A Programming Language for Data Privacy with Accuracy Estimations

ELISABET LOBO-VESGA and ALEJANDRO RUSSO, Chalmers University of Technology
MARCO GABOARDI, Boston University

Differential privacy offers a formal framework for reasoning about the privacy and accuracy of computations on private data. It also offers a rich set of building blocks for constructing private data analyses. When carefully calibrated, these analyses simultaneously guarantee the privacy of the individuals contributing their data, and the accuracy of the data analysis results, inferring useful properties about the population. The compositional nature of differential privacy has motivated the design and implementation of several programming languages to ease the implementation of differentially private analyses. Even though these programming languages provide support for reasoning about privacy, most of them disregard reasoning about the accuracy of data analyses. To overcome this limitation, we present DPella, a programming framework providing data analysts with support for reasoning about privacy, accuracy, and their trade-offs. The distinguishing feature of DPella is a novel component that statically tracks the accuracy of different data analyses. To provide tight accuracy estimations, this component leverages taint analysis for automatically inferring *statistical independence* of the different noise quantities added for guaranteeing privacy. We evaluate our approach by implementing several classical queries from the literature and showing how data analysts can calibrate the privacy parameters to meet the accuracy requirements, and vice versa.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**; • **Information systems → Query languages**; • **Software and its engineering → Functional languages**; **Extensible languages**;

Additional Key Words and Phrases: Accuracy, concentration bounds, differential privacy, databases, Haskell

ACM Reference format:
Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2020. A Programming Language for Data Privacy with Accuracy Estimations. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 6 (June 2021), 42 pages.
https://doi.org/10.1145/3452096

1 INTRODUCTION

Differential privacy (DP) [16] is emerging as a viable solution to release statistical information about the population without compromising data subjects' privacy. A standard way to achieve

This work was initiated by a STINT Initiation grant (IB 2017-77023) and supported by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011), as well as the Swedish research agency Vetenskapsrådet. M. Gaboardi's work was partially funded by the National Science Foundation under grants 1718220 and 1845803.

Authors' addresses: E. Lobo-Vesga and A. Russo, Chalmers University of Technology, Göteborg, Sweden; emails: {elilob, russo}@chalmers.se; M. Gaboardi, Boston University, Boston, MA; email: gaboardi@bu.edu.

0164-0925/2020/06-ART6 $15.00
https://doi.org/10.1145/3452096

DP is by adding some statistical noise to the result of a data analysis. If the noise is carefully calibrated, it provides *privacy* protection for the individuals contributing their data. At the same time, it enables the inference of *accurate* information about the population from which the data are drawn. Thanks to its quantitative formulation, quantifying privacy by means of the parameters ϵ and δ, DP provides a mathematical framework for rigorously reasoning about the *privacy-accuracy trade-offs*. The accuracy requirement is not baked into the definition of DP; instead, it is a constraint made explicit for a specific task at hand when designing a differentially private data analysis.

An important property of DP is *composability*. Multiple differentially private data analyses can be composed with a graceful degradation of the privacy parameters (ϵ and δ). This property allows to reason about privacy as a *budget*: a data analyst can decide how much privacy budget (the ϵ parameter) assigns to each of her analyses. The compositionality aspects of DP motivated the design of several programming frameworks [4, 7, 23, 26, 31, 40, 50, 51, 60, 62–64] and tools [24, 38, 41, 44] to help analysts design their own differentially private consults. At a high level, most of these programming frameworks and tools are based on a similar idea for reasoning about privacy: provide primitives for fundamental differentially private analyses as building blocks, and use composition properties to combine these building blocks. During composition, these systems ensure that the privacy cost of each data analysis sums up and that the total cost does not exceed the privacy budget. The programming frameworks also provide general support to further combine, through programming techniques, the different building blocks and the results of several data analyses. Differently, DP tools are optimized for specific tasks at the price of restricting the kinds of data analyses they can support.

Unfortunately, reasoning about accuracy is *less compositional* than reasoning about privacy. It depends both on the specific task at hand and on the specific accuracy measure that one is interested in offering to data analysts. Despite this, when restricted to specific mechanisms and specific forms of data analyses, one can measure accuracy through estimates given as *confidence intervals*, or error bounds. As an example, most of the standard mechanisms from the DP literature come with theoretical confidence intervals or error bounds that can be exposed to data analysts to allow them to make informed decisions about the consults they want to run. This approach has been integrated in tools such as GUPT [44], PSI [24], and APEx [25]. Users of these tools can specify the target confidence interval they want to achieve, and the tools adjust the privacy parameters accordingly, when sufficient budget is available.[1]

In contrast, all of the programming frameworks proposed so far [4, 7, 23, 26, 31, 40, 50, 51, 60, 62–64] do not offer any support to programmers or data analysts for tracking, and reasoning about, the accuracy of their data analyses. This phenomenon is in large part due to the complex nature of accuracy reasoning, concerning privacy analyses, when designing arbitrary data analyses that users of these frameworks may want to implement and execute. In this work, we address this limitation by building a programming framework for designing differentially private analyses, which supports a compositional form of reasoning about accuracy.

Contribution

Our main contribution is showing how programming frameworks can internalize the use of *probabilistic bounds* [15] for composing different confidence intervals or error bounds, in an automated way. Probabilistic bounds are part of the standard toolbox for the analysis of randomized algorithms. Specifically, they are the tools that DP algorithm designers usually employ for the accuracy analysis of classical mechanisms [17, 19]. Two important probabilistic bounds are the *union*

[1]APEx goes beyond this by also helping users select the right differentially private mechanism to achieve the required accuracy.

bound, which can be used to compose errors with no assumption on the way the random noise is generated, and *Chernoff bound*, which applies to the sum of random noise when the different random variables characterizing noise generation are statistically independent (see Section 5). When applicable, and when the number of random variables grows, Chernoff bound usually gives a much "tighter" error estimation than the union bound.

Barthe et al. [8] have shown how the union bound can be internalized in a Hoare-style logic for reasoning about probabilistic imperative programs, and how this logic can be used to reason in a mechanized way about the accuracy of probabilistic programs, particularly programs implementing differentially private primitives.

Building on this idea, we propose a programming framework where this kind of reasoning is automated and can be combined with reasoning about privacy. Such a framework aims to offer programmers the tools they need to implement differentially private data analyses and explore their privacy-accuracy trade-offs, in a *compositional way*. This framework supports not only the use of union bound as a reasoning principle but also the Chernoff bound when applicable. The insight is that probabilistic bounds relying on probabilistic independence of random variables can be smoothly integrated in a programming framework by using techniques from **information-flow control (IFC)** [54] (in the form of taint analysis [55]). Although these probabilistic bounds are not enough to express every accuracy guarantee one wants to formulate for arbitrary data analyses, they enable the inspection of a large class of user-designed programs. Our approach allows programmers to exploit the compositional nature of both privacy and accuracy, complementing in this way the support provided by tools such as GUPT [44] and PSI [24], which yield confidence intervals estimate only at the level of individual queries, and by APEx [25], which issues confidence intervals estimate only at the level of a query workload for queries of the same type.

The described tool is materialized as a programming framework called *DPella*—an acronym for Differential Privacy in Haskell with accuracy—where data analysts can explore the privacy-accuracy trade-off while writing their differentially private data analyses. DPella provides several basic differentially private building blocks and composition techniques, which can be used by a programmer to design complex differentially private data analyses. The analyses that can be expressed in DPella are *data independent* and can be built using primitives for counting, average, max, and any aggregation of their results.

DPella supports both pure-DP, with parameter ϵ, and approximate-DP, with parameters ϵ and δ. Accordingly, it supports the addition of both Laplace and Gaussian random noise, and the use of sequential or advanced [17] composition, respectively, together with parallel composition for both notions. For clarity, we will mainly focus on ϵ-DP and the Laplace mechanism; however, other variants will be briefly discussed (see Section 5.3). DPella is implemented as a library in the general-purpose language Haskell, a programming language that is well known to support information-flow analyses [36, 53] easily. Furthermore, DPella is designed to be *extensible* by adding new primitives implementing advanced DP routines (see Section 9).

To reason about privacy and accuracy, DPella provides two primitives responsible for interpreting programs (which implement data analyses) symbolically. DPella's symbolic interpretation for privacy consists of decreasing the privacy budget of a query by deducing the required budget of its sub-parts. However, the accuracy interpretation uses as abstraction the **inverse Cumulative Distribution Function (iCDF)** representing an upper bound on the (theoretical) error that the program incurs when guaranteeing DP. A query's iCDF is built out from the iCDFs of its components by using the *union bound* as the elemental composition principle. These interpretations provide overestimates of the corresponding quantities they track. To make these estimates as precisely as possible, DPella uses *taint analysis* to track the injection of noise and identify which variables are *statistically independent*. This information is used by DPella to replace *soundly*, when needed, the union bound with the *Chernoff bound*, something that to the best of our knowledge other program logics [8] or program analyses [56] also focusing on accuracy do not consider. We envision DPella's accuracy estimations to be used in scenarios that align with those considered by tools like GUPT, PSI, and APEx.

In summary, our contributions are as follows:

- We present DPella, a programming framework that allows data analysts to reason compositionally about privacy-accuracy trade-offs.
- We show how to use taint analysis to detect statistical independence of the noise that different primitives add, and how to use this information to achieve better error estimates.
- We inspect DPella's expressiveness and error estimations by implementing PINQ-like queries from previous work [6, 39, 40] and workloads from the matrix mechanism [30, 34, 61].

To present DPella and its components, this document is structured as follows. Section 2 provides a brief background on the notions of privacy and accuracy DPella considers. Section 3 introduces DPella by showcasing its main features through simple examples. Section 4 presents each of DPella's primitives for the construction and execution of queries. Section 5 explains how do we calculate accuracy concentration bounds and the accuracy-aware primitives that can be used by the data analysts. In Section 6, we implement case studies from the literature revealing DPella's advantages and limitations. Section 7 introduces a new primitive that allows data analysts to test DPella's accuracy estimations. Section 8 shows DPella's generalized API that allows data analyst to combine noisy values generated with different mechanisms. Following, in Section 9, we discuss DPella's limitations in detail together with possible extensions to the framework. Last, Section 10 puts DPella in context while contrasting it with other approaches and frameworks.

Highlights. This work builds on our previous paper, "A Programming Framework for Differential Privacy with Accuracy Concentration Bounds" [37], which we have improved in its presentation and complemented with novel contributions summarized as follows:

- Comprehensive description of DPella's components
- Introduction of a new feature to tests DPella's accuracy estimations (Section 7)—this way, analysts will be able to measure the *tightness* of DPella's bound
- API updates including new accuracy combinators (Section 5.1) giving more options to manipulate and modify noisy values without losing information of their accuracy
- Description of (ϵ,δ-)-DP and Gauss mechanism integration (Section 5.3) that showcases DPella's flexibility to host other notions of DP and mechanisms
- Presentation of DPella's generalized API (Section 8), which facilitates the implementation of query plans involving results from different mechanisms.

2 BACKGROUND

DP [16] is a quantitative notion of privacy that bounds how much a single individual's private data can affect the result of a data analysis. More formally, we can define DP as a property of a randomized query $\tilde{Q}(\cdot)$ representing the data analysis, as follows.

Definition 2.1 (Differential Privacy (DP) [16]). A randomized query $\tilde{Q}(\cdot) : \text{db} \to \mathbb{R}$ satisfies ϵ*-DP* if and only if for any two datasets D_1 and D_2 in db, which differ in one row, and for every output set $S \subseteq \mathbb{R}$, we have

$$\Pr[\tilde{Q}(D_1) \in S] \leqslant e^{\epsilon} \Pr[\tilde{Q}(D_2) \in S]. \tag{1}$$

In the preceding definition, the parameter ϵ determines a bound on the distance between the distributions induced by $\tilde{Q}(\cdot)$ when adding or removing an individual from the dataset—the further away they are, the more at risk the privacy of an individual is, and vice versa. In other words, ϵ imposes a limit on the *privacy loss* that an individual can incur in, as a result of running a data analysis.

A standard way to achieve ϵ-DP is adding some carefully calibrated noise to the result of a query. To protect all of the different ways in which an individual's data can affect the result of a query, the noise needs to be calibrated to the maximal change that the result of the query can have when changing an individual's data. This is formalized through the notion of *sensitivity*.

Definition 2.2 (Sensitivity [16]). The (global) sensitivity of a query $Q(\cdot) : \text{db} \to \mathbb{R}$ is the quantity $\Delta_Q = \max\{|Q(D_1) - Q(D_2)|$ for D_1, D_2 differing in one row.

The sensitivity gives a measure of the amount of noise needed to protect one individual's data. In addition, to achieve DP, the choice of the kind of noise that one adds is important. A standard approach is based on the addition of noise sampled from the Laplace distribution.

THEOREM 2.1 (LAPLACE MECHANISM [16]). *Let $Q(\cdot) : \text{db} \to \mathbb{R}$ be a deterministic query with sensitivity Δ_Q. Let $\tilde{Q}(\cdot) : \text{db} \to \mathbb{R}$ be a randomized query defined as $\tilde{Q}(D) = Q(D) + \eta$, where η is sampled from the Laplace distribution with mean $\mu = 0$ and scale $b = \Delta_Q/\epsilon$. Then, $\tilde{Q}$ is ϵ-differentially private.*

Notice that in the preceding theorem, for a given query, the smaller ϵ is, the more noise $\tilde{Q}(\cdot)$ needs to inject to hide the contribution of one individual's data to the result—this protects privacy but degrades how meaningful the result of the query is—and vice versa. In general, the notion of *accuracy* can be defined more formally as follows.

Definition 2.3 (Accuracy (e.g., see, [17])). Given an ϵ-differentially private query $\tilde{Q}(\cdot)$, a target query $Q(\cdot)$, a distance function $d(\cdot)$, a bound α, and the probability β, we say that $\tilde{Q}(\cdot)$ is $(d(\cdot), \alpha, \beta)$-*accurate* with respect to $Q(\cdot)$ if and only if for all of dataset D:

$$\Pr[d(\tilde{Q}(D) - Q(D)) > \alpha] \leqslant \beta. \tag{2}$$

This definition allows one to express data-independent error statements such as follows: with probability at least $1 - \beta$, the query $\tilde{Q}(D)$ diverges from $Q(D)$, in terms of the distance $d(\cdot)$, for less than α. Then, we will refer to α as the *error* and $1 - \beta$ as the *confidence probability*, or simply *confidence*. In general, the lower the β is (i.e., the higher the confidence probability is), the higher the error α is.

As discussed previously, an important property of DP is composability.

THEOREM 2.2 (SEQUENTIAL COMPOSITION [16]). *Let $\tilde{Q}_1(\cdot)$ and $\tilde{Q}_2(\cdot)$ be two queries that are ϵ_1- and ϵ_2-differentially private, respectively. Then, their sequential composition $\tilde{Q}(\cdot) = (\tilde{Q}_1(\cdot), \tilde{Q}_2(\cdot))$ is $(\epsilon_1 + \epsilon_2)$-differentially private.*

THEOREM 2.3 (PARALLEL COMPOSITION [40]). *Let $\tilde{Q}(\cdot)$ be an ϵ-differentially private query and* $\text{data}_1, \text{data}_2$ *be a partition of the set of data. Then, the query $\tilde{Q}_1(D) = (\tilde{Q}(D \cap \text{data}_1), \tilde{Q}(D \cap \text{data}_2))$ is ϵ-differentially private.*

Thanks to the composition properties of DP, we can think about ϵ as a *privacy budget* that one can spend on a given data before compromising the privacy of individuals' contributions to that data. The *global* ϵ for a given program can be seen as the privacy budget for the entire data. This budget can be consumed by selecting the *local* ϵ to "spend" in each intermediate query. Thanks to the composition properties, by tracking the local ϵ that are consumed, one can guarantee that a data analysis will not consume more than the allocated privacy budget.

```
project
├── SchemaDS.hs
├── analysts
│   └── Queries.hs
└── curator
    ├── Execution.hs
    └── dataset.csv
```

Fig. 1. File structure.

Given an ϵ, DPella gives data analysts the possibility to explore how to spend it on different queries and analyze the impact on accuracy. For instance, data analysts might decide to spend "more" epsilon on sub-queries whose results are required to be more accurate while spending "less" on the others. The next examples (inspired by the use of DP in network trace analyses [39]) show how DPella helps quantify what "more" and "less" mean.

3 DPELLA BY EXAMPLE

DPella's model considers two kind of actors: *data curators*, owners of the private dataset who decide the global privacy budget and split it among the *data analysts*, the ones who write queries to mine useful information from the data and spend the budget they received. Analysts are not allowed to directly query the dataset; instead, they need to implement their analyses and send them to the curator who will execute them and give the results back.

From an implementation standpoint, this means that the analyses and their run functions are provided in different files, with different privileges. More specifically, Figure 1 depicts a common file structure for the usage of DPella. File `SchemaDS.hs` contains the schema of the dataset own by the curator; it does not contain private data but only the names of the tables and their respective attributes as a Haskell record type. For example, a dataset containing just one table called `Ages` with two attributes `name` (a `String` value) and `age` (an `Int` value) will be encoded in `SchemaDS.hs` as follows, where (::) is used to describe the type of a term in Haskell:

```
data Ages = AgeRow {name :: String, age :: Int}
```

Since the structure of the dataset is not considered sensitive information, `SchemaDS.hs` can be accessed by both the data owner and data analysts.

File `Queries.hs` contains the analyses that have being implemented by the data analysts; all of these queries should be parameterized by the dataset in which they will be later executed. Analysts will only have access to their implementations and the database schema. Last, file `Execution.hs` implements the run functions for the analyses at `Queries.hs`; this file is owned by the curator and has access to all other files in the directory (in particular, it has access to the real data stored in `dataset.csv`).

3.1 Basic Aggregations

For the following examples, we consider a dataset representing a *tcpdump* trace of packets where each row contains the information indicated by its schema:

```
data Tcpdump = TCPRow {id       :: Integer    , timestamp :: Double
                      , src      :: IP         , dest      :: IP
                      , protocol :: Integer    , size      :: Integer
                      , payload  :: ByteString}
```

3.1.1 Counting. An analyst wanting to know the number of packets sent to *WikiLeaks*, with IP address `195.35.109.53`, can do so by writing a simple eps-differentially private query as follows:

```
import SchemaDS
import AnalystLP

wikileaks :: ϵ → Data 1 Tcpdump → Query (Value Double)
wikileaks eps dataset = do
  byIP ← dpWhere ((≡ 195.35.109.53) ∘ dest) dataset
  dpCount eps byIP
```

First, we import file `SchemaDS` where `Tcpdump`'s description (previously presented) is stored. Then, we import DPella's interface for analysts called `AnalystLP`, where `LP` indicates that we will use the Laplacian mechanism. Subsequently, we implement query `wikileaks`, which takes as input the amount of privacy budget `eps` (of type ϵ) to be spent by the query and the `dataset` (of type `Data 1 Tcpdump`) where it will be computed; when executed, this query will yield results of type `Query (Value Double)`—that is, DPella computations of type `Double` (a more detailed explanation of DPella's types can be found in the following sections). In query `wikileaks`, we use the primitive transformation[2] `dpWhere` to filter all rows whose `dest` attribute has a value equal to `195.35.109.53`; this operation returns a transformed dataset we call `byIP`. We proceed to perform the noisy count using primitive `dpCount` over the filtered dataset `byIP` while spending `eps` amount of privacy budget. The value of `eps` will—internally—determine the magnitude of noise to be added to the real count.

Having this general implementation, an analyst can write specific queries fixing the value of `eps`, for instance:

```
analysis1 = wikileaks 0.5
analysis2 = wikileaks 1
analysis3 = wikileaks 5
```

To execute these analyses, the *data owner* needs to implement a function that loads the required dataset and executes analysts' queries; such a function will look like this:

```
import SchemaDS
import CuratorLP (loadDS, dpEval)
import Queries

runAnalysis :: (Data 1 Tcpdump → Query (Value Double)) → ϵ → IO Double
runAnalysis query bud = do
  ds ← loadDS "hotspot.csv"
  dpEval query ds bud
```

Function `runAnalysis` takes as inputs the function to be executed, called `query`, and the global privacy budget `bud`, returning the randomized count as an `IO Double`. This function calls an auxiliary function `loadDS` (provided by DPella's interface for curators) to read file `hotspot.csv`, which is then saved as a DPella dataset in variable `ds`. Next, it uses DPella's primitive `dpEval` indicating

[2]In Section 4, in our code we will usually use red color for transformations, blue color for aggregate operations, and green color for combinators for privacy and accuracy.

which analysis will be performed, over which dataset, and what the tolerance is for the privacy loss.

Let us assume that `hotspot.csv` has the information of 10,000 packets and 7 of them are directed to WikiLeaks' IP address. Then, when the data owner executes the analysis, she would get results such as follows:

```
>runAnalisis analysis1 20
Value = 15.3
>runAnalisis analysis2 20
Value = 4.8
>runAnalisis analysis3 20
Value = 6.7
```

This clearly exemplifies the effects of the selection of `eps` on the queries' results. Intuitively, the greater the `eps`, the closer we are to the real count of packets.

3.1.2 *Sums.*

Suppose that we are now interested in computing the amount of transmitted data. In other words, we want to sum up the value of `size` column, which indicates the length of the packets in bytes.

In DPella, to compute a sum, we need to first determine the range of the values—our framework supports only integer numbers' ranges (e.g., $[1, 10]$, $[-5, 30]$). This information is needed to automatically calculate the sensitivity of sum queries at compile time—that is, if every value is in the range $[a, b]$, the sensitivity of their addition is $\max\{|a|, |b|\}$. We specify ranges in DPella via the primitive `range`.

```
range :: (IsInt a, IsInt b, IsNat | b-a |, a ⩽ b) ⇒ Range a b
```

This function receives no arguments since the range is indicated at the type level with type constraints of the form `IsNat n` for strictly positive integer numbers, and `IsInt n` for positive and negative integers. Then, to create ranges, we need to use type applications such as follows:

```
range1 = range @(:+: 1) @(:+: 10)
range2 = range @(:-: 5) @(:+: 30)
```

Here, functions `:+:` and `:-:` are used to specify the sign of the range's limits.

For the example of packet size, the data curator indicates that the range of size attribute goes from 40 to 35,000 bytes, and then we define our query as follows:

```
totalBytes eps dataset = do
  dpSum eps (range @(:+: 40) @(:+: 35000)) size dataset
```

Function `totalBytes` uses primitive `dpSum` to compute the noisy sum of `size` attribute—whose values are ranging from 40 to 35, 000 bytes—over the indicated `dataset`. The way this query should be executed does not vary from the execution of the analyses derived from function `wikileaks` and thus is omitted.

Changing the question to focus on an specific protocol might require an adjustment on the range to be specified. For instance, if instead we want to inspect the total amount of data transmitted through Kerberos' authentication protocol, which uses port 88, we should use the fact that this port transmits packets of at most 1,465 bytes. Hence, we will need to update our query accordingly.

```
cdf1 bins eps dataset = do
  sizes  ← dpSelect size dataset
  counts ← sequence [do elems ← dpWhere (⩽ bin) sizes
                        dpCount localEps elems | bin ← bins]
  return (norm∞ counts)
    where localEps = eps / length bins
```

(a) Sequential approach

```
cdf2 bins eps dataset = do
  sizes ← dpSelect ((⩽ max bins) ∘ size) dataset
    -- parts :: Map Integer (Value Double)
  parts ← dpPartRepeat (dpCount eps) bins assignBin sizes
  let counts       = Map.elems parts
      cumulCounts = [add (take i counts) | i ← [1.length counts]]
  return (norm∞ cumulCounts)
```

(b) Parallel approach

Fig. 2. CDF's implementations.

```
totalBytesKerberos eps dataset = do
  kerberos ← dpWhere ((≡ 88) ∘ protocol) dataset
  dpSum eps (range @(:+: 40) @(:+: 1465)) size kerberos
```

In function `totalBytesKerberos`, we will first filter the dataset to obtain the information regarding port 88, then we will perform the noisy sum over the filtered data. Observe that we are defining a query with less global sensitivity than the one implemented in function `totalBytes`; thus, if given the same `eps`, less noise will be added to the results of the analyses deriving from function `totalBytesKerberos`. Having a notion of the order of magnitude in which the result of a sum ranges becomes handy when reasoning about the accuracy of the query. In the following examples, we depict how an analyst can use DPella to inspect the error of her queries, check for miscalculations on the consumption of the privacy budget, and more.

3.2 Cumulative Distribution Function

Considering the same dataset `Tcpdump`, we would like to inspect—in a differentially private manner—the packet's length distribution by computing its **Cumulative Distribution Function (CDF)**, defined as CDF(x) = number of records with value $\leqslant x$. Hence, we are just interested in the values of the attribute `size`. McSherry and Mahajan [39] proposed three different ways to approximate (due to the injected noise) CDFs with DP, and they argued for their different levels of accuracy. For simplicity, we revise two of these approximations to show how DPella can assist in showing the accuracy of these analyses.

3.2.1 Sequential `CDF`. A simple approach to compute the CDF consists of splitting the range of lengths into `bins` and, for each `bin`, counting the number of records that are $\leqslant$ `bin`. A natural way to make this computation differentially private is to add independent Laplace noise to each count.

We show how to do this using DPella in Figure 2(a). We define a function `cdf1` that takes as input the list of `bins` describing size ranges, the amount of budget `eps` to be spent by the entire query, and the `dataset` where it will be computed. For now, we assume that we have a fixed list of bins for packet length. Function `cdf1` uses the primitive transformation `dpSelect` to obtain from the dataset the length of each packet via a selector function; in this case, it is just the column of interest `size`. This computation results in a new dataset: `sizes`. Then, we create a counting query for each bin using the primitive `dpWhere`. This filters all records that are less than the `bin` under consideration ($\leqslant$ `bin`). Finally, we perform a noisy count using primitive `dpCount`. The noise injected by the primitive `dpCount` is calibrated so that the execution of `dpCount` is `localEps`-DP (line 6[3]). The function `sequence` then takes the list of queries and computes them sequentially by collecting their results in a list—to create a list of noisy counts. We then return this list. The combinator $\texttt{norm}_\infty$ in line 5 is used to mark where we want the accuracy information to be collected, but it does not have any impact on the actual result of the CDF.

To ensure that `cdf1` is `eps`-DP, we distributed the given budget `eps` evenly among the subqueries (this is done in lines 4 and 6). However, a data analyst may forget to do so—for example, she can define `localEps` = `eps`, and in this case the final query is (`length bins`) $*$ `eps`-DP, which is a significant change in the query's privacy price. To prevent such budget miscalculations or unintended expenditure of privacy budget, DPella provides the analyst with the function `budget` (see Section 4) that, given a query, statically computes an upper bound on how much budget it will spend. To see how to use this function, consider the function `cdf1` and its modified version `cdf1'` with `localEps` = `eps`. Suppose that we want to compute how much budget will be consumed by running them on a list of 10 bins (identified as $\texttt{bins}_{10}$) and a symbolic dataset `symDataset`. Then, the data analyst can ask this as follows:

```
>budget (cdf1 bins₁₀ 1 symDataset)
ϵ = 1
>budget (cdf1' bins₁₀ 1 symDataset)
ϵ = 10
```

The function `budget` *will not* execute the query; it simply performs an static analysis on the code of the query by symbolically interpreting it. The static analysis uses information encoded by the *type* of `symDataset` (explained in Section 4), which, in this particular case, will be provided by `Tcpdump`'s schema.

DPella also provides primitives to statically explore the accuracy of a query. The function `accuracy` takes a noisy query $\tilde{Q}(\cdot)$ and a probability β and returns an estimate of the (theoretical) error that can be achieved with confidence probability $1 - \beta$. Suppose that we want to estimate the error we will incur in by running `cdf1` with a budget of $\epsilon = 1$ with the same list of bins and symbolic dataset as before, and we want to have this estimate for $\beta = 0.05$ and $\beta = 0.2$, respectively. Then, the data analyst can ask this as follows:

```
>accuracy (cdf1 bins₁₀ 1 symDataset) 0.05
α = 53
>accuracy (cdf1 bins₁₀ 1 symDataset) 0.2
α = 40
```

Since the result of the query is a vector of counts, we measure the error α in terms of ℓ_∞ distance with respect to the CDF without noise. This is the max difference that we can have in a bin due to the noise. The way to read the information provided by DPella is that with confidence 95% and 80%, we have errors 53 and 40, respectively. These error bounds can be used by a data analyst to figure out the exact set of parameters that would be useful for her task.

[3]The casting operation `fromIntegral` is omitted for clarity.

3.2.2 *Parallel* `CDF`.

Another way to compute a CDF is by first generating a histogram of the data according to the bins and then building a cumulative sum for each bin. To make this function private, an approach could be to add noise at the different bins of the histogram rather than to the cumulative sums themselves, so we could use the parallel composition rather than the sequential one [40], which we show how to implement in DPella in Figure 2(b)—where double dashes are used to introduce single-line comments.

In `cdf2`, we first select all packages whose length is smaller than the maximum bin (line 8), and then we partition the data accordingly to the given list of bins (line 10). To do this, we use the `dpPartRepeat` operator to create as many (disjoint) datasets as given `bins`, where each record in each partition belongs to the range determined by a specific `bin`—where the record that belongs is determined by the function `assignBin` :: `Integer` → `Integer`. After creating all partitions, the primitive `dpPartRepeat` computes the given query `dpCount eps` in *each partition*—the name `dpPartRepeat` comes from repetitively calling `dpCount eps` as many times as the partitions we have. As a result, `dpPartRepeat` returns a finite map where the keys are the `bins` and the elements are the noisy count of the records per partition (i.e., the histogram). In what follows (lines 12 and 13), we compute the cumulative sums of the noisy counts using the DPella primitive add, and finally we build and return the list of values denoting the CDF.

The privacy analysis of `cdf2` is similar to the one of `cdf1`. The accuracy analysis, however, is more interesting: first it gets error bounds for each cumulative sum, and then these are used to give an error bound on the maximum error of the vector. For the error bounds on the cumulative sums, DPella uses either the union bound or the Chernoff bound, depending on which one gives the lowest error. For the maximum error of the vector, DPella uses the union bound, similarly to what happens in `cdf1`. A data analyst can explore the accuracy of `cdf2`.

```
>accuracy (cdf2 bins₁₀ 1 symDataset) 0.05
α = 22
>accuracy (cdf2 bins₁₀ 1 symDataset) 0.2
α = 20
```

3.2.3 *Exploring the Privacy-Accuracy Trade-off.*

Let us assume that a data analyst is interested in running a CDF with an error bounded with 90% confidence—that is, with $\beta = 0.1$, having three bins (named $\texttt{bins}_3$), and $\epsilon = 1$. With those assumptions in mind, which implementation should she use? To answer that question, the data analyst can ask DPella:

```
>accuracy (cdf1 bins₃ 1 symDataset) 0.1
 α = 11
>accuracy (cdf2 bins₃ 1 symDataset) 0.1
 α = 12
```

So, the analyst would know that using `cdf1` in this case would give, likely, a lower error. Suppose further that the data analyst realize that she prefers to have a finer granularity and have 10 bins instead of only 3. Which implementation should she use? Again, she can compute as follows:

```
>accuracy (cdf1 bins₁₀ 1 symDataset) 0.1
 α = 46
>accuracy (cdf2 bins₁₀ 1 symDataset) 0.1
 α = 20
```

So, the data analyst would know that using `cdf2` in this case would give, likely, a lower error. One can also use DPella to show a comparison between `cdf1` and `cdf2` in terms of error when we keep

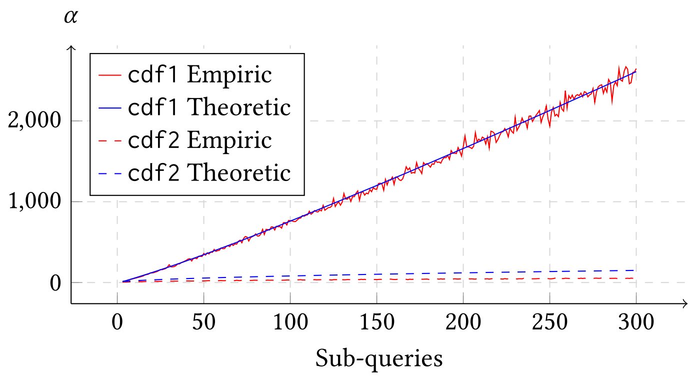

Fig. 3. Error comparison (95% confidence).

the privacy parameter fixed and we change the number of bins, where `cdf2` gives a better error when the number of bins is large [39] as illustrated in Figure 3. In the figure, we also show the empirical error to confirm that our estimate is tight—the oscillations on the empirical `cdf1` are given by the relatively small (300) number of experimental runs we consider.

Now, what if the data analyst chooses to use `cdf2` because of what we discussed before but she realizes that she can afford an error $\alpha \leqslant 50$; what would be the epsilon that gives such α? One of the features of DPella is that the analyst can write a simple program that finds it by repetitively calling `accuracy` with different epsilons—this is one of the advantages of providing a programming framework. These different use cases shows the flexibility of DPella for different tasks in private data analyses.

4 PRIVACY

DPella is designed to help data analysts have an informed decision about how to spend their budget based on exploring the trade-offs between privacy and accuracy. In this section, we introduce DPella's primitives and design principles responsible for ensuring DP of queries written by data analysts.

4.1 Components of the API

Figure 4 shows part of the DPella API. DPella introduces two abstract data types to respectively denote datasets and queries:

```
data Data s r   -- datasets
data Query a    -- queries
```

The attentive reader might have observed that the API also introduces the data type `Value a`. This type is used to capture values resulting from data aggregations. However, we defer its explanation to Section 5 since it is only used for accuracy calculations—for this section, readers can consider the type `Value a` as isomorphic to the type `a`. It is also worth noting that the API enforces an invariant by construction: it is not possible to branch on results produced by aggregations—observe that there is no primitive capable to destruct a value of type `Value a`. Although it might seem restrictive, it enables writing of counting queries, which are the bread and butter of statistical analysis and have been the focus of the majority of the work in DP. Section 9 discusses, however, how to lift this limitation for specific analyses.

```
    -- Transformations (data analyst)
dpWhere      :: (r → Bool) → Data s r → Query (Data s r)
dpGroupBy    :: Eq k  ⇒ (r → k) → Data s r → Query (Data (2*s) (k, [r]))
dpIntersect  :: Eq r  ⇒ Data s₁ r → Data s₂ r → Query (Data (s₁+s₂) r)
dpSelect     :: (r → r′) → Data s r → Query (Data s r′)
dpUnion      :: Data s₁ r → Data s₂ r → Query (Data (s₁+s₂) r)
dpPart       :: Ord k ⇒ (r → k) → Data s r → Map k (Data s r) → Query (Value a))
             → Query (Map k (Value a))
    -- Aggregations (data analyst)
dpCount :: Stb s   ⇒ ϵ → Data s r → Query (Value Double)
dpSum   :: Stb s   ⇒ ϵ → Range a b → (r → Double) → Data s r → Query (Value Double)
dpAvg   :: Stb s   ⇒ ϵ → Range a b → (r → Double) → Data s r → Query (Value Double)
dpMax   :: Eq a    ⇒ ϵ → Responses a → (r → a) → Data 1 r → Query (Value a)
    -- Budget
budget  :: Query a → ϵ
    -- Execution (data curator)
dpEval :: (Data 1 r → Query (Value a)) → [r] → ϵ → IO a
```

Fig. 4. DPella API: Part I.

Values of type `Data s r` represent sensitive datasets with *accumulated stability* `s`, where each row is of type `r`. Accumulated stability, however, is instantiated to type-level positive natural numbers (1, 2, etc.). Stability is a measure that captures the number of rows in the dataset that could have been affected by transformations like selection or grouping of rows. In DP research, stability is associated with dataset transformations rather than with datasets themselves. To simplify type signatures, DPella uses the type parameter s in datasets to represent the accumulated stability of the transformations for which datasets have gone through—as done in the work of Ebadi and Sands [20]. Different from PINQ [40], for example, one novelty of DPella is that it computes stability *statically* using Haskell's type system.

Values of type `Query a` represent *computations*, or queries, that yield values of type `a`. Type `Query a` is a monad [43], and because of this, computations of type `Query a` are built by two fundamental operations:

```
return :: a → Query a
( >>= ) :: Query a → (a → Query b) → Query b
```

The operation `return x` outputs a query that just produces the value `x` without causing side effects (i.e., without touching any dataset). The function (>>=)—called *bind*—is used to sequence queries and their associated side effects. Specifically, `qp >>= f` executes the query `qp`, takes its *result*, and passes it to the function `f`, which then returns a second query to run. Some languages, like Haskell, provide syntactic sugar for monadic computations known as **do** notation. For instance, the program $\mathtt{qp}_1$ >>= $(\lambda \mathtt{x}_1 \rightarrow \mathtt{qp}_2$ >>= $(\lambda \mathtt{x}_2 \rightarrow \mathtt{return}\ (\mathtt{x}_1, \mathtt{x}_2)))$, which performs queries $\mathtt{qp}_1$ and $\mathtt{qp}_2$ and returns their results in a pair, can be written as **do** $\mathtt{x}_1 \leftarrow \mathtt{qp}_1; \mathtt{x}_2 \leftarrow \mathtt{qp}_2; \mathtt{return}\ (\mathtt{x}_1, \mathtt{x}_2)$, which gives a more "imperative" feeling to programs. We split the API into four parts: transformations, aggregations, budget prediction, and execution of queries—see the next section for the description of the API's accuracy components. The first three parts are intended to be used by data analysts, whereas the last one is intended to be *only* used by data curators.[4]

[4]A separation that can be enforced via Haskell modules [57].

```
q :: ϵ → [Color] → Data 1 Double → Query (Map Color Double)
q eps bins dataset = dpPart id dataset dps
  where dps = fromList [(c, λds → dpCount eps dataset) | c ← bins]
        -- dps = fromList [(c, λds → dpCount eps ds) | c ← bins]
```

Fig. 5. DP histograms by using dpPart.

4.2 Transformations

The primitive dpWhere filters rows in datasets based on a predicate function (r → Bool). The created query (of type Query (Data s r)) produces a dataset with the same row type r and accumulated stability s as the dataset given as argument (Data s r). Observe that if we consider two datasets that differ in s rows in two given executions, and we apply dpWhere to both of them, we will obtain datasets that will still differ in at most s rows—thus, the accumulated stability remains the same. The primitive dpGroupBy returns a dataset where rows with the same key are grouped together. The functional argument (of type r → k) maps rows to keys of type k. The rows in the return dataset (Data (2∗*s*) (k, [r])) consist of key-rows pairs of type (k, [r])—syntax [r] denotes the type of lists of elements of type r. What appears on the left-hand side of the symbol ⇒ are type constraints. They can be seen as static demands for the types appearing on the right-hand side of ⇒. Type constraint Eq k demands type k, denoting keys, to support equality; otherwise, grouping rows with the same keys is not possible. The accumulated stability of the new dataset is multiplied by 2 in accordance with stability calculations for transformations [20, 40]—observe that 2∗s is a type-level multiplication done by a type-level function (or type family [21]) ∗. In other words, it is an arithmetic operation computed at compile time. Our API also considers transformations similar to those found in SQL like intersection (dpIntersect), union (dpUnion), and selection (dpSelect) of datasets, where the accumulated stability is updated accordingly. Providing a general join transformation is known to be challenging [10, 31, 40, 45]. The output of a join may contain duplicates of sensitive rows, which makes it difficult to bound the accumulated stability of datasets. However, and similar to PINQ, DPella supports a limited form of joins, where a limit gets imposed on the number of output records mapped under each key to obtain stability. For brevity, we skip its presentation and assume that all of the considered information is contained by the rows of given datasets.

4.3 Partition

Primitive dpPart deserves special attention. This primitive is a mixture of a transformation and aggregations since it partitions the data (transformation) to subsequently apply aggregations on each of them. More specifically, it splits the given dataset (Data s r) based on a row-to-key mapping (r → k). Then, it takes each partition for a given key k and applies it to the corresponding function Data s r → Query (Value a), which is given as an element of a key-query mapping (Map k ((Data s r) → Query (Value a))). Subsequently, it returns the values produced at every partition as a key-value mapping (Query (Map k (Value a))). The primitive dpPartRepeat, used by the examples in Section 3, is implemented as a special case of dpPart, and thus we do not discuss it further.

Partition is one of the most important operators to save privacy budget. It allows running of the same query on a dataset's partitions but only paying for one of them—recall Theorem 2.3. The essential assumption that makes this possible is that every query runs on *disjoint* datasets. Unfortunately, data analysts could ignore this assumption when writing queries.

To illustrate this point, we present the code in Figure 5. Query q produces an ϵ-DP histogram of the colors found in the argument dataset, whose rows are of type Color and variable bins enumerates all possible values of such type. The code partitions the dataset by using the function `id` :: `Color` → `Color` (line 2) and executes the aggregation counting query (`dpCount`) in each partition (line 3)—function `fromList` creates a map from a list of pairs. The attentive reader could notice that `dpCount` is applied to the original `dataset` rather than the partitions. This type of error could lead to a break in privacy as well as inconsistencies when estimating the required privacy budget. A correct implementation consists of executing `dpCount` on the corresponding partition as shown in the commented line 4.

To catch coding errors as the one shown earlier, DPella deploys a static IFC analysis similar to that provided by MAC [52]. IFC ensures that queries run by `dpPart` do not perform queries on shared datasets by attaching provenance labels to datasets `Data s r` indicating to which part of the query they are associated with and propagates that information accordingly.

Coming back to our previous example (see Figure 5), the IFC analysis will assign the provenance of `dataset` in q to the top-level fragment of the query rather than to sub-queries executed in each partition—and DPella will raise an error at compile time when `ds` is accessed by the sub-queries! Instead, if we comment line 3 and uncomment line 4, the query `q` will be successfully run by DPella (when there is enough privacy budget) since every partition is only accessing their own partitioned data (denoted by variable `ds`).

The implemented IFC mechanism is *transparent* to data analysts and curators—that is, they do not need to understand how it works. Analysts and curators only need to know that when the IFC analysis raises an alarm, it is due to a possibe access to non-disjoint datasets when using `dpPart`.

4.4 Aggregations

DPella presents primitives to count (`dpCount`), sum (`dpSum`), and average (`dpAvg`) rows in datasets. These primitives take an argument `eps` :: ϵ, a dataset, and build a Laplace mechanism that is `eps`-differentially private from which a noisy result gets returned as a term of type `Value Double`. The purpose of data type `Value a` is twofold: to encapsulate noisy values of type `a` originating from aggregations of data, and to store information about its accuracy—intuitively, how “noisy” the value is (explained in Section 5). The injected noise of these queries gets adjusted depending on three parameters: the value of type ϵ, the accumulated stability of the dataset `s`, and the sensitivity of the query (recall Definition 2.2). More specifically, the Laplace mechanism used by DPella uses accumulated stability s to scale the noise—that is, it consider b from Theorem 2.1 as $b = s \cdot \frac{\Delta_Q}{\epsilon}$. The sensitivity of DPella’s aggregations are either hard coded into the implementation—similar to what PINQ does—or calculated statically. The sensitivities of `dpSum` and `dpAvg` are determined by the range of the values under consideration—that is, for the indicated `Range a b`, the sensitivity is computed as `max` { |`a`|, |`b`| } and |`b-a`|, respectively. This is enforced by applying a clipping function (`r` → `Double`). This function ensures that the values under scrutiny fall into the interval $[a, b]$ before (and, for `dpAvg`, after) executing the query. The sensitivity of `dpCount` and `dpMax` is set to `1`. To implement the Laplace mechanism, the type constrain `Stb s` in `dpCount`, `dpSum`, and `dpAvg` demands the accumulated stability parameter `s` to be a type-level natural number to obtain a term-level representation when injecting noise. Finally, primitive `dpMax` implements report-noisy-max [17]. This query takes a list of possible responses (`Responses a` is a type synonym for `[a]`) and a function of type `r` → `a` to be applied to every row. The implementation of `dpMax` adds uniform noise to every score—in this case, the amount of rows *voting* for a response—and returns the *response* with the highest noisy score. This primitive becomes relevant to obtain the winner option in elections without singling out any voter. However, it requires the accumulated stability of the dataset to be `1` in order to be sound [8]. DPella guarantees such a requirement by typing: the type of the given dataset as argument is `Data 1 r`.

4.5 Privacy Budget and Execution of Queries

The primitive `budget` statically computes how much privacy budget is required to run a query. It is worth noting that DPella returns an upper bound of the required privacy budget rather than the exact one—an expected consequence of using a type system to compute it and provide early feedback to data analysts. Finally, the primitive `dpEval` is used by data curators to run queries (`Query a`) under given privacy budgets (ϵ), where datasets are just lists of rows (`[r]`). It assumes that the initial accumulated stability is `1` (`Data 1 r`) since the dataset has not yet gone through any transformation, and DPella will automatically calculate the accumulated stability for datasets affected by subsequent transformations via the Haskell type system. This primitive returns a computation of type `IO a`, which in Haskell are the computations responsible for performing side effects—in this case, obtaining randomness from the system to implement the Laplace mechanism.

4.6 Implementation

DPella is implemented as a *deep embedded domain-specific language* in Haskell. Due to such design choice, data analysts can piggyback on Haskell's infrastructure to build queries in a creative way. For instance, it is possible to leverage on any of Haskell's pure functions. The following one-liner (of type `Query [Value Double]`) uses several Haskell functions to filter a dataset `ds` in several (possibly non-disjoint) ways according to a list of predicates `ps` :: [r → `Bool`], and then for each filtered version of `ds`, it performs a noisy count spending `eps` on each count.

```
mapM (flip dpSelect ds >=> dpCount eps) ps
```

The high-order functions `flip`, `mapM`, and (`>=>`) are standard in Haskell and represent a function that switches arguments, the monadic versions of `map`, and the Kleisli arrow, respectively. Despite DPella being a first-order interface, data analysts can use Haskell's high-order functions to compactly describe queries.

5 ACCURACY

DPella uses the data type `Value a` responsible for storing a result of type `a` as well as information about its accuracy. For instance, a term of type `Value Double` stores a noisy number (e.g., coming from executing `dpCount`) together with its accuracy in terms of a bound on the noise introduced to protect privacy.

DPella provides a static analysis capable of computing the accuracy of queries via the following function,

`accuracy` :: `Query (Value a)` → β → α

that takes as an argument a query and returns a function, called *inverse Cumulative Distribution Function* (iCDF), capturing the theoretical error α for a given confidence `1`-β. Function `accuracy` does not execute queries but rather symbolically interprets all of its components to compute the accuracy of the result based on the sub-queries and how data gets aggregated. DPella follows the principle of improving accuracy calculations by detecting statistical independence. For that, it implements taint analysis [55] to track if values were drawn from statistically independent distributions. DPella's primitives involving accuracy calculations are presented in Figure 6 and will be described in the following sections.

5.1 Accuracy Calculations

DPella starts by generating iCDFs at the time of running aggregations based on the following known result of the Laplace mechanism.

```
  -- Accuracy analysis (data analyst)
accuracy :: Query (Value a) → β → α

  -- Norms (data analyst)                        -- Accuracy combinators (data analyst)
norm∞ :: [Value Double] → Value [Double]      add    :: [Value Double] → Value Double
norm2 :: [Value Double] → Value [Double]      sub    :: [Value Double] → Value Double
norm1 :: [Value Double] → Value [Double]      neg    :: Value Double → Value Double
rmsd  :: [Value Double] → Value [Double]      scalar :: Value Double → Double → Value Double
```

Fig. 6. DPella API: Part II.

Definition 5.1 (Accuracy for the Laplace Mechanism). Given a randomized query $\tilde{Q}(\cdot) : \text{db} \rightarrow \mathbb{R}$ implemented with the Laplace mechanism as in Theorem 2.1, we have that

$$\Pr\left[|\tilde{Q}(D) - Q(D)| > \log\left(\frac{1}{\beta}\right) \cdot \frac{\Delta_Q}{\epsilon}\right] \leqslant \beta. \tag{3}$$

Recall that the Laplace mechanism used by DPella utilizes accumulated stability s to scale the noise—that is, it consider b from Theorem 2.1 as $b = s \cdot \frac{\Delta_Q}{\epsilon}$. Consequently, DPella stores the iCDF $\lambda\beta \rightarrow \log\left(\frac{1}{\beta}\right) \cdot s \cdot \frac{\Delta_Q}{\epsilon}$ for the values of type `Value Double` returned by aggregation primitives like `dpCount`, `dpSum`, and `dpAvg`. However, queries are often more complex than just calling aggregation primitives—as shown by `cdf2` in Figure 2(b). In this light, DPella provides combinators responsible to aggregate noisy values while computing its iCDFs based on the iCDFs of the arguments. Figure 6 shows DPella API when dealing with accuracy.

5.1.1 Norms. DPella exposes primitives to aggregate the magnitudes of several error predictions into a *single* measure—a useful tool when dealing with vectors. Primitives `norm`$_\infty$, `norm`$_2$, and `norm`$_1$ take a list of values of type `Value Double`, where each of them carries accuracy information, and produces a *single value* (or vector) that contains a list of elements (`Value [Double]`) whose accuracy is set to be the well-known ℓ_∞-, ℓ_2-, ℓ_1-norms, respectively. Finally, primitive `rmsd` implements *root-mean-square deviation* among the elements given as arguments. In our examples, we focus on using `norm`$_\infty$, but other norms are available for the taste, and preference, of data analysts.

5.1.2 Adding Values. The primitive `add` aggregates values, and to compute accuracy of the addition, it tries to apply the Chernoff bound if all of the values are statistically independent; otherwise, it applies the union bound. More precisely, for the next definitions, we assume that primitive `add` receives n terms v_1 :: `Value Double`, v_2 :: `Value Double`, ..., v_n :: `Value Double`. Importantly, since we are calculating the theoretical error, we should consider random variables rather than specific numbers. The next definition specifies how `add` behaves when applying union bound.

Definition 5.2 (`add` using Union Bound). Given $n \geqslant 2$ random variables V_j with their respective iCDF_j, where $j \in 1 \ldots n$, and $\alpha_j = \text{iCDF}_j(\frac{\beta}{n})$, then the addition $Z = \sum_{j=1}^{n} V_j$ has the following accuracy:

$$\Pr\left[|Z| > \sum_{j=1}^{n} \alpha_j\right] \leqslant \beta. \tag{4}$$

Observe that to compute the iCDF of Z, the formula uses the iCDFs from the operands applied to $\frac{\beta}{n}$. Union bound makes no assumption about the distribution of the random variables V_j.

In contrast, the Chernoff bound often provides a tighter error estimation than the commonly used union bound when adding several statistically independent queries sampled from a Laplace

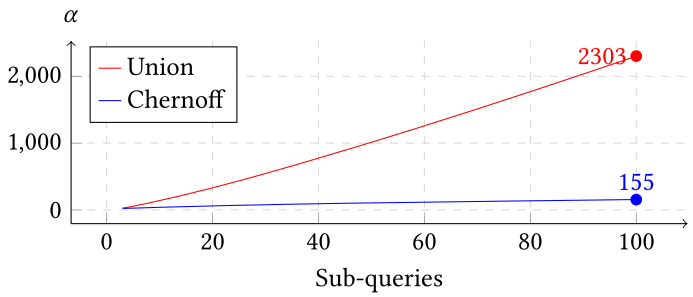

Fig. 7. Union vs. Chernoff bounds.

distribution. To illustrate this point, Figure 7 shows that difference for the `cdf2` function we presented in Section 3 with $\epsilon = 0.5$ (for each DP sub-query) and $\beta = 0.1$. Clearly, the Chernoff bound is asymptotically much better when estimating accuracy, whereas the union bound works best with a reduced number of sub-queries—observe how lines get crossed in Figure 7. In this light, and when possible, DPella computes both union bound and Chernoff bound and selects the tighter error estimation. However, to apply the Chernoff bound, DPella needs to be certain that the events are independent. Before explaining how DPella detects that, we give an specification of the formula we use for Chernoff.

Definition 5.3 (`add` *using the Chernoff Bound [13]).* Given $n \geqslant 2$ *independent random variables* $V_j \sim Lap(0, b_j)$, where $j \in 1 \ldots n$, $b_M = \max\{b_j\}_{j=1\ldots n}$, and $\nu > \max\left\{\sqrt{\sum_{j=1}^{n} b_j^2}, b_M \cdot \sqrt{\ln\left(\frac{2}{\beta}\right)}\right\}$, the addition $Z = \sum_{j=1}^{n} V_j$ has the following accuracy:

$$\Pr\left[|Z| > \nu \cdot \sqrt{8 \cdot \ln\left(\frac{2}{\beta}\right)}\right] \leqslant \beta. \qquad (5)$$

DPella uses the value $\nu = \max\left\{\sqrt{\sum_{j=1}^{n} b_j^2}, b_M \cdot \sqrt{\ln\left(\frac{2}{\beta}\right)}\right\} + 0.00001$ to satisfy the conditions of the preceding definition when applying the Chernoff bound—any other positive increment to the computed maximum works as well.[5] It is worth mentioning that DPella's error estimations for the sums of noisy values rely on available concentration bounds. Hence, even though there exist better approximations for the error of adding random variables (e.g., dependency-dependent bounds for dependent variables [33]), currently, union and Chernoff bounds are the only statistical tools that can be used out of the box.

Last, to support subtraction, DPella provides primitive `neg` responsible for changing the sign of a given value, and `sub` that uses the results of `neg` and `add` to subtract a list of values. In the following, we explain how DPella checks that values come from statistically independent sampled variables.

5.1.3 *Detecting Statistical Independence.*

To detect statistical independence, we apply taint analysis when considering terms of type `Value a`. Specifically, every time a result of type `Value Double` gets generated by an aggregation query in DPella's API (`dpCount`, `dpSum`, etc.), it gets assigned a label indicating that it is *untainted* and thus statistically independent. The label also carries information about the scale of the Laplace distribution from which it was sampled—useful information when applying Definition 5.3. When the primitive `add` receives all untainted values as arguments, the accuracy of the aggregation is determined by the best estimation provided by either the union

[5]There are perhaps other ways to compute the Chernoff bound for the sum of independent Laplace distributions; changing this equation in DPella does not require major work.

```
1 totalCount :: Query (Value Double)
2 totalCount = do
3   v1   ← dpCount 0.3  ds1
4   v2   ← dpCount 0.25 ds2
5   ...
6   v100 ← dpCount 0.5  ds100
7   return (add [v1, v2, ..., v100])
```

Fig. 8. Combination of sub-queries' results.

bound (Definition 5.2) or the Chernoff bound (Definition 5.3). Importantly, values produced by add are considered *tainted* since they depend on other results. When add receives any tainted argument, it proceeds to estimate the error of the addition by just using union bound.

As an example, Figure 8 presents the query plan totalCount, which adds the results of 100 dpCount queries over different datasets, namely $\mathtt{ds}_1, \mathtt{ds}_2, \ldots, \mathtt{ds}_{100}$. (The ... denotes code intentionally left unspecified.) The code calls the primitive add with the results of calling dpCount—we use $[x_1, x_2, x_3]$ to denote the list with elements x_1, x_2, and x_3. What would then be the theoretical error of totalCount? The accuracy calculation depends on whether all values are untainted in line 7. When no dependencies are detected between $\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_{100}$, namely all the values are untainted, DPella applies Chernoff bound to give a tighter error estimation. Instead, for instance, if $\mathtt{v}_3$ were computed as an augmentation of $\mathtt{v}_1$ by a factor of 5, this would be **let** $\mathtt{v}_3$ = scalar $\mathtt{v}_1$ 5. Then, line 7 applies union bound since $\mathtt{v}_3$ is a tainted value—its noise is not freshly sampled but rather inherited from $\mathtt{v}_1$'s noise. With taint analysis, DPella is capable of detecting dependencies among terms of type Value Double and leverages that information to apply different concentrations bounds. The next section formally defines such a procedure.

5.2 Implementation

The accuracy analysis consists of *symbolically* interpreting a given query, calculating the accuracy of individual parts, and then combining them using our taint analysis. We introduce two polymorphic symbolic values: $\mathcal{D}$ :: Data s r and $\mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}]$ :: Value a. Symbolic dataset $\mathcal{D}$ represents concrete datasets arising from data transformations. A symbolic value $\mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}]$ represents concrete values with tags ts and an iCDF that are computed assuming a noise scale s. Tags are used to detect the provenance of symbolic values and when they arise from different *noisy sources*.

Function accuracy takes queries producing results of type Value a. Such queries are essentially built by performing data aggregation queries (e.g., dpCount) preceded by a (possibly empty) sequence of other primitives like data transformations.[6] Figures 9 and 10 show the interesting parts of our analysis. Given a *well-typed* query q :: Query (Value a), accuracy q = iCDF where $\mathtt{q} \rhd \mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}]$ for some s and ts. The rules in Figure 9 are mainly split into two cases: considering data aggregation queries and sequences of primitives glued together with (>>=).

The symbolic interpretation of dpCount is captured by rule DPCOUNT(see Figure 9(a)). This rule populates the iCDF of the return symbolic value with the corresponding error calculations for Laplace as presented in Definition 5.1 (with the scale adjusted with the accumulated stability). Observe that it extracts the stability information from the type of the considered dataset (ds :: Data s r) and attaches a fresh tag indicating an independently generated noisy value. The symbolic interpretation of dpSum and dpAvg proceeds similarly to dpCount, and we thus omit them for brevity.

[6] We ignore the case of return val :: Query (Value a) since the definition of accuracy is trivial for such a case.

$$\textsc{DPCount}\quad \frac{\mathtt{dataset} :: \mathtt{Data\ s\ r} \qquad \mathtt{iCDF} = \lambda\beta \to \log\left(\frac{1}{\beta}\right)\cdot \mathtt{s} \cdot \frac{1}{\epsilon} \qquad \mathtt{t}\ \textit{fresh}}{\mathtt{dpCount}\ \epsilon\ \mathtt{dataset} \rhd \mathcal{S}\left[\mathtt{iCDF}, \mathtt{s}\cdot\frac{1}{\epsilon}, \{\mathtt{t}\}\right]}$$

$$\textsc{DPMax}\quad \frac{\mathtt{dataset} :: \mathtt{Data\ 1\ r} \qquad \mathtt{iCDF} = \lambda\beta \to \frac{4}{\epsilon}\cdot \log\left(\frac{\mathtt{length\ res}}{\beta}\right)}{\mathtt{dpMax}\ \epsilon\ \mathtt{res\ vote\ ds} \rhd \mathcal{S}[\mathtt{iCDF}, \mathtt{0}, \emptyset]}$$

(a) DP-queries

$$\textsc{Seq-Trans}\quad \frac{\mathtt{k}\ \mathcal{D} \rightsquigarrow^{*} \mathtt{next} \qquad \mathtt{next} \rhd \mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}]}{\mathtt{transform} >\!\!>\!= \mathtt{k} \rhd \mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}]}$$

$$\textsc{Seq-Query}\quad \frac{\mathtt{query} \rhd \mathcal{S}[\mathtt{iCDF_q}, s_{\mathtt{q}}, \mathtt{ts_q}] \qquad \mathtt{k}\ (\mathcal{S}[\mathtt{iCDF_q}, s_{\mathtt{q}}, \mathtt{ts_q}]) \rightsquigarrow^{*} \mathtt{next} \qquad \mathtt{next} \rhd \mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}]}{\mathtt{query} >\!\!>\!= \mathtt{k} \rhd \mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}]}$$

(b) Sequential traversal

$$\textsc{Seq-Part}\quad \frac{\begin{array}{cc} (\mathtt{m\ j}\ \mathcal{D} \rightsquigarrow^{*} \mathtt{next}_j)_{j\in dom(\mathtt{m})} & (\mathtt{next}_j \rhd \mathcal{S}[\mathtt{iCDF}_j, s_j, \mathtt{ts}_j])_{j\in dom(\mathtt{m})} \\ \mathtt{m'} = (j \mapsto \mathcal{S}[\mathtt{iCDF}_j, s_j, \mathtt{ts}_j])_{j\in dom(\mathtt{m})} & \mathtt{k\ m'} \rightsquigarrow^{*} \mathtt{next} \qquad \mathtt{next} \rhd \mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}] \end{array}}{\mathtt{dpPart\ sel\ dataset\ m} >\!\!>\!= \mathtt{k} \rhd \mathcal{S}[\mathtt{iCDF}, s, \mathtt{ts}]}$$

(c) Accuracy calculation when partitioning data

Fig. 9. Accuracy analysis implemented by accuracy.

$$\textsc{Union-Bound}\quad \frac{\mathtt{v_j} = \mathcal{S}[\mathtt{iCDF_j}, \mathtt{s_j}, \mathtt{ts_j}] \qquad \alpha_j = \mathtt{iCDF}_j\left(\frac{b}{n}\right) \qquad \mathtt{iCDF} = \lambda\beta \to \sum_{j=1}^{n} \alpha_j}{\mathtt{ub}\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}] \rightsquigarrow \mathtt{iCDF}}$$

$$\textsc{Chernoff-Bound}\quad \frac{\begin{array}{c} \mathtt{v_j} = \mathcal{S}[\mathtt{iCDF_j}, \mathtt{s_j}, \mathtt{ts_j}] \\ s_{\mathtt{M}} = \max\{s_j\}_{j=1\ldots n} \qquad \nu = \max\left\{\sqrt{\sum_{j=1}^{n} s_{\mathtt{j}}^2}, s_{\mathtt{M}} \cdot \sqrt{\ln\left(\frac{2}{\beta}\right)}\right\} + 0.0001 \qquad \mathtt{iCDF} = \lambda\beta \to \nu \cdot \sqrt{8 \cdot \ln\left(\frac{2}{\beta}\right)} \end{array}}{\mathtt{cb}\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}] \rightsquigarrow \mathtt{iCDF}}$$

$$\textsc{Add-Union}\quad \frac{(\exists j.\mathtt{ts}_j = \emptyset) \vee \bigcap_{j=1\ldots n} \mathtt{ts}_j \neq \emptyset}{\mathtt{add}\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}] \rightsquigarrow \mathcal{S}[\mathtt{ub}\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}], \mathtt{0}, \emptyset]}$$

$$\textsc{Add-Chernoff-Union}\quad \frac{\begin{array}{c} \mathtt{v_j} = \mathcal{S}[\mathtt{iCDF_j}, \mathtt{s_j}, \mathtt{ts_j}] \\ (\forall j.\mathtt{ts}_j \neq \emptyset) \qquad \bigcap_{j=1\ldots n} \mathtt{ts}_j = \emptyset \qquad \mathtt{iCDF} = \lambda\beta \to \mathtt{min}\ (\mathtt{ub}\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}]\ \beta)\ (\mathtt{cb}\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}]\ \beta) \end{array}}{\mathtt{add}\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}] \rightsquigarrow \mathcal{S}[\mathtt{iCDF}, \mathtt{0}, \emptyset]}$$

Fig. 10. Calculation of concentration bounds.

Rule DPMax shows the symbolic interpretation of dpMax whose iCDF aligns with the one appearing in the work of Barthe et al. [8]. Observe that the return value is tainted. The reason for that relies on the fact that the result, which is one of the responses in res, contains no noise—it is rather the process that leads to determining the winning response that has been "noisy." In this light, no scale of noise nor distribution can be associated with the response—as we did, for instance, with dpCount.

To symbolically interpret a sequence of primitives, the analysis gets further split into two cases depending on if the first operation to interpret is a transformation or an aggregation, respectively (see Figure 9(b)). Rule Seq-Trans considers the former, where transform can be any of the transformation operations in Figure 4. It simply uses the symbolic value $\mathcal{D}$ to pass it to the continuation k. It can happen that k $\mathcal{D}$ does not match (yet) any part of DPella's API required for our analysis to continue.[7] However, the embedded domain-specific language nature of DPella makes Haskell reduce k $\mathcal{D}$ to the next primitive to be considered, which we capture as k $\mathcal{D} \rightsquigarrow^*$ next—and we know that it will occur thanks to type preservation. We represent $\rightsquigarrow$ ($\rightsquigarrow^*$) to pure reduction(s) in the host language like function application, pair projections, and list comprehension, among others. The analysis then continues symbolically interpreting the next yield instruction. Rule Seq-Query computes the corresponding symbolic value for the aggregation query. The symbolic value is then passed to the continuation, and the analysis continues with the next yield instruction.

Rule Seq-Part shows the symbolic interpretation of dpPart. The argument m :: Map k (Data s r → Query (Value a)) describes the queries to execute once given the corresponding bins. Since these queries produce values, we need to symbolically interpret each of them to obtain their accuracy estimations. The rule applies each of those queries to a symbolic dataset (m j $\mathcal{D}$).[8] The symbolic values yielded by each bin are collected into the mapping m', which is then passed to continuation k in order to continue the analysis on the next yield instruction.

5.2.1 Concentration Bounds. Figure 10 shows the part of our analysis responsible for applying concentration bounds. Rules Union-Bound and Chernoff-Bound define pure functions (reduction $\rightsquigarrow$) that produce the concentration bounds as described in Definitions 5.2 and 5.3, respectively. We define the function add based on two cases. Rule Add-Union produces a symbolic value with an iCDF generated by the union bound (ub $[\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n]$). The symbolic value is tainted, which is denoted by the empty tags ($\emptyset$). The scale 0 denotes that the scale of the noise and its distribution is unknown—adding Laplace distributions does not yield a Laplace distribution. (However, the situation is different with Gaussians; see Section 5.3.) This rule gets exercised when either the list of symbolic values contains a tainted one ($\exists j.\mathsf{ts}_j = \emptyset$) or has not been independently generated ($\bigcap_{j=1\ldots n} \mathsf{ts}_j \neq \emptyset$). Differently, Add-Chernoff-Union produces a symbolic value with an iCDF that chooses the minimum error estimation between union and Chernoff bound for a given β—sometimes union bound provides tighter estimations when aggregating few noisy values (recall Figure 7). This rule triggers when all values are untainted ($\forall j.\mathsf{ts}_j \neq \emptyset$) and independently generated ($\bigcap_{j=1\ldots n} \mathsf{ts}_j = \emptyset$). At first glance, one could believe that it would be enough to use the scale of the noise to track when values are untainted—that is, if the scale is different from 0, then the value is untainted. Unfortunately, this design choice is unsound: it will classify adding a variable twice as an independent sum: **do** $x \leftarrow$ dpCount ϵ ds; return (add $[x, x]$). It is also possible to consider various ways to add symbolic values to boost accuracy. We could easily write a preprocessing function that, for instance, first partitions the arguments into a subset of independently generated values, applies add to them (thus triggering Add-Chernoff-Union), and finally applies add to the

[7]For instance, k $\mathcal{D} = (\lambda x \rightarrow$ dpCount 1 $x)$ $\mathcal{D}$, and thus $((\lambda x \rightarrow$ dpCount 1 $x)$ $\mathcal{D}) \rightsquigarrow^*$ dpCount 1 $\mathcal{D}$.
[8]For simplicity, we assume that maps are implemented as functions

$$\textsc{Norm-Inf}\quad \frac{\mathtt{v_j} = \mathcal{S}[\mathtt{iCDF_j}, \mathtt{s_j}, \mathtt{ts_j}] \qquad \mathtt{iCDF} = \lambda\beta \to \max\left\{\left|\mathtt{iCDF}_j\left(\frac{\beta}{n}\right)\right|\right\}_{j=1\ldots n}}{\mathtt{norm}_\infty\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}] \rightsquigarrow \mathcal{S}[\mathtt{iCDF}, \mathtt{0}, \emptyset]}$$

$$\textsc{Norm-1}\quad \frac{\mathtt{v_j} = \mathcal{S}[\mathtt{iCDF_j}, \mathtt{s_j}, \mathtt{ts_j}] \qquad \mathtt{iCDF} = \lambda\beta \to \sum_{j=1}^{n}\left|\mathtt{iCDF}_j\left(\frac{\beta}{n}\right)\right|}{\mathtt{norm}_1\ [\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n}] \rightsquigarrow \mathcal{S}[\mathtt{iCDF}, \mathtt{0}, \emptyset]}$$

Fig. 11. Calculation of norms.

obtained results (thus triggering Add-Union). The implementation of DPella enables the writing of such functions in a few lines of code.

5.2.2 Norms Calculation. Figure 11 shows our static analysis when computing $\mathtt{norm}_\infty$ and $\mathtt{norm}_1$, respectively. There is nothing special about the rules except to note that the results are symbolic values that are tainted. The reason for this is that norms are designed to condense (in one measure) the error of the list of the arguments. By doing so, it is hard to assign a specific Laplace distribution with sensitivity s to the overall given vector. We simply say that the return symbolic values are tainted—thus, they can only be aggregated by Add-Union in Figure 10.

5.3 Accuracy of Gaussian Mechanism

As mentioned earlier, DPella supports other notions of DP, such as approximate DP, together with the use of the Gaussian mechanism. Specifically, DPella supports a relaxation of the notion of DP known as (ϵ, δ)-DP, formally defined as follows.

Definition 5.4 ((ϵ, δ)-Differential Privacy[16]). A randomized query $\tilde{Q}(\cdot) : \text{db} \to \mathbb{R}$ satisfies *(ϵ, δ)-DP*, with $\epsilon, \delta \geqslant 0$, if and only if for any two datasets D_1 and D_2 in db, which differ in one row, and for every output set $S \subseteq \mathbb{R}$ we have

$$\Pr[\tilde{Q}(D_1) \in S] \leqslant e^\epsilon \Pr[\tilde{Q}(D_2) \in S] + \delta. \tag{6}$$

The main difference between this notion of privacy and the one described in Definition 2.1 is that (ϵ, δ)-DP introduces the probability mass δ that, intuitively, offers a probabilistic notion of privacy loss. More concretely, (ϵ, δ)-DP ensures that for all adjacent datasets, the absolute value of the privacy loss will be bounded by ϵ with probability $1 - \delta$. Observe that when $\delta = 0$, an $(\epsilon, 0)$-DP query satisfies pure ϵ-DP.

A standard implementation of (ϵ, δ)-DP queries is based on the addition of noise sampled from the Gauss distribution—that is, for $Q : \text{db} \to \mathbb{R}$, an arbitrary function with sensitivity Δ_Q (as described in Definition 2.2), the Gaussian mechanism with parameter σ adds noise scaled to $\mathcal{N}(0, \sigma^2)$ to its output. When the noise to be added is calibrated in terms of ϵ, δ, and Δ_Q, the Gaussian mechanism satisfies (ϵ, δ)-DP as stated in the following theorem.

Theorem 5.1 (Gaussian Mechanism [2]). *For any $\epsilon, \delta \in (0, 1)$, the Gaussian output perturbation mechanism with standard deviation $\sigma = \sqrt{2 \cdot \log(\frac{1.25}{\delta})} \cdot \frac{\Delta_Q}{\epsilon}$ is (ϵ, δ)-differentially private.*

Similarly to the Laplace mechanism, to provide bound estimates on the errors caused by the addition of Gaussian noise, DPella keeps track of Gauss' iCDF. By following the general form of accuracy introduced in Definition 2.3, we have the following definition.

$$\textsc{DPCount}\quad \frac{\texttt{dataset} :: \texttt{Data s r} \qquad \sigma = \sqrt{2 \cdot \log\left(\frac{1.25}{\delta}\right)} \cdot \texttt{s} \cdot \frac{1}{\epsilon} \qquad \texttt{iCDF} = \lambda\beta \to \sigma \cdot \sqrt{2 \cdot \log\left(\frac{2}{\beta}\right)} \qquad \texttt{t}\ \textit{fresh}}{\texttt{dpCount}\ \epsilon\ \texttt{dataset} \rhd \mathcal{S}[\texttt{iCDF}, \sigma^2, \{\texttt{t}\}]}$$

(a) Aggregations

$$\textsc{Chernoff-Bound-Gauss}\quad \frac{\texttt{v}_\texttt{j} = \mathcal{S}[\texttt{iCDF}_\texttt{j}, \texttt{s}_\texttt{j}, \texttt{ts}_\texttt{j}] \qquad \texttt{iCDF} = \lambda\beta \to \sqrt{2 \cdot \sum_{j=1}^{n} s_j \cdot \log\left(\frac{1}{\beta}\right)}}{\texttt{cb}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}] \rightsquigarrow \texttt{iCDF}}$$

$$\textsc{Add-Chernoff-Union}\quad \frac{(\forall j.\texttt{ts}_j \neq \emptyset) \qquad \bigcap_{j=1\ldots n} \texttt{ts}_j = \emptyset \qquad \begin{array}{c}\texttt{v}_\texttt{j} = \mathcal{S}[\texttt{iCDF}_\texttt{j}, \texttt{s}_\texttt{j}, \texttt{ts}_\texttt{j}] \\ \texttt{iCDF} = \lambda\beta \to \min(\texttt{ub}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}]\ \beta)(\texttt{cb}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}]\ \beta)\end{array}}{\texttt{add}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}] \rightsquigarrow \mathcal{S}[\texttt{iCDF}, \sum_{\texttt{j}=1}^{\texttt{n}} \texttt{s}_\texttt{j}, \bigcup_{\texttt{j}=1\ldots\texttt{n}} \texttt{ts}_\texttt{j}]}$$

(b) Concentration bounds

Fig. 12. Accuracy analysis for the Gaussian mechanism.

Definition 5.5 (Accuracy for the Gaussian Mechanism). Given a randomized query $\tilde{Q}(\cdot) : \mathrm{db} \to \mathbb{R}$ implemented with the Gaussian mechanism as described previously,

$$\Pr\left[|\tilde{Q}(D) - Q(D)| > \sigma \cdot \sqrt{2 \cdot \log\left(\frac{2}{\beta}\right)}\right] \leqslant \beta, \qquad (7)$$

where the iCDF to be stored by DPella refers to the function $\lambda\beta \to \sigma \cdot \sqrt{2 \cdot \log(\frac{2}{\beta})}$.

From an implementation standpoint, adding the Gaussian mechanism to our framework does not significantly alter the presented primitives, and, in particular, privacy preservation remains (almost) unchanged. The most significant changes can be seen when calculating the accuracy of aggregations and their combinations.

The symbolic interpretation of aggregations is updated accordingly to keep track of Gauss' iCDF, as well as its respective noise scale determined by σ^2 as depicted in Figure 12(a) for the case of `dpCount`. Additionally, Figure 12(b) shows how concentration bounds are applied for the case of the Gaussian mechanism—Union-Bound and Add-Union are omitted since they are the same as the ones in Figure 10. In general, the accuracy analysis for addition of aggregations follows the one presented previously for the Laplace mechanism. The main difference is seen when adding independent values. In this case, we use the well-known fact that the addition of independent normally distributed random variables is also normally distributed. This means that after executing the Add-Chernoff-Union, we do not lose information about the distribution of our result as we used to do under the Laplacian setting. This effect can be seen in the generated symbolic value $\mathcal{S}[\texttt{iCDF}, \sum_{j=1}^{n} s_j, \bigcup_{j=1\ldots n} \texttt{ts}_j]$, where $\sum_{j=1}^{n} s_j$ indicates that the variance of the new value is calculated as the addition of the variances of the components being added, and $\bigcup_{j=1\ldots n} \texttt{ts}_j$ indicates that the new value is statistically dependent of the involved values.

This is a useful feature when combining queries in batches. For instance, Figure 13 shows the query plan `totalCountG` that adds the results of 100 queries—using Gaussian `dpCount` that takes as input the tuple (ϵ, δ) and the dataset—similar to the one presented in Figure 8, but it does so by adding the first half of the queries (line 7), then adding the second half (line 8), and finally returning the addition of the two halves (line 9). How will DPella calculate the theoretical error of `totalCountG`?

```
1 totalCountG :: Query (Value Double)
2 totalCountG = do
3   v1   ← dpCount (0.3 ,1e-5) ds1
4   v2   ← dpCount (0.25,1e-5) ds2
5   ...
6   v100 ← dpCount (0.5 ,1e-3) ds100
7   let h1 = add [v1,  v2,  ...,v50 ]
8   let h2 = add [v51,v52,...,v100]
9   return (add [h1,h2])
```

Fig. 13. Combination in batches.

Table 1. Implemented Literature Examples

Category	Application	Programs
PINQ-like	CDFs [39]	cdf1, cdf2, cdfSmart
	Term frequency [40]	queryFreq, queriesFreq
	Network analysis [39]	packetSize, portSize
	Cumulative sums [6]	cumulSum1, cumulSum2, cumulSumSmart
Counting queries	Range queries via identity, histograms [30], and wavelet [61]	i_n, h_n, y_n

Observe that h_1 and h_2 are constructed as combinations of untainted values, meaning that when performing the additions at lines 7 and 8, the Chernoff bound could be triggered. More importantly, DPella still has information about their distribution. Furthermore, h_1 and h_2 are statistically independent (they do not share sub-queries), so when computing their addition at line 9, Chernoff bound could also be triggered; this could not have been possible under the Laplace mechanism, since once a value is calculated as a combination of values, their distribution becomes unknown and only union bound could be applied. In this sense, the Gaussian mechanism might yield tighter error bounds when dealing with queries that are created in batches, especially when the number of batches is big enough to trigger use of the Chernoff bound.

6 CASE STUDIES

In this section, we will discuss the advantages and limitations of our programming framework. Moreover, we will go in depth into using DPella to analyze the interplay of privacy and accuracy parameters in hierarchical histograms.

6.1 DPella Expressiveness

First, we start by exploring the expressiveness of DPella. For this, we have built several analyses found in the DP literature—see Table 1—which we classify into two categories, *PINQ-like queries* and *counting queries*. The former class allows us to compare DPella expressivity with the one of PINQ and the latter with APEx.

PINQ-like queries. We have implemented most of PINQ's examples [39, 40], such as different versions of CDFs (sequential, parallel, and hybrid) and network tracing-like analyses (e.g., determining the frequency a term or several terms have been searched by the users, and computing port and packet size distribution); additionally, we considered analyses of *cumulative sums* [6], which are queries that share some commonalities with CDFs. The interest over differentially private CDFs and cumulative partial sums applications rely on the existing several approaches to inject noise;

$$\begin{bmatrix}1&0&0&0\\1&1&0&0\\1&1&1&0\\1&1&1&1\\0&1&0&0\\0&1&1&0\\0&1&1&1\\0&0&1&0\\0&0&1&1\\0&0&0&1\end{bmatrix}$$

$\mathbf{W}_{\mathbf{R}_4}$

$$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

$\mathbf{I}_4$

$$\begin{bmatrix}1&1&1&1\\1&1&0&0\\0&0&1&1\\1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

$\mathbf{H}_4$

$$\begin{bmatrix}1&1&1&1\\1&1&-1&-1\\1&-1&0&0\\0&0&1&-1\end{bmatrix}$$

$\mathbf{Y}_4$

Fig. 14. Workload of all range queries and query strategies for four ranges.

such choices will directly impact the accuracy of our results and therefore are ideal to be tested and analyzed in DPella. The structures of these examples closely follow the ones of the CDFs presented in previous sections, which are straightforward implementations. DPella supports these queries naturally since its expressiveness relies on its primitives and, by construction, they follow the ones of PINQ very closely. However, as stated in previous sections, our framework goes a step further and exposes to data analysts the accuracy bound achieved by the specific implementation. This feature allows data analysts to reason about accuracy of the results—without actually executing the query—by varying the strategy of the implementation and the parameters of the query. For instance, in Section 3, we showed how an analyst can inspect the error of a sequential and parallel strategy to compute the CDF of packet lengths. Furthermore, the data analyst can take advantage of DPella being an embedded DSL and write a Haskell function that takes any of the approaches (`cdf1` or `cdf2`) and varies epsilon aiming to certain error tolerance (for a fixed confidence interval), or vice versa. Such a function can be as simple as a brute force analysis or as complex as a heuristic algorithm.

Counting queries. To compare our approach with the tool APEx [25], we consider range queries analyses—a specific subclass of counting queries. APEx uses the *matrix mechanism* [34] to compute counting queries. This algorithm answers a set of linear queries (called the *workload*) by calibrating the noise to specific properties of the workload while preserving DP. More in detail, the matrix mechanism uses some *query strategies* as an intermediate device to answer a workload, returning a DP version of the query strategies (obtained using the Laplace or Gaussian mechanism) from which noisy answers of the workload are derived. The matrix mechanism achieves an almost optimal error on counting queries. To achieve such error, the algorithm uses several non-trivial transformations that cannot be implemented easily in terms of other components. APEx implements it as a black box, and we could do the same in DPella (see Section 9). Instead, in this section, we show how DPella can be directly used to answer sets of counting queries using some of the ideas behind the design of the matrix mechanism, and how these answers improve with respect to answering the queries naively, thanks to the use of partition and the Chernoff bound.

To do this, we have implemented several strategies to answer a specific workload $\mathbf{W}_\mathbf{R}$: the set of all range queries over a domain. Figure 14 illustrates the workload that would be the answer for a frequency count of four ranges. The identity strategy $\mathbf{I}_4$ represents four queries (number of rows) computing the noisy count of each range (number of columns). The hierarchical strategy $\mathbf{H}_4$ contains seven queries representing a binary hierarchy of sums, whereas the wavelet strategy $\mathbf{Y}_4$ contains four queries representing the Haar wavelet matrix.

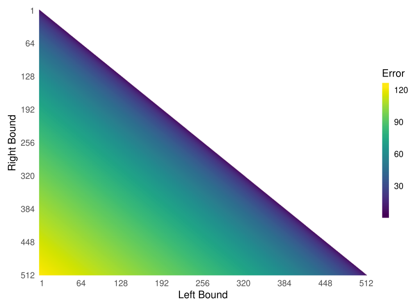

Fig. 15. Error of each range query in $\mathbf{W_R}$ using strategy $\mathbf{I_n}$ with $n = 512, \epsilon = 1$, and $\beta = 0.05$.

Our implementation generates noisy counts, and any possible combination of them will yield (at least) the same error as using strategy $\mathbf{I_4}$. In other words, the most accurate answer for $\mathbf{W_R}$ will be yield by the identity strategy. This is not unexpected, because to use the other queries strategies more efficiently, we would need transformations similar to the ones used in the matrix mechanism.

Figure 15 exposes the error of answering each range query (i.e., each row) in $\mathbf{W_R}$ with strategy $\mathbf{I_n}$ and $n = 512$. Although we use the same kind of plot, this error cannot be directly compared with the one shown in Figure 7 of Li et al. [34], as we use a different error metrics: (α,β)-accuracy vs **mean squared error (MSE)**. Nonetheless, we share the tendency of having lower error on small ranges and significant error on large ranges. Now, since the noisy values that will be added (using the function add) are statistically independent, we can use the Chernoff bound to show that the error is approximately $O(\sqrt{n})$ for each range query, and a maximum error of $O(\sqrt{n \log n})$ for answering any query in $\mathbf{W_R}$. If we compare our maximum error $O(\sqrt{n \log n})$ with the one of the matrix mechanism based on the identity strategy $O(n/\epsilon^2)$, it becomes evident how Chernoff bound is useful to provide tighter accuracy bounds. Unfortunately, as stated previously, the error of strategies $\mathbf{H_n}$ and $\mathbf{Y_n}$ in DPella is not better than the one of the strategy $\mathbf{I_n}$, so we cannot reach the same accuracy the matrix mechanism achieves with these strategies (see Figure 7 of Li et al. [35]). This limitation can be addressed by leveraging the fact that DPella is a programming framework that could be *extended* by adding the matrix mechanism—and some other features—as black-box primitives.

Black-box primitives. To demonstrate the effects of adding black-box operators in DPella, let us consider a rather simple query using primitive `dpMax`. Suppose that there is a highly contagious virus spreading in a state. To reduce this virus's rapid spreading, one might want to alert the population where there are more cases of infections so that they can quarantine. In this scenario, we will consider two sources of information: one that is private, containing information of infected patients such as their identity number and ZIP code (i.e., **data** `RowV = V {id :: String, zip :: ZIP}`), and other that is public, such as the phone numbers of people living in each ZIP code (referred as `contact :: [(ZIP, Phone)]`).

With this information at hand, we implement query `alert` (Figure 16). Function `alert` uses primitive `dpMax` (line 3) to access the private dataset of infected patients and return the ZIP code with more infection cases, which will determine which zone is at risk. Then we obtain the phone number of all residents of such an area using `getPhone` function (lines 6–8). Additionally, in line 4, we function `useIndex` to access ZIP code enclosed in `atRisk` to later be used by function `getPhone`.

Function `useIndex` is then a new combinator with type `useIndex :: (a → b) → Value a → Value b` that has been introduced to be applied only to values generated with `dpMax`. The insight behind its implementation relies on the fact that the output of a `dpMax` computation does not

```
alert :: ε → Data 1 RowV → Query (Value [Phone])
alert eps ds = do
  atRisk ← dpMax eps allZIPs zip ds
  return (useIndex getPhone atRisk)
    where
      getPhone :: ZIP → [Phone]
      getPhone z = [snd info | info ← contact
                                  , fst info ≡ z]
```

Fig. 16. Using dpMax.

```
hierarchical1 [e1, e2, e3] dat = do
  -- h₁ :: Map Gen (Value Double)
  -- h₂ :: Map (Gen, Age) (Value Double)
  -- h3 :: Map (Gen, Age, Nationality) (Value Double)
  h₁ ← byGen        e1 dat
  h₂ ← byGenAge     e2 dat
  h3 ← byGenAgeNat e3 dat
  return (h₁, h₂, h3)
```

(a) Approach I: distribute budget among levels

```
hierarchical2 e dat = do
  h3 ← byGenAgeNat e dat
  h₂ ← level2 h3
  h₁ ← level1 h3
  return (h₁, h₂, h3)
```

(b) Approach II: query most detailed level

Fig. 17. Implementation of hierarchical histograms.

contain noise; therefore, applying any function to its result should not affect its interpretation of accuracy.

From this example, we expect to illustrate the convenience of adding deferentially private algorithms as black boxes in DPella (e.g., dpMax) and their relative smooth integration into the system once their accuracy estimation has been determined. Ultimately, it highlights that integrating these primitive usually cascades into defining new combinators (as useIndex) to further manipulate their outputs.

6.2 Privacy and Accuracy Trade-off Analysis

We study histograms with certain hierarchical structure (commonly seen in Census Bureau analyses) where different accuracy requirements are imposed per level, and where varying one privacy or accuracy parameter can have a *cascade impact* on the privacy or accuracy of others. We consider the scenario where we would like to generate histograms from the Adult database[9] to perform studies on gender balance. The information that we need to mine is not only a histogram of the genders (for simplicity, just male and female) but also how the gender distributes over age, and within that how age distributes over nationality—thus exposing a hierarchical structure of three levels.

Our first approach is depicted in Figure 17(a), where hierarchical1 generates three histograms with different levels of details. This query puts together the results produced by queries byGen, byGenAge, and byGenAgeNationality, where each query generates a histogram of the specified set of attributes. Observe that these sub-queries are called with potentially different epsilons, namely e1, e2, and e3, then under sequential composition, we expect hierarchical1 to be e1+e2+e3-differentially private.

[9]https://archive.ics.uci.edu/ml/datasets/adult.

Table 2. Budgeting with α Tolerances, $\beta = 0.05$, and Total $\epsilon = 3$

Histogram	α Tolerance	Status	ϵ	α
`byGen`	100	✓	0.06	61.48
`byGenAge`	100	✓	0.06	96.13
`byGenAgeNat`	100	✓	0.11	85.74
`byGen`	10	✓	0.41	8.99
`byGenAge`	50	✓	0.16	36.05
`byGenAgeNat`	5	× MaxBud	1	9.43
`byGen`	5	✓	0.76	4.85
`byGenAge`	5	× MaxBud	1	5.76
`byGenAgeNat`	10	✓	0.96	9.82

We proceed to explore the possibilities to tune the privacy and accuracy parameters to our needs. In this case, we want a confidence of 95% for accuracy (i.e., $\beta = 0.05$) with a total budget of 3 ($\epsilon = 3$). We could manually try to take the budget $\epsilon = 3$ and distribute it to the different histograms in many different ways and analyze the implication for accuracy by calling `accuracy` on each sub-query. Instead, we write a small (simple, brute force) optimizer in Haskell that splits the budget uniformly among the queries (i.e., `e1 = 1`, `e2 = 1`, and `e3 = 1`) and tries to find the minimum epsilon that meets the accuracy demands per histogram. In other words, we are interested in minimizing the *privacy loss* at each level bounding the maximum accepted error. The optimizer essentially adjusts the different epsilons and calls `accuracy` during the minimization process. To ensure termination, the optimizer aborts either after a fixed number of calls to `accuracy` or when the local budget e_i is exhausted.

Table 2 shows some of our findings. The first row shows what happens when we impose an error of 100 at every level of detail (i.e., each bar in all of the histograms could be at most $+/-100$ off). Then, we only need to spend a little part of our budget—the optimizer finds the minimum epsilons that adhere to the accuracy constrains. Instead, the second row shows that if we ask to be *gradually* more accurate on more detailed histograms, then the optimizer could fulfill the first two demands and be aborted on the most detailed histogram (`byGenAgeNat`) since it could not find an epsilon that fulfills that requirement—the best we can do is spend all of the budget and obtain an error bound of 9.43. Finally, the last row shows what happens if we want *gradually* tighter error bounds on the less detailed histograms. In this case, the middle layer can be "almost" fulfilled by expending all of the budget and obtaining an error bound of 5.76 instead of 5. Although the results from Table 2 could be acceptable for some data analysts, they might not be for others.

We propose an alternative manner to implement the same query that consists of spending privacy budget *only* for the most detailed histogram. As shown in Figure 17(b), this new approach spends all of the budget `e` on computing `h3` ← `byGenAgeNat e dat`. Subsequently, the algorithm builds the other histograms based on the information extracted from the most detailed one. For that, we add the noisy values of `h3` (using helper functions `level2` and `level1`) creating the rest of the histograms representing the Cartesian products of gender and age, and gender, respectively. This methodology will use `add` and $\texttt{norm}_\infty$ to compute the derived histograms and therefore will not consume more privacy budget. Observe that the query proceeds in a bottom-up fashion—that is, it starts with the most detailed histogram and finishes with the less detailed one. Now that we have two implementations, which one is better? Which one yields the better trade-offs between

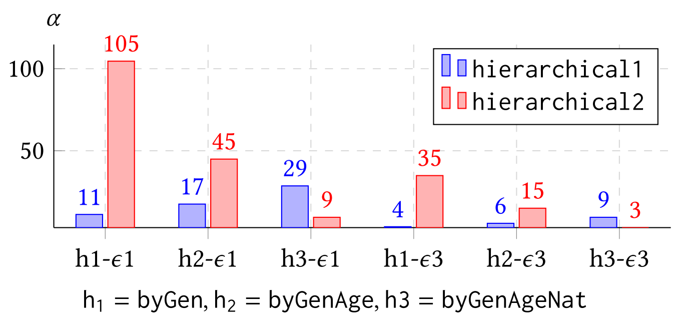

Fig. 18. `hierarchical1` vs. `hierarchical2`.

privacy and accuracy? Figure 18 shows the accuracy of the different levels of histograms (i.e., h_1, h_2, and h3) when fixing $\beta = 0.05$ and a global budget of $\epsilon = 1$ (h1-ϵ1, h2-ϵ2, and h3-ϵ3) and $\epsilon = 3$ (h1-ϵ3, h2-ϵ3, and h3-ϵ3)—we obtained all of this information by running repetitively the function `accuracy`. From the graphics, we can infer that splitting the privacy budget per level often yields more accurate histograms. However, observe the exception at the most detailed histogram h3: as expected, `hierarchical1` will use just one-third of the budget, whereas `hierarchical2` uses all of it, and hence the first approach will return a noisier count.

6.3 *K*-way Marginal Queries on Synthetic Data

When compared with (non-compositional) approaches for estimating accuracy based on synthetic or public data, such as that of Hay et al. [29], the static analysis of DPella can be used in a complimentary manner to quickly (and precisely) estimate privacy and accuracy for a wide range of simple queries. There are certain kinds of queries where it is more convenient to use our static analysis than synthetic data for high-dimensional datasets.

As an example, we focus on the problem of releasing, in a differentially private manner, the k-way marginals of a binary dataset $D \in (\{0,1\}^d)^n$. This is a classical learning problem that has been studied extensively in the DP literature (see [14, 22, 58], among others). A k-marginal query, also called a *k-conjunction*, returns the count of how many individual records in D have $k < d$ attributes set to certain values. For simplicity, we will work with 3-way marginal queries to compare performance between DPella and using synthetic data. The goal of our analysis is to release all 3-way marginals of a dataset. This is implemented through the functions depicted in Figure 19.

Function `allChecks` counts how many records have 3-attributes set to 1. Auxiliary function `combinatory d k` generates k-tuples arising from the combination of indexes $0, 1, \ldots, d$ taken k at the time. In our example, the number of generated tuples is $\binom{\mathtt{dim}}{3}$. For each tuple, `allChecks` filters the rows that have attributes `i`, `j`, and `k` set to 1 (implemented as `dpWhere allOne db`) for then making a noisy count (`dpCount localEps tab`). Last, function `threeMarginal` collects the counts for the different considered attributes and places them into a vector ($\mathtt{norm}_\infty$ `checks`).

We run `threeMarginal` considering a synthetic dataset (db) that has only one row with all of the attributes set to zeros. Setting all of the attributes to zero produces that all of the counts are 0, and thus we are able to measure the noise on each run and accuracy accordingly. We run `threeMarginal` approximately 1,000 times for each dimension to measure the noise magnitude, where we took the `1`-β percentile with β = `0.05` (as we did in many of our case studies). Observe that we have $\binom{\mathtt{dim}}{3}$ queries and thus $\binom{\mathtt{dim}}{3}$ independent sources of noise, which need a high number of runs to be well represented. In general, for this kind of task, one is interested in bounding the max error that can occur in one of the queries (the ℓ_∞ norm over the output). For this task, the

```
 -- Perform all 3-way combinations up to attribute dim
allChecks :: ϵ → Int → Data s BinR → [Query (Value Double)]
allChecks localEps dim db = do
  (i, j, k) ← combinatory (dim-1) 3
  let allOne r = (r !! i) ≡ (r !! j) ≡ (r !! k) ≡ 1
  return (do tab ← dpWhere allOne db
             dpCount localEps tab)

 -- Compute k-way marginals
threeMarginal :: ϵ → Int → Data s BinR → Query (Value [Double])
threeMarginal localEps dim db = do
  checks ← sequence (allChecks localEps dim db)
  return (norm∞ checks)
```

Fig. 19. *K*-way marginal implementation.

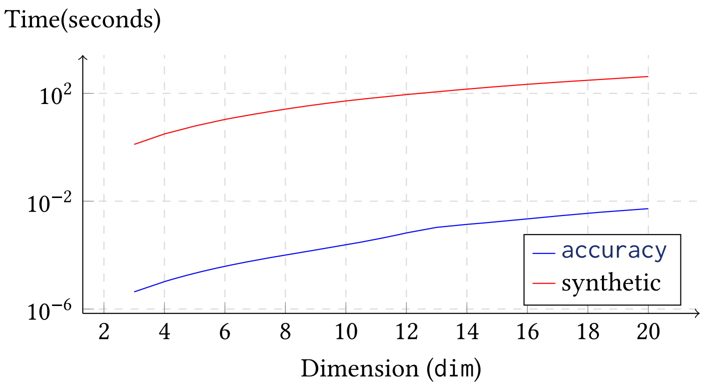

Fig. 20. Performance comparison between accuracy (DPella) and estimating errors using synthetic analysis.

empirical error is well aligned with the theoretical one provided by DPella by calling the function accuracy. The latter is computed by taking a union bound over the error of each individual query. For each query we have a tight bound, and the union bound gives us a tight bound over the max. However, we observe a significant difference in performance.

Figure 20 shows (in log scale) the time difference when calculating accuracy by DPella and on synthetic data when the dimension of the dataset increases. Already in low dimension, the difference in performance is many orders of magnitude in favor of DPella—a tendency that does not change when the dimension goes above 20. The main reason for that comes down to the fact that DPella, as an static analysis, does not execute the filtering dpWhere allOne db (as well as any other transformation, recall Section 5.2), which an approach based on synthetic data should do many times—in our case, 1,000 iterations for each dimension. We expect that for more complex tasks, this difference is even more evident.

7 TESTING ACCURACY

In previous sections, we have seen the usefulness of the accuracy function to inspect queries' error and reason about the trade-offs of privacy and accuracy, among other perks. It is clear then that providing theoretical bounds over the errors of the implemented queries becomes handy to ease and assist data analysts' tasks. However, one might argue that having a theoretical bound is as important as producing a measurement of the *tightness* of such calculations. In this section, we focus on the verification of how close DPella's accuracy calculations are to the real error bounds.

Thanks to DPella's data independence, we have been able to create the primitive `empiric` that allows analysts to compare the theoretical bound (provided by `accuracy`) against an empirical one. It offers a way to compare DPella's estimations for a query against its empirical error while still preserving the privacy of the data subjects.

The primitive `empiric` is a follows:

$$\texttt{empiric} :: (\epsilon \to \texttt{Data 1 r} \to \texttt{Query (Value a)}) \to \texttt{Iter} \to \epsilon \to \beta \to \texttt{IO}\ \alpha$$

Given a query plan (of type $\epsilon \to$ `Data 1 r` $\to$ `Query (Value a)`), a number of iterations (where `Iter` is isomorphic to the type `Int`), a fixed privacy loss ϵ, and confidence β, primitive `empiric` will return the empirical error α_{emp} of the given query using the theoretical error α_{th} provided by `accuracy` with β.

Ideally, α_{emp} should be significantly close to α_{th}. In particular, since `accuracy` yields an *upper bound* of the error, when `empiric` is run multiple times we expect α_{emp} to be less than or equal to α_{th} most of the time. The unsatisfiability of this condition indicates that the probability of being above the theoretic error is higher than anticipated, from which we can deduce that DPella's error estimation is unsound and it does not actually yield an upper bound of the query's accuracy. However, if for most of the runs we observe that $\alpha_{\text{emp}} \ll \alpha_{\text{th}}$, we can infer that DPella's estimations are loose, indicating that we could either increase the confidence or decrease the error.

The procedure followed by `empiric` is fairly simple. First, it executes the given query as many times as indicated over an *empty dataset*; this process clearly does not involve any sensitive information. However, the attentive reader might have noticed that these executions will allow us to inspect the query's noise since they will only return the perturbation to be added—that is, we are sampling as many times as iterations from the Laplace distribution (or Gauss, depending on the mechanism) scaled by the sensitivity of the query under consideration. From the samples, we calculate each empirical error α^i_{emp} either applying the absolute value to the i-th sample if the query's output is a scalar or the specified norm (i.e., ℓ_∞, ℓ_2, ℓ_1, or rmsd) if the output is multi-dimensional. Then, α_{emp} is computed as the $1 - \beta$ percentile of all α^i_{emp}, where $i = 1, \ldots, \text{iter}$.

To illustrate how `empiric` could be used by an analyst, recall the example of the 3-way marginal discussed in the previous section (see Section 6.3). Previously, we claimed that the empirical error of function `threeMarginal` from Figure 19 is well aligned with the theoretical one provided by DPella. This statement can be now verified using the `empiric` primitive. For `localEps = 0.1` and `dim` ranging from 3 to 20, Figure 21 shows the results of calculating the empirical error of `threeMarginal` with β set to 0.05 and iterating 1,000 and 10,000 times. From these results, we can conclude that increasing the number of iterations will stabilize the results, making the analyses easier, and that the empirical error provided for DPella for function `threeMarginal` is indeed very close to the empirical error bound. Moreover, it depicts DPella's soundness, since in both cases (for 1K and 10K iterations) most of the α_{emp} were below α_{th}'s line.

We acknowledge that not all configurations of DPella programs will have an accuracy estimation as tight as the one presented earlier. In particular, one can imagine a scenario where `n` non-independent noisy values are being added. The theoretical error of such a query will be calculated using the union bound, which has been established [25] to be a loose approximation of the sum's error. Thus, when comparing the empirical error of these queries against `accuracy`'s projection, we should expect a greater discrepancy in favor of the empiric calculation. Under these circumstances, analysts can consider modifying a code's structure to take advantage of the Chernoff bound as much a possible or adjust their parameters (ϵ and β).

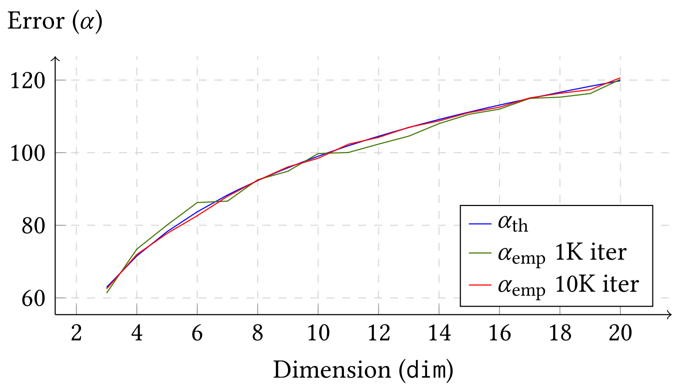

Fig. 21. Results of `empiric` over 3-way marginals.

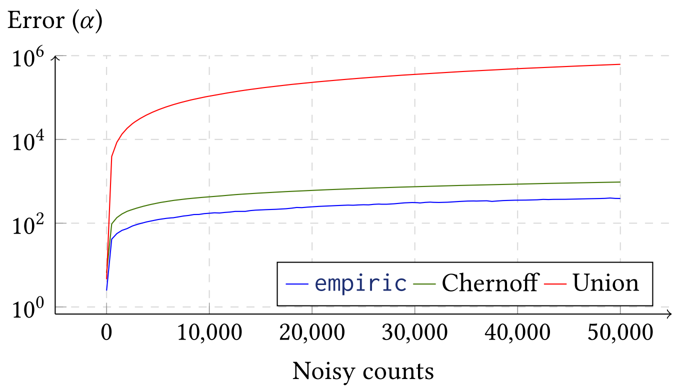

Fig. 22. Results of `empiric` over addition of noisy counts.

Since triggering the Chernoff bound produces the best error estimation when adding several (independent) noisy values, one might ask, how tight is this bound? Figure 22 compares the results of `empiric` over a query adding up to 50,000 independent noisy values with β = `0.2` and `localEps` = `1` against the results of `accuracy` triggering the Chernoff bound; additionally, the estimations under union bound are shown as a baseline for comparison. Clearly, the union bound's theoretical errors are several orders of magnitude apart from the empirical errors, showcasing its overconservative calculations. However, the Chernoff bound provides a much tighter bound that grows by a constant factor w.r.t. the empirical error—the reader should keep in mind that the log scale might make it difficult to appreciate this claim; still, it could be quickly confirmed by analyzing the ratio between the two errors.

We argue that DPella's estimations can be seen as a *quick* initial step into inspecting queries' accuracy that could be further complemented by other techniques of error estimation—integration of such algorithms are left as future work. DPella's analyses are particularly useful when dealing with high-dimensional or more complex tasks since—as shown previously—its estimations do not have computational overheads.

8 API GENERALIZATION

Up to this point, we have seen DPella as a framework where data analysts can implement their differentially private consults using either the Laplace or the Gaussian mechanism, but not both

```
  -- Computations          -- Privacy parameters
data Query p a           data Input p m

  -- Privacy notions       -- Mechanisms
data PureDP              data Laplace
data AproxDP             data Gauss
```

Fig. 23. New types for DPellas's generalization.

```
dpCount :: (Stb s, PrivN p, Mech p m) ⇒ Input p m → Data s r → Query p (Value Double)
dpSum   :: (Stb s, PrivN p, Mech p m) ⇒ Input p m → Range a b → (r → Double) → Data s r
           → Query p (Value Double)
dpAvg   :: (Stb s, PrivN p, Mech p m) ⇒ Input p m → Range a b → (r → Double) → Data s r
           → Query p (Value Double)
dpMax   :: (Eq a,  PrivN p, Mech p m) ⇒ Input p m → Responses a → (r → a) → Data 1 r
           → Query p (Value a)
```

Fig. 24. Updated aggregations.

simultaneously. However, there might be cases where analysts would want to combine queries using Laplace and Gaussian noise. DPella is designed to allow programmers to mix results from different mechanisms as long as they are implemented under the same privacy notion; for instance, only results from mechanisms deploying approximate-DP can be combined within themselves. To support this behavior, DPella's computations are generalized by their privacy notion, for which we introduce the abstract data types presented in Figure 23.

Values of type `Query p a` represent computations yielding outputs of type `a` under the privacy notion `p`. Variable `p` can be instantiated to types `PureDP` or `AproxDP`, and thus a term of type `q :: Query Pure (Value Double)` is a pure differentially private computation whose output has type `Value Double`. As established in Section 4, type `Query p a` is a monad, and then sequencing queries is done through the *bind* function. Hence, to enforce a combination of queries *only* within the same privacy notion, the type of function (>>=) changes to

```
( >>= ) :: Query p a → (a → Query p b) → Query p b
```

where all of the computations involved must share the same privacy notion `p`. This restriction ensures that the principles of composition can be properly applied when combining queries.

To determine which mechanism (or source of noise) will be used in a computation, aggregations are updated to take an argument of type `Input p m` (instead of ϵ or (ϵ, δ)) as depicted in Figure 24. Type variable `p` still refers to the privacy notion, whereas `m` indicates which mechanism should be used to ensure `p`-DP. Variable `m` can be instantiated to types `Laplace` or `Gauss`. Hence, a term of type `arg :: Input AproxDP Gauss` represents an input for a computation guaranteeing approximate-DP using the Gaussian mechanism. The introduction of type `Input p m` allows us to refer to our mechanisms' arguments without specifying them directly. However, when privacy notion and mechanism are chosen, the input gets concretized to either ϵ or (ϵ, δ). More precisely, the implementation of `Input PureDP Laplace` is isomorphic to ϵ, whereas `Input AproxDP Gauss` and `Input AproxDP Laplace` are isomorphic to (ϵ, δ).

Last, new type constraints `PrivN p` and `Mech p m` are introduced in all aggregations (recall Figure 24) to avoid invalid combinations of `p` and `m` (e.g., `q :: Query Bool a` and `arg :: Input PureDP Gauss`).

```
1 hist :: (Stb s, PrivN p, Mech p m) ⇒ [Age] → (Age → Bool) → Input p m → Data s Age
2       → Query p [Value Double]
3 hist bins f arg dataset = do
4   tab   ← dpWhere f dataset
5   parts ← dpPartRepeat (dpCount arg) bins assignBin tab
6   return (Map.elems parts)

 7 mixHist :: Query AproxDP (Value [Double])
 8 mixHist = do
 9   let binsLap   = [5, 10, 15, 20, 25]
10       binsGauss = [50, 55, 60, 65]
11   lapHist   ← hist binsLap   (⩽ 25) (In @AproxDP @Laplace (0.25, 1e-3)) ages
12   gaussHist ← hist binsGauss (⩾ 50) (In @AproxDP @Gauss   (0.5,  1e-3)) ages
13   return (norm∞ (lapHist + gaussHist))
```

Fig. 25. Histogram using Laplace and Gauss mechanisms.

With this new interface, analysts can implement generic programs without specifying which mechanism (and privacy notion) will be used during its execution, these computations can be later used to instantiate specific queries. For instance, Figure 25 presents functions `hist` (lines 1–6), which creates a histogram of ages taking as input a list of bins, a selector function `f`, the general input of the mechanism `arg`, and the dataset. This function filters the given dataset accordingly to the selector functor (line 4), partitions the data into the bins and perform a noisy count on each partition (line 5), and finally returns a list with all noisy counts (line 6). Later, the analyst can call `hist` to define a noisy query under approximate-DP, called `mixHist`, returning a histogram where some bins are computed using the Laplace mechanism (lines 9 and 11), whereas others use the Gaussian mechanism (lines 10 and 12). Observe that we use type applications, such as `In @AproxDP @Laplace (...)` and `In @AproxDP @Gauss (...)`, to specify which privacy notion and mechanism should be used.

8.1 Implementation and Accuracy Estimations

The accuracy analysis of generalized computations closely follow the one defined for the Laplace and Gaussian mechanisms. The main difference is that symbolic values $\mathcal{S}[\mathtt{iCDF}, \mathtt{d}, \mathtt{ts}, \eta]$ now keep track of an extra value, η, representing the distribution from where the noise is drawn. Values of η are limited to L for Laplace, G for Gauss, and U for unknown distributions when noisy values are combined.

The interpretation of aggregations can be summarized by inspecting `dpCount`. Rule DPCount in Figure 26 shows that the value of η is determined by the distribution indicated at the type of input `arg` (i.e., $\eta = \mathtt{m}$). Internally, type `Laplace` gets translated into the value L and type `Gauss` into the value G. With this information, function `noiseScale` computes the scale of Laplace (following Theorem 2.1) or the variance of Gauss (according to Theorem 5.1) depending on the value of η. Similarly, function `errorDist` returns the iCDF of the corresponding distribution.

Major changes occur on the interpretation of combinator add since now the list of values to be added are potentially mixed w.r.t. their distribution. If not done carefully, computing the error of such addition will rarely trigger the Chernoff bound. To maximize the chances of using Chernoff bound, rule Add-Chernoff-Union illustrates how the static analysis splits the symbolic values $\mathtt{v}_j$ into three disjoints sets: values $\mathtt{V}_\mathtt{L}$ are independent Laplacian, values $\mathtt{V}_\mathtt{G}$ are independent Gaussians, and values $\mathtt{V}_\mathtt{U}$ are either non-independent or their distribution is unknown. Each of these groups of values will be added between them to later be used as partial sums for the final result. The

DPCount

$$\dfrac{\begin{array}{c}\texttt{dataset} :: \texttt{Data}\ \texttt{s}\ \texttt{r} \qquad \texttt{arg} :: \texttt{Input}\ \texttt{p}\ \texttt{m} \\ \eta = \texttt{m} \qquad \texttt{sc} = \texttt{noiseScale}\ (\texttt{s}, \texttt{arg}, \eta) \qquad \texttt{iCDF} = \lambda\beta \to \texttt{errorDist}\ (\beta, \texttt{s}, \texttt{arg}, \texttt{sc}, \eta) \qquad \texttt{t}\ \textit{fresh}\end{array}}{\texttt{dpCount}\ \texttt{arg}\ \texttt{dataset} \rhd \mathcal{S}[\texttt{iCDF}, \texttt{sc}, \{\texttt{t}\}, \eta]}$$

Add-Chernoff-Union

$$\dfrac{\begin{array}{c}\texttt{v}_\texttt{j} = \mathcal{S}[\texttt{iCDF}_\texttt{j}, \texttt{s}_\texttt{j}, \texttt{ts}_\texttt{j}, \eta_\texttt{j}] \\ \texttt{d} \in \{\texttt{L}, \texttt{G}\} \qquad \texttt{V}_\texttt{d} = \{\texttt{v}_\texttt{i} \mid \eta_\texttt{i} = \texttt{d} \wedge \bigcap_{\texttt{i}=1\ldots} \texttt{ts}_\texttt{i} = \emptyset\} \qquad \texttt{V}_\texttt{U} = \texttt{v}_\texttt{j} \notin \{\texttt{V}_\texttt{L} \cup \texttt{V}_\texttt{G}\} \\ \texttt{iCDF}_\texttt{d} = \lambda\beta \to \texttt{min}\ (\texttt{ub}\ \texttt{V}_\texttt{d}\ \beta)\ (\texttt{cb}\ \texttt{V}_\texttt{d}\ \beta) \qquad \texttt{iCDF}_\texttt{U} = \texttt{ub}\ \texttt{V}_\texttt{U} \qquad (\texttt{s}, \texttt{ts}, \eta) = \texttt{track}\ (\texttt{s}_\texttt{j}, \texttt{ts}_\texttt{j}, \eta_\texttt{j})\end{array}}{\texttt{add}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}] \rightsquigarrow \mathcal{S}[\texttt{ub}\ [\texttt{V}_\texttt{L}, \texttt{V}_\texttt{G}, \texttt{V}_\texttt{U}], \texttt{s}, \texttt{ts}, \eta]}$$

Fig. 26. Accuracy analysis for mixed mechanisms.

resulting value of the partial sum for values in $\texttt{V}_\texttt{L}$ and $\texttt{V}_\texttt{G}$ will have an error estimation computed as the minimum between union and Chernoff bound—that is, $\texttt{iCDF}_\texttt{d} = \lambda\beta \to \texttt{min}\ (\texttt{ub}\ \texttt{V}_\texttt{d}\ \beta)\ (\texttt{cb}\ \texttt{V}_\texttt{d}\ \beta)$ for $\texttt{d} \in \{\texttt{L}, \texttt{G}\}$. Since values grouped in $\texttt{V}_\texttt{U}$ do not have an associated Chernoff bound, the iCDF of their addition will be determined by union bound $\texttt{iCDF}_\texttt{U} = \texttt{ub}\ \texttt{V}_\texttt{U}$. Last, the iCDF of the addition of all values $\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_n$ is computed as the union bound between the values from $\texttt{V}_\texttt{L}$, $\texttt{V}_\texttt{G}$, and $\texttt{V}_\texttt{U}$. This computation is done in such a way that if all values $\texttt{v}_j$ come from the same distribution $\texttt{d}$, the final iCDF will be the same as presented in Figure 10 for Laplace and Figure 12(b) for Gauss.

To determine the values of s, $\texttt{ts}$, and η, function $\texttt{track}$ checks if all values are untainted ($\forall j.\texttt{ts}_j \neq \emptyset$), independent ($\bigcap_{j=1\ldots n} \texttt{ts}_j = \emptyset$), and Gaussian ($\forall j.\eta_j = \texttt{G}$), and if that is the case, then $s = \sum_{j=1}^{n} s_j$, $\texttt{ts} = \bigcup_{j=1\ldots n} \texttt{ts}_j$, and $\eta = \texttt{G}$ as done previously with the Gaussian mechanism; otherwise, $s = \texttt{0}$, $\texttt{ts} = \emptyset$, and $\eta = \texttt{U}$.

Evidently, combinator add's new behavior will likely yield tighter bounds compared to our previous version since now it procures triggering the Chernoff bound in intermediate stages. One can imagine other optimizations over the interpretation of add to postpone the tainting of values as much as possible. For example, nested additions such as $\texttt{add}\ [\texttt{add}\ [\texttt{v}_1, \texttt{v}_2], \texttt{v}_3, \texttt{v}_4, \texttt{v}_5]$ could be flattened as $\texttt{add}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_5]$, potentially improving their sum's bound, especially when the noisy values come from the Laplace mechanism. We leave the integration of other optimizations as future work.

With this generalization, DPella is also extensible to other notions of privacy (e.g., Rényi-DP [42]) or other mechanisms by simply declaring a new data type, its principles of composition, and a way to sample noise (to compute `noiseScale`) and determine its iCDF (to compute `errorDist`). Additionally, if the Chernoff bound is not provided, DPella will use the union bound instead. All extensions to our framework are delimited and clearly identified thanks to our typed approach.

9 LIMITATIONS AND EXTENSIONS

So far, we have discussed the use of DPella as an API allowing a programmer to implement her own data analyses. However, we foresee DPella to also serve as a "glue" that enables a programmer to integrate arbitrary DP algorithms, as (black-box) building blocks while reasoning about accuracy. In this light, our design supports the introduction of new primitives when some analyses cannot be directly implemented because either the static analysis for accuracy provided by DPella is too conservative or DPella's API building blocks are not enough to express the desired analysis. In the following, we describe several possible such extensions.

The matrix mechanism

As we discussed in the previous section, in some situations DPella allows answering in an accurate way the multiple counting queries in a way that is similar to the matrix mechanism. As an example, DPella estimates accuracy better than the matrix mechanism for the strategy **I**—recall Section 6. However, for other workloads and other strategies, the accuracy provided by DPella is too conservative. To consider other workloads and strategies, the matrix mechanism can be incorporated into DPella as a primitive for answering counting queries. The requirements for this are that the return values are tainted, and that we have an iCDF for it—this can be calculated as in the work of Ge et al. [25]. In general, it is sound to add new primitives that permit a more precise accuracy analysis as long as the return values are tainted, and accuracy information is provided—thus effectively allowing further composition of the primitive with other analyses by means of the union bound.

Branching on noisy values

By construction, DPella programs are not allowed to branch on results produced by aggregations; this restriction has been enforced since computing the (α, β)-accuracy of such programs poses a challenge in terms of the complexity of their error estimation. More specifically, determining the accuracy of a program branching on a noisy value involves the computation of conditional probabilities (together with the notion of conditional independence); we have identified two main difficulties carried by the consideration of this measurement. First, it must account for both branches' error, thus, quickly loosening the bounds as the program's complexity increases. Second, it is challenging to define a general and compositional way of reasoning about the accuracy of the combination of such programs and their independence tracking. To overcome this limitation, we have proposed adding programs that rely on branching over noisy values (e.g., SVT) as black-box primitives in DPella; in the following, we elaborate on this approach.

Primitives with non-compositional privacy analyses

Several DP algorithms have a privacy analysis that does not follow directly by composition. Some well-known examples are report-noisy-max, the **exponential mechanism (EM)**, and the sparse-vector technique (see the work of Dwork and Roth [17] and Barthe et al. [9] for more details). In their natural implementations, these algorithms branch on the result of some noised query's result, and the privacy analyses use some properties of the noise distributions that are not directly expressible in terms of composition of differentially private components. Because DPella's API does not allow branching on the results of noised queries, and because the privacy analyses that DPella supports are based on composition, we cannot implement these analyses directly using the DPella API. However, we can provide them as (black-box) primitives. We already discussed how to integrate report-noisy-max through a primitive `dpMax` (Figure 4). The EM can be incorporated into DPella in a similar way. A subtlety that one has to consider is the fact that the privacy guarantee of EM depends on a bound of the sensitivity of the score function. We handle this by requiring the score function's output to be bound between 0 and 1, bounding the sensitivity to be at most 1. As with `dpMax`, the output of the EM is tainted. The EM is an important mechanism that allows implementation of many other techniques. In particular, we can use the EM to implement the offline version of the sparse vector technique, as discussed in the work of Dwork and Roth [17]. These components allow DPella to support automated reasoning about accuracy for complex algorithms such as the offline version of the MWEM algorithm [27] following an analysis similar to the one discussed in the work of Barthe et al. [8].

Generally, one of the biggest challenges of introducing black-box primitives into DPella is to determine their (α, β)-accuracy. There is a significant disparity in how the accuracy of well-known differentially private routines is determined in the literature; thus, there is no straightforward approach to translate such results into our error measurement—if possible at all. Recent work from Barthe et al. [3] provides a uniform definition of accuracy across differentially private routines; we consider these results as a promising starting point to systematically interpret these algorithms' accuracy calculations into DPella's error measurement.

Online adaptive algorithms

Several DP algorithms have different implementations depending if they work offline, where all of the decisions are made upfront before running the program, or online, where some of the decisions are made while running the program. Online algorithms usually have a more involved control flow that depends on information that is available at runtime. As an example, the online version of the sparse vector technique uses the result of a DP query to decide whether or not to stop the computation (or whether or not to stop giving meaningful answers). These kinds of algorithms usually are based on some re-use of a noised result that corresponds to a tainted value in DPella. Therefore, the current design of DPella cannot support them. As future work, we plan to explore how to integrate these algorithms in DPella.

Improving accuracy through post-processing

Several works have explored the use of post-processing techniques to improve accuracy (e.g., [28, 30, 47]). Most of these works use accuracy measures that differ from the one we consider here and use some specific properties of the particular problem at hand. As an example, the work by Hay et al. [30] describes how to boost accuracy in terms of MSE for DP hierarchical queries by post-processing the DP results by means of some relatively simple optimization. This improvement in accuracy relies on, among other things, the impact that the optimization has on the MSE, which does not directly apply to the α-β notion of accuracy we use here. We expect that, also for the notion of α-β accuracy we use, it is possible to use post-processing for improving accuracy. However, we leave this for future work. Moreover, the reason for us to choose α-β accuracy as the principal notion of accuracy in DPella is because of its compositional nature expressible through the use of probability bounds. It is an interesting future direction to design a similar compositional theory also for other accuracy notions such as MSE. We expect DPella to be extensible to incorporate such a theory, once it is available.

10 RELATED WORK

Programming frameworks for DP

PINQ [40] uses dynamic tracking and sensitivity information to guarantee privacy of computations. Among the frameworks and tools sharing features with PINQ, we highlight Airavat [51], wPINQ [49], DJoin [45], Ektelo [63], Flex [31], and PrivateSQL [32]. In contrast to DPella, none of these works keeps track of accuracy, nor static analysis for privacy or accuracy. As discussed in Section 4, DPella supports a limited form of joins, and it is still able to provide accuracy estimates. We leave as future work supporting more general join operations through techniques similar to the ones proposed in Flex and PrivateSQL. Although several of the components from the frameworks discussed earlier are not supported in the current implementation of DPella, these can be added as black-box primitives, as discussed in Section 9. All of the programming frameworks discussed previously support reasoning about privacy for complex data analyses while neglecting accuracy, whereas DPella supports accuracy but restricts the programming framework to rule out certain analysis (e.g., adaptive ones) for which we do not yet have a general compositional theory.

Tools for DP

In a way similar to DPella, there exist tools that support reasoning about accuracy and restrict the kind of data analyses they support. GUPT [44] is a tool based on the sample-and-aggregate framework for DP [48]. GUPT allows analysts to specify the target accuracy of the output and compute privacy from it—and vice versa. This approach has inspired several of the subsequent works and also our design. The limitations of GUPT are that it supports only analyses that fit in the sample-and-aggregate framework, and it supports only confidence interval estimates expressed at the level of individual queries. In contrast, DPella supports analyses of a more general class, such as the ones we discussed in Section 3 and Section 6, and it also allows reasoning about the accuracy of combined queries rather that just about the individual ones. PSI [24] offers the data analyst an interface for selecting either the level of accuracy that she wants to reach or the level of privacy she wants to impose. The error estimates that PSI provides are similar to the ones that are supported in DPella. However, similarly to GUPT, PSI supports only a limited set of transformations and primitives, it supports only confidence intervals at the level of individual queries, and in its current form it does not allow analysts to submit their own (programmed) queries.

APEx [25] has similar goals as DPella and allows data analysts to write queries as SQL-like statements. However, the model that APEx uses is different from that of DPella. It supports three kind of queries: WCQ (counting queries), ICQ (iceberg counting queries), and TCQ (top-k counting queries). To answer WCQ queries, APEx uses the matrix mechanism (recall Section 6) and applies Monte Carlo simulations to achieve accuracy bounds in terms of α and β, and to determine the least privacy parameter (ϵ) that fits those bounds. We have shown how DPella can be used to answer queries based on the identity strategies using partition and concentration bounds. To effectively answer different workloads and strategies as well as ICQ and TCQ queries, we would need to extend DPella with the matrix mechanism as a black box (recall Section 9). Although APEx supports advanced query strategies, it does not provide the means to reason about combinations of analyses—for example, it does not support reasoning about the accuracy of a query using results from WCQ queries to perform TCQ ones. DPella instead has been designed specifically to support the combination of different queries. As we discussed in Section 9, DPella can be seen as a programming environment that could be combined with some of the analyses supported by tools similar to PSI, GUPT, or APEx to reason about the accuracy of the combined queries.

Formal Calculi for DP

There are several works on enforcing DP relying on different models and techniques. Within this group are Fuzz [50]—a programming language that enforces (pure) DP of computations using a linear type of system that keeps track of program sensitivity—and its derivatives DFuzz [23], Adaptive Fuzz [60], Fuzzi [64], and Duet [46]. Hoare2 [7], a programming language that enforces (pure or approximate) DP using program verification, together with its extension PrivInfer [4] supporting differentially private Bayesian programming, and other systems using similar ideas [1, 9, 59, 62].

Barthe et al. [6] devise a method for proving DP using Hoare logic. Their method uses accuracy bounds for the Laplace mechanism for proving privacy bounds of the Propose-Test-Release mechanism but cannot be used to prove accuracy bounds of arbitrary computations. Later, Barthe et al. [8] develop a Hoare-style logic, named *aHL*, internalizing the use of the union bound for reasoning about probabilistic imperative programs. The authors show how to use aHL for reasoning in a mechanized way about accuracy bounds of several basic techniques such as report-noisy-max, sparse vector, and MWEM. This work has largely inspired our design of DPella but with several differences. First, aHL mixes the reasoning about accuracy with the more classical Hoare-style reasoning. This choice makes aHL very expressive but difficult to automate. DPella instead favors automation over expressivity. As discussed before, the use of DPella to derive accuracy bound is transparent to a programmer thanks to its automation. However, there are mechanisms that can be analyzed using aHL and cannot be analyzed using DPella, such as adaptive online algorithms. Second, aHL supports only reasoning about accuracy but does not support reasoning about privacy. This makes it difficult to use aHL for reasoning about the privacy-accuracy trade-offs. Finally, aHL supports only reasoning using the union bound and does not support reasoning based on the Chernoff bound. This makes DPella more precise on the algorithms that can be analyzed using the Chernoff bound. Barthe et al. [5] use aHL, in combination with a logic supporting reasoning by coupling, to verify differentially private algorithms whose privacy guarantee depends on the accuracy guarantee of some sub-component. We leave exploring this direction for future work. More recently, Smith et al. [56] propose an automated approach for computing accuracy bounds of probabilistic imperative programs. This work shares some similarities with ours. However, it does not support reasoning about privacy, and it only uses the union bound and does not attempt to reason about probabilistic independence to obtain tighter bounds.

Other works

In a recent work, Ligett et al. [36] propose a framework for developing differentially private algorithms under accuracy constraints. This allows one to choose a given level of accuracy first and then find the private algorithm meeting this accuracy. This framework is so far limited to empirical risk minimization problems and is not yet supported by a system.

11 CONCLUSION

DPella is a programming framework for reasoning about privacy, accuracy, and their trade-offs. DPella uses taint analysis to detect probabilistic independence and derive tighter accuracy bounds using Chernoff bounds. We believe that the principles behind DPella (i.e., the use of concentration bounds guided by taint analysis) could generalize for more notions of privacy, such as Rényi-DP [42], concentrated DP [18], zero concentrated DP [12], and truncated concentrated DP [11] (as is done with (ϵ, δ)-DP). As future work, we envision lifting the restriction that programs should not branch on query outputs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for constructive feedback on an earlier version of this work. We would like to thank Gilles Barthe for early feedback on the development of DPella.

REFERENCES

[1] Aws Albarghouthi and Justin Hsu. 2018. Synthesizing coupling proofs of differential privacy. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), Article 58.

[2] Borja Balle and Yu-Xiang Wang. 2018. Improving the Gaussian mechanism for differential privacy: Analytical calibration and optimal denoising. arXiv:1805.06530

[3] Gilles Barthe, Rohit Chadha, Paul Krogmeier, A. Prasad Sistla, and Mahesh Viswanathan. 2021. Deciding accuracy of differential privacy schemes. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–30.

[4] Gilles Barthe, Gian Pietro Farina, Marco Gaboardi, Emilio Jesús Gallego Arias, Andy Gordon, Justin Hsu, and Pierre-Yves Strub. 2016. Differentially private Bayesian programming. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.*

[5] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Advanced probabilistic couplings for differential privacy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.*

[6] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving differential privacy in Hoare logic. In *Proceedings of the IEEE Computer Security Foundations Symposium.*

[7] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *Proceedings of the 42nd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15).* ACM, New York, NY.

[8] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A program logic for union bounds. In *Proceedings of the 43rd International Colloquium on Automata, Languages, and Programming (ICALP'16)*.

[9] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving differential privacy via probabilistic couplings. In *Proceedings of the ACM/IEEE Symposium on Logic in Computer Science*.

[10] Jeremiah Blocki, Avrim Blum, Anupam Datta, and Or Sheffet. 2013. Differentially private data analysis of social networks via restricted sensitivity. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science (ITCS'13)*.

[11] Mark Bun, Cynthia Dwork, Guy N. Rothblum, and Thomas Steinke. 2018. Composable and versatile privacy via truncated CDP. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, New York, NY, 74–86.

[12] Mark Bun and Thomas Steinke. 2016. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Proceedings of the Theory of Cryptography Conference*.

[13] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. 2011. Private and continual release of statistics. *ACM Transactions on Information and System Security* 14, 3 (2011), 26.

[14] Graham Cormode, Tejas Kulkarni, and Divesh Srivastava. 2018. Marginal release under local differential privacy. In *Proceedings of the International Conference on Management of Data (SIGMOD'18)*. 131–146.

[15] Devdatt P. Dubhashi and Alessandro Panconesi. 2009. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press.

[16] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the 3rd Conference on Theory of Cryptography (TCC'06)*. 265–284.

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

[18] Cynthia Dwork and Guy N. Rothblum. 2016. Concentrated differential privacy. arXiv:1603.01887

[19] Cynthia Dwork, Guy N. Rothblum, and Salil P. Vadhan. 2010. Boosting and differential privacy. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS'10)*. 51–60.

[20] Hamid Ebadi and David Sands. 2017. Featherweight PINQ. *Journal of Privacy and Confidentiality* 7, 2 (2017), 159–164.

[21] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[22] Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Zhiwei Steven Wu. 2014. Dual query: Practical private query release for high dimensional data. In *Proceedings of the International Conference on Machine Learning (ICML'14)*.

[23] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[24] Marco Gaboardi, James Honaker, Gary King, Kobbi Nissim, Jonathan Ullman, and Salil P. Vadhan. 2016. PSI (Ψ): A private data sharing interface. arXiv:1609.04340

[25] Chang Ge, Xi He, Ihab F. Ilyas, and Ashwin Machanavajjhala. 2019. APEx: Accuracy-aware differentially private data exploration. In *Proceedings of the International Conference on Management of Data*.

[26] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential privacy under fire. In *Proceedings of the USENIX Security Symposium*.

[27] Moritz Hardt, Katrina Ligett, and Frank McSherry. 2012. A simple and practical algorithm for differentially private data release. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems*.

[28] Moritz Hardt and Kunal Talwar. 2010. On the geometry of differential privacy. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC'10)*.

[29] Michael Hay, Ashwin Machanavajjhala, Gerome Miklau, Yan Chen, and Dan Zhang. 2016. Principled evaluation of differentially private algorithms using DPBench. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*.

[30] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. 2010. Boosting the accuracy of differentially private histograms through consistency. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1021–1032.

[31] Noah M. Johnson, Joseph P. Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (2018), 526–539.

[32] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: A differentially private SQL query engine. *Proceedings of the VLDB Endowment* 12, 11 (July 2019), 1371–1384.

[33] Christoph H. Lampert, Liva Ralaivola, and Alexander Zimin. 2018. Dependency-dependent bounds for sums of dependent random variables. arXiv:1811.01404

[34] Chao Li, Gerome Miklau, Michael Hay, Andrew McGregor, and Vibhor Rastogi. 2015. The matrix mechanism: Optimizing linear counting queries under differential privacy. *VLDB Journal* 24, 6 (2015), 757–781.

[35] P. Li and S. Zdancewic. 2010. Arrows for secure information flow. *Theoretical Computer Science* 411, 19 (2010), 1974–1994.

[36] Katrina Ligett, Seth Neel, Aaron Roth, Bo Waggoner, and Zhiwei Steven Wu. 2017. Accuracy first: Selecting a differential privacy level for accuracy-constrained ERM. arXiv:1705.10829

[37] E. Lobo-Vesga, A. Russo, and M. Gaboardi. 2020. A programming framework for differential privacy with accuracy concentration bounds. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP'20)*. IEEE, Los Alamitos, CA, 1333–1350. DOI: https://doi.org/10.1109/SP40000.2020.00086

[38] Ashwin Machanavajjhala, Daniel Kifer, John M. Abowd, Johannes Gehrke, and Lars Vilhuber. 2008. Privacy: Theory meets practice on the Map. In *Proceedings of the International Conference on Data Engineering (ICDE'08)*.

[39] Frank McSherry and Ratul Mahajan. 2011. Differentially-private network trace analysis. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 123–134.

[40] Frank D. McSherry. 2009. 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)*. ACM, New York, NY. 19–30.

[41] Darakhshan J. Mir, Sibren Isaacman, Ramón Cáceres, Margaret Martonosi, and Rebecca N. Wright. 2013. DP-WHERE: Differentially private modeling of human mobility. In *Proceedings of the IEEE International Conference on Big Data*.

[42] Ilya Mironov. 2017. Rényi differential privacy. In *Proceedings of the 2017 IEEE 30th Computer Security Foundations Symposium (CSF'17)*. IEEE, Los Alamitos, CA.

[43] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92.

[44] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David E. Culler. 2012. GUPT: Privacy preserving data analysis made easy. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*.

[45] Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially private join queries over distributed databases. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*.

[46] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, et al. 2019. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), Article 172.

[47] Aleksandar Nikolov, Kunal Talwar, and Li Zhang. 2013. The geometry of differential privacy: The sparse and approximate cases. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC'13)*.

[48] Kobbi Nissim, Sofya Raskhodnikova, and Adam D. Smith. 2007. Smooth sensitivity and sampling in private data analysis. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC'07)*.

[49] Davide Proserpio, Sharon Goldberg, and Frank McSherry. 2014. Calibrating data to sensitivity in private data analysis: A platform for differentially-private analysis of weighted datasets. *Proceedings of the VLDB Endowment* 7, 8 (2014), 637–648.

[50] Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*.

[51] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and privacy for MapReduce. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*.

[52] Alejandro Russo. 2015. Functional pearl: Two can keep a secret, if one of them uses Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, NY.

[53] A. Russo, K. Claessen, and J. Hughes. 2008. A library for light-weight information-flow security in Haskell. In *Proceedings of the ACM SIGPLAN Symposium on Haskell*. ACM, New York, NY.

[54] A. Sabelfeld and A. C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19.

[55] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. 2016. Explicit secrecy: A policy for taint tracking. In *Proceedings of the IEEE European Symposium on Security and Privacy*. 15–30.

[56] Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace abstraction modulo probability. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), Article 39.

[57] David Terei, Simon Marlow, Simon L. Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell*. 137–148.

[58] Justin Thaler, Jonathan Ullman, and Salil P. Vadhan. 2012. Faster algorithms for privately releasing marginals. In *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming (ICALP'12)*. 810–821.

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

[60] Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A framework for adaptive differential privacy. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), Article 10.

[61] Xiaokui Xiao, Guozhang Wang, and Johannes Gehrke. 2011. Differential privacy via wavelet transforms. *IEEE Transactions on Knowledge and Data Engineering* 23, 8 (2011), 1200–1214.

[62] Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards automating differential privacy proofs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*.

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

[64] Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019. Fuzzi: A three-level logic for differential privacy. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'19)*.

Received July 2020; revised February 2021; accepted February 2021