Privacy in Theory, Bugs in Practice: Grey-Box Auditing of Differential Privacy Libraries

Tudor Cebere
PreMeDICaL team, Inria, Idesp, Inserm, Université de Montpellier
tudor.cebere@inria.fr

David Erb
Technical University of Munich & Oblivious
david.erb@tum.de

Damien Desfontaines
Hiding Nemo
damien@desfontain.es

Aurélien Bellet
PreMeDICaL team, Inria, Idesp, Inserm, Université de Montpellier
aurelien.bellet@inria.fr

Jack Fitzsimons
Oblivious
jack@oblivious.com

Abstract

Differential privacy (DP) implementations are notoriously prone to errors, with subtle bugs frequently invalidating theoretical guarantees. Existing verification methods are often impractical: formal tools are too restrictive, while black-box statistical auditing is intractable for complex pipelines and fails to pinpoint the source of the bug. This paper introduces Re:cord-play, a "gray-box" auditing paradigm that inspects the internal state of DP algorithms. By running an instrumented algorithm on neighboring datasets with identical randomness, Re:cord-play directly checks for data-dependent control flow and provides concrete falsification of sensitivity violations by comparing declared sensitivity against the empirically measured distance between internal inputs. We generalize this to Re:cord-play-sample, a full statistical audit that isolates and tests each component, including untrusted ones. We show that our novel testing approach is both effective and necessary by auditing 12 open-source libraries, including SmartNoise SDK, Opacus, and Diffprivlib, and uncovering 13 privacy violations that impact their theoretical guarantees. We release our framework as an open-source Python package[1], thereby making it easy for DP developers to integrate effective, computationally inexpensive, and seamless privacy testing as part of their software development lifecycle.

Keywords

differential privacy, privacy auditing, implementation security, testing framework

1 Introduction

The proliferation of large-scale data collection and analysis has made protecting individuals' privacy a key challenge for society and technology. In response, differential privacy (DP) has emerged as the de facto standard for private data sharing, offering a rigorous, provable privacy guarantee [28]. A randomized mechanism is said to be differentially private if its output distribution remains statistically indistinguishable when applied to any pair of neighboring datasets, i.e., datasets differing by the data of a single individual. This guarantee ensures that the presence or absence of any individual's data in a dataset has a mathematically bounded impact on the outcome, protecting them from privacy attacks based on released results. In most DP mechanisms, this influence is captured by the *sensitivity* of the underlying query, i.e., the maximum change in the query's output when the data of a single individual is modified. DP also supports building complex algorithms by composing basic *privacy primitives* (e.g., the Laplace, Gaussian, and exponential mechanisms) and applying data-independent post-processing to their outputs. These properties have led to the growing adoption of DP by technology companies and public institutions [2, 3, 49, 93][2], making it a critical component in the modern data ecosystem.

Unfortunately, the conceptual elegance of DP contrasts with the practical difficulty of implementing it correctly. DP guarantees are notoriously susceptible to subtle implementation flaws that can invalidate the claimed protection. These bugs arise from sources including incorrect sensitivity calculations [82], data-dependent pre-

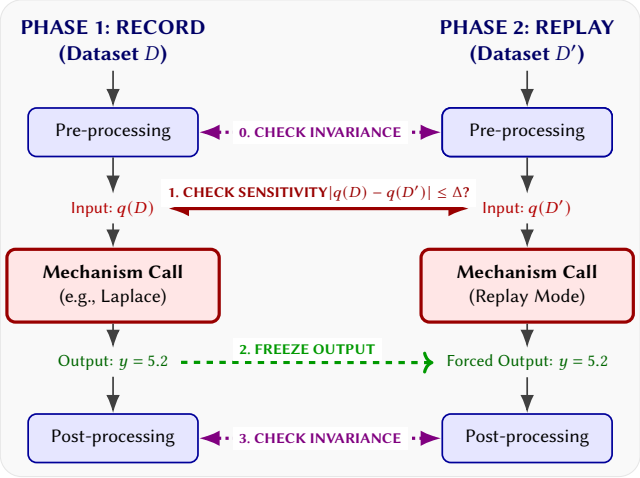

Figure 1: Summary of Re:cord-play highlighting the Record Phase (Dataset D) and Replay Phase (Dataset D').

[1] https://github.com/ObliviousAI/dp-recorder

[2] For a comprehensive list, see the Differential Privacy Deployments Registry [45].

Proceedings on Privacy Enhancing Technologies 2026(3), 467–483

https://doi.org/10.56553/popets-2026-0091

or post-processing [7, 36], mishandled privacy budget composition [55], insecure randomness [24], or floating-point vulnerabilities [42, 48, 63]. This creates a gap between the stated privacy guarantees and the actual protection provided by their implementation.

To address this gap between theoretical promises and practical guarantees, the research community has increasingly turned to *distributional auditing* [6]. This statistical approach aims to empirically falsify privacy guarantees by distinguishing the output distributions of a mechanism on neighboring datasets. The majority of distributional auditing frameworks operate as *black-box* auditors, assessing privacy properties based solely on input-output behavior. This approach can detect privacy violations regardless of their cause, from high-level algorithmic mistakes to low-level implementation bugs, thus providing a robust assessment of a system.

However, such techniques are limited when auditing modern DP algorithms, which are rarely simple primitives but rather complex pipelines that chain heterogeneous components. A typical workflow involves preprocessing (like clipping), adaptive composition of multiple mechanisms, and post-processing of their outputs. Black-box auditing of such pipelines presents two challenges. First, it is often statistically intractable: the final output distribution is the high-dimensional result of a complex process, requiring a prohibitive number of samples to distinguish. Second, it is uninformative: even if a black-box test detects a violation, it flags the entire system as faulty without pinpointing the faulty component.

This leaves developers with few practical options. Formal verification tools [35], while powerful, are typically restrictive, requiring entire pipelines to be rewritten in specialized, provable domain-specific languages (DSLs). This is a barrier to entry for the vast ecosystem of Python or C++ libraries. Thus, a practical gap remains: we lack an automated tool that can pinpoint where and why an implementation fails without the cost of end-to-end auditing.

Our work is motivated by the need for a "privacy debugger" integrated into the software development lifecycle. Rather than relying on manual verification, we propose a solution to detect implementation bugs in Continuous Integration and Delivery (CI/CD) pipelines. We aim to provide a tool as effortless as a standard unit test, yet capable of evaluating highly randomized algorithms.

Contribution. This paper bridges the gap between theoretical privacy guarantees and practical correctness of implementations by introducing a new *gray-box auditing* paradigm. Unlike black-box methods that analyze high-dimensional output distributions, our approach assumes access to the underlying implementation and inspects the internal state of the algorithm mechanisms and the interfaces between its components. Our core insight is that DP implementations are not monolithic: they interleave data-independent logic with calls to data-dependent privacy primitives. We leverage the fact that most privacy violations that arise fall into one of the following categories: (i) data-dependent leakage into data-independent code, (ii) sensitivity miscalculations, (iii) noise miscalibration, and (iv) flawed privacy primitive implementations. Our approach generates concrete, actionable reports that help practitioners identify and resolve common errors. Our contributions can be summarized as follows:

(1) **Deterministic Verification (Re:cord-play):** We leverage a novel record-and-replay instrumentation that benefits from the separation between private primitives and the rest of the processing logic (see Figure 1). By *freezing* the outputs of privacy primitives during execution on the neighboring dataset, we check whether code assumed to be data-independent indeed remains *invariant* across datasets. This allows us to catch leaks of type (i) deterministically: if intermediate variables diverge despite identical mechanism outputs, data has leaked into logic assumed to be data-independent. Simultaneously, it flags sensitivity errors of type (ii) by comparing the declared privacy parameters against the empirically measured distance between the internal inputs fed to each primitive. Additionally, if we assume *vetted, trusted, and correctly implemented* privacy primitives, we can catch errors of type (iii) as well.
(2) **Component-Wise Audit (Re:cord-play-sample):** We extend our framework to handle untrusted or custom primitives via Re:cord-play-sample. By isolating privacy primitives, we perform statistical tests on them rather than the entire pipeline, making the auditing problem computationally cheaper and more tractable. Applying ideas from black-box auditing within our framework, we propose to leverage Privacy Loss Distributions (PLDs) to obtain precise privacy loss estimates for each privacy primitive. This approach allows us to detect issues of type (iii) and (iv), while enabling the composition of both analytically known PLDs and empirically estimated PLDs from arbitrary mechanisms to derive end-to-end privacy bounds.
(3) **Practical Evaluation:** We demonstrate the practical value of our framework by auditing popular open-source DP libraries, including SmartNoise SDK, Synthcity, Opacus, and Diffprivlib, in which we *uncover several previously unknown privacy violations*. These findings range from subtle sensitivity miscalculations in data preprocessing to data-dependent logic flaws. This validates our approach as a practical, effective method for finding real-world bugs that elude current testing and verification techniques. We release our framework as an open-source package, lowering the barrier to test DP code and encouraging practitioners to adopt our effective and computationally inexpensive tool as a component of their software development lifecycle.

The rest of this paper is organized as follows. We discuss background and related work in Section 2. In Section 3, we introduce our auditing techniques, Re:cord-playand Re:cord-play-sample. We present our evaluation setup and the real-world bugs we uncovered in Section 4. We provide practical insights and recommendations for authors of DP software, outline the limitations of our approach, and discuss future work in Section 5, and then conclude.

2 Background & Related Work

2.1 Differential Privacy

Differential Privacy (DP) [28] has become the standard for private data analysis, valued for its rigorous and mathematically provable guarantees. It enables the release of useful aggregate statistics from a dataset while limiting the disclosure of information about any single individual within it. A mechanism's privacy is quantified by how indistinguishable its output distributions are when run on

any two neighboring datasets (i.e., datasets that differ by a single record). More formally, a mechanism is considered differentially private if it satisfies the following definition:

Definition 2.1. A randomized mechanism $\mathcal{M} : \mathcal{D} \to \mathcal{X}$ satisfies (ε, δ)-approximate DP if for every pair of neighboring datasets D, D', and every subset $A \subseteq \mathcal{X}$ of the output space, it holds that:

$$P(\mathcal{M}(D) \in A) \leq e^{\varepsilon} P(\mathcal{M}(D') \in A) + \delta.$$

Here, ε represents the privacy loss parameter, while δ should be a small constant, see [22, 27] for more details. Mechanisms with $\delta = 0$ satisfy pure DP.

The core challenge in implementing any DP mechanism $\mathcal{M}$ is to correctly calibrate the amount of noise, which is directly determined by the mechanism's *sensitivity*. The global sensitivity of a query $f : \mathcal{D} \to \mathbb{R}^k$ measures the maximum possible change in f's output when applied to any two neighboring datasets, D and D' [28]. The specific definition of sensitivity depends on the L_p norm used to measure this change: $\Delta_p f = \max_{D,D'} \|f(D) - f(D')\|_p$.

While our approach is general, we anchor our exposition in three canonical privacy primitives defined by their corresponding sensitivity: (i) the *Laplace Mechanism* achieves pure ε-DP by adding Laplace-distributed noise scaled to the L_1-sensitivity of the query; (ii) the Gaussian Mechanism guarantees (ε, δ)-DP by adding Gaussian noise, with a scale σ calibrated to the L_2-sensitivity; (iii) the Exponential Mechanism [62] serves as a general-purpose tool for discrete outputs, providing ε-DP by selecting an item $r \in \mathcal{R}$ based on a quality score q with probability proportional to $\exp\left(\frac{\varepsilon q(D,r)}{2\Delta_\infty q}\right)$, where $\Delta_\infty q$ is the L_∞-sensitivity of the quality function.

Two fundamental properties make DP a powerful and practical framework. First, immunity to *post-processing*: an analyst cannot weaken the privacy guarantee by performing arbitrary data-independent computations on the output of a DP mechanism. This ensures that once data is released privately, it remains private regardless of subsequent analysis. Second, *composition* allows for managing a "privacy budget" across multiple analyses. Composition theorems [29, 51, 65, 85] state that the total privacy loss from releasing results of multiple DP mechanisms on the same underlying data is a function of the privacy losses of the individual mechanisms. This allows for the construction of complex, multi-step private algorithms from simpler DP building blocks. To track this cumulative privacy loss, *privacy accountants* [1] are used, providing tighter bounds than basic composition theorems [28, 29]. This is often done by converting (ε, δ) into a representation for which composition is tight, such as Rényi DP (RDP) [64] or *PLDs* [26, 41], which can be composed more precisely before being converted back to the final (ε, δ) guarantee.

2.2 Membership Inference Attacks and Auditing

Membership Inference Attacks (MIAs) [7, 44, 46, 58, 79, 88, 89] determine whether a specific individual's data was included in the input dataset of a function, based solely on its observed output. Typically, these attacks operate by comparing the outputs of a mechanism on datasets that either include or exclude a target record. If the outputs are sufficiently different, the adversary can infer the presence of the record with better than random accuracy. The strength of an MIA is typically evaluated by its advantage over random guessing, as visualized by the Receiver Operating Characteristic (ROC) curve. This analysis often focuses on the attack's True Positive Rate (TPR) at low False Positive Rates (FPRs) (e.g., 1% or 0.1%), as this demonstrates the attack's practical effectiveness [17, 76].

In practice, constructing such attacks ranges in difficulty. For simple algorithms, one can often design linear or threshold-based classifiers that succeed [25, 67]. For more complex algorithms, shadow-modeling [79] techniques are employed, which train auxiliary models to simulate the target mechanism and guide inference. These methods, however, are computationally demanding and algorithm-specific, requiring careful design and significant resources.

MIAs are particularly relevant when auditing differentially private algorithms [4, 18, 19, 47, 53, 57, 67, 68, 80, 92] because the DP definition is interpreted as a worst-case bound against membership inference. A mechanism satisfying (ε, δ)-DP ensures that, for any record, the success probability of a membership inference attack is bounded for all FPRs [25, 51, 86], imposing a constraint on the ROC curve on the success probability of MIAs of the form $\text{TPR} \leq e^{\varepsilon}\text{FPR} + \delta$. Thus, the formal DP guarantee provides a direct upper bound on the information leakage that MIAs can exploit. This relationship enables auditing privacy claims by inverting it: the empirical performance of an MIA directly yields a lower bound on the privacy parameters of an algorithm. If an adversary achieves a TPR/FPR pair that violates the theoretical bound, the claimed (ε, δ) is falsified.

We define a *distributional auditing* pipeline (see Algorithm 3 in the Appendix) as a statistical framework that estimates the privacy loss of a mechanism by analyzing its output samples. While typically deployed as a *black-box* (where the adversary observes only final outputs), the underlying statistical methodology remains the same regardless of access. It consists of two main components:

(1) **The Adversary ($\mathcal{A}$):** This component defines the attack strategy. It typically involves two subroutines: (i) a `Rank` function assigning a confidence score that the output was generated by D or D', and (ii) a `Reject` (e.g., a threshold) to classify the output as belonging to $\mathcal{M}(D)$ or $\mathcal{M}(D')$.
(2) **The Auditing Scheme:** This component analyzes the adversary's performance by running the mechanism many times on D and D'. It computes the Type I error rate (α) and Type II error rate (β) and uses these statistics to compute a high-probability lower bound on the $(\hat{\varepsilon}, \delta)$ privacy loss.

2.3 Related Work

Distributional Auditing. A significant amount of the literature has been devoted to auditing DP machine learning (ML) algorithms. For a comprehensive summary, refer to Annamalai et al. [6]. Initially investigated by Jagielski et al. [47], auditing DP-ML is constrained by the fact that the underlying mechanism is computationally intensive and typically involves subsampling [8, 52]. Recent research aims to improve the sample efficiency of these audits [53, 57, 81, 92], audit various DP variants [19, 67], or improve accounting in specific threat models [4, 5, 18]. We draw on standard techniques from this line of work in Section 3.3, applying them to simple privacy primitives with unknown structures.

Optimization and Counterexample Search. A central challenge in distributional auditing is identifying worst-case inputs, or counterexamples, yielding the tightest privacy lower bounds (see e.g. the hierarchy in Nasr et al. [68]). Bichsel et al. [13] introduced DP-Finder, which formulates the search for privacy violations as an optimization problem and uses a smooth surrogate objective to discover high-violation counterexamples. Other approaches include StatDP [23], a semi-black-box tool using hypothesis testing, and DP-Sniper [14], which trains classifiers to distinguish neighboring outputs. Our framework complements these techniques: while these methods excel at optimizing for inputs triggering violations, our work focuses on pinpointing the bugs causing the harm.

Proofs of Privacy. Another approach to auditing leverages Zero-Knowledge Proofs (ZKPs) [38]. While substantial work [11, 15, 66, 77] has been done to certify DP compliance without data access, these proofs are limited in scope. As noted, they strictly demonstrate that the mechanism implemented in practice matches the intended algorithmic description. They do not validate the algorithm's actual behavior on the underlying data, nor can they verify whether the claimed privacy guarantee associated with the intended algorithm is theoretically valid [68, 82]. Therefore, this line of work addresses execution integrity and complements our approach.

Sensitivity-tracking and domain-specific languages (DSLs). A robust method for preventing privacy violations is the use of libraries that allow only a subset of vetted functions. By composing these known transformations, systems can track sensitivity and apply noise mechanisms with formal guarantees. Frameworks such as OpenDP [71] and Tumult Analytics [12] have demonstrated that this "correct-by-construction" approach can scale to multiprocess and multihost environments. However, the primary limitation of this paradigm is its integration into existing environments: integrating these DSLs into pre-existing, large-scale data pipelines often requires a significant rewrite of the underlying semantics. Consequently, many real-world systems continue to rely on ad-hoc or custom implementations that lack these safety guarantees, making the need for easy-to-use testing frameworks such as ours essential.

Formal Verification. A significant body of research in DP has focused on formal languages and static analysis [35, 69, 87, 94]. These systems employ programming language techniques to statically verify that an algorithm formally satisfies the mathematical definitions of DP. The primary benefit of this approach is providing a provable guarantee of privacy, effectively eliminating entire classes of bugs at compile-time. Despite these robust guarantees, such formal languages have seen little practical adoption, primarily due to significant usability and integration barriers. These systems often impose a steep learning curve, requiring practitioners to become specialized in non-mainstream programming languages that are difficult to integrate into existing, mature data science pipelines.

3 Framework

In this section, we introduce our gray-box auditing framework, Re:cord-play and Re:cord-play-sample. We begin by outlining the key insights that make our fast, *structural* verification possible.

3.1 Motivation & Insights

The transition of DP from theoretical research to production software has revealed a significant gap in our testing capabilities. While the mathematical properties of standard privacy primitives (e.g., the Laplace and Gaussian mechanisms) are well-understood, the pipelines that utilize them are increasingly complex and prone to implementation errors.

In practice, most privacy violations do not stem from flaws in the primitives themselves. Rather, they arise from mechanism-independent integration errors, including sensitivity underestimation, incorrect noise scaling, and improper accounting. These are logic bugs, not statistical ones. Still, the standard approach to detecting them remains distributional auditing, which treats the algorithm as a black box and attempts to distinguish its output distributions on neighboring datasets. This approach faces three challenges:

(1) **Statistical Intractability:** As noted by Gilbert and McMillan [37], auditing unconstrained algorithms is computationally intractable. In complex pipelines involving adaptive composition, the output space grows exponentially.
(2) **Complexity of MIA Design:** To detect a violation, an auditor must design a Membership Inference Attack (MIA) capable of distinguishing distributions with high confidence. This requires specialized expertise in threat modeling, extensive hyperparameter tuning, and massive computational resources.
(3) **Lack of Error Source:** Even if the MIA successfully detects a violation, it provides a binary fail signal. It cannot pinpoint whether the error lies in pre-processing, sensitivity calibration, privacy primitives, or post-processing.

A Structural Decomposition Approach. To address this, we move beyond black-box distributional auditing and instead leverage the program's structure to detect integration errors deterministically. Our framework initially assumes that the privacy primitives are correctly implemented—an assumption we later relax. This is a reasonable premise given the availability of community-vetted, open-source libraries such as OpenDP [71] and Tumult Analytics [12]. Then, if a mechanism such as `laplace_mechanism(input, sensitivity, scale)` is correctly implemented, we can focus on how the mechanism is *used* within the broader pipeline.

Observing that DP algorithms interleave private primitives with logic that is not supposed to depend on the private data, we approach the auditing problem through *structural decomposition*. We distinguish between data-dependent functions (the privacy primitives) and data-independent ones (the control flow, pre-processing, and post-processing logic). Data-dependent functions are the only legitimate source of variation between executions on D and D', and all other variations must be explained through sensitivity. To verify this, consider a thought experiment: suppose we overwrite every data-dependent mechanism in a pipeline to return a constant value (frozen output). If we then execute this pipeline on D and D' with the same seed, the program's state must remain identical throughout execution. We define this property as *invariance under D and D'*. If any intermediate value diverges, the function was not data-independent, implying a privacy leak in the processing logic.

This decomposition transforms the auditing problem. Instead of designing MIAs to estimate privacy loss bounds, we need to answer

the following questions: (i) Do the data-independent parts of the pipeline remain *invariant* when the dataset changes, given fixed mechanism outputs? and (ii) Do the inputs fed into the trusted mechanisms respect the desired overall privacy guarantees?

This shift in perspective enables us to transition from computationally expensive attacks to fast unit tests suitable for standard software development pipelines. This logic forms the foundation of Re:cord-play, which we detail in the following section.

3.2 Re:cord-play

Before presenting our method, we first outline the core assumptions and the threat model under which our auditing framework operates.

Assumption 3.1. *We assume the availability of correctly implemented privacy primitives (e.g., Gaussian, Laplace, Exponential mechanisms) for which the privacy accounting rules under composition are well established, and we have access to a correct accountant for them.*

Assumption 3.2. *We assume access to a function that snapshots and restores the state of the system's pseudo-random number generator. We assume that the developers provide this function, as it is becoming increasingly standard for reproducibility purposes.*

Threat Model: Gray-Box Auditing. We adopt a gray-box auditing model in which the adversary is not modeled as a malicious external actor, but rather as a developer who inadvertently introduces implementation errors into the processing logic. We designate this approach as "gray-box" because it relies on a selective visibility of the codebase: we require explicit access to the algorithm's control flow, the intermediate variables passed to privacy primitives, and the state of the pseudo-random number generator. However, we treat the internal implementations of the privacy primitives themselves (e.g., the specific logic within a Laplace mechanism) as a black-box. This distinction is crucial, as it allows us to leverage structural knowledge of the intermediate code.

Intuition. Our approach operates on the principle that in a correctly implemented DP algorithm, the dependency on private data should be restricted to specific, accounted-for primitives. All surrounding control flow and parameter logic must remain data independent. To verify this, we employ a strategy of *isolation*: if we freeze the outputs of the privacy primitives, the entire remaining execution path of the algorithm should be identical for any two neighboring datasets D and D'. Any deviation in the program state outside of these primitives signals that private data has leaked into the "public" logic without paying the necessary privacy budget. This reduces the complex auditing task to two straightforward verifications: (i) that the data-independent logic remains *invariant*, and (ii) that the inputs entering the privacy primitives adhere to their declared sensitivity limits.

Trace Generation and Strict Equality. To implement this verification, Re:cord-play (formalized in Algorithm 1) generates parallel execution traces by instrumenting the target algorithm $\mathcal{A}$. A central hook intercepts all calls to DP primitives. Beyond standard primitives (e.g., Laplace), our framework supports a specialized verification node denoted as `ensure_equality`. This node allows the auditor to explicitly assert that specific intermediate values, such as hyperparameters, metadata-derived constants, or loop bounds, remain equal across runs. This ensures that changes to private data do not affect data-independent components of the computation. The framework operates in two distinct phases:

Phase 1: Recording on Initial Dataset (D). First, we execute the instrumented algorithm $\mathcal{A}$ on an initial dataset, D. The instrumentation hook (Algorithm 1, line 10) operates in `RECORD` mode. For each invocation of a DP primitive $\mathcal{M}$ (or equality check), the hook logs a detailed entry into a trace $\mathcal{T}$, containing: (1) the mechanism type, (2) its parameters ρ (e.g., noise variance σ), (3) the input query data q (specifically q_D), (4) the PRNG state, and (5) the exact output o produced. This generates an ordered log of every stochastic decision point and equality assertion within the execution path.

Phase 2: Replaying on Neighboring Dataset (D'). Next, we execute $\mathcal{A}$ on a neighboring dataset, D' (lines 5-7), with the hook in `REPLAY` mode. When a primitive is called, it performs the following logic:

(1) **Verifies Control-Flow:** It checks that the current call type and order match the corresponding call in the recorded trace $\mathcal{T}$. A mismatch flags a violation, indicating the control flow is data-dependent.
(2) **Verifies Invariance:** If the node is of type `ensure_equality` (lines 27-30), it asserts that the input q matches the recorded input from the D trace. If they differ, it flags a violation. For privacy primitives, invariance is checked on non-sensitive arguments (e.g., declared noise scale).
(3) **Logs Inputs:** It logs the new inputs $q_{D'}$ into a new trace $\mathcal{T}'$.
(4) **Enforces Determinism:** It restores the PRNG state to the *post-execution* state captured during the recording phase and returns the *frozen* output from the D trace.

By forcing the mechanisms to return the same output under both D and D', we isolate the valid source of variation. Finally, the auditor compares the inputs q_D and $q_{D'}$ stored in the traces. If the empirical distance between them exceeds the declared sensitivity (and yet the accounting claimed otherwise), a sensitivity violation is flagged.

In the next section, we generalize Re:cord-playto scenarios where we do not trust the underlying implementation of the privacy primitives, enabling us to also find bugs in their accounting.

3.3 Distributional Audit: Re:cord-play-sample

The lightweight audit in Section 3.2 is a testing tool for verifying consistency and data-independence. However, it relies on Assumption 3.1: the privacy primitives ($\mathcal{M}_i$) themselves are correctly implemented and their accounting is accurate.

In this section, we lift this assumption. We consider a scenario in which the privacy primitives themselves are untrusted because they have a custom, unvetted implementation, are part of a complex legacy system, or use non-standard theoretical privacy accounting.

To handle this, we introduce **Re:cord-play-sample**, extending our lightweight framework into a *gray-box distributional audit*. Unlike black-box approaches that must treat the whole algorithm as opaque, we leverage the Re:cord-play trace to isolate each mechanism and its exact sensitive inputs, and then perform a statistical audit on each isolated component. The core idea is to move from verifying *declared* properties to empirically *estimating* the privacy loss. Our method (formalized in Algorithm 2) works in three phases:

Algorithm 1 Trace Generation via Re:cord-play

Input: Instrumented Algorithm $\mathcal{A}$, neighboring datasets D, D'
Output: Execution traces $\mathcal{T}, \mathcal{T}'$

procedure GenerateTraces($\mathcal{A}, D, D'$)
// Phase 1: Record
  $\mathcal{T} \leftarrow []$ ▷ Initialize trace for D
  SetInstrumentationMode(RECORD, $\mathcal{T}$)
  $\mathcal{A}(D)$ ▷ Run on D; hook populates $\mathcal{T}$
// Phase 2: Replay
  $\mathcal{T}' \leftarrow []$ ▷ Initialize trace for D'
  SetInstrumentationMode(REPLAY, $\mathcal{T}, \mathcal{T}'$)
  $\mathcal{A}(D')$ ▷ Run on D'; hook populates $\mathcal{T}'$
  return $\mathcal{T}, \mathcal{T}'$
end procedure

procedure OnPrimitiveCall($\mathcal{M}, \rho, q$) ▷ Instrumentation Hook
  if mode = RECORD **then**
    if $\mathcal{M}$ = ensure_equality **then**
      $o \leftarrow q$ ▷ Pass-through for equality check
    else
      $o \leftarrow \mathcal{M}(q, \rho)$ ▷ Execute real mechanism
    end if
    $s_{post} \leftarrow$ GetRNGState()
    $\mathcal{T}$.add($(\mathcal{M}, \rho, q, s_{post}, o)$) ▷ Log full state
    return o
  else if mode = REPLAY **then**
    $k \leftarrow |\mathcal{T}'| + 1$ ▷ Current call index
    if $k > |\mathcal{T}|$ **or** $\mathcal{M} \neq \mathcal{T}[k].\mathcal{M}$ **then**
      ▷ Control flow mismatch; traces are now invalid.
      return STOP
    end if
    $(\ldots, s_{post}, o_{rec}) \leftarrow \mathcal{T}[k]$ ▷ Read from D trace
    if $\mathcal{M}$ = ensure_equality **then**
      if $q \neq \mathcal{T}[k].q$ **then** ▷ Verify Invariance
        return STOP
      end if
      $\mathcal{T}'$.add($(\mathcal{M}, q)$) ▷ Log D' inputs/params
      return q
    end if
    $\mathcal{T}'$.add($(\mathcal{M}, \rho, q)$) ▷ Log D' inputs/params
    SetRNGState(s_{post}) ▷ Restore post-call RNG state
    return o_{rec} ▷ Return *frozen* output from D run
  end if
end procedure

(1) **Phase 1: Record & Replay.** See Section 3.2.
(2) **Phase 2: Sample.** With the inputs for each primitive isolated, we now treat each *individual* $\mathcal{M}_i$ as a black box. For each tuple $(\mathcal{M}_i, q_{D,i}, q_{D',i})$ in the trace $\mathcal{T}$, we perform a statistical sampling experiment. We execute $\mathcal{M}_i(q_{D,i})$ N times to generate an empirical output distribution $\hat{P}_i$, and similarly execute $\mathcal{M}_i(q_{D',i})$ N times to generate $\hat{P}'_i$.
(3) **Phase 3: Estimate & Compose.** At this stage, auditing the isolated component $\mathcal{M}_i$ reduces to a standard black-box auditing problem. As a result, our framework can incorporate any existing distributional auditing technique [47, 67, 68]. Building on this flexibility, we propose a specific auditing technique based on Privacy Loss Distributions (PLDs). From each pair of empirical distributions $(\hat{P}_i, \hat{P}'_i)$, we compute an empirical PLD $\mathcal{L}_i$ following the methodology of [26]. Given the set of empirical PLDs $\mathcal{L}_1, \ldots, \mathcal{L}_k$ for the entire execution path, we then use a numerical privacy accountant [26] to compose them and obtain a single end-to-end (ε, δ) bound for the full algorithm $\mathcal{A}$ on this specific path and input pair. Additional details on the PLD estimation procedure are provided in Section B.

This method provides a concrete, empirical estimator of the privacy loss, effectively bypassing the reliance on the algorithm's internal accounting or implementation correctness.

This approach differs from standard black-box distributional auditing, which typically requires re-executing the entire pipeline to generate inputs for each sample. Instead, we leverage the trace to fix inputs for specific privacy primitives, decoupling expensive preprocessing from the audit. By isolating each primitive, we avoid the computational cost of the full end-to-end function and focus on decomposed, cheap-to-execute privacy primitives. This efficiency is critical to our goal of providing a tool suitable for integration into CI/CD pipelines, where audits must be frequent and low-latency.

Interestingly, this framework supports hybrid composition. Because we introduce a novel way of estimating a PLD for an arbitrary mechanism without assuming a priori structure (see Section B), we can interleave *trusted* primitives with *untrusted* ones (audited empirically). This flexibility allows practitioners to allocate their computational auditing budget where it is needed most, minimizing overall testing costs while maximizing coverage.

4 Evaluation

To validate the practicality of our framework, we designed an evaluation benchmark to highlight that our proposed approach is: (i) *Effective:* Our framework detects real, subtle, and diverse classes of DP implementation bugs; and (ii) *Efficient:* Our framework is computationally lightweight. First, we detail the implementation of our auditing tool and our strategy for generating the neighboring datasets (D, D') used to trigger privacy violations. Then, we discuss our findings applying both Re:cord-play and Re:cord-play-sample.

4.1 Implementation

We implemented our framework as a Python library that integrates with the `pytest` testing ecosystem. This design choice positions privacy auditing as a standard component of the software development lifecycle, capable of running alongside unit and integration tests. The core instrumentation logic (the `OnPrimitiveCall` hook from Algorithm 1) uses context variables [74]. Crucially, our framework introduces no side effects or performance overhead in production; tracing is enabled only in the testing environment.

Figure 2 shows a fully self-contained integration example. A Python decorator marks the DP primitives, in this case, just the Laplace Mechanism (line 1). A simple `pytest` test (line 24) instantiates two neighboring datasets D and D', as well as the auditor, which is in `record` mode by default. The private algorithm

Algorithm 2 Distributional Audit via Re:cord-play-sample

Input: Algorithm $\mathcal{A}$, neighboring datasets D, D', Sample count N (for statistical audit), expected (ε, δ)
Output: The empirical privacy guarantee $\hat{\varepsilon}$ at a prespecified δ.

procedure AuditDistributional($\mathcal{A}, D, D', N$)
  Phase 1: Record & Replay (from Section 3.2)
  $\mathcal{T}, \mathcal{T}' \leftarrow$ RunRecordReplay($\mathcal{A}, D, D'$)
  if $\mathcal{T}$ contains control-flow mismatch **then**
    return STOP
  end if
  $\mathcal{L}_{list} \leftarrow []$ ▷ List to store empirical PLDs
// Phase 2: Sample
  for each mechanism call $i = 1 \dots |\mathcal{T}|$ **do**
    $(\mathcal{M}_i, q_{D,i}, q_{D',i}) \leftarrow \mathcal{T}[i]$
    $O_D, O_{D'} \leftarrow [], []$ ▷ Outputs from D and D'
    for $j = 1 \dots N$ **do** ▷ Collect N samples
      O_D.add($\mathcal{M}_i(q_{D,i})$) ▷ Run on D input
      $O_{D'}$.add($\mathcal{M}_i(q_{D',i})$) ▷ Run on D' input
    end for
// Phase 3: Estimate
    $\hat{P}_i, \hat{P}'_i \leftarrow$ BuildEmpiricalDistributions($O_D, O_{D'}$)
    $\mathcal{L}_i \leftarrow$ EstimatePLD($\hat{P}_i, \hat{P}'_i$) ▷ e.g.,[47]
    $\mathcal{L}_{list}$.add($\mathcal{L}_i$)
  end for
// Phase 3: Compose
  $\mathcal{L}_{total} \leftarrow$ NumericalAccountant($\mathcal{L}_{list}$) ▷ e.g., via [26]
  $\hat{\varepsilon} \leftarrow$ GetPrivacyBound($\mathcal{L}_{total}, \delta$)
  return $\hat{\varepsilon}$
end procedure

`private_function` (line 11) is executed twice—once on D and once on D'—with the second execution in `replay` mode (line 32). We use `ensure_equality` (line 13) to check for data-dependent changes in parameters. Afterwards, the auditor checks for mismatches in the execution trace in `validate_records` (line 37) and then starts the `distributional_audit`. Finally, the test concludes by deriving an overall privacy bound (see Appendix B) and asserting whether it exceeds the declared privacy budget. While Re:cord-play (Section 3.2) fixes the mechanism's output during replay, it also checks inputs: it calculates the empirical distance between inputs across the two runs and compares this against the declared sensitivity in `validate_records`. In the example, the input `scaled_count` differs by 2 between runs. Our framework catches the bug and pinpoints the violation. In this case, it detects a sensitivity miscalibration when the `LM` mechanism is invoked for the first time.

The example illustrates the framework's ease of integration into standard software development workflows. The integration consists of two components: (i) writing tests and (ii) marking privacy primitives in the library. For part (i), shown in Figure 2, the auditing logic requires minimal boilerplate code: the developer simply wraps the target algorithm within the Auditor scope, first in `RECORD` mode to capture the execution trace $\mathcal{T}$ on D, and subsequently in `REPLAY` mode to generate $\mathcal{T}'$ on D'. The API automatically handles internal state tracking and side-effect isolation, allowing practitioners to seamlessly perform deterministic sensitivity checks (`validate_records`) and statistical audits (`distributional_audit`) within existing testing pipelines.

Integrating our library is designed to be non-intrusive. For part (ii), developers mark privacy primitives using our `@audit_spec` decorator and check for invariance using the `ensure_equality` functions, as illustrated in Figure 2. The decorator specifies a mapping of the arguments required to check privacy violations. The `audit_spec` takes the following arguments: (i) `kind`, which sets the mechanism's name for debugging and reporting; (ii) `input_arg`, denoting the data that the mechanism applies noise to; (iii) `sensitivity_arg`, denoting an upper bound on the allowed change in `input_arg`; and (iv) `metric_fn`, which defines the distance metric used to compute sensitivity across two runs on neighboring datasets. Optionally, developers may provide (v) a privacy `accountant`. When an accountant is provided, the primitive is treated as trusted, and our tool verifies it using Re:cord-play. Otherwise, as is common for custom mechanisms, the primitive defaults to Re:cord-play-sample, triggering statistical verification. This annotation process is performed only *once* per library. In our experiments, we utilize the numerical accountant from Doroshenko et al. [26].

4.2 Datasets

In this section, we describe the data used to run our auditing. We want to stress that the goal is to keep the execution time and memory usage as small as possible.

Synthetic Data. To create a controlled and reproducible test environment, our synthetic data procedure generates an initial dataset D as a low-dimensional tabular dataset. Using a seeded pseudo-random number generator, it creates a table with a fixed number of records (e.g., $n = 200$) and a small number of categorical attributes (e.g., 2 columns). The values for these attributes are drawn uniformly at random from small, predefined integer ranges.

Neighboring Datasets. A core component of any DP audit is to generate neighboring datasets, as the definition of DP is parameterized by the relationship between D and D'. Our evaluation must test an implementation's robustness against the specific adjacency model (i.e., the definition of "neighbor") it claims to support. DP literature primarily considers two types of dataset adjacency, corresponding to different privacy models and threat scenarios:

(1) **Unbounded Adjacency (Add/Remove):** The most common definition, used when the dataset size n is not public. Two datasets D, D' are neighbors if one can be formed from the other by adding or removing a record. This models the privacy risk to an individual's data based on whether they participate in the analysis or not. The sensitivity of a query (e.g., SUM) under this model is typically its maximum possible contribution (e.g., the clipping bound C).
(2) **Bounded Adjacency (Replace-One):** This definition is standard where dataset size n is public. Two datasets D, D' are neighbors if they both have exactly n records, and D' can be formed by replacing one record in D with any other possible record from the data universe. This models the privacy risk to an individual when they change their data. The sensitivity of a query under this model can be different (e.g., up to 2C for SUM).

Figure 2: End-to-end testing example of a code with a sensitivity miscalibration.

```python
@audit_spec(
    kind="LM",
    input_arg="x",
    sensitivity_arg="sensitivity",
    metric_fn=dist_l1,
    accountant=laplacian_pl
)
def laplace_mechanism(x, sensitivity, epsilon):
    return x + sample_laplacian_noise(scale=
        sensitivity / epsilon)

def private_function(data, multiplier, epsilon)
    :
    # ensure multiplier is not data-dependent
    ensure_equality(m = multiplier)

    # sensitivity = 1 (under add/remove)
    count = len(data)
    # sensitivity = multiplier
    scaled_count = count * multiplier

    # BUG: incorrect sensitivity used
    noisy_count = laplace_mechanism(
        scaled_count, sensitivity=1, epsilon=
        epsilon)
    return noisy_count

def test_simple_audit():
    D = [0,0,0]     # D dataset
    Dp = [0,0,0,0]  # D' dataset
    eps = 1.0
    auditor = Auditor()
    with auditor:
        _ = private_function(D, 2, eps)

    auditor.set_replay()
    with auditor:
        _ = private_function(Dp, 2, eps)

    # Run Re:cord-play
    auditor.validate_records()

    # Run Re:cord-play-sample
    eps_rec = auditor.distributional_audit(
        delta=1e-6, n_samples=1e5)
    assert eps_rec <= eps, "Privacy violation"
```

Adjacency Agnosticism. A key advantage of our framework is its independence from specific adjacency definitions. Although the previous two notions are the most common, alternative notions exist, notably in graph data or machine learning [50]. Our tool enforces no specific model. Instead, it evaluates consistency against any provided neighboring pair (D, D').

Generation Strategies. To instantiate the necessary dataset pairs (D, D') for auditing, we employ a diverse set of heuristics ranging from standard mutations to adversarial edge cases. For unbounded adjacency, we randomly select candidates, while additions are generated via diverse strategies: uniform sampling, marginal sampling, or data duplication. To stress-test numerical stability and overflow handling, we inject values near floating-point limits (e.g., `max(float64)`). Furthermore, we explicitly target domain inference vulnerabilities by sampling values outside the implied min/max range of D. For bounded adjacency, we combine the add and remove operations to form neighbor pairs. As discussed in Section 5.2, dataset generation is outside the scope of this work; our framework is designed to integrate seamlessly with recent advances in this area (e.g., [14]).

4.3 Auditing Results

Before presenting our results, we clarify two points: (i) *Intent*: Our goal is to motivate the adoption of our automated gray-box testing framework, not to criticize library maintainers. The subtleties of DP logic make manual verification challenging, even for experienced teams. (ii) *Completeness*: For each audited implementation, we terminate the audit at the first detected bug. Consequently, our results are not exhaustive; instead, they are meant to illustrate the practical applicability of our framework with minimal engineering effort, rather than to catalog all potential issues.

Reproducibility. We ensure reproducibility by reporting the specific Git commit hash for every codebase audited and associated links in Table 2 of the Appendix. This list serves as the ground truth for our findings, allowing independent verifiers to inspect the exact logic present at the time of the audit.

We selected diverse target implementations from widely used, open-source DP libraries. Our results are summarized in Table 1, with details in Section C. We identified privacy violations ranging from subtle preprocessing flaws to misapplication of parameters. Below, we detail specific bugs and their implications.

4.3.1 SmartNoise SDK. We audited the SmartNoise SDK's private covariance matrix implementation, a key component of algorithms such as Logistic Regression. We identified a bug (via Re:cord-play) where the covariance computation incorrectly uses the *unprocessed* data, even after a sanitization step was intended to be applied. As shown in Figure 3, the function creates `newdata` by sanitization but then computes `trueval` using the original `data`. This is a *sensitivity miscalibration*: the declared sensitivity `self.sens` is calculated assuming censored data, but an uncensored outlier can cause the true sensitivity to be arbitrarily large. Our framework detects this in a case where a neighboring dataset D' contains large outliers. The Re:cord-play hook logs the data-dependent inputs to the noise primitive, q_D and $q_{D'}$. In the validation phase, the framework computes the empirical distance $\|q_D - q_{D'}\|_1$ and finds it is greater than the *declared* sensitivity `self.sens`, returning an error.

4.3.2 Smartnoise SQL. We audited the SmartNoise SQL framework and uncovered a miscalculation in the privacy accountant via Re:cord-play-sample. As shown in Figure 4, the implementation of the homogeneous privacy odometer omits the $\log(1/\delta)$ term required by the advanced composition bound of [31, 51], replacing

Figure 3: Simplified code for the Sensitivity Bug in the SmartNoise SDK Covariance estimation. The declared sensitivity `self.sens` is calculated assuming censored data, but the `covar` function receives the original, uncensored `data`.

```
def release(self, data):
    newdata = censor_data(data)
    newdata = fill_missing(newdata)
    ...
     BUG: 'data' is used instead of 'newdata'
     trueval = covar(data.values.T, self.intercept)
    scale = self.sens / self.epsilon
    val = np.array(true_val) + dp_noise(n=len(
        true_val), noise_scale=scale)
    return list(val)
```

Figure 4: SmartNoise SQL missing $\log(1/\delta)$ factor in the homogeneous odometer.

```
class Odometer:
    def spent(self):
        ...
        BUG: equation missing log term around (1/tol)
        optimal_b = optimal_left_side + epsilon * np.sqrt(
                        2 * self.k * (1/tol)
```

it with a linear $1/\delta$ factor. This error dramatically inflates the "optimal" composition term b, causing it to grow by orders of magnitude for standard parameter ranges. As a result, the intended advanced composition bound is never applied: the accountant always falls back to more conservative alternatives, yielding substantially looser privacy guarantees than theoretically possible. This example highlights a notable capability of our framework: it can be used to detect implementation bugs that are not necessarily privacy violations. By comparing our estimated privacy loss with the accountant's reported upper bound, a significant discrepancy signals a potential bug, even if no privacy violation occurs.

4.3.3 Synthcity. We audited the PrivBayes implementation [95] and found two distinct bugs using Re:cord-play. First, as shown in Figure 5, the code uses the output of the Exponential Mechanism (`candidate_idx`) to index a *private*, un-noised list (`mutual_info_list`), and then uses this private value in a public `if` statement. This constitutes an *Invariant Violation*, as the execution path (whether the `if` block is entered) leaks information. Our framework catches this by setting an `ensure_equality` node on the right-hand term of the `if` condition. During `REPLAY` on D', the hook (Algorithm 1) detects that the sequence of mechanism calls differs from the trace on D. Second, in the utility function (`_laplace_noise_parameter` in Figure 5), we found a *Noise Miscalibration* bug. The Laplace noise scale numerator can become zero if `self.K == n_features`. This disables noise addition entirely, releasing the true value. This bug is detected by both Re:cord-play and Re:cord-play-sample, which report an ε value larger than the specified one. We also audited their AIM [61] implementation and found no issues to report.

Figure 5: Simplified code for the two bugs in Synthcity's DP Bayes algorithm. First, the output of the Exponential Mechanism, `candidate_idx`, is used to index into a private list, and the result is used in a non-private `if` check. Also, `self.K` (e.g., the number of parents) equals `n_features`, and the noise scale becomes zero, completely disabling privacy protection.

```
def _greedy_bayes(self, data):
        ...
        sampling_distribution = self.
            _exponential_mechanism(
            data,
            parents_pair,
            mutual_info)
        candidate_idx = np.random.choice(
            list(range(len(mutual_info))),
            p=sampling_distribution)
        sampled_pair = parents_pair[candidate_idx]
         BUG: indexed in private data
         if self.mi_thresh >= mutual_info[candidate_idx]:
            sampled_pair = network_edge(
                sampled_pair.feature, parents=[])
        ...

def _laplace_noise_parameter(self, n_items: int,
    n_features: int) -> float:
     BUG: self.K can be equal to n_features
     return 2*(n_features - self.K) / (n_items * self.epsilon)
```

4.3.4 Diffprivlib. We audited the `LinearRegression` model, which fits an ordinary least-squares linear regression to training data in a differentially private manner. We found a *Sensitivity Miscalibration* via Re:cord-play in the `_construct_regression_obj` function. For the diagonal values of the quadratic term in the squared error, i.e. `mono_coef_2[i, i]`, the sensitivity is the squared maximum of the lower and upper bound of the feature. However, as shown in Figure 6, only the lower bound is being used. Our framework caught this by checking whether the empirical sensitivity exceeds the recorded sensitivity.

We also uncovered an invariance violation in `diffprivlib`'s `LogisticRegression` classifier via Re:cord-play. Internally, the implementation infers the label set via `classes = np.unique(y)`, so under the add/remove adjacency model, the presence or absence of a single sample from a rare class can change `classes` between neighboring datasets. In our Re:cord-play audit, this causes a control-flow divergence and violates our invariance check. Conceptually, this leaks whether a given label appears at least once in the training data (label-space domain inference). Instead, the set of possible classes should be treated as public and set by the user.

We found no issues to report in the `histogram` tool, `GaussianNB`, `StandardScaler`, `PCA`, and `KMeans` models.

4.3.5 Private-PGM. We audited the JAM [34] algorithm, which utilizes the bounded DP model. Under this definition, changing a single record shifts a marginal histogram by at most 2 in ℓ_1-distance. JAM selects marginals via the Exponential Mechanism, using the

Figure 6: Simplified code of the Sensitivity Miscalibration bug in Diffprivlib's `_construct_regression_obj` function. The computation might incorrectly result in zero and is passed into the Laplace mechanism.

```
def _construct_regression_obj(X, y, bounds_X,
    bounds_y, epsilon, alpha, random_state):
    ...
    for i in range(n_features):
        BUG: lower bound is used twice
        sensitivity = np.max(np.abs([bounds_X[0][i],
                      bounds_X[0][i]])) ** 2
        mech = LaplaceFolded(epsilon=local_epsilon
            , sensitivity=sensitivity, lower=0,
            upper=float("inf"), random_state=
            random_state)
        mono_coef_2[i, i] = mech.randomise(coefs
            [2][i, i])
    ...
```

Figure 7: The accounting bug and the data-dependent control flow bug raised in the MOSTLY AI codebase.

```
def analyze_reduce_numeric(...
    value_protection: bool = True,
    value_protection_epsilon: float | None = None,
    ...) -> dict:
    BUG: both mechanisms spend the same privacy budget
    ... = dp_approx_bounds(..., value_protection_epsilon)
        ...
    ... = dp_non_rare(..., value_protection_epsilon)
    ...

def _analyze_reduce_seq_len(....)
    BUG: np.any(lengths != 1) is a non-private measurement
    if value_protection_epsilon is not None
        and np.any(lengths != 1):
        ... = dp_quantiles(...)
    # dp_quantiles skipped otherwise
```

scoring function:

$$\texttt{model\_error}[m] - \texttt{pub\_errors}[m]$$

Both terms depend on the private inputs and can shift by up to 2. Because these changes can occur in opposite directions, the true sensitivity is 4 (i.e., $|\Delta(\texttt{model\_error}[m] - \texttt{pub\_errors}[m])| \leq 4$). The implementation, however, underestimates this value, leading to a sensitivity miscalibration bug detected via Re:cord-play. This appears to be a regression introduced during the migration to the Private-PGM package, as the original research code correctly sets `score_sensitivity = 4`, see Fuentes et al. [34]. We have also audited their MWEM+PGM, Gaussian+AppGM, MST, and AIM algorithms and found no issues.

4.3.6 MOSTLY AI Engine. Our audit of the MOSTLY AI synthetic data engine identified two classes of issues: incorrect budget composition (detected via Re:cord-play-sample) and data-dependent control flow (detected via Re:cord-play). The numeric encoder applies two separate DP mechanisms to the same column: one for approximating bounds and one for identifying non-rare values, each spending the same amount of budget. Only a single budget is accounted for, although both routines add Laplace noise and therefore each consumes ε. In cases where both branches are activated (e.g., low-cardinality numeric columns), the actual privacy cost is 2ε even though the system reports ε.

Additionally, we observed data-dependent branching, where control-flow decisions rely on raw, non-private statistics. For example, the decision to invoke the private quantile routine depends on whether any sequence length exceeds one (Figure 7). An observer can therefore infer the existence of multi-step sequences solely from which mechanisms are called, as detected by our invariance check.

A similar pattern appears in the numeric encoder, where the unique value count is computed non-privately and compared to a threshold (e.g., < 100) to decide whether to compute per-value counts. Crossing this alters the mechanisms invoked, revealing if the cardinality lies above or below the boundary.

Figure 8: Simplified logic from Opacus's `make_private` function. The `sample_rate` is derived from `len(data_loader.dataset)`, which is a private quantity under the add/remove adjacency model, leading to a parameter instability bug.

```
def make_private(...data_loader: DataLoader...):
    ...
    sample_rate = 1 / len(data_loader)
    BUG: size of the dataset is private under add/remove
    expected_batch_size = int(len(data_loader.dataset)
                              * sample_rate)
    ...
```

4.3.7 Opacus. We audited the `make_private` function, which instruments a model for DP training. As shown in Figure 8, the variable `expected_batch_size` is derived from the private variable `len(data_loader.dataset)`. Under unbounded adjacency (assumed by Poisson sampling in the Subsampled Gaussian Mechanism [8, 9, 52]), the dataset size n is private. Using it for normalization leaks n into the optimizer. While privacy loss decreases as n increases, this constitutes a violation for small datasets, detected via Re:cord-play. Notably, a similar issue was reported over two years ago but remains unpatched [83]. We found no other bugs in their implementation.

4.3.8 dpmm. We audited the `dpmm` library and uncovered an invariance violation in both the `AIM` and `PrivBayes` algorithms via Re:cord-play. When no domain is pre-specified for the marginals, the implementation derives it directly from the private data via

```
_domain = (df.astype(int).max(axis=0) + 1).to_dict()
```

without accounting for it in the privacy budget. The presence of a single record with a large feature value can thus change the inferred domain across neighboring datasets, altering the marginals' supports. In Re:cord-play, this manifests as an invariance violation between the Record and Replay runs. A robust DP implementation should treat the domain as public (provided by the user or a safe default), or estimate it through an explicit differentially private primitive rather than inferring it directly from the private data.

4.3.9 Vulnerability to pathological inputs. We also audited the robustness of the implementations against pathological inputs, specifically $\pm\infty$ and `NaN` values. Correctly handling these inputs under differential privacy is often overlooked, however, they often bypass standard clipping checks (e.g., standard comparison operators with `NaN` often yield false, effectively skipping bounds enforcement), leading to undefined behavior or information leakage. The results, summarized in Table 3, indicate a widespread lack of defensive programming in the Python ecosystem, leading to either unhandled exceptions or the propagation of `NaN` values into the final output, leaking properties of the input data. Notably, *Google Differential Privacy* is the only library among those tested that consistently handles these inputs correctly, either by filtering them out or raising a controlled error before the privacy budget is consumed.

4.3.10 Additional evaluations and scope. We also audited OpenDP's Python bindings for MST and AIM [78], Google Differential Privacy's Clustering algorithm [20], Jax Privacy [10], and PipelineDP [72]. Our framework did not detect any privacy violations in these libraries. We note that our evaluation scope is currently restricted to local Python execution models. Consequently, we excluded libraries implemented in other languages or frameworks that rely on distributed computing backends, such as Tumult Analytics [12] and Privacy on Beam [39]. As noted in Section 5, adapting our framework to distributed environments represents an important direction for future work.

5 Discussion

Our audit of major libraries suggests that ensuring the correctness of mathematical derivations is necessary but not sufficient. While the core privacy mechanisms may be theoretically sound, our findings highlight a distinct class of privacy-impacting bugs that warrant investigation. These violations arise not from the privacy proofs themselves, but from the friction between abstract privacy definitions and practical software engineering. Based on the common failure modes detected by our auditing framework, we offer the following recommendations for developers of DP software.

Prioritize Vetted DSLs. In line with prior work [7, 82], our auditing results demonstrate that ad hoc implementations of differential privacy are notoriously error-prone. Consequently, whenever architectural constraints allow, practitioners should prioritize the use of vetted DSLs such as OpenDP [71] or Tumult Analytics [12]. These frameworks abstract away the complexity of sensitivity tracking and noise calibration, significantly reducing the surface area for the types of implementation bugs we have identified.

Table 1: Summary of vulnerabilities detected across various DP libraries. A (green) cross ✗ indicates that our framework did not catch a bug, while a (red) checkmark ✓ indicates that the framework found a bug.

Implementation	Sensitivity Miscalibration	Noise Miscalibration	Invariant Violation
SmartNoise Synth			
Logistic Regression	✓	✗	✗
SmartNoise SQL			
Odometers	✗	✓	✗
dpmm			
PrivBayes	✗	✗	✓
MST	✗	✗	✗
AIM	✗	✗	✓
Private PGM			
MWEM+PGM	✗	✗	✗
JAM	✓	✗	✗
Gaussian+AppGM	✗	✗	✗
MST	✗	✗	✗
AIM	✗	✗	✗
Synthcity			
PrivBayes	✗	✓	✓
AIM	✗	✗	✗
Diffprivlib			
Linear Regression	✓	✗	✗
Logistic Regression	✗	✗	✓
MostlyAI			
Synthetic Data Engine	✗	✓	✓
Opacus			
DP-SGD	✗	✗	✓
Jax-Privacy			
DP-SGD	✗	✗	✗
Google Differential Privacy			
Clustering	✗	✗	✗
PipelineDP			
DP Engine	✗	✗	✗
OpenDP			
MST	✗	✗	✗
AIM	✗	✗	✗

Unclear neighboring definitions. A recurring source of issues is the ambiguity regarding the adjacency relationship (i.e., the definition of a neighboring dataset). The lack of clear statements can lead to the composition of incompatible mechanisms or incorrect usage. Maintainers should not assume that users understand whether the system provides add/remove or replace-one privacy. Libraries should explicitly encode the adjacency type in the privacy accountant object. A good example is the Google DP accounting library, which requires selecting a neighboring definition [26, 40], as does Tumult Analytics (see [84]). If a system operates under replace-one, the code can treat the dataset size n as a public constant. However, under the add/remove model, developers must treat n as private information. Our audit of Opacus (Figure 8) showed that calculating normalization factors based on `len(dataset)` is a dangerous default. We recommend requiring users to specify the dataset size

explicitly or, if it is not public, privately releasing it (at a privacy cost).

Data-dependent code isolation. We recommend structuring DP algorithms by separating data-dependent and data-independent components. All data-dependent preprocessing (clipping, bounding) should be performed in a distinct phase, and the resulting restricted data should then be passed to an isolated core privacy block that handles aggregation and noise addition. Post-processing can be performed afterward. This separation helps identify any unintended data-dependent dependencies outside the privacy mechanism. Additionally, it simplifies the deployment of our auditing technique, allowing us to instrument functions that access private data separately from those that should remain invariant under a neighboring dataset change.

5.1 Limitations

On the need for Assumption 3.2. This assumption is central to our methodology and a standard requirement for deterministic testing and debugging in scientific computing. Our goal is to verify that any two execution paths on D and D' are control-flow invariant, with differences only arising from the sensitive inputs to the DP primitives. Without controlling randomness, this verification is impossible. A stochastic decision (e.g., `if np.random.rand() > 0.5`) taken after the first DP mechanism call could diverge between the "record" and "replay" runs purely by chance, causing our tool to incorrectly flag a data-dependent control-flow bug. Importantly, DP libraries introduce additional sources of randomness beyond the DP mechanisms themselves—for instance, [20] uses locality-sensitive hashing [21], while Opacus and TensorFlow sample minibatches to compute stochastic gradients.

We stress that Assumption 3.2 is practical for the vast majority of DP libraries, which are written in high-level languages like Python or C++ and rely on standard, stateful PRNGs (e.g., `numpy.random`, `torch.manual_seed`). We however acknowledge two limitations. First, our method cannot audit systems that draw randomness from non-deterministic sources like `/dev/random` or specialized cryptographic hardware (e.g., Intel RDRAND), unless these sources can be mocked or patched at a lower level. Second, its application to highly concurrent or distributed systems (such as Tumult Analytics [12] that relies on Apache Spark [91]) is non-trivial, as it would require controlling and synchronizing the PRNG state across multiple threads or processes, which is outside our current scope.

What does our audit mean? It is important to interpret the results of our audit within the appropriate context. Our Re:cord-play framework is a testing tool, not a formal verifier. This distinction introduces a fundamental trade-off: the methodology is sound but not complete:

(1) **Soundness (Structural Correctness):** If our tool flags a bug, it corresponds to a concrete implementation error. While it is theoretically possible to construct a data-dependent computation outside privacy mechanisms that does not alter the final output distribution (e.g., a redundant operation), such patterns violate the principle of data-independence outside trusted primitives. Thus, the tool provides a proof-of-failure regarding the *structural* integrity of the algorithm.

(2) **Incompleteness (False Negatives):** A successful audit (i.e., finding no bugs) does not constitute a proof of correctness. It only indicates that, for the specific neighboring datasets (D, D') and the execution paths tested, no privacy violations were detected.

This limitation, inherent to empirical auditing, is a manifestation of the test coverage problem. A bug may go undetected if it only appears on an untested execution path (i.e., triggered by a dataset D with specific properties) or if it depends on a specific relationship between D and D' not included in the test suite. Additionally, bugs may be missed if they arise from rare stochastic events, such as low-probability failures (e.g., 0.01%). Random sampling might not trigger the specific conditions required to observe the failure. Thus, while our tool detects common and critical bugs, it cannot prove the absence of all bugs.

Interpreting the errors. While our framework offers much higher granularity than black-box auditing by pinpointing the specific primitive call where a violation occurs, it does not automatically reveal the root cause. For instance, if Re:cord-play flags a sensitivity violation, it conclusively shows that the inputs to a mechanism exceeded the declared bounds, but it cannot explain *why* the preprocessing logic failed to enforce them. The error might result from a missing clipping step or an incorrect variable reference. Our tool provides the necessary evidence, but identifying the underlying flaw in the source code remains a manual debugging task for the developer.

5.2 Future work

Extending to other types of accounting and composition. Currently, our framework assumes adaptive composition [32], flattening the algorithm's structure into a sequence of mechanism calls. While sufficient for basic sequential composition, this view obscures structural dependencies required for other accounting techniques. For example, the trace alone does not reveal whether two mechanisms operate on disjoint data subsets, which would permit *parallel composition* [29]. Similarly, advanced methods such as privacy amplification by iteration [33] rely on the algorithm's semantic structure and the precise sequence of mechanisms. Future work aims to enhance the instrumentation to support these more complex composition techniques.

On the need for `ensure_equality` *nodes.* Currently, the placement of these verification nodes requires manual annotation: developers must explicitly identify which variables (e.g., hyperparameters, loop counters, configuration settings) should remain invariant. Although effective, this process is susceptible to human oversight. In future iterations, we aim to automate this discovery through advanced dynamic instrumentation, enabling the framework to infer and monitor public invariants automatically. This would also reduce modifications to the audited codebase while maintaining robust coverage against data-dependent control-flow leaks.

Advanced dataset crafting. While our current framework relies on basic dataset crafting strategies, we aim to extend the test suite to support advanced dataset crafting techniques (see for example [14]) to enhance bug discovery and testing coverage.

6 Conclusion

In this work, we introduce a novel "gray-box" auditing paradigm that narrows the gap between theoretical claims and the practical guarantees of DP libraries. By shifting the focus from computationally intensive output analysis to precise internal-state inspection, Re:cord-play enables developers to verify sensitivity claims and detect data-dependent behavior with the speed and reliability of a unit test. For scenarios involving unverified components, Re:cord-play-sample bridges deterministic checking and statistical validation, providing a fine-grained assessment of complex pipelines. The discovery of real bugs in major DP libraries highlights the need for improved tooling in the privacy ecosystem. By releasing our framework as an open-source package, we hope to help practitioners to build privacy software that faithfully fulfills its theoretical promises.

7 Ethics and Responsible Disclosure

During our evaluation of our testing framework, we identified privacy vulnerabilities in existing open-source differential privacy libraries. We followed standard responsible disclosure protocols by reporting these findings to the respective library maintainers. We have established a 60-day disclosure embargo to allow sufficient time for patches to be implemented before the details of these vulnerabilities are made public. We will update the text to include information on how the bugs were addressed.

All experiments were conducted locally using purely synthetic data.

Acknowledgments

This work is partially supported by grant ANR-20-CE23-0015 (Project PRIDE), ANR 22-PECY-0002 IPOP (Interdisciplinary Project on Privacy) project of the Cybersecurity PEPR. This work was performed using HPC resources from GENCI–IDRIS (Grant 2023-AD011014018R3).

References

[1] Martín Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 308–318. https://doi.org/10.1145/2976749.2978318 arXiv:1607.00133 [cs, stat].

[2] John M. Abowd. 2018. The U.S. Census Bureau Adopts Differential Privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, London United Kingdom, 2867–2867. https://doi.org/10.1145/3219819.3226070

[3] Ahmet Aktay, Shailesh Bavadekar, Gwen Cossoul, John Davis, Damien Desfontaines, Alex Fabrikant, Evgeniy Gabrilovich, Krishna Gadepalli, Bryant Gipson, Miguel Guevara, et al. 2020. Google COVID-19 community mobility reports: anonymization process description (version 1.1). *arXiv preprint arXiv:2004.04145* (2020).

[4] Meenatchi Sundaram Muthu Selva Annamalai. 2024. It's our loss: No privacy amplification for hidden state DP-SGD with non-convex loss. *arXiv preprint arXiv:2407.06496* (2024).

[5] Meenatchi Sundaram Muthu Selva Annamalai, Borja Balle, Jamie Hayes, and Emiliano De Cristofaro. 2024. To shuffle or not to shuffle: Auditing DP-SGD with shuffling. *arXiv preprint arXiv:2411.10614* (2024).

[6] Meenatchi Sundaram Muthu Selva Annamalai, Borja Balle, Jamie Hayes, Georgios Kaissis, and Emiliano De Cristofaro. 2025. The Hitchhiker's Guide to Efficient, End-to-End, and Tight DP Auditing. *arXiv preprint arXiv:2506.16666* (2025).

[7] Meenatchi Sundaram Muthu Selva Annamalai, Georgi Ganev, and Emiliano De Cristofaro. 2024. " What do you want from theory alone?" experimenting with tight auditing of differentially private synthetic data generation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4855–4871.

[8] Borja Balle, Gilles Barthe, and Marco Gaboardi. 2018. Privacy Amplification by Subsampling: Tight Analyses via Couplings and Divergences. *CoRR* abs/1807.01647 (2018). arXiv:1807.01647 http://arxiv.org/abs/1807.01647

[9] Borja Balle, Gilles Barthe, and Marco Gaboardi. 2020. Privacy Profiles and Amplification by Subsampling. *Journal of Privacy and Confidentiality* 10, 1 (Jan. 2020). https://doi.org/10.29012/jpc.726

[10] Borja Balle, Leonard Berrada, Zachary Charles, Christopher A Choquette-Choo, Soham De, Vadym Doroshenko, Dj Dvijotham, Andrew Galen, Arun Ganesh, Sahra Ghalebikesabi, Jamie Hayes, Peter Kairouz, Ryan McKenna, Brendan McMahan, Aneesh Pappu, Natalia Ponomareva, Mikhail Pravilov, Keith Rush, Samuel L Smith, and Robert Stanforth. 2025. *JAX-Privacy: Algorithms for Privacy-Preserving Machine Learning in JAX*. http://github.com/google-deepmind/jax_privacy

[11] James Bell-Clark, Adrià Gascón, Baiyu Li, Mariana Raykova, and Amrita Roy Chowdhury. 2025. Vϵrity: Verifiable Local Differential Privacy. IACR Cryptology ePrint Archive, Report 2025/851. https://eprint.iacr.org/2025/851

[12] Skye Berghel, Philip Bohannon, Damien Desfontaines, Charles Estes, Sam Haney, Luke Hartman, Michael Hay, Ashwin Machanavajjhala, Tom Magerlein, Gerome Miklau, Amritha Pai, William Sexton, and Ruchit Shrestha. 2022. Tumult Analytics: a robust, easy-to-use, scalable, and expressive framework for differential privacy. https://doi.org/10.48550/arXiv.2212.04133 arXiv:2212.04133 [cs].

[13] 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* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 508–524. https://doi.org/10.1145/3243734.3243863

[14] 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)*. 391–409. https://doi.org/10.1109/SP40001.2021.00081

[15] Ari Biswas and Graham Cormode. 2023. Interactive Proofs For Differentially Private Counting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Copenhagen Denmark, 1919–1933. https://doi.org/10.1145/3576915.3616681

[16] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

[17] Nicholas Carlini, Steve Chien, Milad Nasr, Shuang Song, Andreas Terzis, and Florian Tramer. 2022. Membership Inference Attacks From First Principles. https://doi.org/10.48550/arXiv.2112.03570 arXiv:2112.03570 [cs].

[18] Tudor Cebere, Aurélien Bellet, and Nicolas Papernot. 2024. Tighter privacy auditing of dp-sgd in the hidden state threat model. *arXiv preprint arXiv:2405.14457* (2024).

[19] Karan Chadha, Matthew Jagielski, Nicolas Papernot, Christopher Choquette-Choo, and Milad Nasr. 2024. Auditing private prediction. *arXiv preprint arXiv:2402.09403* (2024).

[20] Alisa Chang, Badih Ghazi, Ravi Kumar, and Pasin Manurangsi. 2021. Locally Private k-Means in One Round. (2021).

[21] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (Brooklyn, New York, USA) *(SCG '04)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/997817.997857

[22] Damien Desfontaines and Balázs Pejó. 2019. Sok: differential privacies. *arXiv preprint arXiv:1906.01337* (2019).

[23] 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.

[24] Yevgeniy Dodis, Adriana López-Alt, Ilya Mironov, and Salil Vadhan. 2012. Differential Privacy with Imperfect Randomness. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Vol. 7417. Springer Berlin Heidelberg, Berlin, Heidelberg, 497–516. https://doi.org/10.1007/978-3-642-32009-5_29 Series Title: Lecture Notes in Computer Science.

[25] Jinshuo Dong, Aaron Roth, and Weijie J Su. 2022. Gaussian differential privacy. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 84, 1 (2022), 3–37.

[26] Vadym Doroshenko, Badih Ghazi, Pritish Kamath, Ravi Kumar, and Pasin Manurangsi. 2022. Connect the dots: Tighter discrete approximations of privacy loss distributions. *arXiv preprint arXiv:2207.04380* (2022).

[27] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *Advances in Cryptology - EUROCRYPT 2006*, Serge Vaudenay (Ed.). Vol. 4004. Springer Berlin Heidelberg, Berlin, Heidelberg, 486–503. https://doi.org/10.1007/11761679_29 Series Title: Lecture Notes in Computer Science.

[28] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography* (New York, NY) *(TCC'06)*. Springer-Verlag, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

[29] Cynthia Dwork and Aaron Roth. 2013. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3-4

(2013), 211–407. https://doi.org/10.1561/0400000042
[30] Cynthia Dwork and Guy N. Rothblum. 2016. Concentrated Differential Privacy. https://doi.org/10.48550/arXiv.1603.01887 arXiv:1603.01887 [cs].
[31] Cynthia Dwork, Guy N Rothblum, and Salil Vadhan. 2010. Boosting and differential privacy. In *2010 IEEE 51st annual symposium on foundations of computer science*. IEEE, 51–60.
[32] Cynthia Dwork, Guy N. Rothblum, and Salil Vadhan. 2010. Boosting and Differential Privacy. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE, Las Vegas, NV, USA, 51–60. https://doi.org/10.1109/FOCS.2010.12
[33] Vitaly Feldman, Ilya Mironov, Kunal Talwar, and Abhradeep Thakurta. 2018. Privacy Amplification by Iteration. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. 521–532. https://doi.org/10.1109/FOCS.2018.00056 arXiv:1808.06651 [cs, stat].
[34] Miguel Fuentes, Brendan C. Mullins, Ryan McKenna, Gerome Miklau, and Daniel Sheldon. 2024. JAM: Joint Adaptive Measurements – Official Code Repository. https://github.com/Miguel-Fuentes/JAM_AiStats. Accessed: 2025-02-06.
[35] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 357–370. https://doi.org/10.1145/2429069.2429113
[36] Georgi Ganev, Meenatchi Sundaram Muthu Selva Annamalai, Sofiane Mahiou, and Emiliano De Cristofaro. 2025. The Importance of Being Discrete: Measuring the Impact of Discretization in End-to-End Differentially Private Synthetic Data. *arXiv preprint arXiv:2504.06923* (2025).
[37] Anna C Gilbert and Audra McMillan. 2018. Property testing for differential privacy. In *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 249–258.
[38] S Goldwasser, S Micali, and C Rackoff. 1985. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (Providence, Rhode Island, USA) *(STOC '85)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/22145.22178
[39] Google. 2025. Privacy on Beam: Differential Privacy in Beam. GitHub repository. https://github.com/google/differential-privacy/tree/main/privacy-on-beam Accessed: 2025-11-29.
[40] Google. n.d.. Differential Privacy Library (Python): dp_event.py. https://github.com/google/differential-privacy/blob/main/python/dp_accounting/dp_accounting/dp_event.py Accessed: 2025-11-29.
[41] Sivakanth Gopi, Yin Tat Lee, and Lukas Wutschitz. 2021. Numerical Composition of Differential Privacy. http://arxiv.org/abs/2106.02848 arXiv:2106.02848 [cs].
[42] Samuel Haney, Damien Desfontaines, Luke Hartman, Ruchit Shrestha, and Michael Hay. 2022. Precision-based attacks and interval refining: how to break, then fix, differential privacy on finite computers. *arXiv preprint arXiv:2207.13793* (2022).
[43] Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. 2019. Diffprivlib: The IBM Differential Privacy Library. https://doi.org/10.48550/arXiv.1907.02444 arXiv:1907.02444 [cs].
[44] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V. Pearson, Dietrich A. Stephan, Stanley F. Nelson, and David W. Craig. 2008. Resolving Individuals Contributing Trace Amounts of DNA to Highly Complex Mixtures Using High-Density SNP Genotyping Microarrays. *PLoS Genetics* 4, 8 (Aug. 2008), e1000167. https://doi.org/10.1371/journal.pgen.1000167
[45] Gary Howarth. 2025. *A Community-Driven Differential Privacy Deployment Registry.* Technical Report NIST IR 8588 ipd. National Institute of Standards and Technology, Gaithersburg, MD. NIST IR 8588 ipd pages. https://doi.org/10.6028/NIST.IR.8588.ipd
[46] Thomas Humphries, Simon Oya, Lindsey Tulloch, Matthew Rafuse, Ian Goldberg, Urs Hengartner, and Florian Kerschbaum. 2023. Investigating Membership Inference Attacks under Data Dependencies. http://arxiv.org/abs/2010.12112 arXiv:2010.12112 [cs].
[47] Matthew Jagielski, Jonathan Ullman, and Alina Oprea. 2020. Auditing Differentially Private Machine Learning: How Private is Private SGD? arXiv:2006.07709 [cs.CR] https://arxiv.org/abs/2006.07709
[48] Jiankai Jin, Eleanor McMurtry, Benjamin IP Rubinstein, and Olga Ohrimenko. 2022. Are we there yet? timing and floating-point attacks on differential privacy systems. In *2022 IEEE Symposium on security and privacy (SP)*. IEEE, 473–488.
[49] Noah Johnson, Joseph P. Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (Jan. 2018), 526–539. https://doi.org/10.1145/3187009.3177733
[50] Peter Kairouz, Brendan Mcmahan, Shuang Song, Om Thakkar, Abhradeep Thakurta, and Zheng Xu. 2021. Practical and Private (Deep) Learning Without Sampling or Shuffling. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 5213–5225. https://proceedings.mlr.press/v139/kairouz21b.html
[51] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. 2015. The composition theorem for differential privacy. In *International conference on machine learning*. PMLR, 1376–1385.
[52] Shiva Prasad Kasiviswanathan, Homin K Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2011. What can we learn privately? *SIAM J. Comput.* 40, 3 (2011), 793–826.
[53] Antti Koskela and Jafar Aco Mohammadi. 2025. Auditing Differential Privacy Guarantees Using Density Estimation. In *2025 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 1007–1026.
[54] Ivona Krchova, Mariana Vargas Vieyra, Mario Scriminaci, and Andrey Sidorenko. 2025. Democratizing Tabular Data Access with an Open-Source Synthetic-Data SDK. arXiv:2508.00718 [cs.LG] https://arxiv.org/abs/2508.00718
[55] Min Lyu, Dong Su, and Ninghui Li. 2017. Understanding the sparse vector technique for differential privacy. *Proceedings of the VLDB Endowment* 10, 6 (Feb. 2017), 637–648. https://doi.org/10.14778/3055330.3055331
[56] Sofiane Mahiou, Amir Dizche, Reza Nazari, Xinmin Wu, Ralph Abbey, Jorge Silva, and Georgi Ganev. 2025. dpmm: Differentially Private Marginal Models, a Library for Synthetic Tabular Data Generation. https://doi.org/10.48550/arXiv.2506.00322 arXiv:2506.00322 [cs].
[57] Saeed Mahloujifar, Luca Melis, and Kamalika Chaudhuri. 2024. Auditing f-Differential Privacy in One Run. *arXiv preprint arXiv:2410.22235* (2024).
[58] Saeed Mahloujifar, Alexandre Sablayrolles, Graham Cormode, and Somesh Jha. 2022. Optimal Membership Inference Bounds for Adaptive Composition of Sampled Gaussian Mechanisms. http://arxiv.org/abs/2204.06106 arXiv:2204.06106 [cs].
[59] Ryan McKenna, Gerome Miklau, and Daniel Sheldon. 2021. Winning the NIST Contest: A scalable and general approach to differentially private synthetic data. https://doi.org/10.48550/arXiv.2108.04978 arXiv:2108.04978 [cs].
[60] Ryan McKenna, Gerome Miklau, and Daniel Sheldon. 2021. Winning the NIST Contest: A scalable and general approach to differentially private synthetic data. *Journal of Privacy and Confidentiality* 11, 3 (2021).
[61] Ryan McKenna, Brett Mullins, Daniel Sheldon, and Gerome Miklau. 2024. AIM: An Adaptive and Iterative Mechanism for Differentially Private Synthetic Data. https://doi.org/10.48550/arXiv.2201.12677 arXiv:2201.12677 [cs].
[62] Frank McSherry and Kunal Talwar. 2007. Mechanism Design via Differential Privacy. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*. 94–103. https://doi.org/10.1109/FOCS.2007.66 ISSN: 0272-5428.
[63] Ilya Mironov. 2012. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 650–661. https://doi.org/10.1145/2382196.2382264
[64] Ilya Mironov. 2017. Renyi Differential Privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 263–275. https://doi.org/10.1109/CSF.2017.11 arXiv:1702.07476 [cs].
[65] Jack Murtagh and Salil Vadhan. 2016. The Complexity of Computing the Optimal Composition of Differential Privacy. In *Theory of Cryptography*, Eyal Kushilevitz and Tal Malkin (Eds.). Vol. 9562. Springer Berlin Heidelberg, Berlin, Heidelberg, 157–175. https://doi.org/10.1007/978-3-662-49096-9_7 Series Title: Lecture Notes in Computer Science.
[66] Arjun Narayan, Ariel Feldman, Antonis Papadimitriou, and Andreas Haeberlen. 2015. Verifiable differential privacy. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, Bordeaux France, 1–14. https://doi.org/10.1145/2741948.2741978
[67] Milad Nasr, Jamie Hayes, Thomas Steinke, Borja Balle, Florian Tramèr, Matthew Jagielski, Nicholas Carlini, and Andreas Terzis. 2023. Tight auditing of differentially private machine learning. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1631–1648.
[68] Milad Nasr, Shuang Song, Abhradeep Thakurta, Nicolas Papernot, and Nicholas Carlini. 2021. Adversary Instantiation: Lower Bounds for Differentially Private Machine Learning. http://arxiv.org/abs/2101.04535 arXiv:2101.04535 [cs].
[69] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 172 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360598
[70] OpenDP. 2025. SmartNoise — OpenDP SmartNoise. https://docs.smartnoise.org/ Accessed: 2025-11-24.
[71] Team OpenDP. 2020. The OpenDP White Paper. https://projects.iq.harvard.edu/files/opendp/files/opendp_white_paper_11may2020.pdf
[72] OpenMined contributors. 2025. *OpenMined/PipelineDP: Python framework for applying differentially private aggregations to large datasets.* https://github.com/OpenMined/PipelineDP GitHub repository; Accessed: 2025-12-01.
[73] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[74] Python Dev. 2025. contextvars — Context Variables. https://docs.python.org/3/library/contextvars.html Accessed: 2025-11-24.
[75] Zhaozhi Qian, Bogdan-Constantin Cebere, and Mihaela van der Schaar. 2023. Synthcity: facilitating innovative use cases of synthetic data in different data

modalities. https://doi.org/10.48550/arXiv.2301.07573 arXiv:2301.07573 [cs].
[76] Shahbaz Rezaei and Xin Liu. 2021. On the Difficulty of Membership Inference Attacks. https://doi.org/10.48550/arXiv.2005.13702 arXiv:2005.13702 [cs].
[77] Ali Shahin Shamsabadi, Gefei Tan, Tudor Ioan Cebere, Aurélien Bellet, Hamed Haddadi, Nicolas Papernot, Xiao Wang, and Adrian Weller. 2024. Confidential-DPproof : CONFIDENTIAL PROOF OF. (2024).
[78] Michael Shoemate, Andrew Vyrros, Chuck McCallum, Raman Prasad, Philip Durbin, Sílvia Casacuberta Puig, Ethan Cowan, Vicki Xu, Zachary Ratliff, Nicolás Berrios, Alex Whitworth, Michael Eliot, Christian Lebeda, Oren Renard, and Claire McKay Bowen. 2026. *OpenDP Library*. https://github.com/opendp/opendp
[79] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 3–18.
[80] Thomas Steinke, Milad Nasr, and Matthew Jagielski. 2023. Privacy Auditing with One (1) Training Run. http://arxiv.org/abs/2305.08846 arXiv:2305.08846 [cs].
[81] Thomas Steinke, Milad Nasr, and Matthew Jagielski. 2023. Privacy auditing with one (1) training run. *Advances in Neural Information Processing Systems* 36 (2023), 49268–49280.
[82] Florian Tramer, Andreas Terzis, Thomas Steinke, Shuang Song, Matthew Jagielski, and Nicholas Carlini. 2022. Debugging Differential Privacy: A Case Study for Privacy Auditing. https://doi.org/10.48550/arXiv.2202.12219 arXiv:2202.12219 [cs].
[83] Tudor Cebere. 2022. Privacy Leakage at low sample size. GitHub Issue #571, repository `meta-pytorch/opacus`. https://github.com/meta-pytorch/opacus/issues/571 Accessed: 2025-11-29.
[84] Tumult Labs. 2025. Tumult Analytics: Privacy Promise. Online Documentation. https://docs.tmlt.dev/analytics/dev/topic-guides/privacy-promise.html Accessed: 2025-11-29.
[85] Salil Vadhan and Wanrong Zhang. 2023. Concurrent Composition Theorems for Differential Privacy. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. ACM, Orlando FL USA, 507–519. https://doi.org/10.1145/3564246.3585241
[86] Larry Wasserman and Shuheng Zhou. 2009. A statistical framework for differential privacy. https://doi.org/10.48550/arXiv.0811.2501 arXiv:0811.2501 [math].
[87] june wunder, Arthur Azevedo de Amorim, Patrick Baillot, and Marco Gaboardi. 2023. Bunched Fuzz: Sensitivity for Vector Metrics. In *Programming Languages and Systems*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 451–478.
[88] Jiayuan Ye, Aadyaa Maddi, Sasi Kumar Murakonda, Vincent Bindschaedler, and Reza Shokri. 2022. Enhanced Membership Inference Attacks against Machine Learning Models. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Los Angeles CA USA, 3093–3106. https://doi.org/10.1145/3548606.3560675
[89] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. 2018. Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting. http://arxiv.org/abs/1709.01604 arXiv:1709.01604 [cs, stat].
[90] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Ghosh, Akash Bharadwaj, Jessica Zhao, Graham Cormode, and Ilya Mironov. 2022. Opacus: User-Friendly Differential Privacy Library in PyTorch. https://doi.org/10.48550/arXiv.2109.12298 arXiv:2109.12298 [cs].
[91] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664
[92] Santiago Zanella-Beguelin, Lukas Wutschitz, Shruti Tople, Ahmed Salem, Victor Rühle, Andrew Paverd, Mohammad Naseri, Boris Köpf, and Daniel Jones. 2023. Bayesian estimation of differential privacy. In *International Conference on Machine Learning*. PMLR, 40624–40636.
[93] Bing Zhang, Vadym Doroshenko, Peter Kairouz, Thomas Steinke, Abhradeep Thakurta, Ziyin Ma, Eidan Cohen, Himani Apte, and Jodi Spacek. 2023. Differentially private stream processing at scale. *arXiv preprint arXiv:2303.18086* (2023).
[94] Danfeng Zhang and Daniel Kifer. 2017. LightDP: towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 888–901. https://doi.org/10.1145/3009837.3009884
[95] Jun Zhang, Graham Cormode, Cecilia M. Procopiuc, Divesh Srivastava, and Xiaokui Xiao. 2017. PrivBayes: Private Data Release via Bayesian Networks. *ACM Trans. Database Syst.* 42, 4, Article 25 (Oct. 2017), 41 pages. https://doi.org/10.1145/3134428

A General Auditing Algorithm

Algorithm 3 describes the general statistical process for privacy auditing.

Algorithm 3 General Privacy Auditing Pipeline

Input: Mechanism $\mathcal{M}$, Adversary $\mathcal{A}$ (with `Rank`, `Reject`), base dataset D, canary x^*, number of runs R, target δ
Output: Empirical privacy lower bound $\hat{\varepsilon}$

$D_0 \leftarrow D$
$D_1 \leftarrow D \cup \{x^*\}$
$S \leftarrow []$ ▷ List to store (score, ground_truth) pairs
for $i = 1$ to R **do**
  $b_i \leftarrow \text{Bernoulli}(0.5)$ ▷ Draw random bit for ground truth
  if $b_i = 0$ **then**
    $o_i \leftarrow \mathcal{M}(D_0)$
  else
    $o_i \leftarrow \mathcal{M}(D_1)$
  end if
  $s_i \leftarrow \mathcal{A}.\textsc{Rank}(o_i, D_0, D_1)$ ▷ Get confidence score
  $S.\text{add}((s_i, b_i))$
end for
$b_{pred} \leftarrow \mathcal{A}.\textsc{Reject}(S, h^*)$
(TN, FP, FN, TP) $\leftarrow \textsc{ComputeStatistics}(S.\text{labels}, b_{pred})$
$(\alpha, \beta) \leftarrow \textsc{ConfInterval}(\text{TN, FP, FN, TP})$ ▷ e.g., CP [47]
$\hat{\varepsilon} \leftarrow \textsc{ConvertToDP}(\alpha, \beta, \delta)$ ▷ e.g., via GDP [67]
return $\hat{\varepsilon}$

B Details on Empirical PLD Estimation

In Re:cord-play-sample, we aim to integrate black-box statistical auditing for *untrusted* privacy mechanisms, enabling verification of the actual privacy loss. Unlike other auditing techniques that either evaluate a single (ε, δ) (e.g. [47, 68]), or that rely on the structure of the underlying mechanism (e.g. [67]) *empirical Privacy Loss Distribution (PLD)*. This allows us to compose empirically audited mechanisms with theoretically trusted ones using standard numerical accountants [26], and to obtain the overall privacy loss for the pipeline for some fixed δ.

Our techniques uses trade-off function, for a clear overview please see [25] and PLDs (see [26, 30]. The estimation procedure consists of three steps: (i) estimating the trade-off function via binary classification, (ii) converting the trade-off to the privacy profile $(\varepsilon, \delta(\varepsilon))$, and (iii) reconstructing a pessimistically discretized PLD using the privacy profile.

Trade-off Function Estimation. Let $\mathcal{M}$ be a mechanism and D, D' be a pair of adjacent datasets. We denote the output distributions as $P = \mathcal{M}(D)$ and $Q = \mathcal{M}(D')$. To empirically estimate the privacy leakage, we interpret the distinguishing problem as a hypothesis testing task between $H_0 : o \sim P$ and $H_1 : o \sim Q$. We generate empirical sets of output samples $S_P = \{o_i\}_{i=1}^n$ drawn from $\mathcal{M}(D)$ and $S_Q = \{o'_i\}_{i=1}^n$ drawn from $\mathcal{M}(D')$.

We train a binary classifier C (e.g., via logistic regression) to discriminate between S_P and S_Q. The performance of C serves as a proxy for the optimal decision rule. We compute the Receiver Operating Characteristic (ROC) curve, yielding a set of error pairs

$\{(\alpha_k, \beta_k)\}_k$. Here, α represents the Type I error (False Positive Rate) and β represents the Type II error (False Negative Rate). These pairs approximate the true trade-off function $f(\alpha)$, formally defined as the minimum Type II error achievable for a given Type I error constraint α:

$$f(\alpha) = \inf_{\phi} \{1 - \mathbb{E}_{o\sim Q}[\phi(o)] \mid \mathbb{E}_{o\sim P}[\phi(o)] \leq \alpha\}, \tag{1}$$

where ϕ represents a rejection rule $0 \leq \phi(\cdot) \leq 1$.

Conversion to the Privacy Profile. With the tradeoff function f at hand, we rely on the duality between f-DP and (ε, δ)-DP established by Dong et al. [25], Proposition 2.12. The privacy profile $\delta(\varepsilon)$ is the convex conjugate of the trade-off function f. We compute the empirical privacy profile $\delta(\varepsilon)$ for a grid of ε values by solving the following optimization:

$$\delta(\varepsilon) = 1 - \inf_{\alpha\in[0,1]} \left(e^{\varepsilon}\alpha + f(\alpha)\right). \tag{2}$$

In our implementation, we interpolate the discrete empirical pairs $\{(\alpha_k, \beta_k)\}$ to approximate $f(\alpha)$ and solve the minimization problem numerically using `scipy.optimize`.

PLD Reconstruction. Finally, to enable composition with other mechanisms, we convert the empirical profile $\delta(\varepsilon)$ into a PLD. For accounting purposes, we require a PLD that *upper bounds* the privacy loss. We construct a discretized PLD defined over a grid of privacy loss values $x_k = k \cdot \Delta x$. For each x_k, we determine the maximum probability mass consistent with the empirical $\delta(\varepsilon)$ curve. Specifically, we employ a "connect-the-dots" strategy that constructs a pessimistic Probability Mass Function (PMF) such that the $\delta(\varepsilon)$ implied by the PMF is always greater than or equal to the empirically measured $\delta(\varepsilon)$. This empirical PLD is then composed with the PLDs of trusted components to derive the final audit result. To do this, we have used the `create_pmf_pessimistic_connect_dots` of Doroshenko et al. [26] in the Google DP library.

C Audited Codebases

SmartNoise SDK [70]. Developed jointly by Microsoft and the OpenDP [71] project, SmartNoise is a widely deployed platform designed for general-purpose analytics. It focuses on tabular data protection and SQL query rewriting. We specifically audited its core synthesizer components and covariance release mechanisms.

dpmm [56]. The `dpmm` library is a specialized toolkit for generating differentially private synthetic data using marginal models. It implements state-of-the-art algorithms such as PrivBayes [95], MST [59], and AIM [61]. Unlike general-purpose libraries, `dpmm` focuses exclusively on preserving the statistical utility of low-dimensional marginals, making it a good candidate for testing the robustness of complex, iterative sampling mechanisms.

Synthcity [75]. Synthcity is a comprehensive framework for synthetic data generation. It includes a vast collection of generative models, ranging from GANs to Bayesian networks. For this audit, we targeted the privacy-preserving modules to verify whether the diverse set of integrated algorithms consistently adheres to their claimed privacy budgets, including their variant of PrivBayes and AIM.

Opacus [90]. Maintained by Meta, Opacus is the de facto standard library for training PyTorch [73] models with DP. It implements Differentially Private Stochastic Gradient Descent (DP-SGD) [1] by hooking into the optimizer step to perform per-sample gradient clipping and noise addition. Given its dominance in deep learning, ensuring the correctness of its gradient computation logic is an exciting task for our auditing framework.

Diffprivlib [43]. IBM's DP Library is a general-purpose toolkit designed to mimic the API of the popular `scikit-learn` library. It provides drop-in replacements for standard machine learning algorithms (e.g., Logistic Regression, Naive Bayes) and pre-processing tools. We selected it to test how standard ML paradigms are adapted for DP and whether the abstraction layers introduce errors.

Jax-privacy [10]. Developed by Google DeepMind, `jax-privacy` leverages the JAX [16] high-performance computing framework to enable DP training for large-scale models. It is designed for research flexibility and performance, often serving as the reference implementation for new privacy-preserving optimization techniques. We audited this library to evaluate privacy enforcement in a functional programming paradigm, which differs significantly from the object-oriented approach of PyTorch-based libraries.

Private-PGM. Next, we turn our attention to the framework widely known for hosting the winning submission of the NIST US-UK PETS Prize, namely Private-PGM ([60]). In the package a number of synthetic data models are defined, including the JAM mechanism.

MostlyAI [54]. MostlyAI offers an enterprise-grade platform for high-fidelity synthetic data generation. We examined their open-source Python client, which facilitates the training of generative models and the generation of synthetic datasets. Our audit focused on the client-side implementation of privacy safeguards and the mechanisms used to interface with their generative engine.

Google Differential Privacy [20]. This repository contains the foundational building blocks for differential privacy, with implementations in C++, Go, and Java. It provides the core algorithms (e.g., Laplace, Gaussian, and Count mechanisms) that power many higher-level tools. We have only audited their clustering mechanism [20], which is fully implemented in Python.

PipelineDP [72]. PipelineDP is a Python framework developed by OpenMined and Google for applying differentially private aggregations to large datasets.

SmartNoise SQL [70]. Distinct from the analysis SDK, SmartNoise SQL serves as the relational database interface for the SmartNoise project. It operates by intercepting standard SQL queries and rewriting them into differentially private equivalents compatible with backend engines like PostgreSQL. We specifically analyzed the `Odometer` mechanism, which tracks, accumulates, and enforces privacy budget limits across sequences of queries.

OpenDP [78]. OpenDP is a community-governed project designed to serve as a trustworthy, verifiable kernel of algorithms for differential privacy. Built primarily in Rust to leverage its strict memory safety and type systems, the library also enables Python-based implementations of complex algorithms like MST and AIM. We

audited these mechanisms to verify that they correctly compose OpenDP's foundational primitives (i.e. transformations and measurements) to maintain privacy guarantees.

Reproducibility and Scope. Table 2 clarifies the specific software artifacts examined in this paper. Given the rapid development of open-source DP tools, reproducibility is essential. To this end, we report the exact commit hashes for the code snapshots we audited.

Table 2: Overview of the audited libraries, with associated versions and targeted modules for reproducibility purposes.

Library	Link	Commit Hash	Relevant Files & Functionality
SmartNoise [70]	GitHub	b4f0058	Covariance Estimator
dpmm [56]	GitHub	e5cd92b	AIM Mechanism Domain, PrivBayes
Synthcity [75]	GitHub	23f322f	PrivBayes
Opacus [90]	GitHub	f17f254	`make_private`
Diffprivlib [43]	GitHub	5d9c9d8	Linear Regression
Jax-privacy [10]	GitHub	2ddd6d9	- (No bugs found)
Private-PGM [60]	GitHub	8022182	JAM
MostlyAI [54]	GitHub	00cfb12	analyze_reduce_numeric
SmartNoise SQL [70]	GitHub	81d1351	Odometer
Google Differential Privacy [20]	GitHub	3cff255	- (No bugs found)
PipelineDP [72]	GitHub	94bacde	- (No bugs found)
OpenDP [78]	GitHub	cc3a18b	- (No bugs found)

D Handling of Adversarial Inputs

Table 3 demonstrates the effect of using adversarial inputs containing NaN values and infinity. While these values were likely out of scope for the original framework, we showcase the widespread implication of not performing tight input validation on mechanisms.

Table 3: Summary of behavior under $-\infty$ and NaN inputs across the evaluated DP libraries. A (green) cross ✗ indicates that the implementation correctly handled the corresponding case, while a (red) checkmark ✓ indicates that our framework found an issue when injecting $\pm\infty$/NaN values.

Implementation	$\pm\infty$ Inputs	NaN Inputs
SmartNoise Synth		
Logistic Regression	✓	✓
dpmm		
PrivBayes	✓	✓
MST	✗	✗
AIM	✓	✓
Synthcity		
PrivBayes	✓	✓
AIM	✗	✗
Diffprivlib		
Linear Regression	✓	✓
Logistic Regression	✓	✓
Opacus		
DP-SGD	✓	✓
Google Differential Privacy		
Clustering	✗	✗

E Acknowledgments

The authors used generative AI-based tools to revise the text, improve flow, and correct typos, grammatical errors, and awkward phrasing. AI-based tools were used in the production of the code as well for documentation writing, test writing, polishing, and for the understanding and discovery of the audited codebases.