6687	9.7	0.9	124

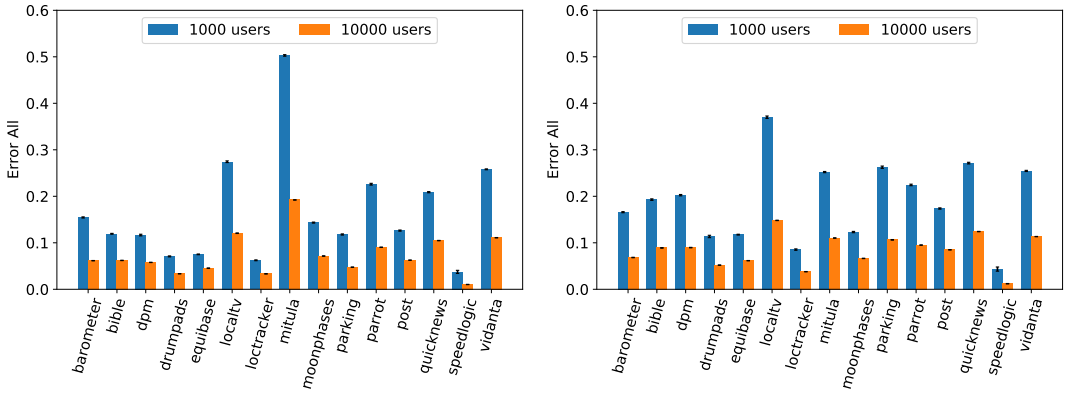
hashing. In particular, hash functions h_k and g_k used in count sketch were implemented by prepending k to the string representation of the trace (which itself is based on the methods ids), computing SHA-256, and taking the appropriate number of bits from the result.

Table 1 shows the details of the subjects used in our experiments. Column "Classes" lists the number of application classes, excluding several well-known third-party Android libraries, e.g., `dagger` and `okio`. The group of columns labeled "Call Chains" describes measurements for the call chain analysis, and the group labeled "Enter/Exit Traces" shows the same measurements for the analysis of enter/exit traces. Column "Total" and "Len_{avg}" show the total number of unique traces across the 1000 local sets T_i and their average length respectively. Column "Time_u" shows the average time (in seconds) to process the local data of a user, as described in Algorithm 1. Column "Time_s" contains the time (in seconds) to identify hot traces from the global sketch at the analysis server, using the approach from Algorithm 2 for $n = 1000$. For both analyses, the costs are practical and suitable for real-world use.

As mentioned in Section 3.3, we initially attempted to perform randomization separately for each covered trace, but incurred high running times for the local randomizer. This led to the development of the optimized approach in Algorithm 1. For example, for the `parrot` app, the naive randomization of call chains and enter/exit traces took 264 seconds per user on average, while the optimized one took 1.7 seconds. We typically observed two orders of magnitude improvement in the running time of the local randomizer.

5.1 Accuracy for All Covered Traces

The first research question we consider is this: *What is the accuracy of estimates for traces that are covered by at least one user?* Note that, from the data in the global sketch, the analytics server cannot directly determine this set of traces. (We address this issue in the next subsection.) However, the knowledge of this accuracy provides a useful baseline. To answer this question, we use a normalized L_1 distance between the vector of true frequencies



(a) Call chains.

(b) Enter/exit traces.

Figure 2 Error of estimates for all covered traces.

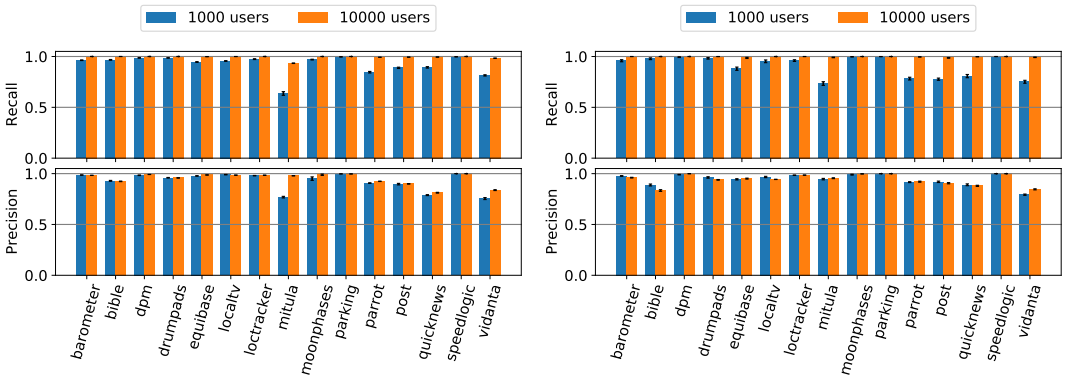
and the vector of their estimates. Specifically, for all t that appear in at least one T_i, we compute the error as $\sum_t |f(t) - \hat{f}(t)| / \sum_t f(t)$. Values close to 0 mean that the estimates are overall close to the real frequencies. Figure 2 shows these measurements for the two values of n. As described in Section 3.5, each run of this experiment (and all later experiments) used a randomly-selected set of size $n/10$ as opt-in users, and then performed the analysis and computed all reported error measurements for the remaining users. For these and all other experiments reported later, we followed a popular approach for statistically-rigorous performance measurements [20]: 30 independent runs of the experiment were performed, and the mean together with the 95% confidence interval are reported. The confidence interval characterizes the variance due to the randomization. In the bar charts, the confidence interval is shown at the top of the corresponding bar. In many cases, the interval is so small (i.e., the variance is so low), that it is hard to see in the figures.

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

5.2 Precision and Recall for Hot Traces

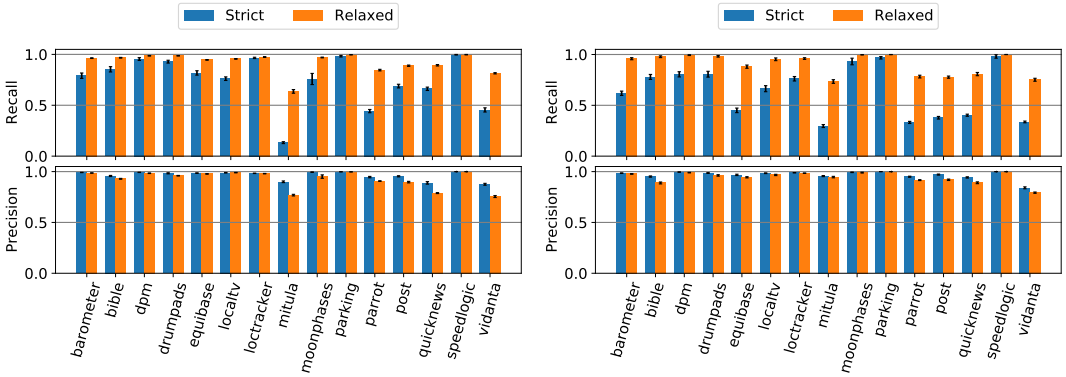
As discussed earlier, the set of all covered traces is not directly known to the analysis server. Section 4 discussed an approach to identify *hot traces*. Our next research question is: *How accurately are the hot traces identified?* The metrics we use to answer this question are recall (what portion of the true hot traces are discovered) and precision (what portion of the reported hot traces are actually hot). We executed Algorithm 3 on the global sketch to identify likely hot traces, with hotness threshold $h = 0.9 \times n$. This was done in 30 independent repetitions of the experiment. The mean values from these experiments and their 95% confidence intervals are shown in Figure 3.

Overall, the results of this experiment provide strong indication that hot traces can indeed be identified accurately with a sufficient number of users. For $n = 1000$, the average recall across the 15 apps is 92.1% and the average precision is 92.5% for call chains, and 90.4% and



(a) Call chains. **(b)** Entry/exit traces.

Figure 3 Recall and precision for hot traces.



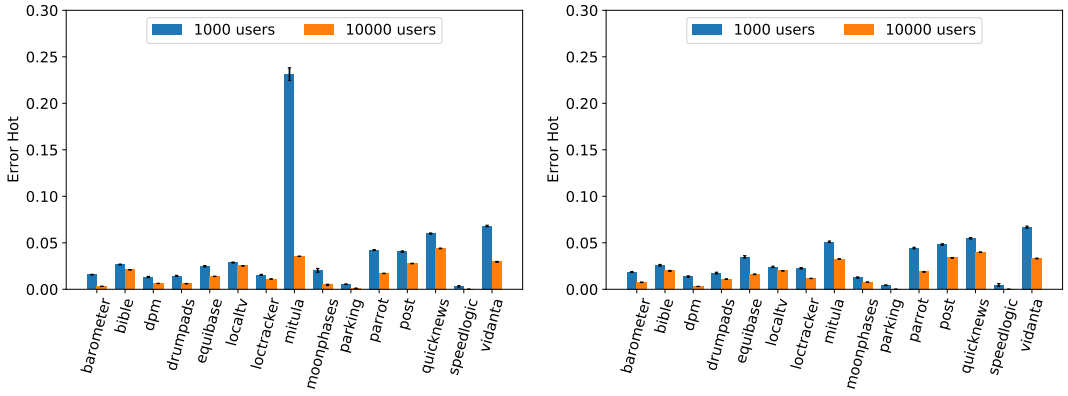
(a) Call chains. **(b)** Enter/exit traces.

Figure 4 Recall and precision for hot traces: strict vs relaxed hotness criterion.

94.5% for enter/exit traces respectively. For $n = 10000$, the recall for call chains increases to 99.3% and the precision to 95.0%; for enter/exit traces, the recall increases to 99.7% and precision decreases slightly to 94.1%. We investigated the apps with the lowest precision and determined that they have a large number of traces whose true frequencies are slightly below the threshold h; some of these almost-hot traces are misclassified as being hot, leading to the lower precision.

One related question is how the design choices for Algorithm 3 affect its precision and recall. In Section 4, we discussed two possible criteria for deciding whether a trace should be considered hot. The "strict" criterion is that a trace's estimate $\hat{f}(t)$ should exceed the hotness threshold h. However, if this estimate is inaccurate and too small, the chain and all other hot chains that have it as prefix will be missed. Thus, in the algorithm we use a "relaxed" criterion which also considers traces t with estimates $h/2 \leq \hat{f}(t) < h$ such that t has at least one extended trace with an estimate that exceeds h. This relaxed criterion was employed when collecting the data in Figure 3.

To understand the effects of this choice, we also measured precision and recall using the strict criterion. Figure 4 shows a comparison between the two criteria for $n = 1000$; the other value for n leads to similar conclusions. As can be seen from these measurements, using the strict criterion results in lower recall. For example, for call chain analysis, three apps have



(a) Call chains. **(b)** Enter/exit traces.

Figure 5 Error of estimates for reported hot traces.

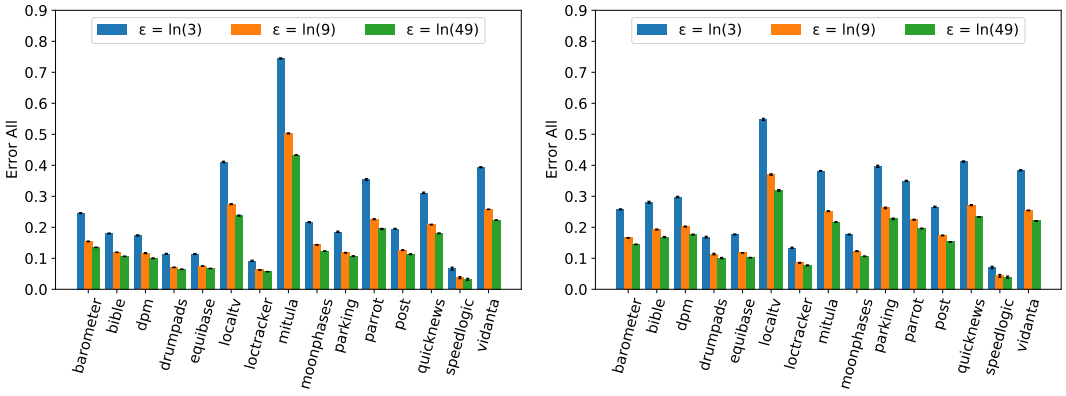
recall less than 50%. Similarly, for enter/exit trace analysis there are six apps with recall below 50%. As expected, the strict criterion does improve precision, but this effect is not very pronounced. Depending on the intended uses of the analysis, the app developers may prefer higher recall or higher precision. Using these two criteria, or variations of them, allows this trade-off to be adjusted as desired.

5.3 Accuracy of Estimates for Reported Hot Traces

For the set of traces reported by Algorithm 3 as likely-hot, we ask following question: *What is the accuracy of estimates for reported hot traces?* Figure 5 shows the error of estimates, using a metric similar to the one used in Figure 2: the sum of $|\hat{f}(t) - f(t)|$ for all reported hot traces t, normalized by the sum of $f(t)$ for those t. Based on these results, the answer to the question is that high accuracy is achieved for the frequency estimates of hot traces. Combined with the high recall demonstrated earlier, our conclusion is that hot traces and their frequencies can be successfully estimated via our differentially-private analysis.

Compared to other apps, in Figure 5a the error for app `mitula` is significantly larger for 1000 users. The underlying reasons are indicated in Figure 2a, where the estimates for this app have large cumulative error for 1000 users. This produces a large number of false positive hot chains (Figure 3a); further, those false positives have significant cumulative error. If we remove the false-positive hot chains from Figure 5a, the cumulative error becomes similar to that for the other apps. The reason for the error in Figure 2a is that there are many more chains in this app compared to the other apps. Further, the distribution of the frequency of these chains is not uniform: there is a large number of low-frequency chains, and the DP approaches produce inaccurate estimates for such chains. This can be solved by increasing the number of users (as can be seen in all figures for 10000 users): even the low-frequency chains now have enough instances to benefit from "random noise cancellation" across a larger number of instances.

It is instructive to compare Figure 5 with Figure 2. Overall, the estimate error for hot traces is smaller than the estimate error for all traces. For example, for $n = 10000$, the average error value in Figure 5a is 1.6%, compared to 7.4% in Figure 2a, and 1.7% vs 8.4% for Figure 5b vs Figure 2b. Theoretically, both count sketch and randomized response tend to favor higher-frequency items. The higher accuracy for hot traces demonstrates that this also holds in practice.



(a) Call chains. **(b)** Enter/exit traces.

Figure 6 Error of estimates for all covered traces for three values of ϵ.

5.4 Privacy Loss Parameter

As discussed earlier, the privacy loss parameter ϵ can be used to tune accuracy/privacy trade-offs. We considered the following question: *To what degree does accuracy change with changes in this parameter?* In existing work, ϵ ranges from 0.01 to 10 [28]. Related work that employs randomized response has used, for example, ln(3), ln(9), and ln(49) [18, 59, 58]. We computed the error for all covered traces for these three values; the results for ln(9) were already presented in Figure 2 and are repeated here. Figure 6 shows these measurements for $n = 1000$; similar trends are seen for the other n value. Overall, with increasing ϵ, the expected accuracy gains are observed but seem to level off. For call chains and enter/exit traces, respectively, the average error across all apps decreases from 25.3% and 28.7% for the smallest value of ϵ to 16.6% and 19.0% for ln(9), and then further to 14.5% and 16.5% for the largest value of the parameter. Based on these results, we consider ln(9) to provide a reasonable trade-off and have used it to present the majority of data in our evaluation. In practical scenarios, the developers can select a small fixed value of ϵ (before deployment), based on data from in-house testing or from real opt-in users, as well as the desired accuracy. Once selected, ϵ provides an upper bound on the privacy loss: for any data, and any two data items, they are guaranteed to be ϵ-indistinguishable. The real workload will affect only the accuracy, not the privacy.

5.5 Summary of Results

Our experimental results can be summarized as follows. First, as illustrated in Figure 2, the frequency estimates have high accuracy, for practical values of ϵ. This results indicates that with good privacy and sufficient number of software users, one can obtain accurate frequency estimates for software traces. Second, based on the results in Figure 3, the set of hot traces can be determined with high precision and recall. The relaxed identification of hot traces is important for achieving this result (Figure 4). Third, the frequency estimates for hot traces are accurate and better than those for the remaining covered traces (Figure 5). Finally, consider the accuracy/privacy trade-off spectrum: from smaller values of ϵ (i.e., stronger privacy) and lower accuracy, to larger values of ϵ and high accuracy. As indicated by Figure 6, after a certain point in this spectrum there do not seem to be significant additional improvements in accuracy.

6 Related Work

Differential privacy

There is a large body of work on both the theory and practice of differential privacy. As already discussed, several approaches based on randomized response consider a single data item per user [18, 52, 8], while we are interested in a set of data items (i.e., a set of locally-covered traces). Differentially-private analysis of software executions has also been studied in prior work [58, 57, 60]. In those efforts the domain of possible items is small, enumerated ahead of time before software deployment, and the randomizer output is straightforward to generate and store. A key distinguishing feature of our work is that the domain of possible traces is either infinite or very large, which requires different randomization techniques. We address this problem by using a count sketch representation. This allows tunable trade-offs between accuracy and representation size, as well as higher accuracy for high-frequency traces. Efficient randomization of simple bitvectors has also been considered [58]. Our efficient randomization (Section 3.3) requires more general reasoning. Because of the small number of possible data items, these prior efforts do not need to explore a large domain in order to identify hot items. In contrast, we need to develop effective search in a domain containing billions of possible traces. We demonstrate how to achieve this using considerations of chain prefixes and suffixes, and illustrate the approach for context-free-language domains by exploring the states of the corresponding automaton (Section 4).

Privacy-preserving techniques in programming languages and software engineering

The programming languages community has investigated techniques for testing and verification of differentially-private algorithms and implementations [56, 53, 36, 61]. Privacy issues are also important for many areas in software engineering, including design [25], testing [24, 9, 50, 32], and defect prediction [44, 45, 31]. Other than the work described earlier, we are not aware of attempts to employ differential privacy techniques in this area. Given the strong theoretical properties of such techniques, and their increasing adoption in industry and government [33, 4, 18, 51, 14], it is a worthwhile research goal to reconsider a range of software engineering techniques using differential privacy machinery.

Analysis of deployed software

Remote analysis of deployed software is an area with a significant body of prior work. As one example, residual coverage monitoring [43] uses coverage information from software users for testing purposes. GAMMA [42] collects data from software users and orchestrates the data collection across program instances. Placement of profiling probes has been considered by several projects [15, 39]. Failure reproduction and debugging are aided by collected data from deployed software [12]. Similarly, researchers have proposed analysis of post-deployment failure reports [38].

Privacy in remote software analysis has been targeted by prior efforts. Anonymization of collected data has taken several forms [17, 13]. As shown by privacy researchers [34, 35], anonymization is not enough to provide strong privacy guarantees. Instead, we consider the principled protection provided by local differential privacy. Remote software analyses from prior work could potentially benefit from developing differentially-private versions for them. Examples of such analyses include impact analysis and regression testing [41], as well as failure analysis [27, 29, 30].

7 Conclusions and Future Work

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

There is a large body of prior work on software analysis that could be revisited with increased emphasis on privacy in general, and differential privacy in particular [42, 41, 27, 12, 13, 29, 30, 38]. Such studies will contribute to broader efforts to integrate privacy-preserving techniques in the analysis of deployed software, in response to growing needs for better privacy of data collection.

References

1 ACM SIGACT/EATCS. Gödel Prize. `https://sigact.org/prizes/g%C3%B6del/citation2017.pdf`, 2017.

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

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

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

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

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

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

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

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

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

11 S.R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *ASE*, pages 429–440. IEEE, 2015.

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

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

14 A. Dajan, A. Lauger, P. Singer, D. Kifer, J. Reiter, A. Machanavajjhala, S. Garfinkel, S. Dahl, M. Graham, V. Karwa, H. Kim, P. Leclerc, I. Schmutte, W. Sexton, L. Vilhuber, and J. Abowd. The modernization of statistical disclosure limitation at the U.S. Census Bureau. `https://www2.census.gov/cac/sac/meetings/2017-09/statistical-disclosure-limitation.pdf`, 2017.

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

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

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

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

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

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

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

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

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

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

25 I. Hadar, T. Hasson, O. Ayalon, E. Toch, M. Birnhack, S. Sherman, and A. Balissa. Privacy by designers: Software developers' privacy mindset. *Empirical Software Engineering*, 23(1):259–289, 2018.

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

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

28 J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *CSF*, pages 398–410, 2014.

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

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

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

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

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

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

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

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

37 Oath. Flurry. `http://flurry.com`.

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

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

40 OpenDP. `https://projects.iq.harvard.edu/opendp`, 2020.

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

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

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

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

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

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

47 Soot. Soot analysis framework. `https://soot-oss.github.io/soot`, 2020.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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