PP-CSA: Practical Privacy-Preserving Software Call Stack Analysis

ZHAOYU WANG, Hong Kong University of Science and Technology, China
PINGCHUAN MA*, Hong Kong University of Science and Technology, China
HUAIJIN WANG, Hong Kong University of Science and Technology, China
SHUAI WANG*, Hong Kong University of Science and Technology, China

Software call stack is a sequence of function calls that are executed during the runtime of a software program. Software call stack analysis (CSA) is widely used in software engineering to analyze the runtime behavior of software, which can be used to optimize the software performance, identify bugs, and profile the software. Despite the benefits of CSA, it has recently come under scrutiny due to concerns about privacy. To date, software is often deployed at user-side devices like mobile phones and smart watches. The collected call stacks may thus contain privacy-sensitive information, such as healthy information or locations, depending on the software functionality. Leaking such information to third parties may cause serious privacy concerns such as discrimination and targeted advertisement.

This paper presents PP-CSA, a practical and privacy-preserving CSA framework that can be deployed in real-world scenarios. Our framework leverages local differential privacy (LDP) as a principled privacy guarantee, to mutate the collected call stacks and protect the privacy of individual users. Furthermore, we propose several key design principles and optimizations in the technical pipeline of PP-CSA, including an encoder-decoder scheme to properly enforce LDP over software call stacks, and several client/server-side optimizations to largely improve the efficiency of PP-CSA. Our evaluation over real-world Java and Android programs shows that our privacy-preserving CSA pipeline can achieve high utility and privacy guarantees while maintaining high efficiency. We have released our implementation of PP-CSA as an open-source project at https://github.com/wangzhaoyu07/PP-CSA for results reproducibility. We will provide more detailed documents to support and the usage and extension of the community.

CCS Concepts: • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: differential privacy, call stack analysis

ACM Reference Format:
Zhaoyu Wang, Pingchuan Ma, Huaijin Wang, and Shuai Wang. 2024. PP-CSA: Practical Privacy-Preserving Software Call Stack Analysis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 139 (April 2024), 30 pages. https://doi.org/10.1145/3649856

1 INTRODUCTION

A software call stack is an event sequence that records the functions being called during the runtime of a software program. Consider a software program that is deployed on an end-user device like a mobile phone, each user interaction with the software program will yield a call stack, which is

*Corresponding author

Authors' addresses: Zhaoyu Wang, Hong Kong University of Science and Technology, Hong Kong, China, zwangjz@cse.ust.hk; Pingchuan Ma, Hong Kong University of Science and Technology, Hong Kong, China, pmaab@cse.ust.hk; Huaijin Wang, Hong Kong University of Science and Technology, Hong Kong, China, hwangdz@cse.ust.hk; Shuai Wang, Hong Kong University of Science and Technology, Hong Kong, China, shuaiw@cse.ust.hk.

ACM 2475-1421/2024/4-ART139
https://doi.org/10.1145/3649856

then sent to remote servers for analysis. Call stack analysis (CSA) is a widely used technique in software engineering [Glerum et al. 2009]. It facilitates analyzing the runtime behavior of software in order to optimize software performance, identify defects, and profile software [Han et al. 2012; Hao et al. 2021; Jin and Orso 2012]. To date, many software applications are deployed on remote end-user devices such as mobile phones. Hence, collecting user call stacks enables developers to continuously gain insights into how users interact with their software, thereby better optimizing the software functionality and performance. By gathering call stacks in accordance with various daily software usages, developers can perform different analytic tasks and create software that is more user-friendly and efficient.

Although CSA offers several advantages, it has recently faced privacy concerns. Call stacks are essentially event sequences that illustrate the software's runtime behavior. In real-world situations, these behaviors can be proprietary and confidential, holding valuable information for the software users. As aforementioned, CSA in real-world scenarios is frequently performed over software apps deployed on user-side devices, such as mobile phones or smart watch devices. Given that on-device scenarios often involve user private data, there is concern that the collected call stacks may reveal a non-trivial amount of user privacy [Hao et al. 2021], such as user health conditions or locations, imposing compliance issues with privacy laws such as the GDPR or CCPA [Goldman 2020; Voigt and Von dem Bussche 2017]. To illustrate the privacy risks of CSA in practice, we study and present a motivating example over real-world Android apps in Sec. 3.

Current Solution. Existing solutions, although laying a solid foundation for this emerging problem, may not adequately address the needs of privacy-preserving CSA in practice. First, despite a significant volume of contributions from the NLP community that aim to privatize texts under local differential privacy (LDP) scheme [Habernal 2021; Igamberdiev and Habernal 2023; Krishna et al. 2021], they are not directly applicable to privatize call stacks. This is primarily because the privatization of text often manifests distinct lexical and syntactic features different from the original text and only retain the semantic meaning. For instance, under the LDP scheme, the text "Nice to meet you" can be privatized as "Good to see you". While this may be generally acceptable for text analysis, it is not the case for call stack analysis. Call stacks, as sequences of function calls, must adhere to the inherent caller-callee associations and the chronological order of function calls. A small perturbation in the call stack may result in a significant change in the call stack semantics, and thus the utility of the privatized call stack is compromised.

Second, we are also aware of recent works [Hao et al. 2021] that employ the randomized response technique to protect the privacy of call stacks from individual users. Despite the encouraging results, we note that [Hao et al. 2021] ignores the inherent semantics-level constraints of program call stacks, resulting in a considerable privacy cost while simultaneously compromising on accuracy. We will discuss the details and compare our method with them from a conceptual perspective in Sec. 3. Accordingly, we present empirical comparisons in Sec. 6.1, illustrating the encouraging performance of our proposed solution.

Our Solution. We present a practical framework, PP-CSA, for privacy-preserving CSA, which offers principled privacy guarantees while maintaining high utility and accuracy for real-world software scenarios. In particular, we design an encoder-decoder pipeline that can privatize call stacks using standard differential privacy (DP) mechanisms (such as Laplace mechanism or Gaussian mechanism). We further propose several key optimizations on both the client and the server sides to improve the performance of PP-CSA. On the client side, we design several call stack compression schemes to reduce the length of call stacks and simplify the privatization process. Moreover, on the server side, we propose several optimizations, including a reachability-enhanced training, call graph-guided decoding, and probabilistic distribution decoding, to largely improve the accuracy of the decoded call stacks without undermining the privacy guarantees. Finally, we aggregate the results from the sampled call stacks to obtain the final analysis result, i.e., to identify frequently used call stacks.

Evaluation Highlight. We implement PP-CSA and evaluate it on real-world Java programs and Android applications. Our evaluation shows that PP-CSA can achieve high accuracy and efficiency, while maintaining high privacy under active attackers. PP-CSA consistently outperforms the state-of-the-art solutions [Hao et al. 2021; Igamberdiev and Habernal 2023] across different settings. We conducted evaluations using different privacy budgets ($\epsilon = 1, 10, 20, 30, 40, 100$). Notably, PP-CSA achieves an average F1 score of 0.816 on the *DaCapo* dataset and 0.795 on the Android dataset, whereas [Hao et al. 2021; Igamberdiev and Habernal 2023] have only about 0.623 and 0.503 F1 scores, respectively. We also show that PP-CSA's optimizations are highly effective and have a "synergistic effect"; enabling full optimizations offers the highest accuracy (on average 9.2% F1 score improvement over that of disabling all optimizations). When considering adversaries who aim to infer the user privacy from the privatized call stacks, attackers can only achieve around 0.2 adversarial uncertainty on accuracy [Hua et al. 2022] under common privacy budgets (e.g., $\epsilon = 40$), a sufficiently high level of user privacy protection. In sum, our work makes the following contributions:

- This research presents a practical and privacy-preserving CSA framework that can be deployed in real-world scenarios. Our framework offers principled privacy guarantees and is scalable to real-world software call traces.
- Our presented framework, PP-CSA, offers local differential privacy (LDP) guarantee. PP-CSA features several key optimizations in the technical pipeline, including the LDP mechanism, the trace compression scheme, and several trace decoding schemes. Our optimizations are tailored for CSA and can effectively enhance the performance.
- Our evaluation over real-world Java/Android programs shows that our privacy-preserving CSA pipeline can achieve high accuracy and efficiency, while preserving high privacy to mitigate the privacy risks of CSA.

Artifact Availability. We have released our artifacts at [artifact 2023]. We will maintain them for future research comparison and usage.

2 PRELIMINARY

In this section, we present the necessary background of this research, including the local differential privacy (LDP) scheme and call stack analysis (CSA). For complete details of DP basics, we direct interested readers to relevant resources [Dwork 2006; Dwork et al. 2014].

2.1 Local Differential Privacy (LDP)

Differential Privacy (DP) has emerged as a standard framework to ensure privacy guarantees in data-processing algorithms [Dwork 2006; Dwork et al. 2014]. This framework allows for statistical analysis of data while safeguarding the privacy of individuals within the dataset. Importantly, DP provides reliable privacy protections, even when faced with arbitrary side information.

Despite its robust privacy safeguards, DP typically presupposes a trusted data curator (e.g., a central server) to execute necessary data analysis and noise addition tasks. This assumption can be limiting in situations where users distrust a centralized authority or when individual privacy protection is paramount. For instance, in CSA, the traces are often collected from user-side devices like mobile phones, and sent back to the software vendor's central server for analysis. In such scenarios, the central server is not trusted by the users, let alone trusting the central server to perform the necessary DP protection over user data. To tackle these issues, Local Differential Privacy (LDP) was proposed as a localized approach for DP.

LDP is a prominent technique in data privacy that gives privacy assurances at the level of the individual while permitting correct data aggregation and statistical analysis. Essentially, LDP imposes the addition of random noise to each data point prior to sharing or analysis. Properly chosen random noise not only assures anonymity, but also ensures the accuracy of statistical analysis. Formally, LDP is defined as follows.

Definition 1 (Local Differential Privacy). *Let $\mathcal{D}$ be a domain of data values, and let R be a domain of responses. A randomized mechanism (or function) $\mathcal{M} : \mathcal{D} \rightarrow R$ is said to be (ε, δ)-locally differentially private if for any two inputs $x, x' \in \mathcal{D}$ and for all $r \in R$, the following inequality holds:*

$$\Pr[\mathcal{M}(x) = r] \leq e^{\varepsilon} \Pr[\mathcal{M}(x') = r] + \delta \tag{1}$$

where $\varepsilon > 0$ and $0 \leq \delta < 1$.

Here, ε is referred to as the privacy budget, and δ is the failure probability. A smaller value of ε indicates a stronger privacy guarantee. Likewise, a smaller value of δ increases the likelihood of the LDP mechanism satisfying the inequality.

LDP uses a randomized mechanism to convert a data value x into a response r, such that the probability of obtaining r from x is close to that of obtaining r from a different data value x'. Using this mechanism, adversaries cannot deduce the original data value x from the response r with high confidence, ensuring *indistinguishability*. Compared to centralized DP, LDP offers more potent privacy guarantees [Balle et al. 2019] (for the same ϵ, δ) and assumes a more practical stance towards the data curator.

2.1.1 Sample LDP. To illustrate the concept of the *indistinguishability* property in LDP, we introduce a simple implementation of LDP known as *Randomized Response* (RR) [Warner 1965]. RR is a conventional technique in statistics, enabling the collection of sensitive information from respondents while preserving their privacy. Suppose a survey asks a sensitive yes/no question, such as "Have you ever used illegal drugs?" Each respondent flips a coin in secret. If the coin lands on heads, they answer truthfully; if it lands on tails, they flip a second coin in secret and answer “yes” if the second coin lands on heads, and “no” if it lands on tails. The respondent then reports the answer to the surveyor. Although the surveyor cannot determine whether the respondent answered truthfully, they can estimate the proportion of respondents who have used illegal drugs. Thus, RR enables statistical analysis of the data while preserving the privacy of individual respondents. It has been proven that RR is a well-formed $(\log 3, 0)$-LDP [Dwork et al. 2014], meaning its $\varepsilon = \log 3$ (a very low privacy budget) and $\delta = 0$.

Despite the simplicity and appealing privacy guarantees of RR, it is primarily used for categorical data (e.g., yes/no questions) and not advisable for sequence data (e.g., call stack traces) [Wang et al. 2019]. As a consequence, existing LDP-based CSA [Hao et al. 2021] relies on a large corpus of call stack traces (e.g., 10K) and a high privacy budget (e.g., $\epsilon > 100$) to achieve a plausible utility. Given call stacks are usually collected during software crashes, expecting a large corpus is not practical and an excessive privacy budget risks privacy loss.

In addition to RR, another representative approach is Gaussian Mechanism [Dwork et al. 2014]. We now define L-2 sensitivity and the Gaussian Mechanism in further detail.

Definition 2 (L-2 Sensitivity). *L-2 sensitivity of f is defined as $\Delta_2 = \max_{d(D,D')=1} \|f(D) - f(D')\|_F$ where $\|\cdot\|_F$ is Frobenius norm.*

Theorem 1 (Gaussian Mechanism). *For function $f : \mathcal{D} \rightarrow \mathbb{R}^k$, $\mathcal{A}$ satisfies (ϵ, δ)-differential privacy if $\mathcal{A}(D) = f(D) + \mathcal{N}(\sigma^2 \boldsymbol{I}_k)$ where $\mathcal{N}(\sigma^2 \boldsymbol{I}_k)$ is a k-dimensional Gaussian noise, $\sigma = \sqrt{2 \ln \frac{1.25}{\delta}} \frac{\Delta_2}{\epsilon}$, and Δ_2 is the L-2 sensitivity of f.*

The Gaussian mechanism was originally designed for central DP, but it can be extended to LDP by using D with exactly one data point. Thus, Δ_2 is the L-2 sensitivity of f on a single data point. As a result, the noise scale σ is proportional to Δ_2 and ϵ. And, loosely speaking, Δ_2 is proportional to the dimension of f's output (i.e., k).

2.2 Call Stack Analysis (CSA)

CSA is a common concept in software engineering [Glerum et al. 2009]. CSA often considers a software execution trace composed of a sequence of *runtime events*. To date, CSA has been widely used to optimize the software performance, identify bugs, and profile the software. For example, the Android ecosystem provides a tool called `systrace` [developer manual 2023] to collect the call stacks of Android apps. The collected call stacks can be used to identify the performance bottlenecks or to identify the bugs in Android apps. Similarly, `FlowDroid` [Arzt et al. 2014], a popular analysis framework of Android apps, uses the collected call stacks to identify the information leakage bugs of Android apps.

Depending on the specific CSA tasks, the "runtime events" can be function calls, system calls, or other program runtime behaviors. Aligned with prior works [Hao et al. 2021], this paper considers an important and cornerstone CSA task, i.e., *hot call stack analysis*, where each event on the trace is a function call. The identified hot traces can be used to optimize the software performance, or to identify the bugs of the software. Formally,

Definition 3 (Hot Call Stack Analysis). *Consider a software distributed over n users, denoted as $u_1, \cdots, u_n$. For each user u_i, let S_i represent the call stack when executing the software on his end device. Then, the frequency $f(S_i)$ of S_i among all users is computed as $f(S_i) = \frac{N(S_i)}{n}$, where $N(S_i)$ denotes the number of S_i's occurrences among call stacks collected over all users. Then, given a threshold $T_{hot} \in [0, 1]$, S_i is considered a hot call stack if $f(S_i) \geq T_{hot}$.*

It is clear that when sending the local data to the untrusted server, the privacy of each user may be leaked (see an example in Sec. 3). Hence, we aim to enable the clients to perturb their local data and share it to an untrusted analysis server without leaking their privacy or undermining the analysis result.

Hot call stack analysis is not only a common but also a cornerstone CSA task. Moreover, considering this task eases an apple-to-apple comparison with prior works [Hao et al. 2021]. Nevertheless, our proposed framework is general and can be applied to other group statistics-based CSA tasks. We leave the exploration of other CSA tasks as future work. See Sec. 7 for more discussions.

Call Trace v.s. Call Stack. We are also aware that call trace analysis [Hao et al. 2021] is a related concept to CSA. In general, call trace refers to the whole sequence of function calls that occurs during the execution of a program, whereas a call stack contains only the currently active function calls on the stack. In this regard, call traces can be lengthy and maintaining them during runtime can be resource-intensive. In contrast, call stacks offer a standard and rather concise representation of the currently active function calls; they are practically valuable for debugging and performance optimization purposes, especially in the scenarios where software is deployed on client-side devices. Nevertheless, our proposed technique itself is generally applicable to call trace analysis as well.

3 RESEARCH MOTIVATION

This section introduces the motivation of this research. We begin by presenting several real-world examples that illustrate the privacy concerns of CSA in practice in Sec. 3.1. Furthermore, we provide our formulation of privacy in the context of CSA and explain the necessity of securing CSA in Sec. 3.2. From a technical perspective, we also discuss the limitations of existing solutions in Sec. 3.3.

3.1 Real-World Examples of Privacy Concerns in CSA

Collecting user call stacks allows developers to understand how users interact with their apps, thereby optimizing both the functionality and performance. By gathering these call stacks, vendors can perform various analytics tasks, leading to more efficient and effective software. However, a significant "side effect" is the potential leakage of sensitive data [Hao et al. 2021].

In real-world scenarios, CSA is frequently performed on software apps deployed on user-side devices like mobile phones. Given that on-device scenarios often involve user data, there is concern that the collected call stacks may reveal a non-trivial amount of sensitive data, such as user health conditions or location, depending on the software's functionality. To illustrate this point, the authors spend manual effort to study a real-world health app collected from the Google Play Store.[1]

The analyzed app is a widely used mobile application designed to support and empower patients throughout their treatment and recovery process. It allows users to log basic health records and vital signs, and also offers a feature to swiftly share this health data with their doctors. We collect this app, and perform a manual analysis to identify potential privacy concerns after first reverse engineering the app code. In particular, upon exploring the app, we identified two particular call stacks that could potentially leak user health information. These two cases are detailed in Listing 1 and 2.

```java
public class AlarmReciever ... {

  public void onReceive(Context context, ...) {
    ...
    Intent i = new Intent(context, OnRecieverMedicineActivity.class);
    Notification n = getNotification(i);
    NotificationManager notificationManager = getNotificationManager(context);
    notificationManager.notify(n);
    ...
  }
}

public class OnRecieverMedicineActivity ... {

  protected void onCreate(...) {
    String bodyText = fetchMedicineInfo(...);
  }
}
```

Listing 1. Code for medicine reminders (simplified to ease understanding).

Our first observed concern arises from the medicine reminder feature. As illustrated in Listing 1, when the designated time for taking medicine approaches, the `onReceive` function of the `AlarmReceiver` class is activated. This function, in turn, invokes the `showNotification` method, which is in charge of displaying the medicine reminder. Within `showNotification`, the function `fetchMedicineInfo` is called to pull the medicine record. Thus, if the call stack successively lists `onReceive`, `showNotification`, and `fetchMedicineInfo`, it hints that the user might have a chronic illness and is on regular medication, possibly exposing sensitive health details. When seeing such a call stack, the server can infer that the user is taking medicine, and may even be

[1]The app name and details have been anonymized. The SHA-256 hash of the app name is 038187dc805631b77cfdc29d73b310ccad26f637307179ba7f78944c82913257.

```
public class AppointmentActivity ... {

  protected void onCreate( ... ) {
    ...
    createAppointment();
    ...
  }

  private void createAppointment() {
    Intent i = new Intent(this, AppointmentEditActivity.class);
    startActivityForResult(i, ACTIVITY_CREATE);
  }
}

public class AppointmentEditActivity ... {

  public void onCreate(...) {
    ...
    long id = addAppointment(when, where, time);
    ...
  }
}
```

Listing 2. Code for medical appointment (simplified to ease understanding).

able to infer the specific medicine that the user is taking, thus resulting in the leakage of sensitive health information.

Similarly, we observe another privacy issue tied to the management of medical appointments. As depicted in Listing 2, the `AppointmentActivity` class oversees the creation of new doctor appointments. When a user chooses to schedule an appointment, the `createAppointment` function is triggered. This function then prompts the `onCreate` method of the `AppointmentEditActivity` class. As a result, when `createAppointment` and `onCreate` consecutively appear in the call stack, it evidently implies that the user is arranging a new doctor's appointment. When collecting trace traces like this, the server can infer that the user is scheduling a new doctor's appointment, and may even be able to infer the specific doctor that the user is visiting. This presumably results in the leakage of sensitive health information and also users' location information to the server side.

3.2 Formulation and Clarification on "Privacy" in the CSA Context

In this section, we present a formulation and clarification on the notion of "privacy" in the context of CSA. We begin by presenting the threat model in our study and introducing the privacy guarantee provided by PP-CSA. Subsequently, we make a clarification on the necessity of securing CSA to further motivate our research. Moreover, we briefly discuss other privacy techniques that may be applicable in this context and explain the distinct privacy guarantee provided by PP-CSA.

Threat Model. In this study, our goal is to address the privacy concerns highlighted in the examples presented in Sec. 3.1. Our threat model is *aligned* with previous research in this field [Hao et al. 2021; Zhang et al. 2020a,b] and this work is not making unrealistic assumptions. In particular, the threat model shared by ours and previous works considers a software system with a client-server architecture, where the client is a user's end device (e.g., a mobile phone) and the server is a remote entity responsible for collecting and analyzing call stacks from the client. It is assumed that the server is honest-but-curious, i.e., the server will follow the protocol but may try to infer the user's private information (his health information) from the LDP-protected call stack data submitted by the client. It is also assumed that the client is trusted, i.e., the client will follow the protocol and submit the LDP-perturbed data to the server.

Privacy Guarantee of PP-CSA. Our proposed solution, PP-CSA, ensures that the central server cannot infer the sensitive attributes related to a specific user (e.g. whether a specific user is taking medicine) with high confidence, while still being able to conduct various normal statistical analysis over the collected call stacks, e.g., for the purpose of software optimization. This guarantee is rigorously provided by LDP, with the degree of privacy assurance quantified by the privacy budget ε, as noted in Sec. 2.1. However, we cannot prevent the server from inferring statistical facts across all users, such as nearly 60% of users regularly taking medicine. This is *not* the focus of our study. And it is important to note that the unit of privacy in our framework primarily focuses on instance-level differential privacy, where each user is expected to report only one call stack. But our work can be easily extended to support the cases of user-level privacy, where the server makes repeated queries to the client to collect a sequence of call stacks. See Sec. 7 for more details.

Despite the potential privacy risks from call stacks, we however treat other possible attacks as beyond our research scope. For example, following the standard client-server model, it is easy for the client and server to mutually authenticate each other (thus avoiding a man-in-the-middle attack), and all the communication between the client and server is encrypted. All these operations have been widely supported by existing secure communication protocols, such as HTTPS and TLS. As a result, the data confidentiality, integrity, and authenticity can be guaranteed, and our focus is on facilitating privacy guarantee of the call stacks.

Why Securing CSA is Particularly Important? Give the aforementioned privacy guarantee, careful readers might challenge the necessity on securing CSA, especially in an enterprise setting.

① First, one may question that when a user installs a health-related app, the vendor may have a prior that the user may be taking medicine or scheduling doctor appointments. However, we clarify that merely installing health-related apps does not necessarily indicate that a user has a specific issue. In contrast, the patterns observed in call stacks related to doctor appointments and medication usage can provide valuable and concrete information, leaking highly sensitive data about users' health conditions.

② Also, one may challenge that the vendor may already be aware of the health conditions of its users, especially if all doctor appointments are coordinated through the vendor. Nevertheless, we underscore that transactional data (e.g., doctor appointment records) is often stringently protected to meet compliance standards. Thus, it is very unlikely that the vendor can access the transactional data in real-world scenarios. In contrast, software call stacks, typically stored as logs, are pervasively collected and analyzed by vendors. They were often considered less sensitive than transactional data, and thus the privacy implications tend to be overlooked. In reality, the permissions for these log data are often more permissive than those for transactional data, aiming to facilitate the debugging process for developers. This renders call stacks more susceptible, underscoring the imperative of securing CSA.

Other Privacy-Enhancing Techniques. To protect against call stack information, some other advanced techniques like secure multi-party computation (MPC) [Yao 1982] or homomorphic encryption (HE) [Gentry 2009] may be required. These techniques focus on completely hiding the data from the server while still allowing it to perform computations. However, these techniques are often not practical for real-world scenarios due to their high computational overhead, and thus are not considered in this study. We leave exploring low-cost solutions of MPC or HE in the context of program analysis as future work. Sec. 8 also reviews relevant works in this direction. Additionally, anonymization transmission services, such as the Tor network [Dingledine et al. 2004] can also be helpful in this context. These services specialize in anonymizing user identities and offer an additional layer of privacy protection. However, it is important to note that the privacy guarantees provided by anonymization transmission services are distinct from our work.

Attackers with background knowledge can still deduce sensitive information from anonymized data (e.g., de-anonymization attacks on Tor [Kwon et al. 2015]) and undermine privacy. In contrast, the DP-based approach in PP-CSA offers a principled and distinct way to limit the abilities of such attackers and safeguard against these types of adversarial inferences [Yang et al. 2012]. By leveraging DP, PP-CSA offers a distinct privacy guarantee that cannot be solely achieved through anonymization. Furthermore, due to the different primary security objectives, integrating PP-CSA with anonymization transmission services can yield stronger privacy protection. We discuss this potential integration in Sec. 7.

3.3 Limitations of Existing DP-Based Solutions

Prior works [Hao et al. 2021; Zhang et al. 2020a,b] have proposed using DP to protect the privacy of individual users in software analytics. In particular, the state-of-the-art work [Hao et al. 2021] encodes each call stack via a hash function (e.g., SHA-256) to a *DP-friendly* fixed-length binary vector, on which randomized response (RR) mechanism is applied by randomly flipping some bits. Then, on the server side, the noisy reports are aggregated to yield an estimation of the frequency of each call stack, identifying the "hot traces." Also, in the NLP community, there are emerging attempts [Habernal 2021; Igamberdiev and Habernal 2023; Krishna et al. 2021] to apply DP to protect the privacy of texts. In essence, these works employ an encoder to map the text into a fixed-length numerical vector, on which Laplace or Gaussian noise is added to each dimension. Then, the noisy vectors are passed to a decoder to denoise and reconstruct the original text. Usually, pre-trained language models (e.g., BERT-family models [Kenton and Toutanova 2019; Lewis et al. 2020]) are used to initialize the encoder and decoder.

Despite the encouraging results of prior works, we clarify that the existing solutions are not practical for real-world scenarios due to the following two reasons: ❶ *adherence to caller-callee relationship*, and ❷ *awareness of epistemic uncertainty*.

Adherence to Caller-Callee Relations. It is worth noting that, unlike free-form texts, call stacks in our context consist of a sequence of function calls, which have to adhere to the legitimate caller-callee relationship of the program. The randomized response mechanism (as well as the count sketch technique used in [Hao et al. 2021]) can asymptotically recover the frequency only when the noisy report also represents a valid item. Given the prior that the frequency of invalid call stacks is strictly zero, it becomes questionable whether the technique used in [Hao et al. 2021] can provide an accurate estimation of call stack frequencies. To illustrate this issue, consider a call stack s that yields an encoding e via a hash function h. Then, the randomized response mechanism is applied to e to yield a noisy encoding e'. However, there does not exist a call stack s' that yields e' via h. As a consequence, the occurrence of s vanishes in the aggregated report, leading to an underestimation of the frequency of s.

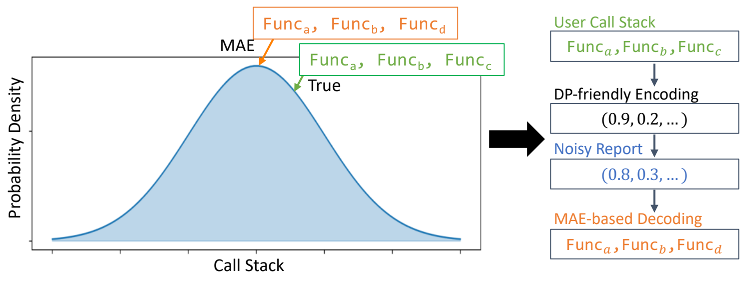

Fig. 1. Illustration of call stack variation issue under LDP.

Awareness of Epistemic Uncertainty. In addition, we argue that, when applying DP to protect the privacy of call stacks, it is important to be aware of the epistemic uncertainty in the decoding process. In particular, the standard text privatization pipeline [Igamberdiev and Habernal 2023] usually reconstructs the text from the noisy vector by simply maximizing the likelihood of the text. However, this approach does not consider the epistemic uncertainty in the decoding process, which may lead to a significant deviation from the original text. In particular, the true call stack does not necessarily correspond to the MLE (maximum likelihood estimation) over the noisy report. In contrast, the noisy report represents a distribution of possible call stacks, and the true call stack is a sample from the distribution. As shown in Figure 1, when merely taking a "point estimation" of the noisy report, we may end up with a call stack that is different from the true one.

4 DESIGN OF PP-CSA

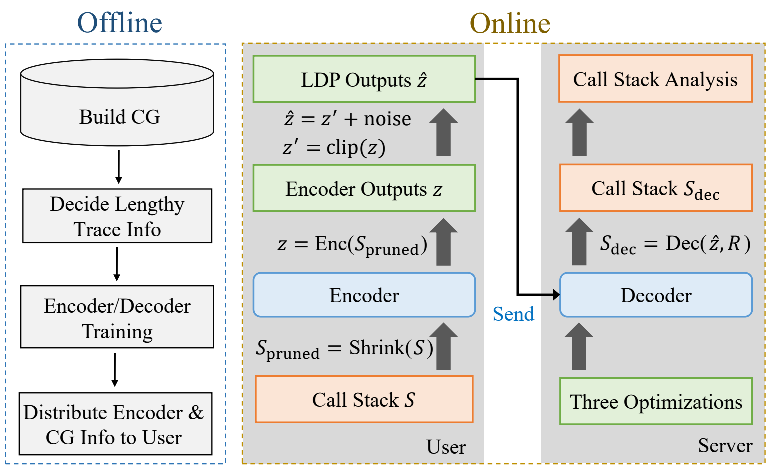
Offline
Online
Build CG
Decide Lengthy Trace Info
Encoder/Decoder Training
Distribute Encoder & CG Info to User
LDP Outputs $\hat{z}$
$\hat{z} = z' + \text{noise}$
$z' = \text{clip}(z)$
Encoder Outputs z
$z = \text{Enc}(S_{\text{pruned}})$
Encoder
$S_{\text{pruned}} = \text{Shrink}(S)$
Call Stack S
User
Send
Call Stack Analysis
Call Stack S_{dec}
$S_{\text{dec}} = \text{Dec}(\hat{z}, R)$
Decoder
Three Optimizations
Server

Fig. 2. Design of PP-CSA.

We present the design overview of PP-CSA in Figure 2. Overall, PP-CSA features both **offline** and **online** phases to achieve privacy-preserving CSA. During the **offline** phase (i.e., before client-side deployment), the developer constructs the call graph of the software of interest, instantiates rules to shrink the call stacks, and trains an encoder-decoder model to enforce LDP (details in Sec. 4.1). Then, the software, together with the trained model and the rules, is deployed to the client side. During the **online** phase, when users execute the software, the client side collects the correspondingly logged call stacks. Note that common software ecosystems like Java and Android have already provided such logging functionality, which can be easily enabled by developers. Then, the rules obtained offline are first applied to shrink the call stacks (details in Sec. 4.1). After that, the encoder model is applied to convert the shrunken call stacks into an LDP-protected representation. The client side then submits the LDP-protected representation to the server side, which is later decoded to recover call stacks. Finally, the server side performs CSA on the call stacks aggregated from multiple users to decide the "hot call stacks". We propose several optimizations to largely improve the efficiency of PP-CSA at this step without sacrificing the privacy guarantees (details in Sec. 4.2). Our empirical evaluation, as will be reported in Sec. 6, shows that PP-CSA can achieve high utility and privacy guarantees while maintaining high efficiency. Before elaborating on the technical details, we first discuss the application scope and assumptions below.

Application Scope. PP-CSA is a general-purpose framework for privacy-preserving CSA. Its technical pipeline is orthogonal to particular software implementation or programming languages, and thus applicable to a wide range of call stacks and software. Our observation shows that real-world software of varying purposes (e.g., medical or financial apps) may result in privacy concerns, and PP-CSA is applicable to all these scenarios. Aligned with previous works [Hao et al. 2021], we focus on CSA, particularly hot stack analysis, of Java and Android programs, given their real-world popularity. Also, using such tasks and languages eases an apples-to-apples comparison with existing works. However, our framework can be easily extended to other scenarios, including different programming languages and many CSA tasks (see our discussion on extensibility in Sec. 7 for more details). Having said that, we note that PP-CSA is not suitable for CSA tasks that require the analysis of individual properties. For example, if the goal is to identify call stacks associated with uncommonly-occurred bugs [Ko and Myers 2008], which requires precisely pinpointing those rare yet buggy call stacks, then PP-CSA is not applicable. As discussed in Sec. 2.1, the fundamental principle of differential privacy is to impede adversarial inferences regarding individual properties, while still providing accurate estimations of group statistics.

4.1 PP-CSA: Offline Analysis

PP-CSA starts by extracting the call graph of the software under analysis. The call graph is a directed graph that represents the function call relations between functions in the software. Each node in the call graph represents a function and each edge represents a calling relation between two functions. Formally, the call graph is defined as follows.

Definition 4 (Call Graph). *A call graph G is a directed graph $G = (F, E)$, where F is the set of nodes (representing all functions) and E is the set of edges (calling relationship between two functions). We denote the set of entry functions of the call graph as $F_{entry} \subseteq F$.*

Given that the current prototype of PP-CSA is implemented for Java and Android programs, we use Soot [OSS 2023] to use Class Hierarchy Analysis (CHA) algorithm and soundly build the call graph.

As already noted in Sec. 2.2, our focused task employs call stacks to encode the program runtime information and facilitate identifying hot call stacks. Formally, a call stack is defined as follows.

Definition 5 (Call Stack). *A call stack s is a stack data structure $s = (f_1, f_2, \ldots, f_n)$, where each f_i is a function involved during the current execution, $f_1 \in F_{entry}$ is an entry function of the program, acting as the starting point for the current execution, and (f_i, f_{i+1}) denotes that function f_i calls f_{i+1} during this execution.*

Shrinking Optimizations. One challenge of CSA is that call stacks can be very long, which is particularly undesirable for privacy-preserving analysis. In general, the longer the call stack, the more challenging it becomes to safeguard its privacy while preserving utility, and also imposes more difficulty in training encoder/decoder. To address this challenge, we propose two shrinking rules to reduce the size of call stacks as follows.

$\mathcal{R}_1$ *Pre-determinable Function Compression.* Intuitively, there may exist redundancy in the call stacks. For example, if function f_i can only be called by function f_j, then the record (f_j, f_i) in a call stack is redundant as it can be deduced from a single record (f_i) (as f_j is the only legitimate caller of f_i). Overall, if there exists exactly one path between two functions in the call graph, then their intermediate functions are redundant on the trace. Therefore, we identify these *pre-determinable* function pairs and compress the call stacks by removing their intermediate functions.

$\mathcal{R}_2$ *Lengthy Stack Cutoff.* Our preliminary study reveals the "long-tail" phenomenon in the call stacks of a program: often, when profiling a real-world Android program with its common workload, a small number of call stacks are exceptionally lengthy. For example, the average call stack length for Eclipse (one of the Java benchmarks used in evaluation) is 6.32, while the longest call stack contains 24 calls. As one can expect, lengthy call stacks are particularly challenging for privacy-preserving analysis given the exponential growth of the domain of all possible call stacks, making encoder/decoder training generally harder. On the other hand, we note that, when considering the specific task of identifying hot stacks, those long and rare call stacks are *unlikely* to be "hot."

We propose to prune those long call stacks to improve the overall performance of PP-CSA. Formally, we define the pruning strategy as follows. Let h be the user-specified frequency of hot call stacks and let $\phi(l)$ be the cumulative distribution function of the call stack length. Then, we define a pruning threshold T, as the inverse of the CDF function: $T = \phi^{-1}(1-h)$, where ϕ^{-1} is the inverse function of the cumulative distribution function $\phi(l)$. This threshold ensures that a fraction h of the hottest call stacks (i.e., those with lengths below T) are retained, while the exceptionally lengthy and rare call stacks above T are trimmed off.

Clarification. It is evident that both $\mathcal{R}_1$ and $\mathcal{R}_2$ are used in the online phase (the "shrink" operator in Figure 2) when the client decides to send a call stack to the server. Nevertheless, we present them here as the information required to apply these rules is obtained in the offline phase in advance. Also, to ease presentation, we leave presenting encoder/decoder design and training in Sec. 4.2, although they are also performed in the offline phase.

4.2 Online Call Trace Protection and Analysis

Sec. 4.1 describes a one-off preprocessing step to establish the call graph and also decide which call stacks to prune or directly discard. Those preparation steps are performed on the server side and can be done in a one-off manner. In line with Figure 2, this section presents the online activities of PP-CSA, which consist of call stacks encoding and decoding, LDP perturbation, and call stack reconstruction and analysis.

4.2.1 Encoder/Decoder Design and LDP Enforcement. At this step, we seek to encode the call stacks into numerical representations to enable the enforcement of LDP. Often, the baseline approach is to use one-hot encoding or hash functions [Hao et al. 2021], and thus converting the call stacks into a fixed length bit vector. Despite the simplicity, this approach is undesirable given that call stacks are with orders by nature and are often lengthy and complex. The resulting encoding would be very sparse and would not be able to achieve high utility.

At this step, we propose to protect call stacks using a sequence-to-sequence encoder-decoder scheme. Overall, an encoder model on the client side converts a call stack into a sequence of numerical vectors where LDP is later enforced. When the server receives the encoded call stack, it employs a decoder model to decode the LDP-perturbed sequence of numerical vectors into a call stack. Below, we describe the design of the encoder and decoder in detail.

Architecture Design. The encoder-decoder model plays a pivot role in PP-CSA. Given a call stack $S = (f_1, f_2, \ldots, f_n)$, after being shrunk by the shrinking rules (as shown in Figure 2), the encoder Enc on the client side computes a latent numerical vector representation z for the call stack, to which the LDP noise is later added (see details below). This noise-added representation $\hat{z}$ is then passed to the server side. Subsequently, $\hat{z}$ together with auxiliary reachability features (see Sec. 4.3) is fed to the decoder Dec, which reconstructs the call stack $\hat{s}$ on the server side.

In essence, we employ an LSTM-based encoder-decoder model [Hochreiter and Schmidhuber 1997], where both encoder and decoders are uni-directional LSTM models. The encoder takes the call stack as input and outputs a context vector. Then, this numerical vector is clipped by norm and noise is added to the clipped vector. To enhance the decoder's accuracy, we accompany the decoder with several optimizations, as will be discussed in Sec. 4.3.

Objective Function. The objective function of the encoder-decoder model is defined as follows:

$$Loss = -\frac{1}{L}\sum_{i=1}^{L} y_i log(p_i) \tag{2}$$

where L is the length of the call stack, y_i is the i-th function name in the call stack, and p_i is the probability of the i-th function name predicted by the decoder. All call stacks are padded to the same length L with a special token. Note that L, representing the maximum length of the call stack handled by PP-CSA, is decided by $\mathcal{R}_2$ in Sec. 4.1.

Clipping by Norm. As stated in Sec. 2.1, the sensitivity of the function is a crucial parameter in determining the appropriate scale of noise to be added. However, the sensitivity of the encoder function Enc can be potentially infinite; because the encoder's output is unbounded, leading to unbounded changes in its output. As a common tactic to address this challenge, we clip the output of the encoder function Enc, namely a latent vector z, by its norm and the clipping constant $C \in \mathbb{R}$. The clipping by norm operation is defined as follows:

$$z' = z \cdot \min\left(1, \frac{C}{||z||_2}\right) \tag{3}$$

This way, the sensitivity of the encoder function Enc is bounded by its norm and the clipping constant, and the Gaussian mechanism can be applied.

LDP Enforcement. With this clipping mechanism in place, we can now calculate its sensitivity, in order to determine the scale of noise to add in the LDP setting. This is outlined in Theorem 2.

Theorem 2. *Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a function as defined in Eqn. 3. The ℓ_2 sensitivity $\Delta_2 f$ of this function is $2C$ [Krishna et al. 2021], where $C \in \mathbb{R} : C > 0$ is the clipping constant and $n \in \mathbb{N}$ is the dimensionality of the vector.*

The noise is then added to this clipped vector, as in Eqn. 4.

$$\hat{z} = z' + \eta \tag{4}$$

where z' is the clipped vector and η is the noise vector with the same dimensionality. Each noise element, denoted as η_i, is drawn i.i.d. from the Gaussian distribution of $\mathcal{N}(0, 2\ln(\frac{1.25}{\delta})\frac{\Delta_2^2}{\epsilon^2})$ according to Theorem 1. In sum, we present the whole process occurred on the client side in Alg. 1.

Algorithm 1: PP-CSA: client-side

```
Input: call stack $S = (f_1, f_2, \dots, f_n)$, privacy budget $\epsilon$, max sequence length $L$, clipping
       constant $C$, threshold $T$
Output: encoded call stack $S_{\text{enc}}$
1 if $\mathcal{R}_2(S, T)$ then
2     return $\perp$ ;                        // $\mathcal{R}_2$ (Sec. 4.1): if too long, directly cutoff
3 else
4     $S_{\text{compress}} = \mathcal{R}_1(S)$;              // $\mathcal{R}_1$ (Sec. 4.1): compress call stack
5     $z = \text{Enc}(S_{\text{compress}})$;                    // encode the call stack
6     $z' = \text{clip}(z, C)$;                              // clip the vector by norm
7     $S_{\text{enc}} = z' + \text{noise}$;                                   // add noise
8     return $S_{enc}$;
9 end
```

Following the standard procedure [Dwork et al. 2014], we note that the encoder part of the PP-CSA satisfies (ϵ, δ)-differential privacy for the Gaussian mechanism. Subsequently, the encoder output is transmitted to the decoder for decoding, where the reconstruction of call stacks takes place. Notably, the entire PP-CSA ensures differential privacy due to its robustness under composition, a property known as "post-processing invariance [Dwork 2006]." This essential property guarantees that even though the encoder output is used as input for further computation or analysis, the overall privacy guarantees remain intact, safeguarding the privacy of individuals in the dataset.

Proposition 1. *Alg. 1 satisfies (ϵ, δ)-differential privacy.*

4.3 Server-Side Optimizations

Our tentative study shows that the vanilla decoder design, although already achieving encouraging results, may still be inaccurate due to distinct characteristics between program call traces and other common sequence data, such as natural language (see empirical comparison in Sec. 6.2). Enforcing the differential privacy over program call traces which have strict semantics is more challenging. In this section, we describe three optimizations (O_1, O_2, and O_3) in the decoder phase which improve the overall performance without introducing much additional cost. In particular, O_1 and O_2 are designed to improve the adherence to caller-callee relationships, while O_3 aims at increasing the awareness of epistemic uncertainty, as discussed in Sec. 3.3. We clarify that these optimizations can be applied together or separately, depending on the specific scenario. Nevertheless, our evaluations show that the combination of these optimizations can achieve the synergistic effect and yield the best performance (see Sec. 6).

O_1 **Reachability-enhanced Training.** Enlightened by the algebraic property of graph adjacency matrix [De la Cruz Cabrera et al. 2019], we propose to incorporate the reachability information of the call graph into the underlying encoder-decoder model. This optimization aims to enhance the model's ability to capture nuances of the program's reachability dynamics, which illustrates how an individual function call triggers subsequent ones during the execution flow. This ability allows the model to better adhere to the caller-callee relationship, as introduced in Sec. 3.3. To quantify reachability, we employ a reachability matrix $R \in \mathbb{R}^{m \times m}$, derived from the matrix exponential representation of an input adjacency matrix A. The matrix exponential — denoted as $\exp(A)$ — is a mathematical operation that extends the concept of exponentiation to square matrices. Precisely, for any assigned matrix A, its corresponding matrix exponential $\exp(A)$ can be devised utilizing its expansion in terms of the Taylor series:

$$R = \exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!} \tag{5}$$

From an intuitive perspective, R_{ij} signifies the connectivity from function f_i to function f_j within the call graph. Considering any particular function f_i, the associated reachability-informed feature is characterized by the respective row vector residing in the reachability matrix, as defined below:

$$r_i = R_{i,\cdot} \tag{6}$$

By integrating the row vector as an auxiliary feature, the model can obtain insight about the reachability of the current function, thereby facilitating its understanding of likely subsequent function calls. The optimization effectively improves the accuracy of the decoding process, as shown in Sec. 6.

O_2 **Call Graph-Guided Decoding.** In the common usage of decoder, the decoder outputs the sequence with the highest likelihood. However, a standard decoding scheme for natural language offers no guarantee that the decoded sequence is a valid call stack sequence (i.e., ensuring adherence to the caller-callee relationship, as noted in Sec. 3.3). As an illustration, given a call graph $G = (V, E)$, if the decoder produces a call stack $(f_1, f_2, \dots, f_n)$ in which $(f_i, f_{i+1}) \notin E$, it implies that the decoder-yielded call stack is not legitimate. To tackle this hurdle, we propose to incorporate the call graph information into the decoding process. In particular, the optimization employs the call graph as an enforced constraint during the decoding phase. Specifically, we only allow the decoder to output a call stack that satisfies the call graph constraints. This way, we ensure that the decoded call stack is valid and realistic to a great extent.

O_3 **Probabilistic Distributional Decoding.** As discussed in Sec. 3.3, for conventional generative models, the decoder aims to produce a "point estimation" of the most likely result [Krishna et al. 2021]. However, in the context of our specific scenario — where differential privacy mechanisms are enforced — an inherent epistemic uncertainty is introduced. As a result, the decoded call stack, as a maximum likelihood estimation, might not faithfully reflect the true call stack. To circumvent this issue, we adopt a "probabilistic distributional decoding" scheme to enable a more holistic estimation of the call stack distribution. Specifically, in the decoder, we sample a collection of outcomes along with their associated probabilities using beam search and reconstruct the empirical distribution accordingly. Subsequently, by aggregating these distribution estimations from a variety of clients, we can approximate the global call stack distribution, thereby catering to the needs of real-world applications. This aggregation process contributes to more robust and reliable insights into the comprehensive estimation of function call stacks.

In sum, we outline the decoding procedure occurred on the server side in Alg. 2. The decoder receives the encoded call stack S_{enc} from the encoder as input and produces a set of decoded call stacks $\mathcal{S}_{\text{dec}}$ along with their associated probabilities. The decoding process is implemented using beam search, a widely used technique for sequence generation tasks. Specifically, we maintain a priority queue B to store potential call stacks during the beam search (line 2). The priority of each call stack is determined by its probability score, guiding the exploration towards more promising sequences.

We initialize the priority queue with the start token, representing the initial state (line 1). The decoder then iteratively processes the current function and its corresponding hidden state, along with the reachability feature, to calculate the probability distribution of the next function (Optimization O_1, line 9). Next, we expand the call stack by considering all possible next functions reachable from the current function, as dictated by the call graph (Optimization O_2, lines 10-14). The number of the valid finished call stacks reaches k, indicating that we have found the top-k most probable call stacks. Finally, we normalize the probability of each call stack using the softmax, providing a probabilistic distributional estimation of the input call stack (Optimization O_3, lines 16-21). We omit finally computing "hot trace" given it is a straightforward process and orthogonal to Alg. 2.

5 IMPLEMENTATION AND EVALUATION SETUP

PP-CSA is implemented in Python and Java, with a total of 5,280 lines of code. The Python code is used to implement the encoder and decoder, and the Java code is used to employ the Soot framework to generate the Java datasets from the *DaCapo* benchmark (version 9.10.1). PyTorch (version 2.0.1) is used to implement the encoder/decoder. We have released a snapshot of our codebase at [artifact 2023] for results reproducibility. We will maintain them and provide further documents to ease future research comparison and extension.

```
Algorithm 2: PP-CSA: Server-side Decoding
  Input: encoded call stack $S_{\text{enc}}$, call graph $G = (V, E)$, reachability matrix
         $R = (r_1{}^T, r_2{}^T, \ldots, r_m{}^T)^T$, search beam size $B$, max output size $k$
  Output: a set of call stack and associated probability $\mathcal{S}_{\text{dec}} = \{(S_1, p_1), (S_2, p_2), \ldots, (S_n, p_n)\}$
1  Initialize $\mathcal{S}_{\text{dec}} = \emptyset$, priority queue $B = (\texttt{start}, 0)$, finished beam set $B_{\text{finished}} = \emptyset$, decoder
     hidden state $h = S_{\text{enc}}$;
2  while $|B_{finished}| < k$ do
3      $S_{\text{curr}}, p_{\text{curr}} = B.\text{pop}()$ ;                // get the most probable call stack
4      $f_{\text{curr}} = \text{last}(S)$;
5      if $f_{curr}$ == end then
6          $B_{\text{finished}} = B_{\text{finished}} \cup (S, p)$;
7          continue;
8      end
9      $h, \mathbf{p} = \text{Dec}(h, f_{\text{curr}}, r_{\text{curr}})$ ;   // $O_1$: use reachability info
10     for $(f_{curr}, f_{next}) \in E$ do                 // $O_2$: call graph-guided decoding
11         $p_{\text{next}} = \mathbf{p}(f_{\text{next}} \mid S) \times p_{\text{curr}}$;
12         $S_{\text{next}} = S \oplus f_{\text{next}}$;
13         Update $B$ with $(S_{\text{next}}, p_{\text{next}})$ and keep top-$B$ elements;
14     end
15 end
16 $B_{\text{top-k}} \leftarrow \textit{top-k}(B)$          // $O_3$: prob. distributional estimation
17 for $(S, p) \in B_{top\text{-}k}$ do
18     $p_{\text{normalized}} \leftarrow \text{softmax}(p)$;
19     $\mathcal{S}_{\text{dec}} = \mathcal{S}_{\text{dec}} \cup (S, p_{\text{normalized}})$;
20 end
21 return $\mathcal{S}_{dec}$
```

5.1 Hyper-parameters

To clarify, we do not heavily tune the hyper-parameters in our implementation, as we aim to show the general feasibility of PP-CSA in real-world scenarios. For the encoder/decoder, we use a single-layer LSTM with 128 hidden units. The embedding size is 300, batch size is 64, and the learning rate is 0.01. The maximum number of epochs is 10. The sample size in the decoding phase is 3. For LDP, we use the Gaussian mechanism with a clipping norm of 5. As a rule-of-thumb [Ponomareva et al. 2023], we use δ as the inverse of the data size in our experiments. We use a threshold $T_{hot} = 0.01$ to identify hot traces (see empirical impacts on this threshold in Sec. 7). Overall, these hyper-parameters are common choices, and with these settings, our evaluations already report promising results. Users may further tune the hyper-parameters to achieve better results if needed.

5.2 Evaluation Setup

Baselines. We compare PP-CSA with the the state-of-the-art tool [Hao et al. 2021], which is the only existing tool that can perform CSA with privacy guarantees. To ease presentation, we refer this tool as SOTA in the rest of the paper. From another perspective, we also compare PP-CSA with DP-BART [Igamberdiev and Habernal 2023], the state-of-the-art DP-based text rewriting system.

Table 1. Statistics of our datasets, stacks and call graphs.

Dataset Name	#Programs	Average #Functions	Average #Stacks	Average Stack Length
Android	15	3,394	4,195	4.03
DaCapo	12	1,866	4,342	4.11

Dataset Name	Average Max Stack Length	Average Degrees per Node	Average #Edges	Average #Entry point
Android	13.07	16.31	178966.6	403
DaCapo	16.58	8.72	79630.5	NA

We show that PP-CSA, with its tailored optimizations, can outperform DP-BART in terms of both accuracy and efficiency. We use both the official release of and SOTA and DP-BART for evaluation. All experiments are run on a machine with Geforce RTX 3090, Intel Core i7-8700 CPU and 32GB RAM.

Datasets. The current PP-CSA is implemented primarily for Java and Android programs. To deliver a fair comparison with SOTA, we reuse the Android apps used in SOTA for evaluation. This dataset, referred to as *Android*, contains 15 real-world Android apps with various sizes and functionalities like games, location tracking, news, music, and so on. It has been analyzed in [Hao et al. 2021] that these real-world Android apps suffer from considerable privacy risks when collecting call stack traces. We also use a common Java dataset, DaCapo Benchmark [Blackburn et al. 2006; University 2021], which contains a set of open-source, real-world Java programs. This dataset, referred to as *DaCapo*, mimics the real-world Java programs with varying complexity, and is widely used in the literature [Bond and McKinley 2005; Thiessen and Lhoták 2017]. We summarize the statistics of our evaluated datasets, including the statistics of the call stacks and call graphs, in Table 1. Note that the average number of entry points in *DaCapo* is not applicable (NA), since these programs are normal Java programs with the main method as a sole entry point. Therefore, we only report this statistic for the *Android* dataset. Below, we introduce the details on call stack collection.

Call Stack Collection. For the Java programs in *DaCapo*, we randomly simulate the execution over its call graph starting from the program entry point and collect call stacks for each program. To do so, we initiate the simulation process from the entry point function on the call graph of a Java program in *DaCapo*. We then conduct a "random walk" on the call graph, where each time we randomly pick a callee function to proceed until there are no more functions that can be called from the current function (i.e., we arrive at a leaf node on the call graph). This forms one chain of function calls, and we randomly choose a prefix on this call chain to form a call stack and include it in our dataset. We then repeat the process with different random seeds, and de-duplicate the collected call stacks until the de-duplicated average call stacks of each program in *DaCapo* approximate that of the *Android* dataset. On average, each program in the *DaCapo* dataset has 4,195 de-duplicated call stacks gathered, as shown in Table 1. This number is close to the 4,342 de-duplicated call stacks collected on average for each program in the *Android* dataset.

As for the Android apps, we directly reuse the call stacks shipped in this dataset. It is disclosed that these call stacks are collected using the Monkey tool [Google 2020] to mimic 1000 users' interactions with those 15 apps in the *Android* dataset. We sampled 100K call stacks for each app from the *Android* dataset, and the statistics of these call stacks are reported in Table 1. Overall, a considerable number of call stacks are collected for each program, and we show that call stacks with comparable sizes are collected and used for evaluation. Note that the SOTA work [Hao et al. 2021] only evaluates the *Android* dataset. We randomly picked 90% of the call stacks for training

Table 2. RQ1: evaluating PP-CSA using *DaCapo* under different privacy budgets. We highlight the best F1 scores for each setting **in bold**.

ϵ	SOTA			DP-BART			PP-CSA		
	P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
100	0.779	0.939	0.814	0.701	0.677	0.680	0.992	0.992	**0.992**
40	0.690	0.873	0.718	0.681	0.592	0.620	0.959	0.986	**0.970**
30	0.659	0.833	0.675	0.667	0.541	0.572	0.910	0.973	**0.931**
20	0.612	0.791	0.642	0.649	0.477	0.508	0.858	0.931	**0.879**
10	0.516	0.832	0.558	0.482	0.376	0.403	0.894	0.732	**0.778**
1	0.357	0.595	0.328	0.021	0.029	0.025	0.786	0.346	**0.348**
Average	0.602	0.811	0.623	0.533	0.449	0.468	0.901	0.827	**0.816**

Table 3. RQ1: evaluating PP-CSA using *Android* under different privacy budgets. We highlight the best F1 scores for each setting **in bold**.

ϵ	SOTA			DP-BART			PP-CSA		
	P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
100	0.578	0.619	0.589	0.894	0.483	0.610	1.000	1.000	**1.000**
40	0.531	0.618	0.553	0.602	0.211	0.264	1.000	1.000	**1.000**
30	0.496	0.561	0.512	0.317	0.154	0.169	0.997	0.977	**0.987**
20	0.491	0.586	0.514	0.207	0.141	0.148	0.985	0.790	**0.874**
10	0.468	0.557	0.497	0.161	0.136	0.138	0.992	0.504	**0.671**
1	0.340	0.395	**0.357**	0.133	0.133	0.133	1.000	0.138	0.238
Average	0.484	0.556	0.503	0.386	0.210	0.244	0.995	0.734	**0.795**

and the remaining 10% for testing. We do not particularly tune the training hyper-parameters, as noted above in this section.

Metrics. Aligned with [Habernal 2021], we use the precision, recall, and F1 scores to assess the performance of PP-CSA and the baseline methods. To clarify, "precision" denotes what portion of the reported hot traces are actually hot, whereas the "recall" denotes what portion of the true hot traces are discovered. The F1 score is the harmonic mean of precision and recall. We also report the running time of PP-CSA and the baseline methods. In RQ3, following the setup in [Hua et al. 2022], we measure the privacy benefit of PP-CSA through the lens of *adversarial uncertainty*. Intuitively, *adversarial uncertainty* indicates the accuracy of an adversary in inferring the original call stack from the released data (i.e., the noisy encoding). In Sec. 6.3, we provide the formal definition of *adversarial uncertainty* in our context.

6 EVALUATION

We have introduced the experimental setup in Sec. 5.2. Our evaluation mainly focuses on three key research questions (RQs):

- **RQ1** what are the performances of PP-CSA under different configurations?
- **RQ2** what are the contributions of each optimization of PP-CSA?
- **RQ3** what are the privacy benefits of PP-CSA?

6.1 RQ1: End-to-End Evaluation

We report the end-to-end evaluation results of PP-CSA in terms of both Java and Android datasets in Table 2 and Table 3, respectively. As aforementioned, we measure the performance of PP-CSA in terms of precision, recall, and the F1 score. Note that data in Table 2 are the average of all Java programs in the DaCapo bench dataset, so are the data in Table 3 for Android apps in the *Drebin* dataset; we discuss how PP-CSA performs on individual traces later in Table 4. It is seen that PP-CSA achieves a highly competitive F1 score across different privacy budgets. PP-CSA notably outperforms two baseline works, **SOTA** and **DP-BART**, in terms of F1 score by 26.3% and 77.4% on average for the Java dataset, and by 43.7% and 206.1% on average for the Android dataset.

We also observe that PP-CSA achieves a (nearly) perfect F1 score when the privacy budget ϵ is set to 40 and 100. Note that while $\epsilon = 100$ can be deemed as a "lenient" privacy budget that may not be commonly adopted in practice, $\epsilon = 40$ is realistic. In comparison, the F1 scores of **SOTA** and **DP-BART** under this setting manifest a considerable space for improvement.

It is seen that with a gradually increasing privacy budget ϵ, PP-CSA achieves higher accuracy. This is expected, because a larger privacy budget ϵ allows PP-CSA to release more information from the input trace, thus enabling the server side to more accurately reconstruct the original trace. Note that on an edge case, when $\epsilon = 1$, PP-CSA achieves a relatively low F1 score in comparison to **SOTA**. Nevertheless, we argue that $\epsilon = 1$ is generally unrealistic in practice, and the low F1 scores are presumably because the noise added by the DP mechanism is too large and the encoding may vanish. Overall, this is a too-strict setting, and all the three approaches achieve low F1 scores under this setting. Overall, we interpret the results reported in Table 2 and Table 3 as highly encouraging, showing the effectiveness of PP-CSA in terms of accuracy. We recommend to set the privacy budget ϵ to 20 or 40, which are realistic and achieve a sufficiently high F1 score. We leave further discussion on the security implication of ϵ in **RQ3**.

Effect of Input Stack Length on PP-CSA Performance. We also measure the performance of PP-CSA in terms of the length of the input stack, whose results are in Table 4. To present a comprehensive study, we use the Eclipse dataset from *DaCapo*, given that it contains traces with varying lengths. In particular, given nearly all traces in this dataset are shorter than 20, we divide the traces into three categories: short (lengthy ≤ 5), medium ($5 <$ length ≤ 10), and long (length $>$ 10).

Table 4. Precision, Recall, F1 score in terms of three kinds of trace lengths.

Length	**Precision**	**Recall**	**F1 score**
long	0.492	0.601	0.541
medium	0.929	0.712	0.806
short	0.966	0.966	0.966

Under a reasonable budget $\epsilon = 30$, we observe that shorter traces lead to the highest F1 score of 0.966, indicating the algorithm's high proficiency in protecting them. The F1 score decreases to 0.806 for medium-length traces, and experiences a further drop to 0.541 for longer traces. Overall, we interpret the results in Table 4 as reasonable, given that longer traces (similar to lengthy natural language text) contain more information and are inherently more difficult to protect under DP. On the other hand, we note that traces longer than 10 are rare in practice, which means that PP-CSA is generally effective in protecting traces in practice. We leave it as future work to shrink the length of the input trace (see discussions in Sec. 7) and improve the accuracy of PP-CSA.

Applicability of PP-CSA across Different Test Data. Furthermore, we evaluate the applicability of PP-CSA across different test data. To do so, we conduct an evaluation of its effectiveness when confronted with call stacks that exhibit different distributions in comparison to the training data. Specifically, we focus on analyzing the top 5% longest call stacks. We ensure that all the test data fall within this category while maintaining the normality of the training data, thus creating an out-of-distribution scenario.

The results of this evaluation are presented in Table 5, where the red color highlights the percentage of performance change compared to the performance achieved under the same distribution settings. For instance, when the ϵ is 40, the precision on *Android* dataset is decreased from 1 to 0.722 ($\frac{0.722-1}{1} \times 100\% = -28\%$). Overall, we observe that PP-CSA exhibits a decrease in performance in the out-of-distribution setting, particularly in the low epsilon regime. This suggests that the model is more likely to misidentify these abnormal call stacks as common call stacks, which is reflected in the low precision values and high recall values. Our current design does not specifically tailor PP-CSA to take into account the out-of-distribution scenarios; this indicates a potential future direction for improvement.

Table 5. Resilience of PP-CSA to different test data for *DaCapo* and *Android*.

ϵ	*DaCapo*						*Android*					
	P		**R**		**F1**		**P**		**R**		**F1**	
100	0.964	-3%	0.992	-0%	0.975	-2%	1.000	-0%	1.000	-0%	1.000	-0%
40	0.755	-21%	0.978	-0.8%	0.840	-13%	0.722	-28%	0.985	-1%	0.815	-18%
30	0.542	-40%	0.971	-0.2%	0.666	-28%	0.469	-53%	0.972	-0.4%	0.601	-39%
20	0.258	-70%	0.813	-12%	0.372	-57%	0.244	-75%	0.895	+13%	0.341	-61%
10	0.171	-81%	0.571	-27%	0.248	-68%	0.169	-83%	0.479	-5%	0.201	-70%

Study the Influence of Entry Points on PP-CSA. We also study how certain characteristics of the benchmarks, specifically the number of entry points, can influence the effectiveness of PP-CSA. We report the results in Figure 3. In short, we do not observe a significant correlation between the number of entry points and the effectiveness of PP-CSA. This observation may be attributed to the proper usage of a virtual entry point during the decoding phase when handling multiple entry points. This virtual entry point serves as the starting point for the decoding process and is connected to all real entry points in the call graph through edges. Consequently, a program with multiple entry points can be treated as having a single entry point. Thus, the number of entry points does not have a significant influence on the overall effectiveness of PP-CSA.

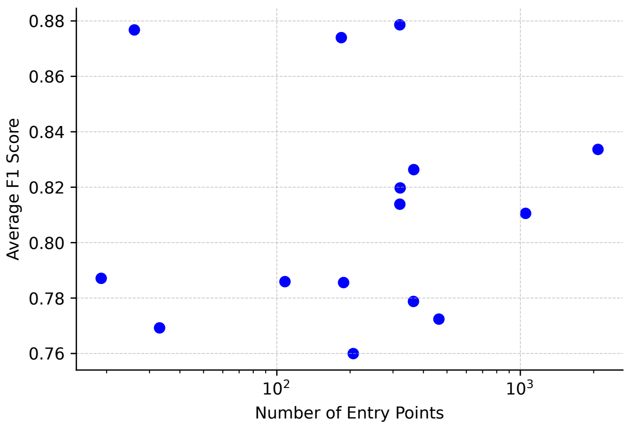

Fig. 3. Performance of PP-CSA under different number of entry points.

Processing Time. We measure and report the average running time of PP-CSA in processing one trace is about 3.84×10^{-3} seconds. In comparison, the average running time of **SOTA** and DP-BART are 6.37×10^{-3} and 2.17×10^{-2} seconds, respectively. We thus interpret the overall processing time

Table 6. RQ2: evaluating the contribution of different optimizations using *DaCapo*. We highlight the best F1 scores for each setting **in bold**.

		100	**40**	**30**	**20**	**10**	**1**	**Average**
	P	0.975	0.966	0.872	0.782	0.749	0.646	0.855
Base	**R**	0.975	0.912	0.930	0.825	0.502	0.118	0.710
	F1	0.974	0.938	0.889	0.787	0.571	0.180	0.723
	P	0.989	0.934	0.901	0.862	0.858	0.796	0.890
$\mathcal{R}_1$	**R**	0.992	0.961	0.942	0.883	0.721	0.238	0.790
	F1	0.991	0.942	0.912	0.863	0.766	0.318	0.798
	P	0.990	0.925	0.871	0.853	0.858	0.735	0.872
$\mathcal{R}_2$	**R**	1.000	0.965	0.927	0.789	0.598	0.229	0.752
	F1	0.995	0.939	0.891	0.810	0.675	0.296	0.768
	P	0.992	0.937	0.902	0.855	0.873	0.713	0.903
Shrinking ($\mathcal{R}_1 + \mathcal{R}_2$)	**R**	0.992	0.972	0.954	0.913	0.721	0.311	0.814
	F1	0.992	0.949	0.919	0.876	0.772	0.324	0.820
	P	1.000	0.946	0.892	0.859	0.783	0.742	0.888
O_1	**R**	1.000	0.971	0.928	0.848	0.637	0.287	0.787
	F1	**1.000**	0.954	0.904	0.847	0.683	0.307	0.791
	P	0.992	0.909	0.896	0.815	0.855	0.759	0.907
O_2	**R**	1.000	0.969	0.954	0.863	0.615	0.295	0.784
	F1	0.996	0.942	0.914	0.825	0.683	0.301	0.796
	P	0.991	0.922	0.886	0.814	0.726	0.685	0.838
O_3	**R**	0.995	0.936	0.932	0.843	0.662	0.291	0.784
	F1	0.993	0.922	0.898	0.813	0.672	0.301	0.783
	P	0.992	0.959	0.910	0.858	0.894	0.786	0.925
All	**R**	0.992	0.986	0.973	0.931	0.732	0.346	0.824
	F1	0.992	**0.970**	**0.931**	**0.879**	**0.778**	**0.348**	**0.830**

of PP-CSA as encouraging, given that it achieves high accuracy with only moderate overhead in terms of processing time.

Answer to RQ1: PP-CSA manifests highly encouraging performance in terms of accuracy and efficiency. Moreover, we recommend using PP-CSA with a privacy budget $\epsilon \in [20, 40]$, which shall achieve a highly competitive accuracy with moderate cost.

6.2 RQ2: Optimizations of PP-CSA

In Sec. 4.1, we propose two client-side shrinking rules to reduce the size of call stacks and, in Sec. 4.3, we propose three optimization strategies on the server side. This section evaluates the effectiveness of these rules and optimizations in improving the accuracy of PP-CSA. Note that those optimizations do not affect PP-CSA's privacy guarantee, which is decided by the privacy budget ϵ. Nevertheless, we clarify that, given the fundamental trade-off between accuracy and privacy, with a fixed accuracy, PP-CSA shall become safer with these optimizations. See our security evaluations in **RQ3**.

At this step, we consider different client-side and server-side optimizations (and their combinations). For each setting, we evaluate the performance of PP-CSA on different privacy budgets, using the entire dataset of *DaCapo* and the *Android*, respectively.

Overall, we can observe that the performance of PP-CSA is generally improved when either client/server-side optimizations are used. Moreover, we observe an encouraging "synergistic effect" such that the accuracy of PP-CSA is further improved when all optimizations are used together

Table 7. RQ2: evaluating the contribution of different optimizations using *Android*. We highlight the best F1 scores for each setting **in bold**.

		100	**40**	**30**	**20**	**10**	**1**	**Average**
Base	**P**	1.000	1.000	0.998	0.952	0.846	0.978	0.960
	R	1.000	0.985	0.875	0.546	0.360	0.116	0.646
	F1	1.000	0.993	0.931	0.675	0.470	0.201	0.712
$\mathcal{R}_1$	**P**	1.000	1.000	1.000	0.975	0.970	0.986	0.988
	R	1.000	1.000	0.971	0.765	0.488	0.140	0.727
	F1	1.000	1.000	0.972	0.855	0.619	0.241	0.781
$\mathcal{R}_2$	**P**	1.000	1.000	1.000	0.970	0.970	0.985	0.987
	R	1.000	1.000	0.903	0.554	0.350	0.124	0.655
	F1	1.000	1.000	0.948	0.690	0.494	0.219	0.725
Shrinking ($\mathcal{R}_1 + \mathcal{R}_2$)	**P**	1.000	1.000	0.995	0.978	0.962	1.000	0.990
	R	1.000	1.000	0.962	0.779	0.465	0.141	0.722
	F1	1.000	1.000	0.978	0.865	0.622	**0.243**	0.785
O_1	**P**	0.999	0.988	0.981	0.955	0.891	1.000	0.969
	R	0.999	0.992	0.826	0.547	0.338	0.138	0.640
	F1	1.000	0.990	0.895	0.686	0.462	0.218	0.708
O_2	**P**	1.000	1.000	0.995	0.974	0.970	1.000	0.990
	R	1.000	0.985	0.919	0.550	0.357	0.134	0.658
	F1	1.000	0.993	0.954	0.687	0.488	0.232	0.726
O_3	**P**	1.000	0.983	0.998	0.976	0.911	1.000	0.981
	R	1.000	0.991	0.882	0.634	0.461	0.141	0.685
	F1	1.000	1.000	0.935	0.762	0.610	0.243	0.758
All	**P**	1.000	1.000	0.997	0.985	0.992	1.000	0.995
	R	1.000	1.000	0.977	0.790	0.504	0.138	0.736
	F1	1.000	**1.000**	**0.987**	**0.874**	**0.671**	0.238	**0.795**

(the "All" row). PP-CSA achieves the best performance in five out of the six settings in Table 6. The remaining setting is the "extreme" case where the privacy budget is extremely large ($\epsilon = 100$). In practice, we do not expect to use such an extreme privacy budget. When comparing to the "Base" setting, we observe that the client-side shrinking rules $\mathcal{R}_1$ — "Pre-determinable Function Compression" — appears to be the most effective one. This is expected, as this rule "compresses" the call stacks by removing the function calls that are pre-determinable, making it easier for the encoder/decoder to identify common patterns in call stacks. This facilitates utility without compromising privacy. Moreover, the other four client/server optimizations also manifest very close and encouraging effectiveness. For instance, we observe that the O_2, call graph-guided decoding, is also highly useful. O_2 effectively "regulates" the decoding process by pruning illegal call stacks that are not in the call graph, contributing to the accuracy improvement.

Answer to RQ2: All optimizations are effective in improving the accuracy of PP-CSA. Moreover, we observe encouraging "synergistic effect" such that the accuracy of PP-CSA is further improved when all server and client optimizations are used together in common privacy budgets. We recommend enabling all optimizations whenever possible in practice.

6.3 RQ3: Privacy Benefit

Sec. 3.2 provides a conceptual discussion on the privacy guarantee of PP-CSA. In essence, the privacy protection offered by PP-CSA aims to impede any potential inferences made by attackers

Table 8. RQ3: Adversarial uncertainty (lower is better).

ϵ	***AU@Acc***	***AU@Prec***	***AU@Rec***	***AU@F1***
100	0.011	0.103	0.106	0.110
40	0.239	0.689	0.693	0.718
30	0.464	0.862	0.880	0.891
20	0.739	0.981	0.980	0.982
10	1.008	0.997	0.998	0.998
1	1.063	1.000	1.000	1.000

regarding individual properties. In this section, we aim to quantitatively evaluate the privacy benefit of PP-CSA. To this end, we first formally instantiate the concept of *adversarial uncertainty* in the context of call stack analysis. Specifically, we consider the following adversarial inference problem: Given the released (noisy) encoding S_{enc} of a call stack S, the goal of the adversary $\mathcal{A}$ is to infer the original call stack S as accurately as possible. In this regard, from a prediction perspective, the adversary $\mathcal{A}$ can be viewed as a classifier performing n-label classification, where n is the number of distinct call stacks in the training dataset. The *adversarial uncertainty* is thus defined as the advantage of accuracy achieved by $\mathcal{A}$ over trivial approach. Formally, we define the adversarial uncertainty on accuracy *AU@Acc* as follows:

$$AU = 1 - \frac{\Pr[\mathcal{A}(S_{\text{enc}}) = S] - Acc_{\text{base}}}{1 - Acc_{\text{base}}} \tag{7}$$

where $\Pr[\mathcal{A}(S_{\text{enc}}) = S]$ is the probability that the adversary $\mathcal{A}$ correctly infers the original call stack S from the encoding and $Acc_{\text{base}} = \max_S(\Pr[S])$ is the accuracy of the trivial approach that always predicts the most frequent call stack. Likewise, we can define the adversarial uncertainty on precision, recall, and F1-score, denoted as *AU@Prec*, *AU@Rec*, and *AU@F1*, respectively. Higher AU indicates lower confidence of the adversary $\mathcal{A}$ and thus better privacy protection. $AU \geq 1$ indicates that the adversary $\mathcal{A}$ is no better than the trivial approach (i.e., always predicting the most frequent call stack). $AU = 0$ indicates that the adversary $\mathcal{A}$ can perfectly infer the original call stack indicating no privacy protection.

We report the evaluation results of PP-CSA in Table 8. Overall, we observe a decreasing trend of AU when the privacy budget ϵ is increased. This is expected, as a larger ϵ implies less noise added to the encoding and thus easier for the adversary to infer the original call stack. In the regime of $\epsilon \in [20, 40]$, we observe that PP-CSA achieves a great trade-off between accuracy and privacy, as the AU is non-trivially high (e.g., *AU@Acc* = 0.239 when $\epsilon = 40$) while PP-CSA attains highly accurate estimation of hot call stacks with F1 score = 0.970 in Table 2. This implies that the adversary cannot confidently infer a notable fraction of individual call stacks while PP-CSA can still accurately estimate the global call stack distribution and thus decide the hot call stacks. This is highly desirable in practice, as we alleviate the privacy concern of individual data while still providing useful statistics to developers for debugging without compromising the utility to a large extent.

Answer to RQ3: PP-CSA achieves a highly encouraging trade-off between privacy and accuracy in the common privacy budget regime ($\epsilon \in [20, 40]$).

7 DISCUSSION

We present the discussion of this paper in the following aspects.

Extensibility. We analyze the extensibility of PP-CSA from the following two aspects. From the language aspect, PP-CSA is currently implemented to handle Java and Android programs. Note that, PP-CSA does *not* rely on any language-specific features, and can be easily extended to other programming languages (e.g., C/C++ or JavaScript). Also, while the current implementation relies on Soot to perform static analysis, PP-CSA can leverage other static analysis tools like WALA [Santos and Dolby 2022] for JavaScript or LLVM [Lattner and Adve 2004] for C/C++. Also, from the analyzed software aspect, PP-CSA can be naturally used to protect the privacy of call stacks collected from different software, e.g., web applications, IoT devices, and smart contracts.

From the analysis aspect, PP-CSA is currently implemented to support hot call trace analysis where each event on the call stack is a user-level function call. Looking ahead, we envision that PP-CSA can be extended to support the analysis of other call stack types, e.g., hot system call analysis where each event on the trace is a system call. This facilitates various security applications like intrusion detection [Lu and Teng 2021; Wunderlich et al. 2020]. Furthermore, recall that the core idea of privacy preservation in PP-CSA is to privatize individual user's call stacks while still providing accurate estimations of group statistics, such as the frequency of each call stack. Thus, PP-CSA can be naturally extended to support other group statistics-based call stack analysis tasks in addition to hot call stack analysis, e.g. bottleneck identification [Tallent et al. 2009] and resource usage analysis [Decker et al. 2018]. However, PP-CSA may not be well-suited for individual property-based call stack analysis tasks, including detecting outlier behavior [Mirgorodskiy et al. 2006] (e.g., unusually long call stacks) or identifying call stacks associated with uncommonly-occurred bugs [Ko and Myers 2008]. Due to the fundamental premise of DP, PP-CSA falls short of providing accurate estimations of individual call stacks.

Moreover, while PP-CSA supports Android programs, it does not leverage domain-specific features of Android programs, such as the Android activity lifecycle and the Android Intent mechanism. We leave it as one future work to take those domain-specific features into account, which may be likely useful to simplify the call stack length and speed up the analysis to a reasonable extent. It is important to note that, with the shortened call stack length, we envision that the accuracy of PP-CSA can be further improved without compromising the privacy guarantees. The rationale has been clarified in Sec. 4.1.

Additional Computation Costs. In comparison to the direct transmission of call stacks to a central server, PP-CSA introduces additional computation cost in both the offline and online phases to ensure the privacy of individual users. During the offline phase, PP-CSA incurs additional computational expenses due to the training of the encoder-decoder model. As for the online phase, PP-CSA introduces additional computational demands stemming from two aspects: the client-side inference of the encoder model and the server-side decoding of the noisy call stacks. In the offline phase, the added computational cost is of minimal concern as it occurs in a non-real-time context. For the online phase, we have conducted a thorough evaluation of the processing time of PP-CSA in Sec. 6.1, which shows that the average running time of PP-CSA to process one call stack is around 3.84×10^{-3} seconds, which is acceptable in real-world scenarios and even faster than the current SOTA method. Overall, we observe that PP-CSA introduces reasonable additional computation cost compared to plain CSA.

Threat To Validity. There exist potential threats that the proposed framework may not adapt to other types of programs. We mitigate this threat to external validity by designing an approach that is language and platform agnostic (as discussed in the above "Extensibility"). Also, PP-CSA is evaluated on a set of real-world Java and Android programs. There exist threats that our evaluation results may not generalize to other test cases. We mitigate this threat by using Java and Android programs containing a diverse set of functionalities. Evaluating PP-CSA on those programs used by prior works also eases a fair comparison with **SOTA** and **DP-BART**.

Additionally, it is important to note that the unit of privacy in our framework primarily focuses on instance-level DP, where each user is expected to report only one call stack. We presume this scenario is a reasonable setup. For instance, when being used for analyzing call stacks that trigger crashes, it is unlikely for a user to experience multiple crashes with different call stacks in a short period. If a user does experience multiple crashes, uploading multiple call stacks may compromise the privacy guarantee. However, it is worth highlighting that our PP-CSA framework can be adapted to support user-level privacy. In this scenario, the server can make repeated queries to the client for multiple call stack information [Wu et al. 2014]. To achieve this, we can divide the total privacy budget into several portions and add noise to each individual call stack using these allocated portions. This approach ensures that the privacy guarantee is preserved for each individual call stack, maintaining user-level privacy.

Table 9. Evaluating different "hot trace" threshold T_{hot}.

T_{hot}	**Precision**	**Recall**	**F1**
0.01	0.925	0.824	0.830
0.005	0.916	0.801	0.807
0.001	0.912	0.739	0.768

Effect of "Hot Call Stack" Threshold. In line with typical DP-based systems, we have evaluated how different ϵ values affect the utility and privacy of PP-CSA. Besides ϵ, another important parameter in PP-CSA is the "hot call stack" threshold T_{hot}, as defined in Def. 3. Recall that T_{hot} is used to determine whether a call stack is "hot" by comparing the frequencies of the call stack with $T_{hot} \times N$ where N denotes the total number of users (following the procedure in [Hao et al. 2021]). Hence, we report how different T_{hot} values may influence the utility and privacy of PP-CSA in Table 9. We report the average precision, recall, and F1 score in different ϵ values on the *DaCapo* dataset. Overall, we observe that PP-CSA is robust to different T_{hot} values, and the utility and privacy of PP-CSA are not notably sensitive to T_{hot}. Nevertheless, when T_{hot} is overly large (e.g., 0.05 in our experiments), we can hardly find any "hot" call stack. Overall, we believe the findings as reasonable. In our evaluation (Sec. 5), we adopt a T_{hot} value as 0.01, and we encourage users to properly decide T_{hot} according to their own needs.

Table 10. Evaluating PP-CSA under low ϵ regime for *DaCapo* and *Android*. We report the F1 scores under each setting in this table.

Benchmark		**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
DaCapo	**SOTA**	0.558	0.535	0.519	0.491	0.480	0.448	0.426	0.409	0.365	0.328
	PP-CSA	**0.778**	**0.698**	**0.644**	**0.631**	**0.569**	**0.524**	**0.466**	**0.414**	**0.367**	**0.348**
Android	**SOTA**	0.497	0.475	0.465	0.451	0.444	0.433	0.415	**0.399**	**0.376**	**0.357**
	PP-CSA	**0.671**	**0.649**	**0.604**	**0.592**	**0.534**	**0.469**	**0.428**	0.355	0.262	0.238

Performance in Low ϵ Regime. To comprehensively compare PP-CSA and **SOTA**, we evaluated the performance of both in the range $\epsilon \in [1, 10]$ as shown in Table 10. PP-CSA consistently outperforms **SOTA** with the exception of the range $\epsilon \in [1, 3]$ for the *Android* dataset. We hypothesize that this is due to the *Android* dataset having a large number of functions and a skewed function call distribution. This characteristic might complicate the encoder-decoder model's capability on accurately learning the call stack distribution. In contrast, **SOTA** uses a count sketch to approximate the call stack distribution and is likely to be more resilient to skewed distributions at low ϵ values. Nevertheless, **SOTA**'s efficacy in this domain remains below par, with F1 scores even falling beneath 0.4.

In general, attaining meaningful utility in low ϵ scenarios is challenging, and many relevant studies, like [Igamberdiev and Habernal 2023; Krishna et al. 2021], prioritize the high ϵ range, focusing more on utility than on privacy. Following this trend, we also focus on the high ϵ range in our main evaluation. And as a general recommendation for practitioners, we discourage performing call stack analysis at low ϵ levels. More importantly, we clarify that a high ϵ value does not inherently imply severe privacy issues in practice. As evidenced in Sec. 6.3, even at $\epsilon = 40$, the level of adversarial uncertainty remains considerably high. These insights lead us to omit the low ϵ regime from our main evaluation in Sec. 6.1.

Potential Integration with other Privacy Enhancing Technques. In addition to the privacy-preserving approach provided by PP-CSA, it is worth considering the potential integration with other privacy-enhancing techniques, such as anonymization transmission services like the Tor network [Dingledine et al. 2004]. These techniques can offer anonymization of user identities, which provides orthogonal privacy guarantees to PP-CSA. Our PP-CSA can be seamlessly built on the basis of Tor. For instance, one possible integration approach is to leverage Tor as the underlying network infrastructure for transmitting the encoded call stacks in PP-CSA. By utilizing Tor, the client in PP-CSA can securely transmit the LDP-protected call stack to the server without revealing the client's identity. This integration would strengthen the overall privacy protection provided by the system, providing both anonymized transmission of call stacks and ensuring differential privacy guarantees.

8 RELATED WORK

In this section, we review the related work on analyzing software in a privacy-preserving manner, the applications of LDP and also how SE and PL techniques are used to improve privacy-preserving software systems.

Privacy-Preserving Software Analysis. Recently, there has been a surge in efforts to secure software analysis using various privacy-enhancing techniques, such as zero-knowledge proof (ZKP), trusted execution environments (TEE), and differential privacy (DP). Fang et al. [Fang et al. 2021] introduce a ZKP-based method for intra- and inter-procedural abstract interpretation. Cheesecloth offers a ZKP-based solution to verify real-world software vulnerabilities [Cuéllar et al. 2023]. Tramer et al. [Tramer et al. 2017] propose an SGX-based method to host a bug bounty program, wherein the proof-of-concept (PoC) of a software exploit is confirmed inside a secure enclave. In the section "Applications of LDP", we delve deeper into LDP's usage in software profiling.

Applications of LDP. LDP stands out as a key privacy protection method. For instance, RAPPOR, an LDP-based mechanism, is employed by Google to collect popular domain data from users in the Chrome browser [Erlingsson et al. 2014]. Tech giants like Apple, Microsoft, and Uber have employed LDP to enhance the privacy of their data collection services [Cormode et al. 2018; Near 2018]. Specifically in software profiling, LDP has been utilized for collecting software node coverage, mobile app event frequency, and call traces [Hao et al. 2021; Zhang et al. 2020a,b].

SE & PL for Privacy-Preserving Software. The software engineering and programming language (SE & PL) communities have delved into testing, verifying, and debugging techniques tailored for privacy-preserving software systems. Notably, recent efforts have focused on employing program verification and type-system based approaches to verify the correctness of ZKP [Liu et al. 2023; Pailoor et al. 2023]. Most of the identified bugs, stemming from erroneous implementations of ZKP programs, can be exploited by adversaries. FedDebug [Gill et al. 2023] introduces a federated debugging framework, enabling users to pinpoint which federated client contributes to the central model's misbehavior. Similarly, MPCDiff [Pang et al. 2024] deploys differential testing to compare a machine learning model with its privacy-preserving counterpart shielded by multi-party computation (MPC). Also, Ding et al. [Ding et al. 2018] utilize differential testing to unearth bugs in differential privacy (DP) programs. Tools such as DP-finder [Bichsel et al. 2018] and DP-sniper [Bichsel et al. 2021] either leverage random testing or a neural network-based testing oracle to reveal violations of DP guarantees. Apart from the analysis aspect, the PL community also explored the optimization of privacy-preserving software systems. For instance, Levy et al. [Levy et al. 2023] and Ishaq et al. [Ishaq et al. 2019] present a toolchain for compiling and vectorizing MPC programs. Roy et al. [Roy et al. 2021] and Wang et al. [Wang et al. 2021] show the usage of program synthesis techniques in generating DP-protected programs from their unprotected counterparts. Some research focuses on programming language design, testing and verification of oblivious RAM (ORAM) protocols [Darais et al. 2019; Liu et al. 2015; Ma et al. 2022].

9 CONCLUSION

We propose PP-CSA, a novel privacy-preserving CSA approach that can be deployed in real-world scenarios. PP-CSA is based on LDP and incorporates several key optimizations in its technical pipeline. These optimizations include an encoder-decoder scheme to enforce LDP and a call stack compressing and matching algorithm to mitigate utility-privacy trade-offs. Our evaluation demonstrates the efficacy of the privacy-preserving call stack analysis pipeline implemented by PP-CSA. It achieves high levels of utility and privacy guarantees while maintaining high efficiency. We further show that our proposed optimizations are effective and have a "synergistic effect"; enabling full optimizations offers the highest accuracy improvement. We also mimic adversaries who aim to infer the user privacy from the PP-CSA-protected call stacks, and illustrate that attackers can only achieve a very low attack accuracy under our common settings. We conclude the paper by discussing various extensions, some design considerations, limitations, and future work directions. Overall, our work aligns with the trajectory of privacy-preserving techniques in the programming language and software engineering communities, extending the realm to the call stack analysis and introducing innovative enhancements to tackle the unique challenges in this domain. We envision that our work can inspire further investigations into privacy protection in software systems.

DATA-AVAILABILITY STATEMENT

We have released our artifacts at [artifact 2023]. Our evaluation results reported in the paper can be fully reported using the released artifacts. Looking ahead, We will maintain them for future research comparison and usage. We will also enhance the artifacts by providing more detailed documents and use cases to support the usage and extension of the community.

ACKNOWLEDGMENTS

This research was supported in part by a RGC CRF grant under the contract C6015-23GF. We are grateful to the anonymous reviewers for their valuable comments.

REFERENCES

Research artifact. 2023. PP-CSA. https://github.com/wangzhaoyu07/PP-CSA.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269. https://doi.org/10.1145/2594291.2594299

Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. 2019. The privacy blanket of the shuffle model. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part II 39*. Springer, 638–667. https://doi.org/10.1007/978-3-030-26951-7_22

Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. 2018. Dp-finder: Finding differential privacy violations by sampling and optimization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 508–524. https://doi.org/10.1145/3243734.3243863

Benjamin Bichsel, Samuel Steffen, Ilija Bogunovic, and Martin Vechev. 2021. Dp-sniper: Black-box discovery of differential privacy violations using classifiers. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 391–409. https://doi.org/10.1109/SP40001.2021.00081

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Michael D Bond and Kathryn S McKinley. 2005. Continuous path and edge profiling. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 11–pp. https://doi.org/10.1109/MICRO.2005.16

Graham Cormode, Somesh Jha, Tejas Kulkarni, Ninghui Li, Divesh Srivastava, and Tianhao Wang. 2018. Privacy at scale: Local differential privacy in practice. In *Proceedings of the 2018 International Conference on Management of Data*. 1655–1658. https://doi.org/10.1145/3183713.3197390

Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. 2023. Cheesecloth:{Zero-Knowledge} Proofs of Real World Vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6525–6540.

David Darais, Ian Sweet, Chang Liu, and Michael Hicks. 2019. A language for probabilistically oblivious computation. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–31. https://doi.org/10.1145/3371118

Omar De la Cruz Cabrera, Mona Matar, and Lothar Reichel. 2019. Analysis of directed networks via the matrix exponential. *J. Comput. Appl. Math.* 355 (2019), 182–192. https://doi.org/10.1016/J.CAM.2019.01.015

Normann Decker, Boris Dreyer, Philip Gottschling, Christian Hochberger, Alexander Lange, Martin Leucker, Torben Scheffel, Simon Wegener, and Alexander Weiss. 2018. Online analysis of debug trace data for embedded systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 851–856. https://doi.org/10.23919/DATE.2018.8342124

Android developer manual. 2023. systrace. https://developer.android.com/topic/performance/tracing/command-line.

Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting violations of differential privacy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 475–489. https://doi.org/10.1145/3243734.3243818

Roger Dingledine, Nick Mathewson, Paul F Syverson, et al. 2004. Tor: The second-generation onion router.. In *USENIX security symposium*, Vol. 4. 303–320.

Cynthia Dwork. 2006. Differential privacy. In *Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II 33*. Springer, 1–12. https://doi.org/10.1007/11787006_1

Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407. https://doi.org/10.1561/0400000042

Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1054–1067. https://doi.org/10.1145/2660267.2660348

Zhiyong Fang, David Darais, Joseph P Near, and Yupeng Zhang. 2021. Zero knowledge static program analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2951–2967. https://doi.org/10.1145/3460120.3484795

Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178. https://doi.org/10.1145/1536414.1536440

Waris Gill, Ali Anwar, and Muhammad Ali Gulzar. 2023. FedDebug: Systematic Debugging for Federated Learning Applications. In *ICSE*. https://doi.org/10.1109/ICSE48619.2023.00053

Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 103–116. https://doi.org/10.1145/1629575.1629586

Eric Goldman. 2020. An introduction to the california consumer privacy act (ccpa). *Santa Clara Univ. Legal Studies Research Paper* (2020).

Google. 2020. Monkey: UI/Application exerciser for Android. https://developer.android.com/studio/test/other-testing-tools/monkey.

Ivan Habernal. 2021. When differential privacy meets NLP: The devil is in the detail. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 1522–1528. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.114

Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 145–155. https://doi.org/10.1109/ICSE.2012.6227198

Yu Hao, Sufian Latif, Hailong Zhang, Raef Bassily, and Atanas Rountev. 2021. Differential privacy for coverage analysis of software traces. *Leibniz international proceedings in informatics* 194 (2021). https://doi.org/10.4230/LIPICS.ECOOP.2021.8

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/NECO.1997.9.8.1735

Yiqing Hua, Armin Namavari, Kaishuo Cheng, Mor Naaman, and Thomas Ristenpart. 2022. Increasing adversarial uncertainty to scale private similarity testing. In *31st USENIX Security Symposium (USENIX Security 22)*. 1777–1794.

Timour Igamberdiev and Ivan Habernal. 2023. DP-BART for Privatized Text Rewriting under Local Differential Privacy. In *Findings of the Association for Computational Linguistics: ACL 2023*. 13914–13934. https://doi.org/10.18653/V1/2023.FINDINGS-ACL.874

Muhammad Ishaq, Ana L Milanova, and Vassilis Zikas. 2019. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1539–1556. https://doi.org/10.1145/3319535.3339818

Wei Jin and Alessandro Orso. 2012. Bugredux: Reproducing field failures for in-house debugging. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 474–484. https://doi.org/10.1109/ICSE.2012.6227168

Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT*. 4171–4186. https://doi.org/10.18653/V1/N19-1423

Amy J Ko and Brad A Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*. 301–310. https://doi.org/10.1145/1368088.1368130

Satyapriya Krishna, Rahul Gupta, and Christophe Dupuy. 2021. ADePT: Auto-encoder based Differentially Private Text Transformation. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. 2435–2439. https://doi.org/10.18653/V1/2021.EACL-MAIN.207

Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. 2015. Circuit fingerprinting attacks: Passive deanonymization of tor hidden services. In *24th USENIX Security Symposium (USENIX Security 15)*. 287–302.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

Benjamin Levy, Muhammad Ishaq, Benjamin Sherman, Lindsey Kennard, Ana Milanova, and Vassilis Zikas. 2023. Combine: compilation and backend-independent vectorization for multi-party computation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2531–2545. https://doi.org/10.1145/3576915.3623181

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880. https://doi.org/10.18653/V1/2020.ACL-MAIN.703

Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices* 50, 4 (2015), 87–101. https://doi.org/10.1145/2775054.2694385

Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. 2023. Certifying Zero-Knowledge Circuits with Refinement Types. *arXiv preprint arXiv:2304.07648* (2023).

Yijun Lu and Shaohua Teng. 2021. Application of sequence embedding in host-based intrusion detection system. In *2021 IEEE 24th international conference on computer supported cooperative work in design (CSCWD)*. IEEE, 434–439. https://doi.org/10.1109/CSCWD49262.2021.9437683

Pingchuan Ma, Zhibo Liu, Yuanyuan Yuan, and Shuai Wang. 2022. NeuralD: Detecting Indistinguishability Violations of Oblivious RAM With Neural Distinguishers. *IEEE Transactions on Information Forensics and Security* 17 (2022), 982–997. https://doi.org/10.1109/TIFS.2022.3155274

Alexander V Mirgorodskiy, Naoya Maruyama, and Barton P Miller. 2006. Problem diagnosis in large-scale computing environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 88–es. https://doi.org/10.1145/1188455.1188548

Joe Near. 2018. Differential privacy at scale: Uber and berkeley collaboration. In *Enigma 2018 (Enigma 2018)*.

Soot OSS. 2023. Soot – A Java Optimization Framework. https://github.com/soot-oss/soot.

Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. 2023. Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1510–1532. https://doi.org/10.1145/3591282

Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2024. MPCDiff: Testing and Repairing MPC-Hardened Deep Learning Models.. In *NDSS*.

Natalia Ponomareva, Hussein Hazimeh, Alex Kurakin, Zheng Xu, Carson Denison, H Brendan McMahan, Sergei Vassilvitskii, Steve Chien, and Abhradeep Guha Thakurta. 2023. How to dp-fy ml: A practical guide to machine learning with differential privacy. *Journal of Artificial Intelligence Research* 77 (2023), 1113–1201. https://doi.org/10.1613/JAIR.1.14649

Subhajit Roy, Justin Hsu, and Aws Albarghouthi. 2021. Learning differentially private mechanisms. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 852–865. https://doi.org/10.1109/SP40001.2021.00060

Joanna CS Santos and Julian Dolby. 2022. Program analysis using WALA (tutorial). In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1819–1819. https://doi.org/10.1145/3540250.3569449

Nathan R Tallent, John M Mellor-Crummey, Laksono Adhianto, Michael W Fagan, and Mark Krentel. 2009. Diagnosing performance bottlenecks in emerging petascale applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/1654059.1654111

Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis *(PLDI 2017)*. 263–277. https://doi.org/10.1145/3062341.3062359

Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 19–34. https://doi.org/10.1109/EUROSP.2017.28

Anstralian National University. 2021. DaCapo Bench. https://www.dacapobench.org/.

Paul Voigt and Axel Von dem Bussche. 2017. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing* 10, 3152676 (2017), 10–5555.

Tianhao Wang, Ninghui Li, and Somesh Jha. 2019. Locally differentially private heavy hitter identification. *IEEE Transactions on Dependable and Secure Computing* 18, 2 (2019), 982–993. https://doi.org/10.1109/TDSC.2019.2927695

Yuxin Wang, Zeyu Ding, Yingtai Xiao, Daniel Kifer, and Danfeng Zhang. 2021. DPGen: Automated program synthesis for differential privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 393–411. https://doi.org/10.1145/3460120.3484781

Stanley L Warner. 1965. Randomized response: A survey technique for eliminating evasive answer bias. *J. Amer. Statist. Assoc.* 60, 309 (1965), 63–69. https://doi.org/10.1080/01621459.1965.10480775

Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 204–214. https://doi.org/10.1145/2610384.2610386

Sarah Wunderlich, Markus Ring, Dieter Landes, and Andreas Hotho. 2020. Comparison of system call representations for intrusion detection. In *International Joint Conference: 12th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2019) and 10th International Conference on EUropean Transnational Education (ICEUTE 2019) Seville, Spain, May 13th-15th, 2019 Proceedings 12*. Springer, 14–24. https://doi.org/10.1007/978-3-030-20005-3_2

Yin Yang, Zhenjie Zhang, Gerome Miklau, Marianne Winslett, and Xiaokui Xiao. 2012. Differential privacy in data publication and analysis. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 601–606. https://doi.org/10.1145/2213836.2213910

Andrew C Yao. 1982. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE, 160–164. https://doi.org/10.1109/SFCS.1982.38

Hailong Zhang, Yu Hao, Sufian Latif, Raef Bassily, and Atanas Rountev. 2020a. Differentially-private software frequency profiling under linear constraints. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–24. https://doi.org/10.1145/3428271

Hailong Zhang, Sufian Latif, Raef Bassily, and Atanas Rountev. 2020b. Differentially-private control-flow node coverage for software usage analysis. In *USENIX Security Symposium (USENIX Security)*.

Received 21-OCT-2023; accepted 2024-02-24