Thesis for the Degree of Licentiate of Philosophy

A Programming Language for Data Privacy with Accuracy Estimations

Elisabet Lobo Vesga

Department of Computer Science and Engineering
Chalmers University of Technology

Göteborg, Sweden 2020

A Programming Language for Data Privacy with Accuracy Estimations
Elisabet Lobo Vesga

Technical Report 214L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Information Security

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2020

Abstract

Differential privacy offers a formal framework for reasoning about 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 analyses results, inferring useful properties about the population. The compositional nature of differential privacy has motivated the design and implementation of several programming languages aimed at helping a data analyst in programming differentially private analyses. However, most of the programming languages for differential privacy proposed so far provide support for reasoning about privacy but not for reasoning about the accuracy of data analyses. To overcome this limitation, in this work 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 which statically tracks the accuracy of different data analyses. In order to make tighter 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 figure out the best manner to calibrate privacy to meet the accuracy requirements.

Keywords: accuracy, concentration bounds, differential privacy, functional programming, databases, haskell

Acknowledgments

First, I would like to express my sincere appreciation to Alejandro Russo, my supervisor, for his patient guidance, inspiring enthusiasm, and constant encouragement. I would also like to thank my co-supervisor and collaborator, Marco Gaboardi, for his detailed lessons on probability and invaluable mentorship.

My thanks are extended to my friends and colleagues for their fellowship, the easy laughs, and insightful conversations. It is an honor to share this experience with all of you.

I wish to acknowledge the support, prayers, and great love of my family who set me off on this journey a long time ago. Last but not least, a special recognition goes to my partner Alejandro Gómez for being my rock from the very beginning. I will always be grateful to have you by my side.

Contents

Introduction

Motivation

Differential privacy (DP) [17] is emerging as a viable solution to release statistical information about the population without compromising data subjects' privacy. A standard way to achieve DP is adding some statistical noise to the result of a data analysis. If the noise is carefully calibrated, it provides a *privacy* protection for the individuals contributing their data, and 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 in the definition of DP, rather it is a constraint that is made explicit for a specific task at hand when a differentially private data analysis is designed.

An important property of DP is *composeability*: 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) to assign to each of her analyses. The compositionality aspects of DP motivated the design of several programming frameworks [38, 49, 48, 25, 21, 4, 5, 61, 58, 31, 60, 62] and tools [36, 42, 39, 23] with built-in basic data analyses to help analysts to 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: use some primitives for basic tasks in DP as building blocks, and use composition properties to combine these building blocks making sure that the privacy cost of each data analysis sum up and that the total cost does not exceed the privacy budget. Programming frameworks such as [38, 49, 48, 25, 21, 4, 5, 61, 58, 31, 60, 62] also provide general support to further combine, through programming techniques, the different building blocks and the results of the different data analyses. Differently,

tools such as [36, 42, 39, 23] are optimized for specific tasks at the price of restricting the kinds of data analyses they can support.

Unfortunately, this simple approach for privacy cannot be directly applied to accuracy. Reasoning about accuracy is *less compositional* than reasoning about privacy, and 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 differential privacy literature come with theoretical confidence intervals or error bounds that can be exposed to data analysts in order to allow them to take informed decisions about the consults they want to run. This approach has been integrated in tools such as GUPT [42], PSI [23], and Apex [24]. Users of these tools, can specify the target confidence interval they want to achieve, and the tools adjust accordingly the privacy parameters, when sufficient budget is available[1].

In contrast, all the programming frameworks proposed so far [38, 49, 48, 25, 21, 4, 5, 61, 58, 31, 60, 62] 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, with respect to privacy analyses, when designing arbitrary data analyses that users of these frameworks may want to program and run. In this work, we address this limitation by building a programming framework for designing differentially private analysis which also supports a compositional form of reasoning about accuracy.

Background

We start by providing a brief background on the notions of privacy and accuracy DPella considers. Differential privacy [17] 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

[1] Apex actually goes beyond this by also helping user by selecting the right differentially private mechanism to achieve the required accuracy.

define differential privacy as a property of a randomized query $\tilde{Q}(\cdot)$ representing the data analysis, as follow.

Definition 1. Differential Privacy (DP)[17] *A randomized query* $\tilde{Q}(\cdot) : \mathrm{db} \to \mathbb{R}$ *satisfies* ϵ-differential privacy *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 definition above, 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 farther 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-differential privacy is adding some carefully calibrated noise to the result of a query. To protect all 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. [17] *The* (global) sensitivity *of a query* $Q(\cdot) : \mathrm{db} \to \mathbb{R}$ *is the quantity* $\Delta_Q = \max\{|Q(D_1) - Q(D_2)| \text{ for } D_1, D_2 \text{ differing in one row}$

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

Theorem 1. Laplace Mechanism [17] *Let* $Q(\cdot) : \mathrm{db} \to \mathbb{R}$ *be a deterministic query with sensitivity* Δ_Q*. Let* $\tilde{Q}(\cdot) : \mathrm{db} \to \mathbb{R}$ *be a randomized query defined as* $\tilde{Q}(D) = Q(D) + \eta$*, where* η *is sample 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 theorem above, for a given query, the smaller the ϵ is, the more noise $\tilde{Q}(\cdot)$ needs to inject in order 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 3. Accuracy, see e.g.[15] *Given an ϵ-differentiallly 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 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: with probability at least $1 - \beta$ the query $\tilde{Q}(D)$ diverge 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 previously discussed, an important property of differential privacy is composeability.

Theorem 2. Sequential Composition [17] *Let $\tilde{Q}_1(\cdot)$ and $\tilde{Q}_2(\cdot)$ be two queries which 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 3. Parallel Composition [38] *Let $\tilde{Q}(\cdot)$ be a ϵ-differentially private query. and $\mathrm{data}_1, \mathrm{data}_2$ be a partition of the set of data. Then, the query $\tilde{Q}_1(D) = (\tilde{Q}(D \cap \mathrm{data}_1), \tilde{Q}(D \cap \mathrm{data}_2))$ is ϵ-differentially private.* Thanks to the composition properties of differential privacy, 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.

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 subqueries which 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 [37]) show how DPella helps to quantify what "more" and "less" means.

Contribution

The main contribution of this thesis is showing how programming frameworks can internalize the use of *probabilistic bounds* [14] for composing different confidence intervals or error bounds, in an automated way. Probabilistic bounds are part of the classical toolbox for the analysis of randomized algorithms, and are the tools that differential privacy algorithms designers usually employ for the accuracy analysis of classical mechanisms [15, 18]. Two important probabilistic bounds are the *union bound*, that 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 3). 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, and in particular of programs implementing differentially private primitives.

Building on this idea, this thesis proposes a programming framework where this kind of reasoning is automated, and can be combined with reasoning about privacy. The aim of such framework is to offer programmers the tools that they need for implementing 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 use of the Chernoff bound. 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 [52] (in the form of taint analysis [53]). While these probabilistic bounds are not enough to express every accuracy guarantee one wants to express for arbitrary data analyses, they allow the analysis of a large class of user-designed programs. The approach to be presented in this thesis allows programmers to exploit the compositional nature of both privacy and utility, comple-

menting in this way the support provided by tools such as GUPT [42], PSI [23], which yield confidence intervals estimate only at the level of individual queries, and by Apex [24], which issue 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 as well as 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 use of both Laplace and Gaussian noise, and the use of sequential or advanced [15] composition, respectively, together with parallel composition for both notions. For clarity, this thesis will mainly focus on ϵ-DP and will present the use of the Laplace mechanism, however other variants will be discussed briefly (see Section 3.3). DPella is implemented as a library in the general purpose language Haskell; a programming language that is well-known to easily support information-flow analyses [34, 50]. Furthermore, DPella is designed to be *extensible* through the addition of new primitives (see Section 6).

To reason about privacy and accuracy, DPella provides two primitives responsible to symbolically interpret programs (which implement data analyses). DPella's symbolic interpretation for privacy consists on decreasing the privacy budget of a query by deducing the required budget of its sub-parts. On the other hand, 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. The iCDF of a query is build out of the iCDFs of the different components, by using as a basic composition principle the *union bound*. These interpretations provide overestimates of the corresponding quantities that they track. In order to make these estimates as precise as possible, DPella uses *taint analysis* to track the use of noise to identify which variables are *statistically independent*. This information is used by DPella to *soundly* replace, when needed, the union bound with the *Chernoff bound*, something that to the best of our knowl-

edge other program logics or program analyses also focusing on accuracy, such as [8] and [54], do not consider. We envision DPella's accuracy estimations to be used in scenarios which align with those considered by tools like GUPT, PSI, and Apex.

In summary, this thesis contributions are:

- Present DPella, a programming framework that allows data analysts to reason compositionaly about privacy-accuracy trade-off.
- 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.
- Inspect DPella's expressiveness and error estimations by implementing PINQ-like queries from previous work [37, 38, 3] and workloads from the matrix mechanism [33, 28, 59].

Thesis structure

This thesis is an extended version of the work "A Programming Framework for Differential Privacy with Accuracy Concentration Bounds" to be published in *The 41st IEEE Symposium on Security and Privacy (S&P 2020), San Francisco, USA, May 2020.* The referred paper was created in collaboration with Alejandro Russo and Marco Gaboardi.

This document is structured as follows. Section 1 introduces DPella by showcasing its main features though simple examples. Section 2 presents each of DPella's primitives for the construction and execution of queries. Section 3 explains how do we calculate accuracy concentration bounds and the accuracy-aware primitives that can be used by the data analysts. On Section 4 we implement case studies from the literature revealing DPella's advantages and limitations. Section 5 introduces a new primitive that allows data analysts to test DPella's accuracy estimations. Following, on Section 6 we discuss DPella's limitations in detail together with possible extensions to the framework. Lastly, Section 7 puts DPella in context while contrasting it with other approaches and frameworks.

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.

1

DPella by example

DPella's model considers two kind of actors: *data curators*, owners of the private dataset that 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 database, instead, they need to implement their analyses and send them to the curator which will execute them and give the results back.

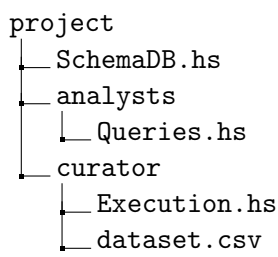

Figure 1: File structure

From an implementation standpoint, it 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 `SchemaDB.hs` contains the schema of the database own by the curator, it does not contain private data, only the names of the tables and their respective attributes as a Haskell record type. For example, a database containing just one table called `Ages` with two attributes `name` (a `String` value) and `age` (an `Int` value), will be encoded in `SchemaDB.hs` as follows, where (::) is used to describe the type of a term in Haskell:

$$\textbf{data}\ \texttt{Ages} = \texttt{AgeRow}\ \{\texttt{name} :: \texttt{String}, \texttt{age} :: \texttt{Int}\}$$

Since the structure of the database is not considered sensitive information, `SchemaDB.hs` can be accessed by both, the data owners 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. Lastly, file `Execution.hs` implements the run functions for the analyses at `Queries.hs`, this file is own 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`.

1.1 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
                      }
```

1.1.1 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 SchemaDB
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 `SchemaDB` 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 could 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 that we have called `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 execute analysts' queries, such a function will look like:

```
import SchemaDB
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 auxiliar function `loadDS` (provided by DPella's interface for curators) to read file `hotspot.csv`

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

which is then saved as a DPella's dataset in variable `ds`. Next, it uses DPella's primitive `dpEval` indicating which analysis will be perform, over which dataset, and what's the tolerance for the privacy loss.

Let's assume `hotspot.csv` have the information of $10,000$ packets and 7 of them where directed to wikileaks' IP address. Then, when the data owner executes the analysis she would get results such as:

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

Which 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.

1.1.2 Sums

Suppose we are now interested in computing the amount of transmitted data. This is, 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 determine first the range of the values—our framework supports only natural numbers' ranges, e.g., $[1, 10]$, $[5, 10]$, etc. This information is needed to automatically calculate the sensitivity of sum queries at compile time, i.e., if every value is in the range $[a, b]$, the sensitivity is $b - a$. The way to specify ranges in DPella is via the primitive range.

```
range :: (Nat a, Nat b, Nat (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 `Nat n`). To create ranges, we need to use type applications, e.g.,

```
range1 = range @1 @10
range2 = range @5 @10
```

For our specific case, the data curator indicates that the range of the size of packets go from 40 to $35,000$ bytes, 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 35000 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`, 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 of 1465 bytes. Hence, we will need to update our query accordingly

```
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 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 out for miscalculations on the consumption of the privacy budget, and more.

1.2 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 $\text{CDF}(x) =$ number of records with value $\leqslant x$. Hence, we are just interested in the values of the attribute `size`.

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

(a) Sequential approach

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

(b) Parallel approach

Figure 2: CDF's implementations

McSherry and Mahajan [37] 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.

1.2.1 Sequential `CDF`

A simple approach to compute the CDF consists in splitting the range of lengths into `bins` and, for each `bin`, count 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 2a. We define a function `cdf1` which 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 packets' length. `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 7 [3]). The function `sequence` then takes the list of queries and compute them sequentially 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 6 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`-differential privacy, we distributed the given budget `eps` evenly among the sub-queries (this is done in lines 4 and 7). However, a data analyst may forget to do so, e.g., 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 2) 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 a its modified version `cdf1'` with `localEps` = `eps`. Suppose that we want to compute how much budget will be consumed by running it on a list of bins of size 10 (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 2), that, 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

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

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 above, and we want to have this estimate for $\beta = 0.05$ and $\beta = 0.2$, respectively. Then, the data analyst can ask this as follow:

```
>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.

1.2.2 Parallel `CDF`

Another way to compute a CDF is by first generating an 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 that we could use the parallel composition, rather than the sequential one [37], which we show how to implement in DPella in Figure 2b. —where double-dashes are used to introduce single-line comments.

In `cdf2`, we first select all the packages whose length is smaller than the maximum bin, and then we partition the data accordingly to the given list of bins. To do this, we use `dpPartRepeat` operator to create as many (disjoint) datasets as given `bins`, where each record in each partition belongs to the range determined by an specific `bin`—where the record 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 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 14–16), 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, 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
```

1.2.3 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, i.e., 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:

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

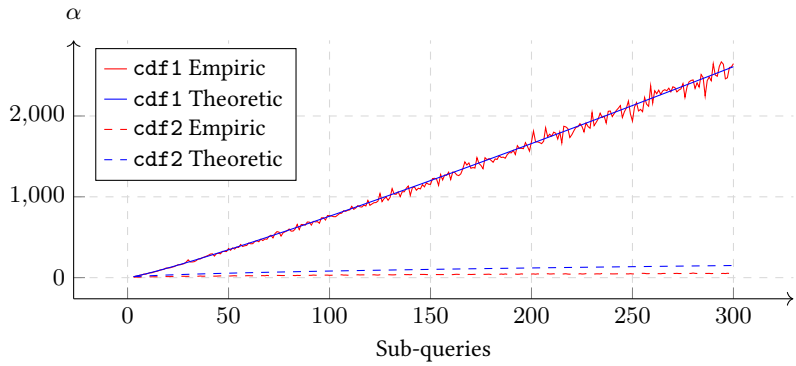

Figure 3: Error comparison (95% confidence)

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 the privacy parameter fixed and we change the number of bins, where `cdf2` gives a better error when the number of bins is large [37] 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 relative small (300) number of experimental runs we consider.

Now, what if the data analyst choose to use `cdf2` because of what we discussed before but she realizes that she can afford an error $\alpha \leqslant 50$; what would be then the epsilon that gives such α? One of the feature 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.

2

Privacy

DPella is designed to help data analysts to 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 to ensure differential privacy of queries written by data analysts.

2.1 Components of the API

Figure 4 shows part of 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 for Section 3 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 noticing 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`. While it might seem restrictive, it enables to write 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 6 discusses, however, how to lift this limitation for specific analyses.

Values of type `Data s r` represent sensitive datasets with *accumulated stability* `s`, where each row is of type `r`. Accumulated stability, on the other hand, is instantiated to type-level positive natural numbers,

```
  -- 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
```

Figure 4: DPella API: Part I

i.e., 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. In order 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 [19]. Different than, e.g., PINQ [38], 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 [41], 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 $(\gg\!=)$—called *bind*—is used to sequence queries and their associated side-effects. Specifically, `qp` $\gg\!=$ `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 $\texttt{qp}_1 \gg\!= (\lambda \texttt{x}_1 \to \texttt{qp}_2 \gg\!= (\lambda \texttt{x}_2 \to \texttt{return}\ (\texttt{x}_1, \texttt{x}_2)))$, which performs queries $\texttt{qp}_1$ and $\texttt{qp}_2$ and returns their results in a pair, can be written as **do** $\texttt{x}_1 \leftarrow \texttt{qp}_1; \texttt{x}_2 \leftarrow \texttt{qp}_2; \texttt{return}\ (\texttt{x}_1, \texttt{x}_2)$ which gives a more "imperative" feeling to programs. We split the API in four parts: transformations, aggregations, budget prediction, and execution of queries—see next section for the description of API's accuracy components. The first three parts are intended to be used by data analysts, while the last one is intended to be *only* used by data curators[4].

2.2 Transformations

The primitive `dpWhere` filters rows in datasets based on a predicate functions (`r` $\to$ `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 which 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 `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` $\to$ `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 $\Rightarrow$ are type constraints. They can be seen as static demands for the types appearing on the right-hand side of $\Rightarrow$. 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 [38, 19]—observe that `2*s` is a type-level multiplication done by a type-level function (or type family [20]) `*`. Our API also considers transformations similar to those found in SQL like intersection

[4] A separation that can be enforced via Haskell modules [55]

(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 [38, 43, 9, 30]. The output of a join may contain duplicates of sensitive rows, which makes 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 in order to obtain stability. For brevity, we skip its presentation and assume that all the considered information is contained by the rows of given datasets.

2.3 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 1, 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 to run the same query on a dataset's partitions but only paying for one of them—recall Theorem 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, which rows are of type Color and variable bins enumerates all the 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 errors could lead to break privacy as well as inconsistencies

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

Figure 5: DP-histograms by using `dpPart`

when estimating the required privacy budget. A correct implementation consists on executing `dpCount` on the corresponding partition as shown in the commented line 4.

To catch coding errors as the one shown above, DPella deploys an static information-flow control (IFC) analysis similar to that provided by MAC [51]. 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, i.e., they do not need to understand how it works. Analysts and curators only need to know that, when the IFC analysis raises an alarm, is due to a possibly access to non-disjoint datasets when using `dpPart`.

2.4 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 which is `eps`-differentially private from which a noisy result gets return as a term of type `Value Double`. The purpose of data type `Value a` is two fold: to encapsulate noisy values of type `a` originating from aggregations of data, and to store informa-

tion about its accuracy—intuitively, how "noisy" the value is (explained in Section 3). 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). More specifically, the Laplace mechanism used by DPella uses accumulated stability s to scale the noise, i.e., it consider b from Theorem 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 e.i., for the indicated `Range a b`, the sensitivity is computed as `b-a`. This is enforced by applying a clipping function (`r` $\rightarrow$ `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 in order to obtain a term-level representation when injecting noise. Finally, primitive `dpMax` implements report-noisy-max [15]. This query takes a list of possible responses (`Responses a` is a type synonym for `[a]`) and a function of type `r` $\rightarrow$ `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 that the accumulated stability of the dataset to be `1` in order to be sound [8]. DPella guarantees such requirement by typing: the type of the given dataset as argument is `Data 1 r`.

2.5 Privacy budget and execution of queries

The primitive `budget` statically computes how much privacy budget is required to run a query. It is worth notice 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 as `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's type system. This primitive returns a computation of type `IO a`, which in Haskell are computations responsible to perform side-effects—in this case, obtaining randomness from the system in order to implement the Laplace mechanism.

2.6 Implementation

DPella is implemented as a *deep embedded domain-specific language* (EDSL) 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]`) shows how to write a query that generates possibly non-disjoint datasets from `ds` :: `Data` s r based on different criteria for then performing a counting.

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

Variable `eps` is the epsilon to spend in each counting while `fs` :: $[r \rightarrow$ `Bool`$]$ is the criteria list. The high-order functions `flip`, `mapM`, and `(>=>)` are standard in Haskell and represent a function who 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.

3

Accuracy

DPella uses the data type `Value a` responsible to store 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 an static analysis capable to compute the accuracy of queries via the following function

$$\texttt{accuracy} :: \texttt{Query}\ (\texttt{Value}\ \texttt{a}) \to \beta \to \alpha$$

which 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 interpret all of its components in order 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 [53] in order to track if values were drawn from statistically independent distributions.

3.1 Accuracy calculations

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

Definition 3.1.1 (Accuracy for the Laplace mechanism). *Given a randomized query $\tilde{Q}(\cdot) : \mathrm{db} \to \mathbb{R}$ implemented with the Laplace mechanism as in Theorem 1, we have that*

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

```
  -- Accuracy analysis (data analyst)
accuracy :: Query (Value a) → β → α
  -- Norms (data analyst)
norm∞ :: [Value Double] → Value [Double]
norm2 :: [Value Double] → Value [Double]
norm1 :: [Value Double] → Value [Double]
rmsd  :: [Value Double] → Value [Double]
  -- Accuracy combinators (data analyst)
add   :: [Value Double] → Value Double
neg   :: Value Double → Value Double
```

Figure 6: DPella API: Part II

Recall that the Laplace mechanism used by DPella utilizes accumulated stability s to scale the noise, i.e., it consider b from Theorem 1 as $b = s \cdot \frac{\Delta_Q}{\epsilon}$. Consequently, DPella stores the iCDF $\lambda\beta \to \log(1/\beta) \cdot s \cdot \frac{\Delta_Q}{\epsilon}$ 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 2b. 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.

3.1.1 Norms

DPella exposes primitives to aggregate the magnitudes of several errors predictions into a *single* measure—a useful tool when dealing with vectors. Primitives $\texttt{norm}_\infty$, $\texttt{norm}_2$, and $\texttt{norm}_1$ 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 $\texttt{norm}_\infty$, but other norms are available for the taste, and preference, of data analysts.

3.1.2 Adding values

The primitive `add` aggregates values and, in order to compute accuracy of the addition, it tries to apply the Chernoff bound if all the values are

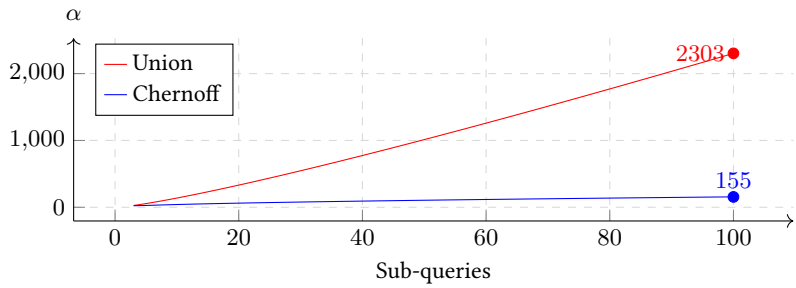

Figure 7: Union vs. Chernoff bounds

statistically independent; otherwise, it applies the union bound. More precisely, for the next definitions we assume that primitive `add` receives n terms $\mathtt{v}_1 :: \mathtt{Value}\ \mathtt{Double}$, $\mathtt{v}_2 :: \mathtt{Value}\ \mathtt{Double}$, ... , $\mathtt{v}_n :: \mathtt{Value}\ \mathtt{Double}$. 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 3.1.2 (`add` using union bound). *Given $n \geqslant 2$ random variables V_j with their respective $iCDF_j$, where $j \in 1 \ldots n$, and $\alpha_j = iCDF_j(\frac{\beta}{n})$, then the addition $Z = \sum_{j=1}^{n} V_j$ has the following accuracy:*

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

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 distribution*. To illustrate this point, Figure 7 shows that difference for the `cdf2` function we presented in Section 1 with $\epsilon = 0.5$ (for each DP sub-query) and $\beta = 0.1$. Clearly, the Chernoff bound is asymptotically much better when estimating accuracy, while 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 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 3.1.3 (`add` using Chernoff bound [12]). *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 $\nu > max\{\sqrt{\sum_{j=1}^{n} b_j^2}, b_M \sqrt{\ln \frac{2}{\beta}}\}$, then the addition $Z = \sum_{j=1}^{n} V_j$ has the following accuracy:*

$$\Pr[|Z| > \nu \cdot \sqrt{8 \ln \tfrac{2}{\beta}}] \leqslant \beta \tag{5}$$

DPella uses the value $\nu = max\{\sqrt{\sum_{j=1}^{n} b_j^2}, b_M \sqrt{\ln \frac{2}{\beta}}\} + 0.00001$ to satisfy the conditions of the definition above when applying the Chernoff bound—any other positive increment to the computed maximum works as well[5].

Lastly, to support subtraction, DPella provides primitive `neg` responsible to change the sign of a given value. We next explain how DPella checks that values come from statistically independent sampled variables.

3.1.3 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 (i.e., `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—a useful information when applying Definition 3.1.3. 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 bound (Definition 3.1.2) or the Chernoff bound (Definition 3.1.3). 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 hundred `dpCount` queries over different datasets, namely $\texttt{ds}_1$, $\texttt{ds}_2$, ..., $\texttt{ds}_{100}$. (The `...` denotes code intentionally left unspecified.) The code calls the primitive `add` with

[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])
```

Figure 8: Combination of sub-queries results

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 it be then the theoretical error of `totalCount`? The accuracy calculation depends on whether all the values are untainted in line 7. When no dependencies are detected between $\mathtt{v}_1$, $\mathtt{v}_2$, ..., v_{100}, namely all the values are untainted, DPella applies Chernoff bound in order to give a tighter error estimation. Instead, for instance, if $\mathtt{v}_3$ was computed as an aggregation of $\mathtt{v}_1$ and $\mathtt{v}_2$, e.g., **let** $\mathtt{v}_3 = \mathtt{add}\ [\mathtt{v}_1, \mathtt{v}_2]$, then line 7 applies union bound since $\mathtt{v}_3$ is a tainted value. With taint analysis, DPella is capable to detect dependencies among terms of type `Value Double`, and leverages that information to apply different concentrations bounds.

In the next Section, we proceed to formally define our accuracy analysis.

3.2 Implementation

The accuracy analysis consists on *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 a `iCDF` which is 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 primi-

tives like data transformations[6]. Figures 9 and 10 show the interesting parts of our analysis. Given a *well-typed* query $\texttt{q} :: \texttt{Query (Value a)}$, $\texttt{accuracy q} = \texttt{iCDF}$ where $\texttt{q} \triangleright \mathcal{S}[\texttt{iCDF}, s, \texttt{ts}]$ for some s and ts. The rules in 9 are mainly split into two cases: considering data aggregation queries and sequences of primitives glued together with $(\gg\!=)$.

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

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

(a) DP-queries

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

$$\textsc{Seq-Query}\quad \frac{\texttt{query} \triangleright \mathcal{S}[\texttt{iCDF}_\texttt{q}, s_\texttt{q}, \texttt{ts}_\texttt{q}] \qquad \texttt{k}\ (\mathcal{S}[\texttt{iCDF}_\texttt{q}, s_\texttt{q}, \texttt{ts}_\texttt{q}]) \rightsquigarrow^* \texttt{next} \qquad \texttt{next} \triangleright \mathcal{S}[\texttt{iCDF}, s, \texttt{ts}]}{\texttt{query} \gg\!= \texttt{k} \triangleright \mathcal{S}[\texttt{iCDF}, s, \texttt{ts}]}$$

(b) Sequential traversal

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

(c) Accuracy calculation when partitioning data

Figure 9: Accuracy analysis implemented by accuracy

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

The symbolic interpretation of `dpCount` is captured by rule DP-Count—see Figure 9a. This rule populates the `iCDF` of the return symbolic value with the corresponding error calculations for Laplace as presented in Definition 3.3.2 (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.

Rule DPMax shows the symbolic interpretation of `dpMax` whose `iCDF` aligns with the one appearing in [8]. Observe that the return value is tainted. The reason for that relies in the fact that the result, which is one of the responses in `res`, contains no noise—it is rather the process that lead to determining the winning response which has been "noisy." In this light, no scale of noise nor distribution can be associated to 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 if the first operation to interpret is a transformation or an aggregation, respectively—see Figure 9b. 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 EDSL nature of DPella makes Haskell's to 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, list comprehension, etc. 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

[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` $\mathcal{D}$.

symbolic dataset ($\texttt{m}\ \texttt{j}\ \mathcal{D}$) [8]. The symbolic values yield 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.

3.2.1 Concentration Bounds

Figure 10 shows the part of our analysis responsible to apply concentration bounds. Rules Union-Bound and Chernoff-Bound define pure functions (reduction $\rightsquigarrow$) which produce the concentration bounds as described in Definitions 3.1.2 and 3.1.3, respectively. We define the function `add` based on two cases. Rule Add-Union produces a symbolic value with a `iCDF` generated by the union bound ($\texttt{ub}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}]$). 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 do not yield a Laplace distribution. (However, the situation is different with Gaussians, see Section 3.3)

Union-Bound

$$\frac{\texttt{v}_\texttt{j} = \mathcal{S}[\texttt{iCDF}_\texttt{j}, \texttt{s}_\texttt{j}, \texttt{ts}_\texttt{j}] \qquad \alpha_j = \texttt{iCDF}_j(\frac{b}{n}) \qquad \texttt{iCDF} = \lambda\beta \rightarrow \sum_{j=1}^{n} \alpha_j}{\texttt{ub}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}] \rightsquigarrow \texttt{iCDF}}$$

Chernoff-Bound

$$\frac{\texttt{v}_\texttt{j} = \mathcal{S}[\texttt{iCDF}_\texttt{j}, \texttt{s}_\texttt{j}, \texttt{ts}_\texttt{j}] \qquad \texttt{v}_\texttt{M} = max\ \{s_\texttt{j}\}_{j=1\ldots n} \qquad \nu = max\{\sqrt{\textstyle\sum_{j=1}^{n} s_\texttt{j}^2}, \texttt{v}_\texttt{M}\sqrt{\ln \tfrac{2}{\beta}}\} + 0.0001 \qquad \texttt{iCDF} = \lambda\beta \rightarrow \nu \cdot \sqrt{8 \ln \tfrac{2}{\beta}}}{\texttt{cb}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}] \rightsquigarrow \texttt{iCDF}}$$

Add-Union

$$\frac{(\exists j \cdot \texttt{ts}_j = \emptyset) \vee \bigcap_{j=1\ldots n} \texttt{ts}_j \neq \emptyset}{\texttt{add}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}] \rightsquigarrow \mathcal{S}[\texttt{ub}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}], 0, \emptyset]}$$

Add-Chernoff-Union

$$\frac{\texttt{v}_\texttt{j} = \mathcal{S}[\texttt{iCDF}_\texttt{j}, \texttt{s}_\texttt{j}, \texttt{ts}_\texttt{j}] \qquad (\forall j \cdot \texttt{ts}_j \neq \emptyset) \qquad \bigcap_{j=1\ldots n} \texttt{ts}_j = \emptyset \qquad \texttt{iCDF} = \lambda\beta \rightarrow \texttt{min}\ (\texttt{ub}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}]\ \beta)\ (\texttt{cb}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}]\ \beta)}{\texttt{add}\ [\texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_\texttt{n}] \rightsquigarrow \mathcal{S}[\texttt{iCDF}, 0, \emptyset]}$$

Figure 10: Calculation of concentration bounds

[8] For simplicity, we assume that maps are implemented as functions

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

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

Figure 11: Calculation of norms

This rule gets exercised when either the list of symbolic values contains a tainted one ($\exists j \cdot \mathtt{ts}_j = \emptyset$) or have not been independently generated ($\bigcap_{k=1\ldots n} \mathtt{ts}_j \neq \emptyset$). Differently, Add-Chernoff-Union produces a symbolic value with a `iCDF` which 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 the values are untainted ($\forall j \cdot \mathtt{ts}_j \neq \emptyset$) and independently generated ($\bigcap_{j=1\ldots n} \mathtt{ts}_j = \emptyset$). At a first glance, one could believe that it would be enough to use the scale of the noise to track when values are untainted, i.e., 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 pre-processing function which, for instance, firstly partitions the arguments into subset of independently generated values, applies `add` to them (thus triggering Add-Chernoff-Union), and finally applies `add` to the obtained results (thus triggering Add-Union). The implementation of DPella enables to write such functions in a few lines of code.

3.2.2 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 which are tainted. The reason for that 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 an 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.

3.3 Accuracy of Gaussian mechanism

As aforementioned, DPella supports other notions of differential privacy—such as approximate differential privacy—together with the use of the Gaussian mechanism. Specifically, DPella supports a relaxation of the notion of differential privacy known as (ϵ, δ)-DP, formally defined as follow.

Definition 3.3.1 ((ϵ, δ)-Differential Privacy[17]). *A randomized query $\tilde{Q}(\cdot) : \mathrm{db} \to \mathbb{R}$ satisfies* (ϵ, δ)-differential privacy*, 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 Theorem 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, this is, for $Q : \mathrm{db} \to \mathbb{R}$ an arbitrary function with sensitivity Δ_Q (as described in Definition 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 on the following theorem.

Theorem 3.3.1 (Gaussian Mechanism [2]). *For any $\epsilon, \delta \in (0, 1)$, the Gaussian output perturbation mechanism with standard deviation $\sigma = \Delta_Q \sqrt{2 \log(\frac{1.25}{\delta})} / \epsilon$ is (ϵ, δ)-differentially private*

Similarly as with the Laplace mechanism, to provide bound estimates on the errors caused by the addition of Gaussian noise, DPella keeps track of Gauss' *inverse Cummulative Distribution Function* (iCDF). By following the general form of accuracy introduced in Definition 2, we have that:

DPCount

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

(a) Aggregations

Chernoff-Bound-Gauss

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

Add-Chernoff-Union

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

(b) Concentration bounds

Figure 12: Accuracy analysis for Gaussian mechanism

Definition 3.3.2 (Accuracy for the Gaussian mechanism). *Given a randomized query* $\tilde{Q}(\cdot) : \mathrm{db} \to \mathbb{R}$ *implemented with the Gaussian mechanism as previously described, then*

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

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

From an implementation standpoint, adding the Gaussian mechanism to our framework does not alter significantly the presented primitives, and, in particular, privacy's 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 12a for the case of `dpCount`. Additionally, Figure 12b 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

```
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])
```

Figure 13: Combining sub-queries under Gaussian mechanism

when adding independent values. In this case, we use the well-known fact 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 an useful feature when combining queries in batches, for instance, Figure 13 shows the query plan `totalCountG` that adds the results of hundred 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 9), then the second half (line 10), and finally returning the addition of the two halves (line 12). How will DPella calculate the theoretical error of `totalCountG`?

Observe that $\texttt{h}_1$ and $\texttt{h}_2$ are constructed as combinations of untainted values, meaning that when performing the additions at lines 7-8, the Chernoff bound could be triggered. More importantly, DPella still have information about their distribution. Furthermore, $\texttt{h}_1$ and $\texttt{h}_2$ 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, specially when the number of batches is big enough to trigger the use of the Chernoff bound.

4

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.

4.1 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, while the latter with APEx.

PINQ-like queries We have implemented most of PINQ's examples [38, 37], such as, different versions of CDFs (sequential, parallel, and hybrid) and network tracing-like analyses (such as determining the frequency a term or several terms have been searched by the users, and computing port's and packets' size distribution); additionally, we considered analyses of *cumulative sums* [3]—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, 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 follow closely 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 PINQ's ones 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

Category	Application	Programs
PINQ-like	CDFs [37]	`cdf1`, `cdf2`, `cdfSmart`
	Term frequency [38]	`queryFreq`, `queriesFreq`
	Network analysis [37]	`packetSize`, `portSize`
	Cumulative sums [3]	`cumulSum1` `cumulSum2` `cumulSumSmart`
Counting queries	Range queries via Identity, Histograms [28], and Wavelet [59]	`i_n` `h_n` `y_n`

Table 1: Implemented literature examples

specific implementation. This feature allows data analyst to reason about accuracy of the results—without actually executing the query—by varying i) the strategy of the implementation ii) the parameters of the query. For instance, in Section 1, we have shown 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 an heuristic algorithm.

Counting queries To compare our approach with the tool APEx [24], we consider range queries analyses—an specific subclass of counting queries. APEx uses the *matrix mechanism* [33] 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 differential privacy. 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 which 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 6). Instead, in this section we show how

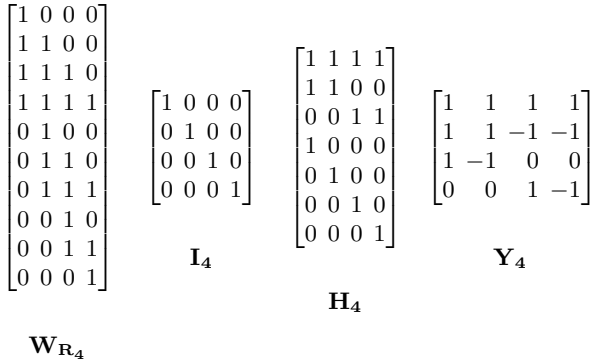

Figure 14: Workload of all range queries and query strategies for 4 ranges

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 an specific workload $\mathbf{W_R}$: the set of all range queries over a domain. Figure 14 illustrates the workload that would be answer for a frequency count of four ranges. The identity strategy $\mathbf{I_4}$, represents 4 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, while the wavelet strategy $\mathbf{Y_4}$ contains four queries representing the Haar wavelet matrix.

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 more accurate answer for $\mathbf{W_R}$ will be yield by the identity strategy. This is not unexpected, since in order 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 $n = 512$. While we use the same kind of plot, this error cannot be directly compared with the one shown in Figure 7 of [33], since we use a different error metrics: (α,β)-accuracy vs 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,

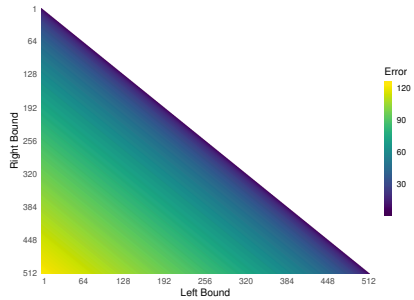

Figure 15: Error of each range query in $\mathbf{W_R}$ using strategy $\mathbf{I_n}$ with $n = 512, \epsilon = 1$, and $\beta = 0.05$

we can use the Chernoff bound to show that the error is approximately $\mathcal{O}(\sqrt{n})$ for each range query, and a maximum error of $\mathcal{O}(\sqrt{n \log n})$ for answering any query in $\mathbf{W_R}$. If we compare our maximum error $\mathcal{O}(\sqrt{n \log n})$ with the one of the matrix mechanism based on the identity strategy $\mathcal{O}(n/\epsilon^2)$, it becomes evident how Chernoff bound is useful to provide tighter accuracy bounds. Unfortunately, as previously stated, 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 [33]). This limitation can be addressed by leveraging the fact that DPella is a programming framerwork that could be *extended* by adding the matrix mechanism—and some other features—as black-box primitives.

4.2 Privacy and accuracy trade-off analysis

We study histograms with certain hierarchical structure (commonly seen in Census Bureaus 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 an 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.

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

```
1 hierarchical₁ [e₁, e₂, e₃] dat = do
2   -- h₁ :: Map Gen (Value Double)
3   -- h₂ :: Map (Gen, Age) (Value Double)
4   -- h₃ :: Map (Gen, Age, Nationality) (Value Double)
5   h₁ ← byGen          e₁ dat
6   h₂ ← byGenAge       e₂ dat
7   h₃ ← byGenAgeNat    e₃ dat
8   return (h₁, h₂, h₃)
```

(a) Hierarchical histogram I: distribute budget among the levels

```
9  hierarchical₂ e dat = do
10   h₃ ← byGenAgeNat e dat
11   h₂ ← level₂ h₃
12   h₁ ← level₁ h₃
13   return (h₁, h₂, h₃)
```

(b) Hierarchical histogram II: spend budget only on the most detailed histogram

Figure 16: Implementation of hierarchical histograms

Our first approach is depicted in Figure 16a, where `hierarchical`$_1$ 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 an histogram of the specified set of attributes. Observe that these sub-queries are called with potentially different epsilons, namely $\mathtt{e}_1$, $\mathtt{e}_2$, and $\mathtt{e}_3$, then under sequential composition, we expect `hierarchical`$_1$ to be $\mathtt{e}_1$+$\mathtt{e}_2$+$\mathtt{e}_3$-differentially private.

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., $\mathtt{e}_1 = 1$, $\mathtt{e}_2 = 1$, and $\mathtt{e}_3 = 1$, 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

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

Table 2: Budgeting with α tolerances, $\beta = 0.05$, & total $\epsilon = 3$

adjusts the different epsilons and calls `accuracy` during the minimization process. To ensure termination, the optimizer aborts after a fixed number of calls to `accuracy`, or when the local budget $\mathtt{e_i}$ 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 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 adheres 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 aborted on the most detailed histogram (`byGenAgeNat`) since it could not find an epsilon that fulfills that requirement—the best we can do is spending all the budget and obtain and 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 the budget and obtaining an error bound of 5.76 instead of 5. While 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 which consists on spending privacy budget *only* for the most detailed histogram. As shown in Figure 16b, this new approach spends all the budget `e` on calling $\mathtt{h_3} \leftarrow$ `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 $\mathtt{h_3}$ (using helper functions $\mathtt{level_2}$ and $\mathtt{level_1}$) creating the rest of the histograms representing the Cartesian products of gender and age, and gender, respectively. These methodology will use `add` and $\mathtt{norm}_\infty$ to compute the de-

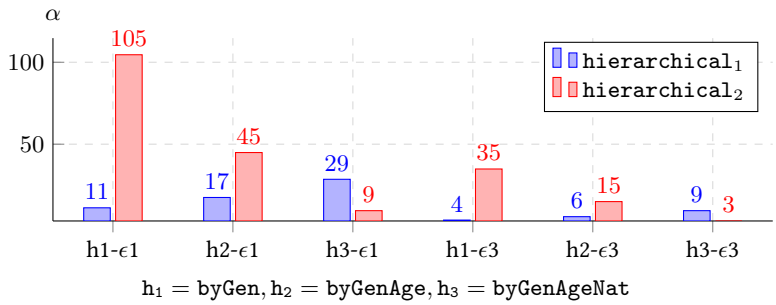

Figure 17: `hierarchical`$_1$ vs. `hierarchical`$_2$

rived histograms, and therefore will not consume more privacy budget. Observe that the query proceeds in a bottom-up fashion, i.e., 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 privacy and accuracy? Figure 17 shows the accuracy of the different level of histograms, i.e., $\texttt{h}_1$, $\texttt{h}_2$, and $\texttt{h}_3$, 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 this information by running repetitively the function `accuracy`. Form the graphics, we can infer that the splitting of the privacy budget per level often gives rise to more accurate histograms. However, observe the exception when $\epsilon = 3$ for `hierarchical`$_2$: in this case, `hierarchical`$_1$ will use an $\epsilon = 1$ in that histogram so it will receive a more noisy count than using $\epsilon = 3$.

4.3 K-way marginal queries on synthetic data

When compared with (non-compositional) approaches for estimating accuracy based on synthetic or public data, such as [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 kind 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 extensively studied in the DP literature, see [56, 22, 13] among others. A k-marginal query, also called a k-conjunction, returns the count of how many in-

```
1   -- Perform all 3-way combinations up to attribute dim
2  allChecks :: ϵ → Int → Data s BinR → [Query (Value Double)]
3  allChecks localEps dim db = do
4    (i, j, k) ← combinatory (dim-1) 3
5    let allOne r = (r !! i) ≡ (r !! j) ≡ (r !! k) ≡ 1
6    return (do tab ← dpWhere allOne db
7               dpCount localEps tab
8           )

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

Figure 18: K-way marginal implementation

dividual 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 the 3-way marginals of a dataset. This is implemented through the functions depicted in Figure 18.

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 $\binom{\texttt{dim}}{3}$. For each tuple, `allChecks` filters the rows which have attributes `i`, `j`, and `k` set to 1 (implemented as `dpWhere allOne db`) for then making a noisy count (`dpCount localEps tab`). Lastly, function `threeMarginal` collects the counts for the different considered attributes and places them into a vector (`norm∞ checks`).

We run `threeMarginal` considering a synthetic dataset (`db`) which has only *1 row with all the attributes set to zeros*. Setting all the attributes to zero produces that all the counts are 0, thus we are able to measure the noisy on each run and accuracy accordingly. We run `threeMarginal` approx. 1000 times for each dimension to measure the noisy magnitude, where we took the `1-`β percentile with $\beta =$ `0.05` (as we did in many of our case studies). Observe that we have $\binom{\texttt{dim}}{3}$ queries and so $\binom{\texttt{dim}}{3}$ independent sources of noise, which need an high number of runs to

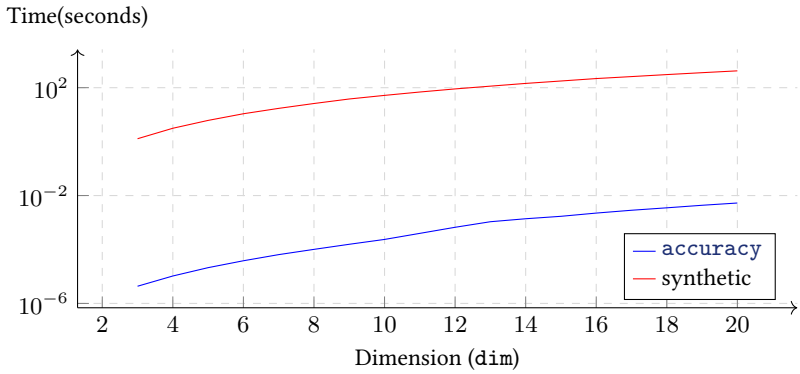

Figure 19: Performance comparison between `accuracy` (DPella) and estimating errors using synthetic analysis

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 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 different in performance.

Figure 19 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 that DPella, as an static analysis, do not execute the filtering `dpWhere allOne db` (as well as any other transformation, recall Section 3.2) which an approach based on synthetic data should do many times—in our case 1000 iterations for each dimension. We expect that for more complex tasks this difference is even more evident.

5

Testing accuracy

In previous sections we have seen the usefulness of `accuracy` function to inspect queries' error, 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 quantification of how tight are DPella's estimations for a query while still preserving the privacy of the data subjects. The primitive `empiric` is a follows:

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

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_{th}; primitive `empiric` will return the empirical confidence β_{emp} of the given query using the theoretical error provided by `accuracy` with β_{th}.

Ideally, β_{emp} should be significantly close to β_{th}. In particular, since `accuracy` yields an *upper bound* of the error, when `empiric` is run multiple times we expect β_{emp} to be less or equal than β_{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

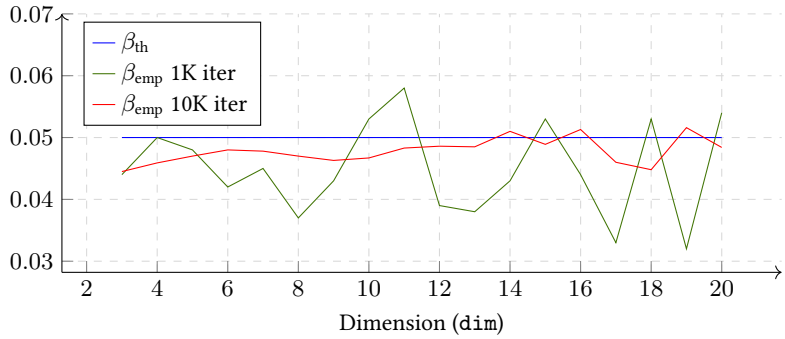

Figure 20: Results of `empiric` over 3-way marginals

actually yield an upper bound of the query's accuracy. On the other hand, if for most of the runs we observe that $\beta_{\text{emp}} \ll \beta_{\text{th}}$, 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. Firstly, 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 query's noise since they will only return the perturbation to be added, this 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 obtain the empirical errors. Secondly, it computes the theoretical error α_{th} using primitive `accuracy` with confidence β_{th}. Lastly, it computes β_{emp} as the percentage of empirical errors that are above the theoretical one. Formally, let α^i_{emp} be the i-th empirical error and x_i be a binary variable describing if $\alpha^i_{\text{emp}} > \alpha_{\text{th}}$, then we can calculate β_{emp} as follow.

$$\beta_{\text{emp}} = \frac{\sum_{i=1}^{\text{iter}} x_i}{\text{iter}}$$

In other words, β_{emp} describes the likelihood that the empirical error α_{emp} is grater than the theoretical one α_{th}, provided by DPella.

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 4.3). Previously, we claimed that the empirical error of function `threeMarginal` from Figure 18 is well aligned with the theoretical one provided by DPella. This statement can be now verified using

`empiric` primitive. For `localEps` $=$ `0.1` and `dim` ranging from 3 to 20, Figure 20 shows the results of calculating the empirical confidence of `threeMarginal` with β_{th} set to 0.05 and iterating 1000 and 10000 times. From these results we can conclude that increasing the number of iterations will stabilize the results, making the anlayses easier, and that, the empirical error provided for DPella for function `threeMarginal` is indeed very close tho the empirical error bound. Moreover, it depicts DPella's soundness, since in both cases (for 1K and 10K iterations), most of the β_{emp} where below β_{th}'s line.

6

Limitations & Extensions

We have discussed so far the use of DPella as an API allowing a programmer to implement her own data analyses. However, we foreseen DPella to also serve as a "glue" which 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 (i) the static analysis for accuracy provided by DPella is too conservative, or (ii) DPella's API building blocks are not enough to express the desired analysis. Below, we describe several possible such extensions.

The matrix mechanism (MM)

As we discussed in the previous section, in some situations DPella allows to answer in an accurate way multiple counting queries in a way that is similar to the MM. As an example, DPella estimates accuracy better then MM for the strategy **I**—recall Section 4. However, for other workloads and other strategies the accuracy provided by DPella is too conservative. To consider other workloads and strategies, the MM 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 iCDFs for it—this can be calculated as in [24]. In general, it is sound to add new primitives which permit a more precise accuracy analysis as long as the return values are tainted, and an accuracy information is provided—thus effectively allowing to further compose the primitive with other analyses by means of the union bound.

Primitives with non-compositional privacy analyses

Several DP-algorithms have a privacy analysis which does not follow directly by composition. Some well-known examples are report-noisy-max, the exponential mechanism, and the sparse-vector technique—see [15] and [7] 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 to branch on the results of noised queries, and because the privacy analyses that DPella support 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 exponential mechanism (EM) can be incorporated into DPella in a similar way. One subtleties 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 EM is tainted. The EM is an important mechanism which allows to implement many other techniques. In particular, we can use EM to implement the offline version of the sparse vector technique, as discussed in [15]. 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 [8].

Online adaptive algorithms

Several DP-algorithms have different implementations depending if they work offline—where all the decision are taken upfront before running the program—or online—where some of the decision are taken while running the program. Online algorithms usually have a more involved control flow which depend on information that are available at runtime. As an example, the online version of the sparse vector technique uses the result of a DP query to decide whether to stop or not the computation (or whether to stop or not giving meaningful answers). These kind of algorithms usually are based on some re-use of a noised result which

correspond to a taint value in DPella. So, the current design of DPella cannot support them. We plan to explore as future how to integrate these algorithms in DPella.

Improving accuracy through post-processing

Several works have explored the use of post-processing techniques to improve on accuracy, e.g. [28, 26, 45]. Most of these works use accuracy measure 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. [28] describes how to boost accuracy in terms of *Mean Squared Error* (MSE) for DP hierarchical queries by post-processing the DP results by means of some relatively simple optimization. This improvement in accuracy relies among other things on 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 improve accuracy. However, we leave this for future works. Moreover, the reason for us to chose 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.

7

Related work

Programming frameworks for DP

PINQ [38] uses dynamic tracking and sensitivity information to guarantee privacy of computations. Among the frameworks and tools sharing features with PINQ we highlight: Airavat [49] ; wPINQ [47]; DJoin [43]; Ektelo [60]; Flex [30]; 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 2, DPella supports a limited form of joins, and it is still able to provide accuracy estimates. We leave as future work to support more general join operations through techniques similar to the ones proposed in Flex and PrivateSQL. While several of the components from the frameworks discussed above are not supported in the current implementation of DPella, these can be added as black-box primitives, as we discussed in Section 6. All the programming frameworks discussed above 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 have a general compositional theory, yet.

Tools for DP

In a way similar to DPella, there exist tools which support reasoning about accuracy and restrict the kind of data analyses they support. GUPT [42] is a tool based on the sample-and-aggregate framework for differential privacy [46]. GUPT allows analysts to specify the target accuracy of the output, and compute privacy from it—or 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 intervals 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 1 and Section 4, and it also allows to reason about the accuracy of combined queries, rather that just about the individual ones. PSI [23] offers to 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 [24] has similar goals as DPella and it allows data analysts to write queries as SQL-like statements. However, the model that APEx uses is different from DPella's. 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 4) and applies a 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 answer effectively 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 6). While APEx supports advanced query strategies, it does not provide means to reason about combinations of analyses, e.g., it does not support reasoning about the accuracy of a query using results from WCQs queries to perform TCQs ones. DPella instead has been designed specifically to support the combination of different queries. As we discussed in Section 6, 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 in order to reason about the accuracy of the combined queries.

Formal Calculi for DP

There are several works on enforcing differential privacy relying on different models and techniques. Within this group are Fuzz [48]—a programming language which enforces (pure) differential privacy of computations using a linear type system which keeps track of program

sensitivity—and its derivatives DFuzz [21], Adaptive Fuzz [58], Fuzzi [62], and Duet [44]. Hoare2 [4], a programming language which enforces (pure or approximate) differential privacy using program verification, together with its extension PrivInfer [5] supporting differentially private Bayesian programming; and other systems using similar ideas [7, 1, 61, 57].

Barthe et al. [3] devise a method for proving differential privacy 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. On the other hand, there are mechanisms that can be analyzed using aHL and cannot be analyzed using DPella, e.g. adaptive online algorithms. Second, aHL supports only reasoning about accuracy but it does not support reasoning about privacy. This makes it difficult to use aHL for reasoning about the privacy-accuracy tradeoffs. Finally, aHL supports only reasoning using the union bound and it 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 [6] 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 works. More recently, Smith et al. [54] propose an automated approach for computing accuracy bounds of probabilistic imperative programs. This work shares some similarities with our. However, it does not support reasoning about privacy, and it only uses the Union Bound and do not attempt to reason about probabilistic independence to obtain tighter bounds.

Other works

In a recent work, Ligett et al. [35] propose a framework for developing differentially private algorithms under accuracy constraints. This allows one to chose a given level of accuracy first, and then finding the private algorithm meeting this accuracy. This framework is so far limited to empirical risk minimization problems and it is not supported by a system, yet.

Conclusions

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 the principles behind DPella, i.e., the use of concentration bounds guided by taint analysis, could generalize for more notions of privacy such as Renyi-DP [40], concentrated differential privacy [16], zero concentrated differential privacy [10], or truncated concentrated differential privacy [11] (as done with (ϵ, δ)-DP). As future work, we envision lifting the restriction that programs should not branch on query outputs.

Bibliography

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

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

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

[4] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *POPL'15*. ACM, 2015.

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

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

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

[8] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. A program logic for union bounds. In *International Colloquium on Automata, Languages, and Programming, ICALP*, volume 55 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[9] Jeremiah Blocki, Avrim Blum, Anupam Datta, and Or Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *Innovations in Theoretical Computer Science, ITCS*, 2013.

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

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

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

[13] Graham Cormode, Tejas Kulkarni, and Divesh Srivastava. Marginal release under local differential privacy. In *Proc. of International Conference on Management of Data, SIGMOD*, pages 131–146, 2018.

[14] Devdatt P Dubhashi and Alessandro Panconesi. *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, 2009.

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

[16] Cynthia Dwork and Guy N Rothblum. Concentrated differential privacy. *arXiv preprint arXiv:1603.01887*, 2016.

[17] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, pages 265–284, 2006. ISBN 3-540-32731-2, 978-3-540-32731-8.

[18] Cynthia Dwork, Guy N. Rothblum, and Salil P. Vadhan. Boosting and differential privacy. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 51–60, 2010.

[19] Hamid Ebadi and David Sands. Featherweight PINQ. *Privacy and Confidentiality*, 7(2), 2017.

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

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

[22] Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Zhiwei Steven Wu. Dual query: Practical private query

release for high dimensional data. In *Proc. International Conference on Machine Learning, ICML*, 2014.

[23] Marco Gaboardi, James Honaker, Gary King, Kobbi Nissim, Jonathan Ullman, and Salil P. Vadhan. PSI (Ψ): a private data sharing interface. *CoRR*, abs/1609.04340, 2016.

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

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

[26] Moritz Hardt and Kunal Talwar. On the geometry of differential privacy. In *Proc. of the 42nd ACM Symposium on Theory of Computing, STOC*, 2010.

[27] Moritz Hardt, Katrina Ligett, and Frank McSherry. 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*, 2012.

[28] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. Boosting the accuracy of differentially private histograms through consistency. *PVLDB*, 3(1), 2010.

[29] Michael Hay, Ashwin Machanavajjhala, Gerome Miklau, Yan Chen, and Dan Zhang. Principled evaluation of differentially private algorithms using DPBench. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016.

[30] Noah M. Johnson, Joseph P. Near, and Dawn Song. Towards practical differential privacy for SQL queries. *PVLDB*, 11(5), 2018.

[31] Noah M. Johnson, Joseph P. Near, and Dawn Song. Towards practical differential privacy for SQL queries. *PVLDB*, 11(5), 2018.

[32] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. PrivateSQL: A differentially private SQL query engine. *Proc. VLDB Endow.*, 12 (11):1371–1384, July 2019. ISSN 2150-8097.

[33] Chao Li, Gerome Miklau, Michael Hay, Andrew McGregor, and Vibhor Rastogi. The matrix mechanism: optimizing linear counting queries under differential privacy. *VLDB J.*, 24(6), 2015.

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

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

[36] Ashwin Machanavajjhala, Daniel Kifer, John M. Abowd, Johannes Gehrke, and Lars Vilhuber. Privacy: Theory meets practice on the map. In *Proc. International Conference on Data Engineering, ICDE*, 2008.

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

[38] Frank D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*. ACM, 2009.

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

[40] Ilya Mironov. Rényi differential privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 2017.

[41] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[42] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David E. Culler. GUPT: privacy preserving data analysis made easy. In *Proc. ACM SIGMOD International Conference on Management of Data, SIGMOD*, 2012.

[43] Arjun Narayan and Andreas Haeberlen. DJoin: Differentially private join queries over distributed databases. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. USENIX Association, 2012.

[44] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. ISSN 2475-1421.

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

[46] Kobbi Nissim, Sofya Raskhodnikova, and Adam D. Smith. Smooth sensitivity and sampling in private data analysis. In *Proc. Annual ACM Symposium on Theory of Computing*, 2007.

[47] Davide Proserpio, Sharon Goldberg, and Frank McSherry. Calibrating data to sensitivity in private data analysis. *PVLDB*, 7(8), 2014.

[48] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, 2010.

[49] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *Proc. USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2010.

[50] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symp. on Haskell*. ACM Press, 2008.

[51] Alejandro Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.

[52] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

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

[54] Calvin Smith, Justin Hsu, and Aws Albarghouthi. Trace abstraction modulo probability. *PACMPL*, 3(POPL), 2019.

[55] David Terei, Simon Marlow, Simon L. Peyton Jones, and David Mazières. Safe Haskell. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 137–148, 2012.

[56] Justin Thaler, Jonathan Ullman, and Salil P. Vadhan. Faster algorithms for privately releasing marginals. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP*, pages 810–821, 2012.

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

[58] Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. A framework for adaptive differential privacy. *PACMPL*, 1(ICFP), 2017.

[59] Xiaokui Xiao, Guozhang Wang, and Johannes Gehrke. Differential privacy via wavelet transforms. *IEEE Trans. Knowl. Data Eng.*, 23 (8), 2011.

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

[61] Danfeng Zhang and Daniel Kifer. LightDP: towards automating differential privacy proofs. In *Proc. ACM SIGPLAN Symp. on Principles of Programming Languages*, 2017.

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