DDUO: General-Purpose Dynamic Analysis for Differential Privacy

Chike Abuah[1], Alex Silence[1], David Darais[2], and Joseph P. Near[1]

[1]Computer Science Department, University of Vermont
[2]Galois, Inc.

2021 IEEE 34th Computer Security Foundations Symposium (CSF) | 978-1-7281-7607-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSF51468.2021.00043

***Abstract*—Differential privacy enables general statistical analysis of data with formal guarantees of privacy protection at the individual level. Tools that assist data analysts with utilizing differential privacy have frequently taken the form of programming languages and libraries. However, many existing programming languages designed for compositional verification of differential privacy impose significant burden on the programmer (in the form of complex type annotations). Supplementary library support for privacy analysis built on top of existing general-purpose languages has been more usable, but incapable of pervasive end-to-end enforcement of sensitivity analysis and privacy composition.**

We introduce DDUO, a *dynamic analysis* for enforcing differential privacy. DDUO is usable by non-experts: its analysis is automatic and it requires no additional type annotations. DDUO can be implemented *as a library* for existing programming languages; we present a reference implementation in Python which features moderate runtime overheads on realistic workloads. We include support for several data types, distance metrics and operations which are commonly used in modern machine learning programs. We also provide initial support for tracking the sensitivity of data transformations in popular Python libraries for data analysis.

We formalize the novel core of the DDUO system and prove it sound for sensitivity analysis via a logical relation for metric preservation. We also illustrate DDUO's usability and flexibility through various case studies which implement state-of-the-art machine learning algorithms.

I. Introduction

Differential privacy has achieved prominence over the past decade as a rigorous formal foundation upon which diverse tools and mechanisms for performing private data analysis can be built. The guarantee of differential privacy is that it protects privacy at the individual level: if the result of a differentially private query or operation on a dataset is publicly released, any individual present in that dataset can claim *plausible deniability*. This means that any participating individual can deny the presence of their information in the dataset based on the query result, because differentially private queries introduce enough random noise to make the result indistinguishable from that of the same query run on a dataset which actually *does not* contain the individual's information. Additionally, differential privacy guarantees are resilient against any form of *linking attack* in the presence of *auxiliary information* about individuals.

High profile tech companies such as Google have shown a commitment to differential privacy by developing projects such as RAPPOR [54] as well as several open-source privacy-preserving technologies [28, 29, 48]. Facebook recently released an unprecedented social dataset, protected by differential privacy guarantees, which contains information regarding people who publicly shared and engaged with about 38 million unique URLs, as an effort to help researchers study social media's impact on democracy and the 2020 United States presidential election [39, 33, 34, 23, 24]. The US Census Bureau has also adopted differential privacy to safeguard the 2020 census results [2].

Both static and dynamic tools have been developed to help non-experts write differentially private programs. Many of the static tools take the form of statically-typed programming languages, where correct privacy analysis is built into the soundness of the type system. However, existing language-oriented tools for compositional verification of differential privacy impose significant burden on the programmer (in the form of additional type annotations) [41, 26, 40, 18, 53, 50, 9, 11, 12, 10, 44, 5, 52, 47, 14, 19, 46] (see Section IX for a longer discussion).

The best-known dynamic tool is PINQ [36], a dynamic analysis for sensitivity and privacy. It features an extensible system which allows non-experts in differential privacy to execute SQL-like queries against relational databases. However, PINQ comes with several restrictions that limit its applicability. For example, PINQ's expressiveness is limited to a subset of the SQL language for relational databases. Methods in PINQ are assumed to be side-effect free, which is necessary to preserve their privacy guarantee.

We introduce DDUO, a dynamic analysis for enforcing differential privacy. DDUO is usable by non-experts: its analysis is automatic and it requires no additional type annotations. DDUO can be implemented *as a library* for existing programming languages; we present a reference implementation in Python. Our goal in this work is to answer the following four questions, based on the limitations of PINQ:

- Can a PINQ-style dynamic analysis extend to base types in the programming language, to allow its use pervasively?
- Is the analysis sound in the presence of side effects?
- Can we use this style of analysis for complex algorithms like differentially private gradient descent?
- Can we extend the privacy analysis beyond pure ϵ-differential privacy?

We answer all four questions in the affirmative, building on PINQ in the following ways:

- DDuo provides a dynamic analysis for base types in a general purpose language (Python). DDuo supports general language operations, such as mapping arbitrary functions over lists, and tracks the sensitivity (stability) and privacy throughout.
- Methods in DDuo are not required to be side-effect free and allow programmers to mutate references inside functions which manipulate sensitive values.
- DDuo supports various notions of sensitivity and arbitrary distance metrics (including L_1 and L_2 distance).
- DDuo is capable of leveraging advanced privacy variants such as (ϵ, δ) and Rényi differential privacy.

Privacy analysis is reliant on *sensitivity* analysis, which determines the scale of noise an analyst must add to values in order to achieve any level of privacy. Dynamic analysis for differential privacy is thus a dual challenge:

Dynamic sensitivity analysis. Program sensitivity is a (hyper)property quantified over two runs of a program with related inputs (sources). A major challenge for dynamic sensitivity analysis is the ability to bound sensitivity, ensuring that the metric preservation property is satisfied, by only observing *a single run* of the program. In addition, an analysis which is performed on a specific input to the program must generalize to future possible arbitrary inputs.

The key insight to our solution is attaching sensitivity environments and distance metric information to *values* rather than variables. Our approach provides a sound upper bound on global sensitivity even in the presence of side effects, conditionals, and higher-order functions. We present a proof using a *step-indexed logical relation* which shows that our sensitivity analysis is sound.

Dynamic privacy analysis. To implement a dynamic privacy analysis, we leverage prior work on privacy filters and odometers [42]. This work, originally designed for the adaptive choice of privacy parameters, can also be used as part of a dynamic analysis for privacy analysis. We view each application of a privacy mechanism (e.g. the Laplace mechanism) as a *global privacy effect* on total privacy cost, and use privacy filters and odometers to track total privacy cost.

We implemented these features in a Python prototype of DDuo via object proxies and other pythonic idioms. We implement several case studies to showcase these features and demonstrate the usage of DDuo in practice. We also provide integrations with several popular Python libraries for data and privacy analysis.

Contributions. In summary, this paper makes the following contributions:

- We introduce DDuo, a *dynamic* analysis for enforcing differential privacy, and a reference implementation as a Python library [1].
- We formalize a subset of DDuo in a core language model, and prove the soundness of DDuo's dynamic sensitivity analysis (as encoded in the model) using a step-indexed logical relation.
- We present several case studies demonstrating the use of DDuo to build practical, verified Python implementations of complex differentially private algorithms.

[1]The reference implementation is available here: https://github.com/uvm-plaid/dduo-python

The rest of the paper is organized as follows. First we provide some background knowledge regarding the field of differential privacy (Section II). We provide an overview of our work and the necessary tradeoffs (Section III). We illustrate the usefulness and power of DDuo through some worked examples (Section IV). We then discuss some of the nuances of dynamic sensitivity (Section V) and dynamic privacy tracking (Section VI). We present the formalization of DDuo and prove the soundness of our sensitivity analysis (Section VII). We provide several case studies demonstrating the usefulness of DDuo in practice (Section VIII). Finally we outline related work (Section IX) and conclude (Section X).

II. Background

Differential Privacy. Differential privacy is a formal notion of privacy; certain algorithms (called *mechanisms*) can be said to *satisfy* differential privacy. Intuitively, the idea behind a differential privacy mechanism is that: given inputs which differ in the data of a single individual, the mechanism should return statistically indistinguishable answers. This means that the data of any one individual should not have any significant effect on the outcome of the mechanism, effectively protecting privacy on the individual level. Formally, differential privacy is parameterized by the privacy parameters ϵ, δ which control the strength of the guarantee.

When we say "neighboring" inputs, this implies two inputs that differ in the information of a single individual. However, formally we can defer to some general *distance metric* which may take several forms. We then say that according to the distance metric, the distance between two databases must have an upper bound of 1. Variation of the distance metric has led to several other useful, non-standard forms of differential privacy in the literature.

Definition II.1 (Differential privacy)**.** *Given a distance metric $d_A \in A \times A \rightarrow \mathbb{R}$, a randomized algorithm (or* mechanism*) $\mathcal{M} \in A \rightarrow B$ satisfies (ϵ, δ)-differential privacy if for all $x, x' \in A$ such that $d_A(x, x') \leq 1$ and all possible sets $S \subseteq B$ of outcomes, $Pr[\mathcal{M}(x) \in S] \leq e^{\epsilon} Pr[\mathcal{M}(x') \in S] + \delta$.*

Differential privacy is *compositional*: running two mechanisms $\mathcal{M}_1$ and $\mathcal{M}_2$ with privacy costs of (ϵ_1, δ_1) and (ϵ_2, δ_2) respectively has a total privacy cost of $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$. *Advanced composition* [20] improves on this composition bound for iterative algorithms; several variants of differential privacy (e.g. Rényi differential privacy [37] and zero-concentrated differential privacy [16]) have been developed that improve the bound even further. Importantly, sequential composition theorems for differential privacy do not necessarily allow the privacy parameters to be chosen *adaptively*, which presents a special challenge in our setting—we discuss this issue in Section VI.

III. Overview of DDuo

DDuo is a *dynamic* analysis for enforcing differential privacy. Our approach does not require static analysis of programs, and allows DDuo to be implemented as a library for programming languages like Python. DDuo's dynamic analysis has complete access to run-time information, so it does not require the programmer to write any additional type annotations—in many cases, DDuo can verify differential privacy for essentially unmodified Python programs (see the case studies in Section VIII). As a Python library, DDuo is easily integrated with popular libraries like Pandas and NumPy.

Threat model. We assume an "honest but fallible" programmer—that is, the programmer *intends* to produce a differentially private program, but may unintentionally introduce bugs. We assume that the programmer is *not* intentionally attempting to subvert DDuo's enforcement approach. Our reference implementation is embedded in Python, an inherently dynamic language with run-time features like reflection. In this setting, a malicious programmer or privacy-violating third-party libraries can bypass our dynamic monitor and extract sensitive information directly. We allow several common side-effects such as reference mutation, printing, reading/writing files, etc. Note that printing/writing sensitive values in DDuo will reveal the type of the value, but not the actual value. Data-independent exceptions can be safely used in our system, however our model must explicitly avoid data-dependent exceptions such as division-by-zero errors. Terminated programs can be rerun safely (while consuming the privacy budget) because our analysis is independent of any sensitive information (our metatheory implies that sensitivity of a value is itself not sensitive). We also do not address side-channels, including execution time. Like existing enforcement approaches (PINQ, OpenDP, Diffprivlib), DDuo is intended as a tool to help well-intentioned programmers produce correct differentially private algorithms.

Soundness of the analysis. We formalize our dynamic sensitivity analysis and prove its soundness in Section VII. Our formalization includes the most challenging features of the dynamic setting—conditionals and side effects—and provides evidence that our Python implementation will be effective in catching privacy bugs in real programs. DDuo relies on existing work on privacy filters and odometers (discussed in Section VI), whose soundness has been previously established, for tracking privacy cost.

IV. DDuo by Example

This section introduces the DDuo system via examples written using our reference Python implementation.

Data Sources. Data sources are wrappers around sensitive data that enable tracking of privacy information in the DDuo python library. Each data source is associated with an identifying string, such as the name of the input file the data was read from. Data sources can be created manually by attaching an identifying string (such as a filename) to a raw value (such as a vector). Or, data sources be created automatically upon loading data through DDuo's custom-wrapped third party APIs, such as pandas. Note that our API can be easily modified to account for initial sensitivities greater than 1 when users have multiple datapoints in the input data.

```
from dduo import pandas as pd
df = pd.read_csv("data.csv")
df
```

Sensitive(<'DataFrame'>, $\{\textit{data.csv} \mapsto 1\}$, L_∞)

A `Sensitive` value is returned. `Sensitive` values represent sensitive information that cannot be viewed by the analyst. When a `Sensitive` value is printed out, the analyst sees (1) the type of the value, (2) its *sensitivity environment*, and (3) its *distance metric*. The latter two components are described next. The analyst is *prevented* from viewing the value itself.

Sensitivity & distance metrics. Function sensitivity is a scalar value which represents how much a change in a function's input will change the function's output. For example, the binary addition function $f(x, y) = x + y$ is 1-sensitive in both x and y, because changing either input by n will change the sum by n. The function $f(x) = x + x$, on the other hand, is 2-sensitive in its argument x, because changing x by n changes the function's output by $2n$. Sensitivity is key to differential privacy because it is directly proportional to the amount of noise we must add to the output of a function to make it private.

DDuo tracks the sensitivity of a value to changes in the program's inputs using a *sensitivity environment* mapping input data sources to sensitivities. Our example program returned a `Sensitive` value with a sensitivity environment of $\{\textit{data.csv} \mapsto 1\}$, indicating that the underlying value is 1-sensitive in the data contained in $\textit{data.csv}$. The DDuo library tracks and updates the sensitivity environments of `Sensitive` objects as operations are applied to them. For example, adding a constant value to the elements of the DataFrame results in no change to the sensitivity environment.

```
df + 5 # no change to sensitivity environment
```

Sensitive(<'DataFrame'>, $\{\textit{data.csv} \mapsto 1\}$, L_∞)

Adding the DataFrame to *itself* doubles the sensitivity, in the same way as the function $f(x) = x + x$.

```
df + df   # doubles the sensitivity
```

Sensitive(<'DataFrame'>, $\{\textit{data.csv} \mapsto 2\}$, L_∞)

Finally, multiplying the DataFrame by a constant scales the sensitivity, and multiplying the DataFrame by *itself* results in *infinite* sensitivity.

```
( df * 5, df * df)
( Sensitive(<'DataFrame'>, {data.csv ↦ 5},  L∞),
  Sensitive(<'DataFrame'>, {data.csv ↦ ∞}, L∞) )
```

The *distance metric* component of a `Sensitive` value describes *how* to measure sensitivity. For simple numeric functions like $f(x) = x + x$, the distance between two possible inputs x and x' is simply $|x - x'|$ (this is called the *cartesian metric*). For more complicated data structures (e.g. DataFrames), calculating the distance between two values is more involved. The L_∞ metric used in our example calculates the distance between two DataFrames by measuring how many rows are different (this is one standard way of defining "neighboring databases" in differential privacy). DDUO's handling of distance metrics is detailed in Section V-B.

Privacy. DDUO also tracks the *privacy* of computations. To achieve differential privacy, programs add noise to sensitive values. The Laplace mechanism described earlier is one basic mechanism for achieving differential privacy by adding noise drawn from the Laplace distribution (DDUO provides a number of basic mechanisms, including the Gaussian mechanism). The following expression counts the number of rows in our example DataFrame and uses the Laplace mechanism to achieve ϵ-differential privacy, for $\epsilon = 1.0$.

```
dduo.laplace(df.shape[0], ϵ=1.0)
```

```
9.963971319623278
```

The result is a *regular Python value*—the analyst is free to view it, write it to a file, or do further computation on it. Once the correct amount of noise has been added, the principle of *post-processing* applies, and so DDUO no longer needs to track the sensitivity or privacy cost of operations on the value.

When the Laplace mechanism is used multiple times, their privacy costs *compose* (i.e. the ϵs "add up" as described earlier). DDUO tracks *total* privacy cost using objects called *privacy odometers* [42]. The analyst can interact with a privacy odometer object to learn the total privacy cost of a complex computation.

```
with dduo.EpsOdometer() as odo:
  _ = dduo.laplace(df.shape[0], ϵ = 1.0)
  _ = dduo.laplace(df.shape[0], ϵ = 1.0)
  print(odo)
```

```
Odometer_ϵ({data.csv ↦ 2.0})
```

Printing the odometer's value allows the analyst to view the privacy cost of the program with respect to each of the data sources used in the computation. In this example, two differentially private approximations of the number of rows in the dataframe *df* are computed, each with a privacy cost of $\epsilon = 1.0$. The total privacy cost of running the program is therefore $2 \cdot \epsilon = 2.0$.

DDUO also allows the analyst to place upper bounds on total privacy cost (i.e. a privacy *budget*) using privacy *filters* [42]. Privacy odometers and filters are discussed in detail in Section VI.

V. Dynamic Sensitivity Tracking

DDUO implements a *dynamic sensitivity analysis* by wrapping values in `Sensitive` objects and calculating sensitivities as operations are performed on these objects. Type systems for sensitivity [41, 26] construct a sensitivity environment for each program expression; in the static analysis setting, a sensitivity environment records the expression's sensitivity with respect to each of the variables currently in scope.

DDUO attaches sensitivity environments to *values* at runtime: each `Sensitive` object holds both a value and its sensitivity environment. As described earlier, DDUO's sensitivity environments record a value's sensitivity with respect to each of the program's data sources. Formally, the sensitivity of a single-argument function f in its input is defined as:

$$\text{sens}(f) \triangleq \text{argmax}_{x,y}\left(\frac{d(f(x), f(y))}{d(x,y)}\right)$$

Where d is a *distance metric* over the values x and y could take (distance metrics are discussed in Section V-B). Thus, a sensitivity environment $\{a \mapsto 1\}$ means that if the value of the program input a changes by n, then the value of $f(a)$ will change by at most n.

A. Bounding the Sensitivity of Operations

Operations on `Sensitive` objects are defined to perform the same operation on the underlying values, and *also* construct a new sensitivity environment for the operation's result. For example, DDUO's `__add__` operation sums both the underlying values *and* their sensitivity environments:

```
def __add__(self, other):
    assert self.metric == other.metric
    return dduo.Sensitive(self.value + other.value,
                          self.senv + other.senv,
                          self.metric)
```

The sum of two sensitivity environments is defined as the element-wise sum of their items. For example:

$$\{a \mapsto 2, b \mapsto 1\} + \{b \mapsto 3, c \mapsto 5\} = \{a \mapsto 2, b \mapsto 4, c \mapsto 5\}$$

The DDUO library provides sensitivity-aware versions of Python's basic numeric operations (formalized in Section VII). We have also defined sensitivity-aware versions of commonly-used library functions, including the Pandas functions used in Section IV, and subsets of NumPy and Scikit-learn.

B. Distance Metrics

At the core of the concept of sensitivity is the notion of distance: how far apart we consider two information sources to be from each other. For scalar values, the following two distance metrics are often used:

- Cartesian (absolute difference) metric: $d(x, y) = |x - y|$
- Discrete metric: $d(x, y) = 0 \textit{ if } x = y; 1 \textit{ otherwise}$

For more complex structures—like lists and dataframes—we can use distance metrics on *vectors*. Two commonly-used metrics for vectors x and y of equal length are:

- $L_1(d_i)$ metric: $d(x, y) = \sum_{x_i, y_i \in x, y} d_i(x_i, y_i)$

- $L_2(d_i)$ metric: $d(x,y) = \sqrt{\sum_{x_i,y_i \in x,y} d_i(x_i,y_i)^2}$

Both metrics are parameterized by d_i, a metric for the vector's elements. In addition to these two, we use the shorthand L_∞ to mean $L_1(d)$, where d is the cartesian metric defined above. The L_∞ metric works for any space with equality (e.g. strings), and measures the *number of elements where x and y differ.*

The definition of differential privacy is parameterized by a distance metric that is intended to capture the idea of two inputs that *differ in one individual's data*. Database-oriented algorithms typically assume that each individual contributes exactly one row to the database, and use the L_∞ metric to define neighboring databases (as we did in Section IV).

Distance metrics can be manipulated manually through operations such as clipping, a technique commonly employed in differentially private machine learning. DDUO tracks distance metrics for `Sensitive` information, which can allow for automatic conservation of the privacy budget while providing more accurate query analysis.

Lists and arrays are compared by one of the L_1, L_2, or L_∞ distance metrics. The choice of distance metric is important when defining sensitivity and thus privacy. For example, the Laplace mechanism can only be used with the L_1 metric, while the Gaussian mechanism can be used with either L_1 or L_2.

C. Conditionals & Side Effects

Conditionals and other branching structures are challenging for any sensitivity analysis, but they present a particular challenge for our dynamic analysis. Consider the following conditional:

```
if df.shape[0] == 10:
  return df.shape[0]
else:
  return df.shape[0] * 10000
```

Here, the two branches have *different* sensitivities (the *else* branch is 10,000 times more sensitive in its data sources than the *then* branch). Static sensitivity analyses handle this situation by taking the maximum of the two branches' sensitivities (i.e. they assume the worst-case branch is executed), but this approach is not possible in our dynamic analysis.

In addition, special care must be taken when a sensitive value appears in the guard position (as in our example). Static analyses typically scale the branches' sensitivity by the sensitivity of the guard; in practice, this approach results in *infinite sensitivity for conditionals with a sensitive guard.*

To retain soundness in our dynamic analysis, DDUO requires that *conditional guards contain no sensitive values*. A run-time error is thrown if DDUO finds a sensitive value in the guard position (as in our example above). Disallowing sensitive guards makes it possible to ignore branches that are not executed: the guard's value remains the same under neighboring program inputs, so the program follows the same branch for neighboring executions. This approach does not limit the set of useful programs we can write, since conditionals with sensitive guards yield infinite sensitivities even under a precise static analysis.

Since DDUO attaches sensitivity environments to *values* (instead of variables), the use of side effects does not affect the soundness of the analysis. When a program variable is updated to reference a new value, that value's sensitivity environment remains attached. DDUO handles many common side-effect-based patterns used in Python this way; for example, DDUO correctly infers that the following program results in the variable `total` holding a value that is 20 times more sensitive than `df.shape[0]`.

```
total = 0
for i in range(20):
  total = total + df.shape[0]
```

For side effects, our dynamic analysis is more capable than type-based static analysis, due to the additional challenges arising in the static setting (e.g. aliasing). We have formalized the way DDUO handles side effects and conditionals, and proved the soundness of our sensitivity analysis; our formalization appears in Section VII.

VI. Dynamic Privacy Tracking

DDUO tracks privacy cost *dynamically*, at runtime. Dynamic privacy tracking is challenging because the dynamic analysis has no visibility into code that is *not executed*. For example, consider the following conditional:

```
if dduo.gauss(ϵ=1.0, δ=1e-5, x) > 5:
  print(dduo.gauss(ϵ=1.0, δ=1e-5, y))
else:
  print(dduo.gauss(ϵ=100000000000.0, δ=1e-5, y))
```

The executed branch of this conditional depends on the result of the first call to `dduo.gauss`, which is non-deterministic. The two branches use different privacy parameters for the remaining calls to `dduo.gauss`; in other words, the privacy parameter for the second use of the Gaussian mechanism is chosen *adaptively*, based on the results of the first use. Sequential composition theorems for differential privacy [20] are typically stated in terms of *fixed* (i.e. non-adaptive) privacy parameters, and do not apply if the privacy parameters are chosen adaptively.

A static analysis of this program will consider *both* branches, and most analyses will produce an upper bound on the program's privacy cost by combining the two (i.e. taking the maximum of the two ϵ values). This approach avoids the issue of adaptively-chosen privacy parameters.

A dynamic analysis, by contrast, *cannot* consider both branches, and must bound privacy cost by analyzing *only* the branch that is executed. Sequential composition does not apply directly when privacy parameters are chosen adaptively, so ignoring the non-executed branch in a dynamic analysis of privacy would be *unsound*.

A. Privacy Filters & Odometers

Privacy *filters* and *odometers* were originally developed by Rogers et al. [42] specifically to address the setting in which privacy parameters are selected adaptively. Winograd-Cort et al. [50] used privacy filters and odometers as part of the Adaptive Fuzz framework, which integrates both dynamic analysis (for composing privacy mechanisms) and static analysis (for bounding the cost of individual mechanisms). Recently, Feldman and Zrnic [25] developed filters and odometers for Rényi differential privacy [37].

Privacy odometers can be used to obtain a running upper bound on total privacy cost at any point in the sequence of adaptive mechanisms, and to obtain an overall total at the end of the sequence. A function $\texttt{COMP}_{\delta_g} : \mathbb{R}^{2k}_{\geq 0} \to \mathbb{R} \cup \{\infty\}$ is called a *valid privacy odometer* [42] for a sequence of mechanisms $\mathcal{M}_1, \ldots, \mathcal{M}_k$ if for all (adaptively-chosen) settings of $(\epsilon_1, \delta_1), \ldots, (\epsilon_k, \delta_k)$ for the individual mechanisms in the sequence, their composition satisfies $(\texttt{COMP}_{\delta_g}(\cdot), \delta_g)$-differential privacy. In other words, $\texttt{COMP}_{\delta_g}(\cdot)$ returns a value for ϵ that upper-bounds the privacy cost of the adaptive sequence of mechanisms. A valid privacy odometer for sequential composition in (ϵ, δ)-differential privacy can be defined as follows (Rogers et al. [42], Theorem 3.6):

$$\texttt{COMP}_{\delta_g}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k) = \begin{cases} \infty & \textit{if } \sum_{i=1}^{k} \delta_i > \delta_g \\ \sum_{i=1}^{k} \epsilon_i & \textit{otherwise} \end{cases}$$

Privacy filters allow the analyst to place an upper bound (ϵ_g, δ_g) on the desired privacy cost, and halt the computation immediately if the bound is violated. A function $\texttt{COMP}_{\epsilon_g,\delta_g} : \mathbb{R}^{2k}_{\geq 0} \to \{\texttt{HALT}, \texttt{CONT}\}$ is called a *valid privacy filter* [42] for a sequence of mechanisms $\mathcal{M}_1, \ldots, \mathcal{M}_k$ if for all (adaptively-chosen) settings of $(\epsilon_1, \delta_1), \ldots, (\epsilon_k, \delta_k)$ for the individual mechanisms in the sequence, $\texttt{COMP}_{\epsilon_g,\delta_g}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k)$ outputs $\texttt{CONT}$ only if the sequence satisfies (ϵ_g, δ_g)-differential privacy (otherwise, it outputs $\texttt{HALT}$ for the first mechanism in the sequence that violates the privacy cost bound). A valid privacy filter for sequential composition in (ϵ, δ)-differential privacy can be defined as follows (Rogers et al. [42], Theorem 3.6):

$$\texttt{COMP}_{\epsilon_g,\delta_g}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k) = \begin{cases} \texttt{HALT} & \textit{if } \sum_{i=1}^{k} \delta_i > \delta_g \textit{ or } \sum_{i=1}^{k} \epsilon_i > \epsilon_g \\ \texttt{CONT} & \textit{otherwise} \end{cases}$$

It is clear from these definitions that the odometer and filter for sequential composition under (ϵ, δ)-differential privacy yield the same bounds on privacy loss as the standard theorem for sequential composition [20] (i.e. there is no "cost" to picking the privacy parameters adaptively).

Rogers et al. [42] also define filters and odometers for *advanced composition* under (ϵ, δ)-differential privacy ([42], §5 and §6); in this case, there *is* a cost. In exchange for the ability to set privacy parameters adaptively, filters and odometers for advanced composition have slightly worse constants than the standard advanced composition theorem [20] (but are asymptotically the same).

B. *Filters & Odometers in* DDUO

DDUO's API allows the programmer to explicitly create privacy odometers and filters, and make them active for a specific part of the program (using Python's *with* syntax). When an odometer is active, it records a running total of the total privacy cost, and it can be queried to return this information to the programmer.

```
with dduo.EdOdometer(max_delta = 10e-5) as odo:
  _ = dduo.gauss(df.shape[0], ϵ = 1.0, δ = 10e-6)
  _ = dduo.gauss(df.shape[0], ϵ = 1.0, δ = 10e-6)
  print(odo)
```

```
Odometer_(ϵ, δ) ({data.csv ↦ (2.0, 20^−6)})
```

When a filter is active, it tracks the privacy cost for individual mechanisms, and halts the program if the filter's upper bound on privacy cost is violated.

```
with dduo.EdFilter(ϵ = 1.0, δ = 10e-6) as odo:
  print('1:', dduo.gauss(df.shape[0], ϵ=1.0, δ=10e-6))
  print('2:', dduo.gauss(df.shape[0], ϵ=1.0, δ=10e-6))
```

```
1: 10.5627
Traceback (most recent call last):
  ...
  dduo.PrivacyFilterException
```

In addition to odometers and filters for sequential composition under (ϵ, δ)-differential privacy (such as `EdFilter` and `EdOdometer`), DDUO provides odometers and filters for advanced composition (`AdvEdFilter` and `AdvEdOdometer`) and Rényi differential privacy (`RenyiFilter` and `RenyiOdometer`, which follow the results of Feldman and Zrnic [25]).

C. *Loops and Composition*

Iterative algorithms can be built in DDUO using Python's standard looping constructs, and DDUO's privacy odometers and filters take care of ensuring the correct form of composition. Parallel composition is also available—via functional mapping. Advanced composition can be achieved via special advanced composition filters and odometers exposed in the DDUO API. For example, the following simple loop runs the Laplace mechanism 20 times, and its total privacy cost is reflected by the odometer:

```
with dduo.EpsOdometer() as odo:
  for i in range(20):
    dduo.laplace(df.shape[0], ϵ = 1.0)
  print(odo)
```

```
Odometer_ϵ ({data.csv ↦ 20.0})
```

To use advanced composition instead of sequential composition, we simply replace the odometer with a different one:

```
with dduo.AdvEdOdometer() as odo:
  for i in range(20):
    dduo.gauss(df.shape[0], ϵ = 0.01, δ = 0.001)
```

D. Mixing Variants of Differential Privacy

The DDuo library includes support for pure ϵ-differential privacy, (ϵ, δ)-differential privacy, and Rényi differential privacy (RDP). Programs may use all three variants together, convert between them, and compose mechanisms from each.

We demonstrate execution of a query while switched to the Rényi differential privacy variant using pythonic "with" syntax blocks. For programs that make extensive use of composition, this approach yields significant improvements in privacy cost. For example, the following program uses the Gaussian mechanism 200 times, using Rényi differential privacy for sequential composition; the total privacy cost is automatically converted into an (ϵ, δ)-differential privacy cost after the loop finishes.

```
with dduo.EdOdometer(max_delta = 1e-4) as odo:
  with dduo.RenyiDP(1e-5):
    for x in range(200):
      noisy_count = dduo.renyi_gauss(α = 10,
        ϵ=0.2, df.shape[0])
  print(odo)
```

```
Odometer_(ϵ, δ)({data.csv ↦ (41.28, 1⁻⁵)})
```

VII. Formal Description of Sensitivity Analysis

In DDuo we implement a novel dynamic analysis for *function sensitivity*, which is a relational (hyper)property quantified over two runs of the program with arbitrary but related inputs. In particular, our analysis computes function sensitivity—a two-run property—after only observing *one* execution of the program. Only observing one execution poses challenges to the design of the analysis, and significant challenges to the proof, all of which we overcome. To overcome this challenge in the design of the analysis, we first disallow branching control flow which depends on any sensitive inputs; this ensures that any two runs of the program being considered for the purposes of privacy will take the same branch observed by the dynamic analysis. Second, we disallow sensitive input-dependent arguments to the "scalar" side of multiplication; this ensures that the dynamic analysis' use of that argument in analysis results is identical for any two runs of the program being considered for the purposes of privacy. Our dynamic analysis for function sensitivity is *sound*—meaning that the true sensitivity of a program is guaranteed to be equal or less than the sensitivity reported by DDuo's dynamic monitor—and we support this claim with a detailed proof.

Formalism Approach. We formalize the correctness of our dynamic analysis for function sensitivity using a *step-indexed big-step semantics* to describe the dynamic analysis, a *step-indexed logical relation* to describe the meaning of function sensitivity, and a proof by induction and case analysis on program syntax to show that dynamic analysis results soundly predict function sensitivity. A *step-indexed* relation is a relation $\mathcal{R} \in A \rightarrow B \rightarrow \mathsf{prop}$ whose definition is stratified by a natural number index n, so for each level n there is a new relation $\mathcal{R}_n$. Typically, the relation $\mathcal{R}_0$ is defined $\mathcal{R}_0(x, y) \triangleq \mathsf{true}$, and the final relation of interest is $\hat{\mathcal{R}} \triangleq \bigcap_n \mathcal{R}_n$, i.e., $\hat{\mathcal{R}}(x, y) \iff \forall n.\ \mathcal{R}_n(x, y)$. Step-indexing is typically used—as we do in our formalism—when the definition of a relation would be not well founded in its absence. The most common reason a relation definition might be not well-founded is the use of self-reference without any decreasing measure. When a decreasing measure exists, self-reference leads to well-founded recursion, however when a decreasing measure does not exist, self-reference is not well-founded. When using step-indexing, self-reference is allowed in the definition of $\mathcal{R}_n$, but only for the relation at strictly lower levels, so $\mathcal{R}_{n'}$ when $n' < n$; this is well-founded because the index n becomes a decreasing measure for the self-reference. In this way, step-indexing enables self-reference without any existing decreasing measure by introducing a new decreasing measure, and maintains well-foundedness of the relation definition.

A *logical* relation is one where the definition of relation on function values (or types) is extensional, essentially saying "when given related inputs, the function produces related outputs". This definition is self-referential and not well-founded, and among common reasons to introduce step-indexing in programming language proofs. As the relation $\mathcal{R}$ is stratified with a step-index to $\mathcal{R}_n$, so must the definition of the semantics, so for a big-step relation $e \Downarrow v$ (relating an expression e to its final value v after evaluation) we stratify as $e \Downarrow_n v$. Also, because the definition of a logical relation *decrements* the step-index for the case of function values, we *increment* the step-index in the semantic case for function application. These techniques are standard from prior work [4], and we merely summarize the key ideas here to give background to our reader.

Formal Definition of Dynamic Analysis. We model language features for arithmetic operations ($e \odot e$), conditionals ($\mathtt{if0}(e)\{e\}\{e\}$), pairs ($\langle e, e\rangle$ and $\pi_i(e)$), functions ($\lambda x.\ e$ and $e(e)$) and references ($\mathtt{ref}(e)$, $!e$ and $e \leftarrow e$); the full language is shown in Figure 1. There is one base value: $r@^{\Sigma}_{m}$ for a real number result r tagged with dynamic analysis information Σ—the sensitivity analysis for the expression which evaluated to r—and m—the metric associated with the resulting value r. The sensitivity analysis Σ—also called a *sensitivity environment*—is a map from sensitive sources $o \in \mathsf{source}$ to how sensitive the result is w.r.t. that source. Our formalism includes two base metrics $m \in \mathsf{metric}$: $\mathtt{diff}$ and $\mathtt{disc}$ for absolute difference ($|x - y|$) and discrete distance (0 if $x = y$ and 1 otherwise) respectively—and two derived metrics: $\top$ and $\bot$ for the smallest metric larger than each base metric and largest metric smaller than each base metric, respectively. Each metric is commonly used when implementing differentially private algorithms. Pair values ($\langle v, v\rangle$), closure values ($\langle \lambda x.\ e \mid \rho\rangle$) and reference values ($\ell$) do not contain dynamic analysis information.

Our dynamic analysis is described formally as a big-step relation $\rho \vdash \ulcorner \sigma, e \lrcorner \Downarrow_n \ulcorner \sigma, v \lrcorner$ where $\rho \in \mathsf{var} \rightharpoonup \mathsf{value}$ is the lexical environment mapping lexical variables to values, $\sigma \in \mathsf{loc} \rightharpoonup \mathsf{value}$ is the dynamic environment (i.e., the heap, or store) mapping dynamically allocated references to values, e is the expression being executed, and v is the resulting runtime value which also includes dynamic analysis information. We write gray box corners around the "input" configuration $\ulcorner \sigma, e \lrcorner$ and the "output" configuration $\ulcorner \sigma, v \lrcorner$ to aid readability. The index n is for step-indexing, and tracks the number of function applications which occurred in the process of evaluation. We show the full definition of the dynamic analysis in Figure 2.

Consider the following example:

$$\{x \mapsto 21@^{\{o \mapsto 1\}}_{\texttt{diff}}\} \vdash \varnothing, (x + x) \Downarrow_0 42@^{\{o \mapsto 2\}}_{\texttt{diff}}$$

This relation corresponds to a scenario where the program to evaluate and analyze is $x + x$, the variable x represents a sensitive source value o, we want to track sensitivity w.r.t. the absolute difference metric, and the initial value for x is 21. This information is encoded in an initial environment $\rho = \{x \mapsto 21@^{\{o \mapsto 1\}}_{\texttt{diff}}\}$. The result value is 42, and the resulting analysis reports that e is 2-sensitive in the source o w.r.t. the absolute difference metric. This analysis information is encoded in the return value $42@^{\{o \mapsto 2\}}_{\texttt{diff}}$. Because no function applications occur during evaluation, the step index n is 0.

Formal Definition of Function Sensitivity. Function sensitivity is encoded through multiple relation definitions:

1) $\rho_1, \sigma_1, e_1 \sim^{\Sigma}_{n} \rho_2, \sigma_2, e_2$ holds when the input triples ρ_1, σ_1, e_1 and ρ_2, σ_2, e_2 evaluate to output stores and values which are related by Σ. Note this definition decrements the step-index n, and is the constant relation when $n = 0$.
2) $r_1 \sim^{r}_{m} r_2$ holds when the difference between real numbers r_1 and r_2 w.r.t. metric m is less than r.
3) $v_1 \sim^{\Sigma}_{n} v_2$ holds when values v_1 and v_2 are related for initial distance Σ and step-index n. The definition is by case analysis on the syntactic category for values, such as:
   a) The relation on base values $r_1@^{\Sigma_1}_{m_1} \sim^{\Sigma}_{n} r_2@^{\Sigma_2}_{m_2}$ holds when Σ_1, Σ_2, m_1 and m_2 are pairwise equal, and when r_1 and r_2 are related by $\Sigma \cdot \Sigma_1$, where Σ is the initial distances between each input source o, and Σ_1 is how much r_1 and r_2 are allowed to differ as a linear function of input distances Σ, and where this function is applied via vector dot product $\cdot$.
   b) The relation on pair values $\langle v_{11}, v_{12} \rangle \sim^{\Sigma}_{m} \langle v_{21}, v_{22} \rangle$ holds when each element of the pair are pairwise related.
   c) The relation on function values $\langle \lambda x.\ e_1 \mid \rho_1 \rangle \sim^{\Sigma}_{n} \langle \lambda x.\ e_2 \mid \rho_2 \rangle$ holds when each closure returns related output configurations when evaluated with related inputs.
   d) The relation on locations $\ell_1 \sim^{\Sigma}_{n} \ell_2$ holds when the two locations are equal.
4) $\rho_1 \sim^{\Sigma}_{n} \rho_2$ holds when lexical environments ρ_1 and ρ_2 map all variables to related values.
5) $\sigma_1 \sim^{\Sigma}_{n} \sigma_2$ holds when dynamic environments σ_1 and σ_2 map all locations to related values.

Note that the definitions of $\rho_1, \sigma_1, e_1 \sim^{\Sigma}_{n} \rho_2, \sigma_2, e_2$ and $v \sim^{\Sigma}_{n} v$ are mutually recursive, but are well founded due to the decrement of the step index in the former relation. We show the full definition of these relations in Figure 1.

The function sensitivity of an expression is encoded first as a statement about expressions respecting relatedness, that is, returning related outputs when given related inputs, i.e., (assuming no use of the store) if $\rho_1 \sim^{\Sigma}_{n} \rho_2$ and $\rho_1 \vdash \varnothing, e \Downarrow_{n_1} \varnothing, v_1$ and $\rho_2 \vdash \varnothing, e \Downarrow_{n_2} \varnothing, v_2$ then $n_1 = n_2$ and $v_1 \sim^{\Sigma}_{n-n_1} v_2$. When instantiated to base types, we have: if $\rho_1 \sim^{\Sigma}_{n} \rho_2$ and $\rho_1 \vdash \varnothing, e \Downarrow_{n_1} \varnothing, r_1@^{\Sigma_1}_{m_1}$ and $\rho_2 \vdash \varnothing, e \Downarrow_{n_2} \varnothing, r_2@^{\Sigma_2}_{m_2}$ then $n_1 = n_2$, $\Sigma_1 = \Sigma_2$, $m_1 = m_2$ and $r_1 \sim^{\Sigma \cdot \Sigma_1}_{m_1} r_2$. The fully general form of this property is called *metric preservation*, which is the main property we prove in our formal development.

Metric Preservation. Metric preservation states that when given related initial configurations and evaluation outputs, then those outputs are related. Outputs include result values, as well as dynamic analysis results, and the relationship that holds demonstrates the soundness of the analysis results.

Theorem VII.1 (Metric Preservation)**.**

$$\begin{array}{rll} \textit{If:} & \rho_1 \sim^{\Sigma}_{n} \rho_2 & \textit{(H1)} \\ \textit{And:} & \sigma_1 \sim^{\Sigma}_{n} \sigma_2 & \textit{(H2)} \\ \textit{Then:} & \rho_1, \sigma_1, e \sim^{\Sigma}_{n} \rho_2, \sigma_2, e & \end{array}$$

That is, either $n = 0$ or $n = n' + 1$ and…

$$\begin{array}{rll} \textit{If:} & n_1 \leq n' & \textit{(H3)} \\ \textit{And:} & \rho_1 \vdash \ulcorner \sigma_1, e \lrcorner \Downarrow_{n_1} \ulcorner \sigma'_1, v_1 \lrcorner & \textit{(H4)} \\ \textit{And:} & \rho_2 \vdash \ulcorner \sigma_2, e \lrcorner \Downarrow_{n_2} \ulcorner \sigma'_2, v_2 \lrcorner & \textit{(H5)} \\ \textit{Then:} & n_1 = n_2 & \textit{(C1)} \\ \textit{And:} & \sigma'_1 \sim^{\Sigma}_{n'-n_1} \sigma'_2 & \textit{(C2)} \\ \textit{And:} & v_1 \sim^{\Sigma}_{n'-n_1} v_2 & \textit{(C3)} \end{array}$$

Proof. See detailed proof in the extended version of this paper [3]. □

Instantiating Metric Preservation. Metric preservation is not enough on its own to demonstrate sound dynamic analysis of function sensitivity. Suppose we execute the dynamic analysis on program e with initial environment ρ, yielding a final store σ, base value r, sensitivity environment Σ, metric m and step-index n as a result:

$$\rho \vdash \varnothing, e \Downarrow_n \sigma, r@^{\Sigma}_{m}$$

To know the sensitivity of e is to know a bound on two *arbitrary* runs of e, that is, using two arbitrary environments ρ_1 and ρ_2. Does Σ tell us this? Remarkably, it does, with one small condition: ρ_1 and ρ_2 must agree with ρ on all non-sensitive values. This is not actually limiting: a non-sensitive value is essentially auxiliary information; they are constants and fixed for the purposes of sensitivity and privacy.

We can encode the relationship that environments ρ and ρ_1 agree on all non-sensitive values as $\rho \sim^{\Sigma'} \rho_1$ for any Σ', and we allow for environments ρ and ρ_1 to differ on

$b \in \mathbb{B}$ $\qquad n \in \mathbb{N}$ $\qquad i \in \mathbb{Z}$ $\qquad r \in \mathbb{R}$ $\qquad x \in \mathsf{var}$

$o \in \mathsf{source}$		sensitive sources	$q \in \hat{\mathbb{R}}$	$::= r \mid \infty$	ext. reals
$\ell \in \mathsf{loc}$		reference locations	$\odot \in \mathsf{binop}$	$::= + \mid \ltimes \mid \rtimes$	operations
$e \in \mathsf{expr}$	$::= x$	variables	$m \in \mathsf{metric}$	$::= \mathtt{diff}$	absolute difference
	$\mid r$	real numbers		$\mid \mathtt{disc}$	discrete
	$\mid e \odot e$	arith. operations		$\mid \bot$	bot metric
	$\mid \mathtt{if0}(e)\{e\}\{e\}$	cond. branching		$\mid \top$	top metric
	$\mid \langle e, e \rangle$	pair creation	$v \in \mathsf{value}$	$::= r@^{\Sigma}_{m}$	tagged base value
	$\mid \pi_i(e)$	pair access		$\mid \langle v, v \rangle$	pair
	$\mid \lambda x.\ e$	function creation		$\mid \langle \lambda x.\ e \mid \rho \rangle$	function (closure)
	$\mid e(e)$	function application		$\mid \ell$	location (pointer)
	$\mid \mathtt{ref}(e)$	reference creation	$\rho \in \mathsf{env}$	$\triangleq \mathsf{var} \rightharpoonup \mathsf{value}$	value environment
	$\mid !e$	reference read	$\sigma \in \mathsf{store}$	$\triangleq \mathsf{loc} \rightharpoonup \mathsf{value}$	mutable store
	$\mid e \leftarrow e$	reference write	$\Sigma \in \mathsf{senv}$	$\triangleq \mathsf{source} \rightharpoonup \hat{\mathbb{R}}$	sens. environment

$$\boxed{\rho, \sigma, e \sim^{\Sigma}_{n} \rho, \sigma, e}$$

$$\begin{aligned}
\rho_1, \sigma_1, e_1 \ \sim^{\Sigma}_{0}\ \rho_2, \sigma_2, e_2 &\overset{\triangle}{\iff} \mathsf{true} \\
\rho_1, \sigma_1, e_1 \sim^{\Sigma}_{n+1} \rho_2, \sigma_2, e_2 &\overset{\triangle}{\iff} \forall n_1 \leq n, n_2, \sigma_1', \sigma_2', v_1, v_2. \\
&\qquad \rho_1 \vdash \ulcorner \sigma_1, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma_1', v_1 \lrcorner \ \wedge\ \rho_2 \vdash \ulcorner \sigma_2, e_2 \lrcorner \Downarrow_{n_2} \ulcorner \sigma_2', v_2 \lrcorner \\
&\quad \Rightarrow n_1 = n_2 \ \wedge\ \sigma_1' \sim^{\Sigma}_{n-n_1} \sigma_2' \ \wedge\ v_1 \sim^{\Sigma}_{n-n_1} v_2
\end{aligned}$$

$$\boxed{r \sim^{r}_{m} r}$$

$$\begin{aligned}
r_1 \sim^{r}_{\mathtt{diff}} r_2 &\overset{\triangle}{\iff} |r_1 - r_2| \leq r &\qquad r_1 \sim^{r}_{\bot} r_2 &\overset{\triangle}{\iff} r_1 \sim^{r}_{\mathtt{diff}} r_2 \ \wedge\ r_1 \sim^{r}_{\mathtt{disc}} r_2 \\
r_1 \sim^{r}_{\mathtt{disc}} r_2 &\overset{\triangle}{\iff} \begin{cases} 0 \leq r & \textit{if} \quad r_1 = r_2 \\ 1 \leq r & \textit{if} \quad r_1 \neq r_2 \end{cases} &\qquad r_1 \sim^{r}_{\top} r_2 &\overset{\triangle}{\iff} r_1 \sim^{r}_{\mathtt{diff}} r_2 \ \vee\ r_1 \sim^{r}_{\mathtt{disc}} r_2
\end{aligned}$$

$$\boxed{v \sim^{\Sigma}_{n} v}$$

$$\begin{aligned}
r_1@^{\Sigma_1}_{m_1} \sim^{\Sigma}_{n} r_2@^{\Sigma_2}_{m_2} &\overset{\triangle}{\iff} \Sigma_1 = \Sigma_2 \ \wedge\ m_1 = m_2 \ \wedge\ r_1 \sim^{\Sigma \cdot \Sigma_1}_{m_1} r_2 \\
\langle v_{11}, v_{12} \rangle \sim^{\Sigma}_{n} \langle v_{21}, v_{22} \rangle &\overset{\triangle}{\iff} v_{11} \sim^{\Sigma}_{n} v_{21} \ \wedge\ v_{21} \sim^{\Sigma}_{n} v_{22} \\
\langle \lambda x.\ e_1 \mid \rho_1 \rangle \sim^{\Sigma}_{n} \langle \lambda x.\ e_2 \mid \rho_2 \rangle &\overset{\triangle}{\iff} \forall n' \leq n, v_1, v_2, \sigma_1, \sigma_2.\ \sigma_1 \sim^{\Sigma}_{n'} \sigma_2 \ \wedge\ v_1 \sim^{\Sigma}_{n'} v_2 \\
&\quad \Rightarrow \sigma_1, \{x \mapsto v_1\} \uplus \rho_1, e_1 \sim^{\Sigma}_{n'} \sigma_2, \{x \mapsto v_2\} \uplus \rho_2, e_2 \\
\ell_1 \sim^{\Sigma}_{n} \ell_2 &\overset{\triangle}{\iff} \ell_1 = \ell_2
\end{aligned}$$

$$\boxed{\begin{array}{c} \rho \sim^{\Sigma}_{n} \rho \\ \sigma \sim^{\Sigma}_{n} \sigma \end{array}}$$

$$\begin{aligned}
\rho_1 \sim^{\Sigma}_{n} \rho_2 &\overset{\triangle}{\iff} \forall x \in (\mathrm{dom}(\rho_1) \cup \mathrm{dom}(\rho_2)).\ \rho_1(x) \sim^{\Sigma}_{n} \rho_2(x) \\
\sigma_1 \sim^{\Sigma}_{n} \sigma_2 &\overset{\triangle}{\iff} \forall \ell \in (\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2)).\ \sigma_1(\ell) \sim^{\Sigma}_{n} \sigma_2(\ell)
\end{aligned}$$

Fig. 1. Formal Syntax & Step-indexed Logical Relation.

any sensitive value while agreeing on non-sensitive values as $\rho \sim^{\{o \mapsto \infty\}} \rho_1$. Under such an assumption, Σ and m are sound dynamic analysis results for two arbitrary runs of e, i.e., under environments ρ_1 and ρ_2, so long as one of those environments agrees with ρ—the environment used to compute the dynamic analysis. We encode this property formally as the following corollary to metric preservation:

Corollary VII.1.1 (Sound Dynamic Analysis for Sensitivity).

If:	$n_1 < n,\ n_2 < n$ *and* $n_3 < n$	*(H1)*
And:	$\rho \sim^{\{o \mapsto \infty\}}_{n} \rho_1$	*(H2)*
And:	$\rho \vdash \ulcorner \varnothing, e \lrcorner \Downarrow_{n_1} \ulcorner \sigma, r@^{\Sigma}_{m} \lrcorner$	*(H3)*
And:	$\rho_1 \sim^{\Sigma'}_{n} \rho_2$	*(H4)*
And:	$\rho_1 \vdash \ulcorner \varnothing, e \lrcorner \Downarrow_{n_2} \ulcorner \sigma_1, r_1@^{\Sigma_1}_{m_1} \lrcorner$	*(H5)*
And:	$\rho_2 \vdash \ulcorner \varnothing, e \lrcorner \Downarrow_{n_3} \ulcorner \sigma_2, r_2@^{\Sigma_2}_{m_2} \lrcorner$	*(H6)*
Then:	$r_1 \sim^{\Sigma' \cdot \Sigma}_{m_1} r_2$	*(C1)*

Proof.
By Metric Preservation, *(H2)*, *(H1)*, *(H3)* and *(H5)* we have $\Sigma_1 = \Sigma$ and $m_1 = m$. By Metric Preservation, *(H4)*, *(H1)*, *(H5)* and *(H6)* we have proved the goal *(C1)*. ☐

$$\rceil\_\lceil^{r} \in \hat{\mathbb{R}} \rightarrow \mathbb{R} \qquad \rceil\_\lceil^{r} \in \text{senv} \rightarrow \text{sens}$$

$$\rceil r' \lceil^{r} \triangleq \begin{cases} 0 & \textit{if} \quad r' = 0 \\ r & \textit{if} \quad r' \neq 0 \end{cases} \qquad \rceil \Sigma \lceil^{r}(o) \triangleq \rceil \Sigma(o) \lceil^{r}$$

$$\text{alloc}(\mathcal{L}) \notin \mathcal{L} \in \wp(\text{loc}) \qquad \mathbf{Z} = \{o \mapsto 0\}$$

$$\boxed{\rho \vdash \ulcorner \sigma, e \lrcorner \Downarrow_{n} \ulcorner \sigma, v \lrcorner}$$

$$\textsc{Var}\ \frac{}{\rho \vdash \ulcorner \sigma, x \lrcorner \Downarrow_{0} \ulcorner \sigma, \rho(x) \lrcorner} \qquad \textsc{Real}\ \frac{}{\rho \vdash \ulcorner \sigma, r \lrcorner \Downarrow_{0} \ulcorner \sigma, r@^{\mathbf{Z}}_{\texttt{disc}} \lrcorner} \qquad \textsc{Fun}\ \frac{}{\rho \vdash \ulcorner \sigma, \lambda x.\ e \lrcorner \Downarrow_{0} \ulcorner \sigma, \langle \lambda x.\ e \mid \rho \rangle \lrcorner}$$

$$\textsc{Plus}\ \frac{\rho \vdash \ulcorner \sigma, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma', r_1@^{\Sigma_1}_{m_1} \lrcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \lrcorner \Downarrow_{n_2} \ulcorner \sigma'', r_2@^{\Sigma_2}_{m_2} \lrcorner}{\rho \vdash \ulcorner \sigma, e_1 + e_2 \lrcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', (r_1 + r_2)@^{\Sigma_1+\Sigma_2}_{m_1 \sqcup m_2} \lrcorner}$$

$$\textsc{Times-L}\ \frac{\rho \vdash \ulcorner \sigma, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma', r_1@^{\mathbf{Z}}_{m_1} \lrcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \lrcorner \Downarrow_{n_2} \ulcorner \sigma'', r_2@^{\Sigma_2}_{m_2} \lrcorner}{\rho \vdash \ulcorner \sigma, e_1 \ltimes e_2 \lrcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', (r_1 \times r_2)@^{r_1\Sigma_2}_{m_2} \lrcorner}$$

$$\textsc{Times-R}\ \frac{\rho \vdash \ulcorner \sigma, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma', r_1@^{\Sigma_1}_{m_1} \lrcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \lrcorner \Downarrow_{n_2} \ulcorner \sigma'', r_2@^{\mathbf{Z}}_{m_2} \lrcorner}{\rho \vdash \ulcorner \sigma, e_1 \rtimes e_2 \lrcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', (r_1 \times r_2)@^{r_2\Sigma_1}_{m_1} \lrcorner}$$

$$\textsc{IfZ-T}\ \frac{\rho \vdash \ulcorner \sigma, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma', r_1@^{\mathbf{Z}}_{m_1} \lrcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \lrcorner \Downarrow_{n_2} \ulcorner \sigma'', v_2 \lrcorner \qquad r_1 = 0}{\rho \vdash \ulcorner \sigma, \texttt{if0}(e_1)\{e_2\}\{e_3\} \lrcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', v_2 \lrcorner}$$

$$\textsc{IfZ-F}\ \frac{\rho \vdash \ulcorner \sigma, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma', r_1@^{\mathbf{Z}}_{m_1} \lrcorner \qquad \rho \vdash \ulcorner \sigma', e_3 \lrcorner \Downarrow_{n_2} \ulcorner \sigma'', v_3 \lrcorner \qquad r_1 \neq 0}{\rho \vdash \ulcorner \sigma, \texttt{if0}(e_1)\{e_2\}\{e_3\} \lrcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', v_3 \lrcorner}$$

$$\textsc{Pair}\ \frac{\rho \vdash \ulcorner \sigma, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma', v_1 \lrcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \lrcorner \Downarrow_{n_2} \ulcorner \sigma'', v_2 \lrcorner}{\rho \vdash \ulcorner \sigma, \langle e_1, e_2 \rangle \lrcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', \langle v_1, v_2 \rangle \lrcorner} \qquad \textsc{Proj}\ \frac{\rho \vdash \ulcorner \sigma, e \lrcorner \Downarrow_{n} \ulcorner \sigma', \langle v_1, v_2 \rangle \lrcorner}{\rho \vdash \ulcorner \sigma, \pi_{n'}(e) \lrcorner \Downarrow_{n} \ulcorner \sigma', v_{n'} \lrcorner}$$

$$\textsc{Ref}\ \frac{\rho \vdash \ulcorner \sigma, e \lrcorner \Downarrow_{n} \ulcorner \sigma', v \lrcorner \qquad \ell = \text{alloc}(\text{dom}(\sigma))}{\rho \vdash \ulcorner \sigma, \texttt{ref}(e) \lrcorner \Downarrow_{n} \ulcorner \{\ell \mapsto v\} \uplus \sigma', \ell \lrcorner} \qquad \textsc{Read}\ \frac{\rho \vdash \ulcorner \sigma, e \lrcorner \Downarrow_{n} \ulcorner \sigma', \ell \lrcorner}{\rho \vdash \ulcorner \sigma, !e \lrcorner \Downarrow_{n} \ulcorner \sigma', \sigma'(\ell) \lrcorner}$$

$$\textsc{Write}\ \frac{\rho \vdash \ulcorner \sigma, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma', \ell \lrcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \lrcorner \Downarrow_{n_2} \ulcorner \sigma'', v \lrcorner}{\rho \vdash \ulcorner \sigma, e_1 \leftarrow e_2 \lrcorner \Downarrow_{n_1+n_2} \ulcorner \sigma''[\ell \mapsto v], v \lrcorner}$$

$$\textsc{App}\ \frac{\rho \vdash \ulcorner \sigma, e_1 \lrcorner \Downarrow_{n_1} \ulcorner \sigma', \langle \lambda x.\ e \mid \rho \rangle \lrcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \lrcorner \Downarrow_{n_2} \ulcorner \sigma'', v \lrcorner \qquad \{x \mapsto v\} \uplus \rho \vdash \ulcorner \sigma'', e \lrcorner \Downarrow_{n_3} \ulcorner \sigma''', v' \lrcorner}{\rho \vdash \ulcorner \sigma, e_1(e_2) \lrcorner \Downarrow_{n_1+n_2+n_3+1} \ulcorner \sigma''', v' \lrcorner}$$

Fig. 2. Formal Big-step, Step-indexed Semantics and Metafunctions.

Note that the final results are related using Σ—the analysis result derived from an execution under ρ—while r_1 and r_2 are derived from executions under unrelated (modulo auxiliary information) environments ρ_1 and ρ_2.

In simpler terms, this corollary shows that even though the dynamic analysis only sees one particular execution of the program, it is accurate in describing the sensitivity of the program—even though the notion of sensitivity considers two arbitrary runs of the program, including those whose inputs differ entirely from those used in the dynamic analysis.

VIII. Implementation & Case Studies

We have developed a reference implementation of DDuo as a Python library, using the approaches described in Sections IV, V, and VI.

A major goal in the design of DDuo is seamless integration with other libraries. Our reference implementation provides initial support for NumPy, Pandas, and Sklearn. DDuo provides hooks for tracking both sensitivity and privacy, to simplify integrating with additional libraries.

We present case studies which focus on demonstrating DDuo's (1) similarity to regular Python code, (2) applicability

Name	Type			Conditions
laplace :	$(\epsilon : \mathbb{R}, value : \mathbb{R})$	$\rightarrow \mathbb{R}$	*where*	$\text{priv}(\text{laplace}(\epsilon, value)) \triangleq \epsilon$
gauss :	$(\epsilon : \mathbb{R}, \delta : \mathbb{R}, value : \mathbb{R})$	$\rightarrow \mathbb{R}$	*where*	$\text{priv}(\text{gauss}(\epsilon, \delta, value)) \triangleq (\epsilon, \delta)$
ed_odo :	$(f : A \rightarrow B, in : A)$	$\rightarrow (out : B, (\epsilon, \delta) : (\mathbb{R}, \mathbb{R}))$	*where*	$\text{priv}(\text{ed\_odo}(f, in)) \triangleq \text{priv}(f(in))$
renyi_odo :	$(f : A \rightarrow B, in : A)$	$\rightarrow (out : B, (\alpha, \epsilon) : (\mathbb{R}, \mathbb{R}))$	*where*	$\text{priv}(\text{renyi\_odo}(f, in)) \triangleq \text{priv}(f(in))$
ed_filter :	$(f : A \rightarrow B, in : A, (\epsilon, \delta) : (\mathbb{R}, \mathbb{R}))$	$\rightarrow (out : B)$	*where*	$(\epsilon, \delta) \geq \text{priv}(f(in))$
renyi_filter :	$(f : A \rightarrow B, in : A, (\alpha, \epsilon) : (\mathbb{R}, \mathbb{R}))$	$\rightarrow (out : B)$	*where*	$(\alpha, \epsilon) \geq \text{priv}(f(in))$
conv_renyi :	$(f : A \rightarrow B, in : A, \delta : \mathbb{R})$	$\rightarrow (out : B)$	*where*	$\text{priv}(f(...)) = (\alpha, \epsilon)$
			and	$\text{conv}(\alpha, \epsilon, \delta) = (\epsilon, \delta)$
svt :	$(\epsilon : \mathbb{R}, qs : [A \rightarrow B], data : [A], t : \mathbb{R})$	$\rightarrow \mathbb{N}$	*where*	for q in qs, $sens(q) = 1$
			and	$\text{priv}(svt(\epsilon, qs, data, t)) \triangleq \epsilon$
exp :	$(\epsilon : \mathbb{R}, q : A \rightarrow B, data : [A])$	$\rightarrow \mathbb{N}$	*where*	$\text{priv}(exp(\epsilon, q, data)) \triangleq \epsilon$
map :	$(f : A \rightarrow B, in : [A])$	$\rightarrow [B]$	*where*	$\text{sens}(\text{map}(f, \_)) \triangleq \text{sens}(f(\_))$

$priv$: denotes the privacy leakage of a program given by dynamic analysis; $sens$: denotes the sensitivity of a program given by dynamic analysis; $conv$: represents the conversion equation from renyi to approximate differential privacy
Types are written as follows: the $\rightarrow$ symbol is used to seperate the domain and range of a function, either of which may be given as an atomic type such as a natural number ($\mathbb{N}$), or as a tuple which is a comma-seperated list of types surrounded by parentheses, or as a symbol (A) indicating parametric polymorphism (generics). In some cases, types may also be accompanied with a placeholder name (ϵ : $\mathbb{R}$) for further qualification in the $where$ clause.

Fig. 3. Core API Methods

to complex algorithms, (3) easy integration with existing libraries. Although our approach is automatic, DDuo is able to compute privacy leakage bounds that match those of bespoke privacy-preserving algorithms. In this section we focus on the Noisy Gradient Descent case study, other case studies have been moved to the extended version of this paper [3] due to space requirements. We introduce new *adaptive* variants of algorithms that stop early when possible to conserve privacy budget. These variants cannot be verified by prior work using purely static analyses, because their privacy parameters are chosen adaptively.

Run-time overhead. Run-time overhead is a key concern in DDuo's instrumentation for dynamic analysis. Fortunately, experiments on our case studies suggest that the overhead of DDuo's analysis is generally low. Table I presents the run-time performance overhead of DDuo's analysis as a percentage *increase* of total runtime. The worst overhead time observed in our case studies was less than 60%.

In certain rare cases, DDuo's overhead can be much higher. For example, mapping the function `lambda x: x + 1` over a list of 1 million numbers takes 160x longer under DDuo than in standard Python. The overhead in this case comes from a combination of factors: first, DDuo's `map` function, itself implemented in Python, is much slower than Python's built-in `map` operator; second, DDuo's `map` function requires the creation of a new `Sensitive` object for each element of the list—a slow operation in Python.

Fortunately, the same strategies for producing high-performance Python code *without* privacy also help reduce DDuo's overhead. Python's performance characteristics have prompted the development of higher-performance libraries like NumPy and Pandas, which essentially provide data-processing combinators that programmers compose. By providing sensitivity annotations for these libraries, we can re-use these high-performance implementations and avoid creating extra objects. As a result, none of our case studies demonstrates the worst-case performance overhead described above.

Technique	Ref.	Libraries Used	Overhead
Noisy Gradient Descent	[13]	NumPy	6.42%
Multiplicative Weights (MWEM)	[30]	Pandas	14.90%
Private Naive Bayes Classification	[45]	DiffPrivLib	12.44%
Private Logistic Regression	[17]	DiffPrivLib	56.33%

TABLE I
LIST OF CASE STUDIES INCLUDED WITH THE DDUO IMPLEMENTATION.

Case study: gradient descent with NumPy. Our first case study (Figure 4) is a simple machine learning algorithm based on [13] implemented directly with DDuo-instrumented NumPy primitives. The remaining case studies appear in the Appendix.

Given a dataset X which is a list of feature vectors representing training examples, and a vector y which classifies each element of X in a finite set, gradient descent is the process of computing a model (a linear set of weights) which most accurately classifies a new, never seen before training example, based on our pre-existing evidence represented by the model.

Gradient descent works by first specifying a loss function that computes the effectiveness of a model in classifying a given dataset according to its known labels. The algorithm then iteratively computes a model that minimizes the loss function, by calculating the gradient of the loss function and moving the model in the opposite direction of the gradient.

One method of ensuring privacy in gradient descent involves adding noise to the gradient calculation, which is the only part of the process that is exposed to the private training data. In order to add noise to the gradient, it is convenient to bound its sensitivity via clipping to some L2 norm. In this example, clipping occurs in the `gradient_sum` function

```
def dp_gradient_descent(iterations, alpha, eps):
  eps_i = eps/iterations
  theta = np.zeros(X.shape[1])
  with dduo.RenyiFilter(alpha,eps_max):
    with dduo.RenyiOdometer((alpha, eps)) as odo:
      noisy_count = dduo.renyi_gauss(α=alpha,
        ϵ=eps,X.shape[0])
        priv_acc = 0
        acc_diff = 1
      while acc_diff > 0.05:
        grad_sum = gradient_sum(theta,
          X_train, y_train, sensitivity)
        noisy_grad_sum = dduo.gauss_vec(grad_sum,
          α=alpha,ϵ=eps_i)
        noisy_avg_grad = noisy_grad_sum/noisy_count
        theta = np.subtract(theta, noisy_avg_grad)
        priv_acc_curr = dduo.renyi_gauss(alpha,
          eps_acc, accuracy(theta))
        acc_diff =  priv_acc_curr - priv_acc
        priv_acc = priv_acc_curr
      print(odo)
    return theta

theta = dp_gradient_descent(iterations,
  α=alpha, ϵ=epsilon)
acc = dduo.renyi_gauss(alpha,
  eps_acc, accuracy(theta))
print(f"final accuracy: {acc}")
```

```
Odometer_(α, ϵ) ({data.csv ↦ (10.0, 2.40)})
final accuracy: 0.753
```

Fig. 4. Gradient Descent with NumPy

before summation.

The original implementation of this algorithm [13] was based on the advanced composition theorem. Advanced composition improves on sequential composition by providing much tighter privacy bounds over several iterations, but requires the analyst to fix the number of iterations up front, regardless of how many iterations the gradient descent algorithm actually takes to converge to minimal error.

We present a modified version based on *adaptive* Renyi differential privacy which provides not only a tighter analysis of the privacy leakage over several iterations, but also allows the analyst to halt computation adaptively (conserving the remaining privacy budget) once a certain level of model accuracy has been reached, or loss has been minimized. We introduce random noise to the accuracy calculation because it is a computation on the sensitive input training dataset in this case.

IX. Related Work

Dynamic Enforcement of Differential Privacy. The first approach for dynamic enforcement of differential privacy was PINQ [36]. Since then several works have been based on PINQ, such as Featherweight PINQ [21] which models PINQ formally and proves that any programs which use its simplified PINQ API are differentially private. ProPer [22] is a system (based on PINQ) designed to maintain a privacy budget for each individual in a database system, and operates by silently dropping records from queries when their privacy budget is exceeded. UniTrax [38] follows up on ProPer: this system allows per-user budgets but gets around the issue of silently dropping records by tracking queries against an abstract database as opposed to the actual database records. These approaches are limited to an embedded DSL for expressing relational database queries, and do not support general purpose programming.

A number of programming frameworks for differential privacy have been developed as libraries for existing programming languages. DPella [35] is a Haskell library that provides static bounds on the accuracy of differentially private programs. Diffprivlib [32] (for Python) and Google's library [49] (for several languages) provide differentially private algorithms, but do not track sensitivity or privacy as these algorithms are composed. ϵktelo [51] executes programmer-specified *plans* that encode differentially private algorithms using framework-supplied building blocks.

Dynamic Information Flow Control. Our approach to dynamic enforcement of differential privacy can be seen as similar to work on dynamic information flow control (IFC) and taint analysis [7]. The sensitivities that we attach to values are comparable to IFC labels. However, dynamic IFC typically allows the programmer to branch on sensitive information and handles implicit flows dynamically. DDuo prevents branching on sensitive information, similar to an approach taken for preventing side-channels in the "constant time" programming discipline for cryptographic code [8]. Another connection with this line of work is that our use of the logical relations proof technique for a differential privacy theorem is similar to the usage of this technique for noninterference theorems [43, 31, 1, 15, 6, 27].

Static Verification of Differential Privacy. The first approach for static verification of differential privacy was Fuzz [41], which used *linear types* to analyze sensitivity. DFuzz [26] adds dependent types, and later work [18, 40] extends the approach to (ϵ, δ)-differential privacy. Fuzzi [53] integrates a Fuzz-like type system with an expressive program logic. *Adaptive Fuzz* [50] combines a Fuzz-style *static* type system for sensitivity analysis with a *dynamic* privacy analysis using privacy filters and odometers; *Adaptive Fuzz* is most similar to our approach, but uses a static sensitivity analysis. All of these approaches require additional type annotations.

A second category is based on *approximate couplings* [9]. The apRHL [11, 12], apRHL$^+$ [10], and span-apRHL [44] relational logics are extremely expressive, but less amenable to automation. Albarghouthi and Hsu [5] use an alternative approach based on constraint solving to synthesize approximate couplings. A third approach is based on *randomness alignments*; LightDP [52] and ShadowDP [47] take this approach. Randomness alignments are effective for verifying low-level mechanisms like the sparse vector technique. The latter two categories are generally restricted to first order imperative programs.

Dynamic Testing for Differential Privacy. A recent line of work [14, 19, 46, 49] has resulted in approaches for *testing* differentially private programs. These approaches generate a series of neighboring inputs, run the program many times on the neighboring inputs, and raise an alarm if a counterexample is found. These approaches do not require type annotations, but do require running the program many times.

X. Conclusion

We have presented DDuo, a dynamic analysis that supports general-purpose differentially private programming with an emphasis on machine learning. We have formalized the sensitivity analysis of DDuo and proven its soundness using a step-indexed logical relation. Our case studies demonstrate the utility of DDuo by enforcing adaptive variants of several differentially private state-of-the-art machine learning algorithms from the ground up, while integrating with some of Python's most popular libraries for data analysis. It is our hope that the usability and strong guarantees of DDuo will inspire data analysts, technology corporations, researchers, and students in computer science to continue to build a community and culture of verifiable data privacy.

Acknowledgments

The authors would like to thank the CSF reviewers for their helpful comments. This work was supported in part by DARPA & SPAWAR under contract N66001-15-C-4066, and the National Science Foundation under award CCF-1901278. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. Association for Computing Machinery.

[2] John M. Abowd. The u.s. census bureau adopts differential privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '18, page 2867, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Chike Abuah, Alex Silence, David Darais, and Joe Near. Dduo: General-purpose dynamic analysis for differential privacy, 2021.

[4] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Programming Languages and Systems*, pages 69–83, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] Aws Albarghouthi and Justin Hsu. Synthesizing coupling proofs of differential privacy. *PACMPL*, 2(POPL):58:1–58:30, 2018.

[6] Maximilian Algehed and Jean-Philippe Bernardy. Simple noninterference from parametricity. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[7] Thomas H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS '09*, 2009.

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. System-level non-interference of constant-time cryptography. part I: model. *J. Autom. Reason.*, 63(1):1–51, 2019.

[9] Gilles Barthe, Thomas Espitau, Justin Hsu, Tetsuya Sato, and Pierre-Yves Strub. Relational $\star$-liftings for differential privacy. *Logical Methods in Computer Science*, 15(4), 2019.

[10] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 749–758, New York, NY, USA, 2016. Association for Computing Machinery.

[11] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 97–110, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3), November 2013.

[13] Raef Bassily, Adam Smith, and Abhradeep Thakurta. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 464–473. IEEE, 2014.

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

[15] William J. Bowman and Amal Ahmed. Noninterference for free. *SIGPLAN Not.*, 50(9):101–113, August 2015.

[16] Mark Bun and Thomas Steinke. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of Cryptography Conference*, pages 635–658. Springer, 2016.

[17] K. Chaudhuri and C. Monteleoni. Privacy-preserving logistic regression. In *NIPS*, 2008.

[18] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. Probabilistic relational reasoning via metrics. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–19. IEEE, 2019.

[19] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. Detecting violations of differ-

ential privacy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 475–489, 2018.

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

[21] Hamid Ebadi and David Sands. Featherweight pinq, 2015.

[22] Hamid Ebadi, David Sands, and Gerardo Schneider. Differential privacy: Now it's getting personal. volume 50, 01 2015.

[23] Georgina Evans and Gary King. Statistically valid inferences from differentially private data releases, with application to the facebook urls dataset, Working Paper.

[24] Georgina Evans, Gary King, Margaret Schwenzfeier, and Abhradeep Thakurta. Statistically valid inferences from privacy protected data, Working Paper.

[25] Vitaly Feldman and Tijana Zrnic. Individual privacy accounting via a renyi filter. *arXiv preprint arXiv:2008.11193*, 2020.

[26] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 357–370, 2013.

[27] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[28] Miguel Guevara. Enabling developers and organizations to use differential privacy, Sep 2019.

[29] Miguel Guevara, Mirac Vuslat Basaran, Sasha Kulankhina, and Badih Ghazi. Expanding our differential privacy library, Jun 2020.

[30] Moritz Hardt, Katrina Ligett, and Frank McSherry. A simple and practical algorithm for differentially private data release. In *Advances in Neural Information Processing Systems*, pages 2339–2347, 2012.

[31] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 365–377, New York, NY, USA, 1998. Association for Computing Machinery.

[32] Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. Diffprivlib: the ibm differential privacy library. *arXiv preprint arXiv:1907.02444*, 2019.

[33] Daniel Kifer, Solomon Messing, Aaron Roth, Abhradeep Thakurta, and Danfeng Zhang. Guidelines for implementing and auditing differentially private systems, 2020.

[34] Gary King and Nathaniel Persily. Unprecedented facebook urls dataset now available for academic research through social science one, Feb 2020.

[35] Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. A programming framework for differential privacy with accuracy concentration bounds. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 411–428. IEEE, 2020.

[36] Frank D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 19–30, New York, NY, USA, 2009. Association for Computing Machinery.

[37] Ilya Mironov. Rényi differential privacy. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 263–275. IEEE Computer Society, 2017.

[38] Reinhard Munz, Fabienne Eigner, Matteo Maffei, Paul Francis, and Deepak Garg. Unitrax: Protecting data privacy with discoverable biases. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 278–299, Cham, 2018. Springer International Publishing.

[39] Chaya Nayak. New privacy-protected facebook data for independent research on social media's impact on democracy, Feb 2020.

[40] Joseph P Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, et al. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[41] Jason Reed and Benjamin C Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 157–168, 2010.

[42] Ryan M. Rogers, Salil P. Vadhan, Aaron Roth, and Jonathan Ullman. Privacy odometers and filters: Pay-as-you-go composition. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1921–1929, 2016.

[43] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. In S. Doaitse Swierstra, editor, *Programming Languages and Systems*, pages 40–58, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[44] T. Sato, G. Barthe, M. Gaboardi, J. Hsu, and S. Katsumata. Approximate span liftings: Compositional semantics for relaxations of differential privacy. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14, June 2019.

[45] J. Vaidya, B. Shafiq, A. Basu, and Y. Hong. Differentially private naive bayes classification. In *2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volume 1, pages 571–576, 2013.

[46] Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng

Zhang. Checkdp: An automated and integrated approach for proving differential privacy or finding precise counterexamples. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 919–938, 2020.

[47] Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. Proving differential privacy with shadow execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 655–669, 2019.

[48] Royce J. Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. Differentially private SQL with bounded user contribution. *CoRR*, abs/1909.01917, 2019.

[49] Royce J Wilson, Celia Yuxing Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. Differentially private sql with bounded user contribution. *Proceedings on Privacy Enhancing Technologies*, 2020(2), 2020.

[50] Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. A framework for adaptive differential privacy. *Proc. ACM Program. Lang.*, 1(ICFP):10:1–10:29, 2017.

[51] Dan Zhang, Ryan McKenna, Ios Kotsogiannis, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. Ektelo: A framework for defining differentially-private computations. In *Proceedings of the 2018 International Conference on Management of Data*, pages 115–130, 2018.

[52] Danfeng Zhang and Daniel Kifer. Lightdp: towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 888–901, 2017.

[53] Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C Pierce, and Aaron Roth. Fuzzi: A three-level logic for differential privacy. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–28, 2019.

[54] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, Scottsdale, Arizona, 2014.