A Programming Framework for OpenDP[*][†]

Marco Gaboardi[‡] Michael Hay[§] Salil Vadhan[¶]

May 11, 2020

Abstract

In this working paper, we propose a programming framework for the library of differentially private algorithms that will be at the core of the OpenDP open-source software project, and recommend programming languages in which to implement the framework.

Note: The current version of this paper is intended as a proposal for discussion at the May 2020 OpenDP Community Meeting, rather than as a scholarly paper that identifies novel contributions. In particular, proper credits to and comparisons with prior work are still missing (but will be added in a future version).

[*]CC BY 4.0: `https://creativecommons.org/licenses/by/4.0/`

[†]We thank the rest of the OpenDP Design Committee (Mercè Crosas, James Honaker, Gary King, Aleksandra Korolova, and Ilya Mironov) for numerous discussions and feedback that contributed to this paper.

[‡]Department of Computer Sciences, Boston University. Supported by NSF through grants 1565365 and 1845803.

[§]Department of Computer Sciences, Colgate University. Supported by NSF through grant 1409125 and by DARPA and SPAWAR under contract N66001-15-C-4067.

[¶]School of Engineering and Applied Sciences, Harvard University. Supported by a grant from the Sloan Foundation and a Simons Investigator Award.

Contents

1 Goals

In this paper, we propose a programming framework for the library of differentially private algorithms that will be at the core of the OpenDP open-source software project, and recommend programming languages in which to implement the framework.

There are a number of goals we seek to achieve with this programming framework and language choice:

Extensibility We would like the OpenDP library to be able to expand and advance together with the rapidly growing differential privacy literature, through external contributions to the codebase. This leads to a number of the other desiderata listed below.

Flexibility The programming framework should be flexible enough to incorporate the vast majority of existing and future algorithmic developments in the differential privacy literature. It should also be able to support many variants of differential privacy. These variants can differ in the type of the underlying sensitive datasets and granularity of privacy (e.g. not only tabular datasets with record level-privacy, but also graph datasets with node-level privacy and datastreams with user-level privacy), as well as in the distance measure between probability distributions on adjacent datasets (e.g. not only pure and approximate differential privacy, but also Rényi and concentrated differential privacy).

Verifiability External code contributions need to be verified to actually provide the differential privacy properties they promise. We seek a programming framework that makes it easier for the OpenDP Editorial Board and OpenDP Committers to verify correctness of contributions. Ideally, most contributions will be written by combining existing components of the library with built-in composition primitives, so that the privacy properties are automatically derived. Human verification should mostly be limited to verifying mathematical proofs for new building blocks, which should usually be small components with clear specifications. At the same time, if an error is later found in a proof for a building block, it should be possible to correct that component locally and have the correction propagate throughout all the code that makes use of that building block.

Programmability The programming framework should make it relatively easy for programmers and researchers to implement their new differentially private algorithms, without having to learn entirely new programming paradigms or having to face excessive code annotation burdens.

Modularity The library should be composed of modular and general-purpose components that can be reused in many differentially private algorithms without having to rewrite essentially the same code. This supports extensibility, verifiability, and programmability.

Usability It should be easy to use the OpenDP Library to build a wide variety of DP Systems that are useful for OpenDP's target use cases. These may have varying front-end user interfaces (e.g. programming in Python notebooks versus a graphical user interface) and varying back-end storage, compute, and security capabilities.

Efficiency It should be possible to compile algorithms implemented in the programming framework to execute efficiently in the compute and storage environments that will occur in the aforementioned systems.

Utility It is important that the algorithms in the library expose their utility or accuracy properties to users, both prior to being executed (so that "privacy loss budget" is not wasted on useless computations) and after being executed (so that analysts do not draw incorrect statistical conclusions). When possible, algorithms should expose uncertainty measures that take into account both the noise due to privacy and the statistical sampling error.

In Sections 2–6, we build up the ideas in our proposed framework by starting with the ideas underlying existing differential privacy systems like PINQ [8], Ektelo [13], and Fuzz [11], and explaining how we generalize them to achieve the above goals. We present the framework with both mathematical specifications and with

Python-like code snippets. The code snippets are meant solely for illustrative purposes, and not meant to be a proposal for actual implementation (which is discussed more in Sections 7 and 8).

2 Stable Transformations on Datasets and Pure DP

In the most standard model of differential privacy and in systems like PINQ, a sensitive dataset x is represented as a *multiset* of records, each of which is from some data domain $\mathcal{X}$, which we'll write as $x \in \text{MultiSets}(\mathcal{X})$. For example, for a dataset where each record has a student's last name, age, and GPA, we might have $\mathcal{X} = \{A, B, \ldots, Z\}^* \times \mathbb{N} \times \mathbb{R}$. The *distance* between two datasets $x, x' \in \text{MultiSets}(\mathcal{X})$ is the size of their symmetric difference $d_{\text{Sym}}(x, x') = |x \Delta x'|$.[1] We say that x and x' are *adjacent*, written $x \sim x'$, if $d_{\text{Sym}}(x, x') \leq 1$, i.e. x and x' differ by adding or removing at most one record. The two main kinds of operators in PINQ and Ektelo are *measurements* and *transformations*.

2.1 Measurements

A *measurement* M is a randomized mapping from datasets to outputs of an arbitrary type. That is, $M : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathcal{Y}$, where we are using the squiggly arrow $\rightsquigarrow$ to denote a randomized function. A measurement M is *ε-DP* if for every pair of adjacent datasets $x \sim x' \in \text{MultiSets}(\mathcal{X})$, the random variables $Y = M(x)$ and $Y' = M(x')$ have the property that for every set $T \subseteq \mathcal{Y}$,

$$\Pr[Y \in T] \leq e^{\varepsilon} \cdot \Pr[Y' \in T].$$

Equivalently, we have $D_\infty(Y||Y') \leq \varepsilon$, where

$$D_\infty(Y||Y') = \sup_{T \subseteq \mathcal{Y}: \Pr[Y' \in T] > 0} \ln\left(\frac{\Pr[Y \in T]}{\Pr[Y' \in T]}\right).$$

Yet another equivalent formulation (known as the "group privacy" property of pure differential privacy) is that for every pair of (not necessarily adjacent datasets $x, x' \in \text{MultiSets}(\mathcal{X})$,

$$D_\infty(Y||Y') \leq \varepsilon \cdot d_{\text{Sym}}(x, x') \tag{1}$$

An example of a measurement operator is a noisy sum. Suppose $\mathcal{X} = [L, U]$ is an interval of real numbers. Then the following is an ε-DP measurement $\text{NoisySum}_{L,U,\varepsilon} : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathbb{R}$:

$$\text{NoisySum}_{L,U,\varepsilon}(x) = \sum_{z \in x} z + \text{Lap}(S/\varepsilon), \text{ where } S = \max\{|L|, |U|\}$$

where $\text{Lap}(S/\varepsilon)$ represents a draw from the Laplace distribution with scale S/ε and the summation $\sum_{z \in x}$ is a summation with multiplicity.[2]

Here we see that a measurement operator M and its privacy properties are specified by three attributes:

1. The data domain $\mathcal{X}$, whereby the measurement operator expects inputs from $\text{MultiSets}(\mathcal{X})$.

2. The privacy loss parameter ε.

3. The randomized function from $\text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathcal{Y}$.

The specification of the input data domain $\mathcal{X}$ is important because the privacy properties of M rely on the assumption that the inputs x come from $\text{MultiSets}(\mathcal{X})$. As above it is often useful to make use of rich data types, like the interval $[L, U]$. (When using the library, one should of course minimize the assumptions

[1] A multiset $x \in \text{MultiSets}(\mathcal{X})$ can be specified by its *histogram* $h_x : \mathcal{X} \to \mathbb{N}$, where $h_x(z)$ is the number of occurrences of z in $\mathcal{X}$. Then $d_{\text{Sym}}(x, x')$ equals the ℓ_1 distance between h_x and $h_{x'}$, i.e. $\sum_z |h_x(z) - h_{x'}(z)|$.

[2] That is, $\sum_{z \in x} z = \sum_{z \in [L,U]: h_z > 0} h_x(z) \cdot z$, where h_x is the histogram of x.

about the data that are used for privacy. See Section 7 for discussion on how this can be done.) We don't include the output data domain $\mathcal{Y}$ as an attribute, because without loss of generality we can think of $\mathcal{Y}$ as the countably infinite set of all possible objects representable in our programming language, so it need not be an explicitly specified attribute of M.

In code, our noisy sum measurement operator could be implemented as follows:

```
1  class Measurement:
2    input_domain
3    privacy_loss
4    function
5
6  def MakeNoisySum(L: float, U: float, epsilon: float):
7      if L > U or epsilon < 0: raise Exception('Invalid parameters')
8      input_domain = bounded_float(L,U)
9      privacy_loss = epsilon
10     def function(data):
11       data_sum = sum(data)
12       s = max(abs(L), abs(U))
13       z = Laplace(s/epsilon,0)
14       return data_sum + z
15     return Measurement(input_domain,privacy_loss,function)
16
17   # Example
18   l, u, eps = # ... assign to some values
19   NoisySum = MakeNoisySum(l, u, eps)
```

Figure 1: Measurement Example: `NoisySum`. We define `Measurement` as a class with three attributes and an implicit constructor, but we could have defined it just as some kind of *struct* with three attributes. We use `class` as one such example, but this should not be read as advocating object-oriented languages (discussed further in Section 8). `NoisySum` is a `Measurement` with all parameters fixed (the values are assigned on 18). The `MakeNoisySum` function acts like a constructor that allows a programmer to specify values for the parameters, discussed further in Section 3. Our code examples make some assumptions: we assume the availability of complex data types such as `bounded_float(L,U)`, and of operations over multisets (implicitly defined), like the `sum` operator shown. We also pretend that floating-point arithmetic is a faithful implementation of real-number arithmetic, which it is certainly not and indeed this is known to lead to failures of differential privacy [9]. This is only for pedagogical purposes, to convey the ideas of the programming framework with familiar examples, but in Section 7 we advocate that the OpenDP library entirely avoid floating-point arithmetic.

2.2 Transformations

In PINQ, a *transformation* is a (deterministic) mapping from datasets to datasets. That is, $T : \mathrm{MultiSets}(\mathcal{X}) \to \mathrm{MultiSets}(\mathcal{Y})$. For $c \in \mathbb{R}$, we say that T is *c-stable* if for all $x, x' \in \mathrm{MultiSets}(\mathcal{X})$,

$$d_{\mathrm{Sym}}(T(x), T(x')) \leq c \cdot d_{\mathrm{Sym}}(x, x'). \tag{2}$$

An example of a stable transformation is *clamping*. Suppose $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = [L, U]$ and define $\mathrm{clamp}_{L,U} : \mathcal{X} \to \mathcal{Y}$ by

$$\mathrm{clamp}_{L,U}(z) = \begin{cases} U & \text{if } z > U \\ z & \text{if } z \in [L, U] \\ L & \text{if } z < L. \end{cases}$$

Then we can define a 1-stable transformation $T : \text{MultiSets}(\mathcal{X}) \to \text{MultiSets}(\mathcal{Y})$ by:

$$T(x) = \{\text{clamp}_{L,U}(z) : z \in x\} \quad \text{(with multiplicity)}.$$

Here we see that a transformation T and its stability properties are specified by four attributes:

1. The data domain $\mathcal{X}$ for the input datasets.
2. The data domain $\mathcal{Y}$ for the output datasets.
3. The stability parameter c.
4. The function from $\text{MultiSets}(\mathcal{X}) \to \text{MultiSets}(\mathcal{Y})$.

Here we need to specify the output domain $\mathcal{Y}$ because a transformation should promise that if the input dataset x is in $\text{MultiSets}(\mathcal{X})$, then the output dataset $T(x)$ is in $\text{MultiSets}(\mathcal{Y})$. This is crucial for *chaining* as described later.

In code, the clamping transformation could be implemented as follows:

```python
class Transformation:
  input_domain
  output_domain
  stability
  function

def MakeClamp(L: float, U: float):
    if L > U: raise Exception('Invalid parameters')
    input_domain = float
    output_domain = bounded_float(L,U)
    stability = 1
    def function(data):
      def clamp(x): return max(min(x, U), L)
      return map(clamp, data)
    return Transformation(input_domain,output_domain,stability,function)

# Example
Clamping = MakeClamp(l, u)  # where l, u are constants defined previously
```

Figure 2: Transformation Example: Clamping. We use the same style as in Figure 1 for measurements. We define `Transformation` as a class with four attributes. `Clamping` is a concrete transformation with fixed parameters, constructed via `MakeClamp`. We use here a function `map` to apply the `clamp` function to every element of the multiset.

2.3 Chaining

From transformations and measurements defined as above, we can build up more complex transformations and measurements through various operators that combine them. One way of combining measurements and transformations is through *chaining*, which is simply function composition (but in the differential privacy literature the term "composition" has a different meaning, described below). There are two forms of chaining:

TT Chaining: If $S : \text{MultiSets}(\mathcal{X}) \to \text{MultiSets}(\mathcal{Y})$ is a c-stable transformation, and $T : \text{MultiSets}(\mathcal{Y}) \to \text{MultiSets}(\mathcal{Z})$ is a d-stable transformation, then $T \circ S : \text{MultiSets}(\mathcal{X}) \to \text{MultiSets}(\mathcal{Z})$ is a cd-stable transformation. Notice that the S and T need to be compatible in the sense that the data domain of the output dataset of S must match the data domain of the input datasets of T.

MT Chaining: Similarly, if T : MultiSets($\mathcal{X}$) $\rightarrow$ MultiSets($\mathcal{Y}$) is a c-stable transformation, and M : MultiSets($\mathcal{Y}$) $\rightsquigarrow \mathcal{Z}$ is an ε-DP measurement, then $M \circ T$: MultiSets($\mathcal{X}$) $\rightsquigarrow \mathcal{Z}$ is a $c\varepsilon$-DP measurement. Again, this requires compatibility of the intermediate data domain between T and M.

To illustrate the latter (chaining of a transformation and measurement), consider $T = \text{clamp}_{L,U}$ and $M = \text{NoisySum}_{L,U,\varepsilon}$. Since $\text{clamp}_{L,U}$: MultiSets($\mathbb{R}$) $\rightarrow$ MultiSets($[L, U]$) is a 1-stable transformation and $\text{NoisySum}_{L,U,\varepsilon}$: MultiSets($[L, U]$) $\rightsquigarrow \mathbb{R}$ is ε-DP, their composition

$$\text{NoisyClampedSum}_{L,U,\varepsilon} = \text{NoisySum}_{L,U,\varepsilon} \circ \text{clamp}_{L,U} : \text{MultiSets}(\mathbb{R}) \rightarrow \mathbb{R}$$

is ε-DP.

In code, we can implement chaining as an operator that takes two transformation objects that are compatible and produces a new transformation object, or similarly for a transformation object and a measurement object. The example above of chaining $\text{clamp}_{L,U}$ and $\text{NoisySum}_{L,U,\varepsilon}$ can be illustrated in code as follows:

```
def ChainingTT(trans_2: Transformation, trans_1: Transformation):
# makes new transformation: trans_2(trans_1( . ))
  if (trans_1.output_domain!=trans_1.input_domain): raise Exception('Domain mismatch')
  input_domain = trans_1.input_domain
  output_domain = trans_2.output_domain
  stability = trans_1.stability*trans_2.stability
  def function(data): return trans_2.function(trans_1.function(data))
  return Transformation(input_domain,output_domain,stability,function)

def ChainingMT(meas: Measurement, trans: Transformation):
# makes new measurement: meas(trans( . ))
  if (trans.output_domain!=meas.input_domain): raise Exception ('Domain mismatch')
  input_domain = trans.input_domain
  privacy_loss = trans.stability*meas.privacy_loss
  def function(data): return meas.function(trans.function(data))
  return Measurement(input_domain,privacy_loss,function)

# Example
NoisyClampedSum=ChainingMT(NoisySum, Clamping)
```

Figure 3: Chaining. We define two chaining operations, chaining of two transformations, and chaining of a transformation and a measurement. These functions check that the input transformations/measurements are certified to be correct (see Section 3) and that the input and output domains are compatible and they raise an exception otherwise.

2.4 Composition

Informally, the *Basic Composition Theorem* of differential privacy says that if we execute an ε_1-DP mechanism on a dataset followed by an ε_2-DP mechanism on the same dataset, the result is $(\varepsilon_1 + \varepsilon_2)$-DP. This sort of a composition principle has two main applications:

1. It allows for tracking cumulative privacy loss when analysts make many differentially private queries on a dataset. This is what gives rise to the notion of a "privacy loss budget" in differential privacy, where we can prevent exceeding a desired total privacy loss bound ε by tracking the accumulated privacy loss and refusing to answer any queries that would result in exceeding the overall budget of ε.

2. It allows for building more complex DP mechanisms from multiple executions of simpler mechanisms, deriving an upper bound on the privacy loss of the "outer mechanism" as the sum of the privacy losses of the the "inner mechanisms".

In the simplest, "nonadaptive" form of basic composition, we have an ε_1-DP mechanism $M_1 : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathcal{Y}$ and an ε_2-DP mechanism $M_2 : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathcal{Z}$, and obtain an $(\varepsilon_1 + \varepsilon_2)$-DP mechanism $M : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathcal{Y} \times \mathcal{Z}$ by $M(x) = (M_1(x), M_2(x))$, where M_1 and M_2 use independent randomness. For example, if we wanted to compute a differentially private clamped mean, we could take $M_1 = \text{NoisyClampedSum}_{L,U,\varepsilon}$ and $M_2 = \text{NoisyClampedSum}_{1,1,\varepsilon}$ to obtain an (2ε)-DP NoisyPair that provides us with both a noisy clamped sum (from M_1) and a noisy estimate of the size of the dataset (from M_2).

In code, we can implement such basic composition as an operator on measurement objects as follows:

```
def Compose(meas_1: Measurement,meas_2: Measurement)
  if (meas_1.input_domain!=meas_2.input_domain): raise Exception('Domain mismatch')
  input_domain = meas_1.input_domain
  privacy_loss = meas_1.privacy_loss+meas_2.privacy_loss
  def function(data): return (meas_1.function(data),meas_2.function(data))
  return Measurement(input_domain,privacy_loss,function)

# Example

# We first define a new measurement: a noisy count that is built from the
# clamping, noisy sum, and chaining operations previously introduced
ClampToOne = MakeClamp(1, 1)
NoisySumOfOnes = MakeNoisySum(1, 1, eps)
NoisyCount=ChainingMT(NoisySumOfOnes, ClampToOne)


# With this we can now define NoisyPair
NoisyPair=Compose(NoisyClampedSum, NoisyCount)
```

Figure 4: Compose. We check that the input domains of the two measurements we want to compose are compatible, otherwise we throw an exception. In the example, we define a `NoisyCount` measurement by reusing previously defined operations and then compose `NoisyClampedSum` and `NoisyCount`.

However, it is often useful to use more sophisticated, "adaptive" forms of composition, where the choice of the mechanism M_2 depends on the result of $M_1(x)$, and we also allow even more mechanisms $M_3, M_4, \ldots$ to be chosen adaptively. This kind of flexibility is clearly important when allowing an analyst to interactively query a dataset protected by differential privacy. To support this, PINQ and other DP systems often manage the privacy budget and composition at a higher layer that sits above the basic transformations and measurements. (Indeed, PINQ also handles chaining at that higher level, rather than as an operator that produces new transformations and measurements.)

2.5 Post-processing

An important property of differential privacy is that it is closed under post-processing. That is, if $M : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathcal{Y}$ is ε-DP and $f : \mathcal{Y} \to \mathcal{Z}$ is any function, then $f \circ M : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathcal{Z}$ is ε-DP.

Now we can actually produce a differentially private mean by composing the NoisyPair mechanism above with a division operation $f(a, b) = a/b$. In code:

```python
def Postprocess(meas: Measurement,funct):
  input_domain = meas.input_domain
  privacy_loss = meas.privacy_loss
  def function(data): return funct(meas.function(data))
  return Measurement(input_domain,privacy_loss,function)

# Example
# We can define an auxiliary divide function
def divide(x,y): return x/y
# We can now redefine NoisyMean using Postprocess.
NoisyMean=Postprocess(NoisyPair,divide)
```

Figure 5: Postprocessing. Notice that we don't require the `input_domain` of the function to match the `output_domain` of the measurement. In fact, measurements don't have an `output_domain` attribute. From the privacy perspective this is not required. This can create a mismatch which can generate an exception, but this exception would not create any privacy issue. Using a statically typed language can help to avoid such situations.

3 Verifying Privacy Properties

Our goal is to ensure that the only measurements and transformations that can be constructed by the OpenDP Library have mathematically proven privacy properties, based on either:

- A custom proof, which is provided by the contributor and is verified by a human (on the OpenDP editorial board) or by a computer (for components that are amenable to formal verification techniques), or
- An automatically derived proof, if the new component is obtained by combining components that already exist in the library (using combination primitives that exist in the library).

Looking at the examples in Section 2, we see that it is rare that a measurement or transformation would be provided as a single, stand-alone object in the library. Rather they are given as parameterized families of objects, like $\text{clamp}_{L,U}$ is parameterized by L and U and $\text{NoisySum}_{L,U,\varepsilon}$ is parameterized by L, U, and ε. So we really need verification procedures for *measurement families* and *transformation families*. In our code examples, we implemented measurement (respectively, transformation) families as constructors that take the parameters and output a measurement (respectively a transformation), or an exception if the parameters are invalid. Thus we propose:

> Code should only be accepted to the library if there is a proof that (1) it can only ever construct *valid* measurements or transformations, where valid means that the measurement (resp. transformation) respects the privacy-loss bound (resp., stability bound) promised in its attributes, (2) it does not modify any measurements or transformations that have been constructed, and (3) it does not modify code in the library.
>
> The proof can assume (by induction) that all measurements and transformations given as inputs or constructed by existing code in the library are valid. In particular, if the new code does not *directly* construct any measurements or transformations on its own (but only using existing code to do so), does not modify any measurements or transformations that have been constructed, and does not modify code in the library, then it should be possible to verify its validity automatically.

Evaluating the code examples from Section 2 with respect to this principle, note that the various checks they do on their input parameters (e.g. checking that $L \leq U$ and $\varepsilon \geq 0$) are crucial for ensuring that is impossible for them to generate invalid measurements or transformations.

Although the principle above ensures that all measurements and transformations constructed by the library are valid and no further verification is necessary, it might be of interest to decorate measurements and transformations with the "provenance" of their proof of correctness (showing how the proofs of all the library methods used to construct them are stitched together).

4 Varying Types and Distance Measures

4.1 Private Data Types

In the basic PINQ-like framework described above, all sensitive data is represented as a multiset, and the granularity of privacy is captured by the symmetric difference metric d_{Sym}. As realized in Fuzz [11], it is very useful to allow for other sensitive data types and metrics. There are two benefits to such a generalization:

1. Not all sensitive data comes in the form a multiset of records, where the desired granularity of privacy is adding or removing one record. For example, for network data, the sensitive dataset is often a graph (sometimes augmented with labels) and the granularity of privacy may refer to modifications at the level of a node or an edge.

2. Even if the original dataset comes in the form of multiset of records, a differentially private algorithm may produce intermediate data representations (e.g. a histogram). Being able to reason about distances in such intermediate representations allows decomposing differentially private algorithms into smaller modular and reusable components.

Now a transformation $T : \mathcal{X} \to \mathcal{Y}$ can have arbitrary input and output types $\mathcal{X}$ and $\mathcal{Y}$, which need not be multisets. For a given metrics $d_{\mathcal{X}}$ and $d_{\mathcal{Y}}$ on $\mathcal{X}$ and $\mathcal{Y}$, we say that T is a *c-stable transformation from $d_{\mathcal{X}}$ to $d_{\mathcal{Y}}$* if all $x, x' \in \mathcal{X}$, we have $d_{\mathcal{Y}}(T(x), T(x')) \leq c \cdot d_{\mathcal{X}}(x, x')$, generalizing Inequality (2). Similarly, we say that a measurement $M : \mathcal{X} \rightsquigarrow \mathcal{Y}$ is *ε-DP with respect to $d_{\mathcal{X}}$* if for all $x, x' \in \mathcal{X}$, $D_\infty(M(x)||M(x')) \leq \varepsilon \cdot d_{\mathcal{X}}(x, x')$, generalizing Inequality (1).

Now, a transformation should also have the metrics $d_{\mathcal{X}}$ and $d_{\mathcal{Y}}$ associated with it, and a measurement the metric $d_{\mathcal{X}}$. Then chaining works as before, except we must also check compatibility of the intermediate metrics.

As an illustration, we can decompose our $\mathrm{NoisySum}_{L,U,\varepsilon}$ as a chaining of two components $\mathrm{BoundedSum}_{L,U}$: $\mathrm{MultiSets}([L, U]) \to \mathbb{R}$ and $\texttt{BaseLap}_{\max\{|L|,|U|\}/\varepsilon} : \mathbb{R} \rightsquigarrow \mathbb{R}$, where

$$\mathrm{BoundedSum}_{L,U}(x) = \sum_{z \in x} z \text{ (with multiplicity)},$$

and

$$\texttt{BaseLap}_\sigma(y) = y + \mathrm{Lap}(\sigma).$$

The ε-DP property of NoisySum follows from the facts that BoundedSum is $\max\{|L|, |U|\}$-stable from d_{Sym} to $d_{\mathbb{R}}$, where $d_{\mathbb{R}}(a, b) = |a - b|$, and $\texttt{BaseLap}_\sigma$ is $(1/\sigma)$-DP with respect to $d_{\mathbb{R}}$.

In code:

```python
class Measurement:
  input_metric   # --- new ---
  input_domain
  privacy_loss
  function

class Transformation:
  input_metric   # --- new ---
  input_domain
  output_metric  # --- new ---
  output_domain
  stability
  function

def ChainingMT(meas: Measurement, trans: Transformation):
  if (trans.output_domain!=meas.input_domain
      or trans.output_metric!=meas.input_metric)): raise Exception('Domain/metric mismatch')
  input_metric = trans.input_metric
  input_domain = trans.input_domain
  privacy_loss = trans.stability*meas.privacy_loss
  def function(data): return meas.function(trans.function(data))
  return Measurement(input_metric,input_domain,privacy_loss,function)

def MakeBaseLap(sigma: float):
  if sigma < 0: raise Exception('Invalid parameter')
  input_metric = dist_real
  input_domain = float
  privacy_loss = 1/sigma
  def function (data): return data + Laplace(sigma,0)
  return Measurement(input_metric,input_domain,privacy_loss,function)

def MakeBoundedSum(L: float, U: float):
    if U > L: raise Exception('Invalid parameters')
    input_metric = dist_sym
    input_domain = multiset(bounded_float(L,U))
    output_metric = dist_real
    output_domain = float
    stability = max(abs(L), abs(U))
    def function (data): return sum(data)
    return Transformation(input_metric,input_domain,output_metric,output_domain,
                          stability,function)

# Example
BaseLaplace = MakeBaseLap(sig)  # sig is some constant
BoundedSum = MakeBoundedSum(l, u) # l,u defined previously
NoisySum=ChainingMT(BaseLaplace, BoundedSum)
print(NoisySum.privacy_loss) # prints max(abs(l),abs(u))/sig
```

Figure 6: Private Data Types. We re-define `Measurement` and `Transformation` to include metrics. We also re-define the chaining operations to also check metric compatibility (`ChainingTT` omitted but similar). Notice how the `input_domain` of the BoundedSum transformation is now more explicit about the type, which declares that we expect a multiset of bounded floats.

4.2 Privacy Relations

In the definitions of “pure” differential privacy above, we measure the distance between the output distributions $M(x)$ and $M(x')$ using “max-divergence” $D_\infty(M(x)||M(x'))$. In the differential privacy literature, a number of other measures have been proposed, motivated by the greater utility they afford and/or their superior composition properties.

For example, with *approximate differential privacy*, we say that M is (ε, δ)-DP if for all $x \sim x'$, we have $D^\delta_\infty(M(x)||M(x')) \leq \varepsilon$, where D^δ_∞ is the *smoothed max-divergence*. This means that for all sets T, we have:

$$\Pr[M(x) \in T] \leq e^\varepsilon \cdot \Pr[M(x') \in T] + \delta.$$

Note that here privacy is not measured by a single number but a pair of numbers (ε, δ). Moreover, there need not be a single optimal choice of (ε, δ) for a given mechanism, but rather a Pareto curve, where for every $\varepsilon > 0$, we can try to identify the minimum δ for which the mechanism is (ε, δ)-DP. A similar phenomenon occurs with other notions such as concentrated differential privacy and Rényi differential privacy.

Thus, it is no longer sufficient to express privacy as a linear relationship between input distances and output distances, like we did before when we required $D_\infty(M(x)||M(x')) \leq \varepsilon \cdot d_\mathcal{X}(x, x')$. Instead, for a measurement $M : \mathcal{X} \rightsquigarrow \mathcal{Y}$ a metric $d_\mathcal{X}$ on $\mathcal{X}$ and a similarity measure D on probability distributions, we assert the privacy properties of M by a *relation* $R(d_\text{in}, d_\text{out})$ that takes an input distance d_in and an output distance d_out and certifies that “for all $x, x' \in \mathcal{X}$, if x is d_in-close to x' under $d_\mathcal{X}$, then $M(x)$ is d_out-close to $M(x')$ with respect to D.”

Let us illustrate with the example of the Gaussian mechanism $\text{BaseGauss}_\sigma : \mathbb{R} \rightsquigarrow \mathbb{R}$, defined as

$$\text{BaseGauss}_\sigma(y) = y + N(0, \sigma).$$

It is known that if $|y - y'| \leq d$, then for every $\delta > 0$,

$$D^\delta_\infty(\text{BaseGauss}_\sigma(y)||\text{BaseGauss}_\sigma(y')) \leq \frac{d}{\sigma} \cdot \sqrt{2 \ln\left(\frac{1.25}{\delta}\right)},$$

provided that the right-hand side is at most 1.

$$R(d_\text{in}, (\varepsilon, \delta)) = \begin{cases} \texttt{true} & \text{if } \min\{\varepsilon, 1\} \geq (d_\text{in}/\sigma)\sqrt{2\ln(1.25/\delta)} \\ \texttt{false} & \text{otherwise} \end{cases}$$

To capture this in code, we modify our Measurement class from before in the following ways:

1. In addition to an input metric $d_\mathcal{X}$, we also have an attribute that corresponds to the privacy measure D.
2. The pure DP constant is replaced with a privacy relation.

```
class Measurement:
  input_metric
  input_domain
  output_measure
  privacy_relation        # --- new (replaces privacy_loss)
  function

def MakeBaseGauss(sigma: float):
  if sigma < 0: raise Exception
  input_metric = dist_real
  input_domain = float
  output_measure = approxDP
  def privacy_relation (d_in: float,d_out: float*float):
    return ((d_in/sigma)*sqrt(2*ln(1.25/d_out[2]))) <= min(d_out[1],1)
  def function (data):
    z = Laplace(sigma,0)
    return data + z
  return Measurement(input_metric,input_domain,output_measure,privacy_relation,function)

# Example
BaseGauss = MakeBaseGauss(sig)
```

Figure 7: Privacy relations. We re-define `Measurement` by changing the `privacy_loss` attribute to a `privacy_relation` attribute. In `BaseGauss`, the `privacy_relation` function returns true if the distance in input and the distance in output satisfy the constraints imposed by the correctness of the Gaussian mechanism for approximate differential privacy – we use a data type `approxDP` as the associated `output_measure`. The relation is intentionally given as *callable* code; as other mechanisms that use the measurement may need to evaluate the relation during execution. The notation `float*float` means the expected type is pair of floats.

4.3 Stability Relations

Having generalized to arbitrary, multi-parameter privacy measures, it is natural to allow the same flexibility for our distance measures on the sensitive data types. An example that has arisen in the differential privacy literature is that of *joins* [10, 5, 7, 8]. Let $x \in \text{MultiSets}(\mathcal{K} \times \mathcal{A})$ and $y \in \text{MultiSets}(\mathcal{K} \times \mathcal{B})$ be two datasets over records that share a common key from domain $\mathcal{K}$. The *join* of x and y over $\mathcal{K}$ is a multiset $x \bowtie_{\mathcal{K}} y \in \text{MultiSets}(\mathcal{K} \times \mathcal{A} \times \mathcal{B})$ defined as

$$x \bowtie_{\mathcal{K}} y = \{(k, a, b) : (k, a) \in x, (k, b) \in y\} \text{ (with multiplicity)}$$

A join has unbounded stability with respect to both x and y, so a solution is to first apply a *truncation* operation to each dataset. Define $\text{Trunc}_{\mathcal{K},\ell}(x)$ to be a dataset obtained by discarding records from x so that for every $k \in \mathcal{K}$, the number of records of the form $(k, \cdot)$ in $\text{Trunc}_{\mathcal{K},\ell}(x)$ (counting multiplicity) is at most ℓ. (Specifically, $\text{Trunc}_{\mathcal{K},\ell}(x)$ sorts the elements $(k, a_1), (k, a_2), \ldots$ according to a fixed ordering on $\mathcal{A}$ and discards all but the first ℓ records.) Similarly define $\text{Trunc}_{\mathcal{K},m}(y)$. Then we define

$$\text{BoundedJoin}_{\ell,m}(x, y) = \text{Trunc}_{\mathcal{K},\ell}(x) \bowtie_{\mathcal{K}} \text{Trunc}_{\mathcal{K},m}(y).$$

Inputs of BoundedJoin are *pairs* of multisets, so it is useful to measure distances by a pair of numbers. Specifically, we say that (x, y) is (c, d)-*close* to (x', y') under $d_{\text{Sym}} \times d_{\text{Sym}}$ if $|x\Delta x'| \le c$ and $|y\Delta y'| \le d$. This notion arises naturally in DP applications when x and y may each be derived from some underlying sensitive dataset z using transformations of stability c and d, respectively. Then we can express the stability property of BoundedJoin as follows: if (x, y) is (c, d)-close to (x', y') under $d_{\mathrm{Sym}} \times d_{\mathrm{Sym}}$, then BoundedJoin$_{\ell,m}(x, y)$ is $2(c \cdot m + \ell \cdot d + 2c \cdot d)$-close to BoundedJoin$_{\ell,m}(x', y')$ under d_{Sym} [7]. This can be expressed in terms of a stability relation as follows:

$$R((c,d),e) = \begin{cases} \texttt{true} & \text{if } e \geq 2(c \cdot m + \ell \cdot d + 2c \cdot d) \\ \texttt{false} & \text{otherwise} \end{cases}$$

4.4 Multiple Distance Measures at Once

Additionally, we need not have a fixed pair of input and output distance measures associated with a given transformation or measurement operator, as the same operator can have its stability measured according to multiple metrics and we can exploit relationships between those metrics to automatically translate between them. For example, it is known that any mechanism that is ρ-zCDP (where zCDP stands for "zero-concentrated differential privacy" [2]) is also $(\rho + 2\sqrt{\rho \ln(1/\delta)}, \delta)$-DP for every $\delta > 0$. Similarly, any two vectors $u, v \in \mathbb{R}^d$ that are α-close in ℓ_1 norm are also $\sqrt{d} \cdot \alpha$-close in ℓ_2 norm. To support this, we no longer have our input and output metrics as fixed attributes of our transformations and measurements, but we take them as input to the privacy or stability relations. That is, for a measurement operator $M : \mathcal{X} \to \mathcal{Y}$ we now have a privacy relation $R((d_{\mathcal{X}}, d_{\mathrm{in}}), (D, d_{\mathrm{out}}))$ such that whenever $R((d_{\mathcal{X}}, d_{\mathrm{in}}), (D, d_{\mathrm{out}}))$ is true, the following holds:

1. $d_{\mathcal{X}}$ is a (possibly multidimensional) distance measure on $\mathcal{X}$, and d_{in} is a valid distance bound under $d_{\mathcal{X}}$.
2. D is a (possibly multidimensional) distance measure on probability distributions on arbitrary sets, and d_{out} is a valid distance bound under D.
3. for all datasets $x, x' \in \mathcal{X}$, if x and x' are d_{in}-close under $d_{\mathcal{X}}$, then the probability distributions $M(x)$ and $M(x')$ are d_{out}-close under D.

For example, when $\mathcal{X} = \mathrm{MultiSets}(\mathcal{Z})$, then $R((d_{\mathrm{Sym}}, 1), (D_\infty, \varepsilon))$ being true is asserts that M is ε-DP. For the BaseGauss$_\sigma : \mathbb{R} \to \mathbb{R}$ mechanism defined above, we can assert its privacy properties under zCDP and approximate DP with a relation defined as follows:

$$R(d_{\mathcal{X}}, d_{\mathrm{in}}), (D, d_{\mathrm{out}})) = \begin{cases} \texttt{true} & \text{if } d_{\mathcal{X}} = d_{\mathbb{R}},\ D = \text{zCDP},\ d_{\mathrm{in}}, d_{\mathrm{out}} \in \mathbb{R}, \text{ and } d_{\mathrm{out}} \geq d_{\mathrm{in}}^2/2\sigma^2 \\ \texttt{true} & \text{if } d_{\mathcal{X}} = d_{\mathbb{R}},\ D = \text{approxDP},\ d_{\mathrm{in}} \in \mathbb{R},\ d_{\mathrm{out}} = (\varepsilon, \delta) \in \mathbb{R} \times \mathbb{R}, \\ & \quad \text{and } \min\{\varepsilon, 1\} \geq (d_{\mathrm{in}}/\sigma)\sqrt{2\ln(1.25/\delta)} \\ \texttt{false} & \text{otherwise} \end{cases}$$

In code:

```
def MakeBaseGauss(sigma: float):
  # ... same as before except relation now takes distance measures as input ...
  def privacy_relation ((m_in,d_in),(m_out,d_out)):
    if m_in==dist_real and type(d_in)==float:
      if m_out==approxDP and type(d_out)==float*float:
        return ((d_in/sigma)*sqrt(2*ln(1.25/d_out[1]))) <= min(d_out[0],1)
      elif m_out==zCDP and type(d_out)==float:
        return (d_in^2/(2*sigma^2)) <= d_out
      else return false
  return Measurement(input_metric,input_domain,output_measure,relation,function)

# Example
BaseGauss = MakeBaseGauss(sig)
```

Figure 8: Privacy relations for multiple distance measures at once. We redefine `MakeBaseGauss` so that it produces a `privacy_relation` that takes distance measures as input in addition to the distances

The BaseGauss example illustrates that the privacy relations are only meant to be *sound* but are not necessary *complete*. That is, if the privacy relation says true, it should be considered a certificate that the measurement or transformation has the claimed privacy or stability properties. However, if the privacy relation says false, it only means that no claim is made about the property. For example, using the aforementioned relationship between zCDP and approximate DP, it follows that the Gaussian mechanism is (ε, δ)-DP whenever

$$\varepsilon \geq (d_{\text{in}}^2/2\sigma^2) + (d_{\text{in}}/\sigma)\sqrt{2\ln(1/\delta)},$$

which is satisfied by some values of $d_{\text{out}} = (\varepsilon, \delta)$ that do not satisfy the relation above. This example also illustrates the eventual possibility of having some automated translation between different distance measures. That is, we can encode the relationships between different distance measures in the OpenDP library, and have it automatically check whether a privacy or stability property can be derived from other ones.

4.5 Chaining

Let us consider how chaining behaves with these stability and privacy relations. Specifically, how do we derive the relation for a chained transformation or measurement from the relations for its component parts? Let $T : \mathcal{X} \to \mathcal{Y}$ be a transformation with stability relation R_T and $M : \mathcal{Y} \rightsquigarrow \mathcal{Z}$ be a measurement with privacy relation R_M. Intuitively, the privacy of $M \circ T$ should be derived from the fact that (a) close inputs in $\mathcal{X}$ map to close elements of $\mathcal{Y}$ under T, and (b) close elements of $\mathcal{Y}$ map to close probability distributions under M. Note that we are free to choose any notion of closeness on $\mathcal{Y}$ for which these two statements hold. So the following would be a valid derivation of a privacy relation for the chained measurement $M \circ T : \mathcal{X} \rightsquigarrow \mathcal{Z}$:

$$R_{M\circ T}((d_{\mathcal{X}}, d_{\text{in}}), (D, d_{\text{out}})) = \begin{cases} \texttt{true} & \text{if } \exists (d_{\mathcal{Y}}, d_{\text{mid}})\ R_T((d_{\mathcal{X}}, d_{\text{in}}), (d_{\mathcal{Y}}, d_{\text{mid}})) \wedge R_M((d_{\mathcal{Y}}, d_{\text{mid}}), (D, d_{\text{out}})) \\ \texttt{false} & \text{otherwise} \end{cases}$$

But in general it is not feasible to implement such a relation, as there may be many choices for the pair $(d_{\mathcal{Y}}, d_{\text{mid}})$ — indeed, infinitely many if d_{mid} is vector of real numbers. So, when chaining, we must provide a hint function that specifies the intermediate notion of closeness to be used: $(d_{\mathcal{Y}}, d_{\text{mid}}) = \text{hint}((d_{\mathcal{X}}, d_{\text{in}}), (D, d_{\text{out}}))$, and we instead use the following relation:

$$R_{M\circ T}((d_{\mathcal{X}}, d_{\text{in}}), (D, d_{\text{out}})) = \begin{cases} \texttt{true} & \text{if } R_T((d_{\mathcal{X}}, d_{\text{in}}), \text{hint}((d_{\mathcal{X}}, d_{\text{in}}), (D, d_{\text{out}}))) \wedge R_M(\text{hint}((d_{\mathcal{X}}, d_{\text{in}}), (D, d_{\text{out}})), (D, d_{\text{out}})) \\ \texttt{false} & \text{otherwise} \end{cases}$$

Note that the soundness of this relation does not depend on the choice of the hint function. A poorly chosen hint function may result in a relation that outputs `false` more often than necessary, but will not make the relation output `true` when the chained measurement does not have the specified privacy property (assuming the relations R_M and R_T are sound).

Let's illustrate this by chaining $\text{BoundedSum}_{L,U}$ and BaseGauss_σ to obtain $\text{GaussSum}_{L,U,\sigma}$. We know that $\text{BoundedSum}_{L,U} : \text{MultiSets}([L,U]) \to \mathbb{R}$ is a $\max\{|L|,|U|\}$-stable transformation from d_{Sym} to $d_{\mathbb{R}}$, so here a natural hint function to use would set:

$$\text{hint}((d_{\text{Sym}}, d_{\text{in}}), (D, d_{\text{out}})) = (d_{\mathbb{R}}, \max\{|L|,|U|\} \cdot d_{\text{in}}).$$

In code:

```
def ChainingMT(trans: Transformation,meas: Measurement,hint):
  if trans.output_domain!=meas.input_domain: raise Exception('domain mismatch')
  input_metric  = trans.input_metric
  input_domain = trans.input_domain
  def function(data): return meas.function(trans.function(data))
  output_measure = meas.output_measure              # --- new ---
  def privacy_relation((m_in,d_in),(m_out,d_out)):  # --- new ---
    m_mid, d_mid = hint((m_in,d_in),(m_out,d_out))
    return (trans.stability_relation((m_in,d_in),(m_mid,d_mid))
            and meas.privacy_relation((m_mid,d_mid),(m_out,d_out)))
  return Measurement(input_metric,input_domain,output_measure,privacy_relation,function)

def MakeBoundedSum(L: float, U: float):
  # ... same as before except relation replaces stability constant ...
  def stability_relation((m_in,d_in),(m_out,d_out)) =
    if m_in==dist_sym and type(d_in)==float and m_out==dist_real and type(d_out)==float:
      return (max(abs(L),abs(U)) <= d_out)
  return Transformation(input_metric,input_domain,output_metric,output_domain,
                          stability_relation,function)

# Example
def hintGaussSum((m_in,d_in)(m_out,d_out)):
  if (m_in==dist_sym and c==approxDP): return (dist_real, max(abs(L),abs(U))*b)
BoundedSum = MakeBoundedSum(l, u)
BaseGauss = MakeBaseGauss(sig)
GaussSum=ChainingMT(BoundedSum,BaseGauss,hintGaussSum)
```

Figure 9: Chaining revisited. `ChainingMT` has been revised to define a new `privacy_relation` function that simply computes the conjunction of the relation checks for the operations being chained, with a `hint` on the intermediate distance and metric supplied by the caller. As an example, `GaussSum` is made through chaining.

We anticipate that most cases will be like the above, where the hint is obtained by just using the stability constant of the transformation. Indeed, when a transformation is stable, this information should also be provided by the relation, so that the hint function can specify just the distance measure $d_{\mathcal{Y}}$ and need not calculate the number d_{mid}. More generally, if $d_{\mathcal{Y}}$ is a univariate, monotone distance measure (i.e. where if y and y' are d-close under $d_{\mathcal{Y}}$ and $d' > d$, then y and y' are also d'-close under $d_{\mathcal{Y}}$), then a hint for d_{mid} can also be automatically obtained by doing a binary search on $R_T((d_{\mathcal{X}}, d_{\text{in}}), (d_{\mathcal{Y}}, \cdot))$.

4.6 Assumptions on the Measures

A crucial assumption about the privacy measures D is that they satisfy *post-processing*: if X and X' are random variables distributed that are d-close with respect to D and f is any function, then $f(X)$ and $f(X')$

are also d-close with respect to D. (In particular, D should be a well-defined distance measure on probability distributions over all representable objects, and not specific to their domain $\mathcal{X}$.) This allows us to implement our post-processing operator on measurements just as before.

Beyond post-processing, the proposed framework does not need to assume anything else about the measures. In particular, they need not be metrics in the standard mathematical sense. However, it may be useful to annotate the distance measures with additional properties that may be useful to exploit in some privacy analyses:

1. Symmetry: if x and x' are d-close under $d_{\mathcal{X}}$, then x' and x are d-close under $d_{\mathcal{X}}$. (Note that some of the privacy measures like D_∞ are most natural as non-symmetric measures.)

2. Monotonicity: whenever x and x' are d-close under $d_{\mathcal{X}}$ and $d' > d$, then x and x' are d'-close under $d_{\mathcal{X}}$.

3. Triangle Inequality: if x and x' are d-close under $d_{\mathcal{X}}$ and x' and x'' are d'-close under $d_{\mathcal{X}}$, then x and x'' are $(d + d')$-close under $d_{\mathcal{X}}$.

4. Nonnegativity: if x and x' are d-close under $d_{\mathcal{X}}$, then $d \geq 0$.

(Some of these only make sense when $d_{\mathcal{X}}$ measures privacy by a single real number, or at least in some ordered domain with an addition operation.) For example, the measure D_∞ used in defining pure differential privacy does not satisfy symmetry, but it does satisfy monotonicity, the triangle inequality, and nonnegativity, and the triangle inequality in particular underlies the "group privacy" property of pure differential privacy.

4.7 Composition

Let us illustrate how the basic composition of two DP mechanisms can be analyzed with different privacy measures. Composing an $(\varepsilon_1, \delta_1)$-DP mechanism and an $(\varepsilon_2, \delta_2)$-DP mechanism yields an $(\varepsilon_1+\varepsilon_2, \delta_1+\delta_2)$-DP mechanism. Composing a ρ_1-zCDP and a ρ_2-zCDP mechanism yields a $(\rho_1 + \rho_2)$-zCDP mechanism. Let $M_1 : \mathcal{X} \rightsquigarrow \mathcal{Y}$ and $M_2 : \mathcal{X} \to \mathcal{Z}$ be measurements with privacy relations R_1 and R_2. Then we can construct a privacy relation R for their composition $M : \mathcal{X} \rightsquigarrow \mathcal{Y} \times \mathcal{Z}$ where $M(x) = (M_1(x), M_2(x))$ as follows:

$$R_M((d_{\mathcal{X}}, d_{\text{in}}), (D, d_{\text{out}})) = \begin{cases} \texttt{true} & \text{if } D = \texttt{puredp} \text{ and } \exists\varepsilon_1\ R_1((d_{\mathcal{X}}, d_{\text{in}}), (D, \varepsilon_1) \wedge R_2((d_{\mathcal{X}}, d_{\text{in}}), (D, d_{\text{out}} - \varepsilon_1) \\ \texttt{true} & \text{if } D = \text{zCDP and } \exists\rho_1\ R_1((d_{\mathcal{X}}, d_{\text{in}}), (D, \rho_1) \wedge R_2((d_{\mathcal{X}}, d_{\text{in}}), (D, d_{\text{out}} - \rho_1) \\ \texttt{true} & \text{if } D = \text{approxDP, } d_{\text{out}} = (\varepsilon, \delta) \in \mathbb{R} \times \mathbb{R}, \text{ and} \\ & \ \exists\varepsilon_1\ R_1((d_{\mathcal{X}}, d_{\text{in}}), (D, (\varepsilon_1, \delta/2))) \wedge R_2((d_{\mathcal{X}}, d_{\text{in}}), (D, (\varepsilon - \varepsilon_1, \delta/2))) \\ \texttt{false} & \text{otherwise} \end{cases}$$

This composition relation is somewhat unsatisfactory because of the existential quantifiers (which can be eliminated by doing a binary search) and because the composition for approximate DP assumes that the δ is split evenly between the two measurements. Later we will describe a much more general "interactive" composition primitive that allows for the privacy parameters of the individual measurements to be precisely specified.

5 Interactive Measurements

The framework described before assumes that measurements are one-shot randomized functions $M : \mathcal{X} \rightsquigarrow \mathcal{Y}$. However, many of the useful primitives in the differential privacy literature, such as Adaptive Composition, the Sparse Vector Technique, and Private Multiplicative Weights are actually *interactive* mechanisms, which allow one to ask an adaptive sequence of queries about the dataset. Having a library that supports such interactive measurements is useful both for enabling the design of interactive query systems for end users, as well as tools for the design of even noninteractive differentially private algorithms (such as differentially private gradient descent).

Here we will describe the basic theory and implementation of interactive measurements. For sake of exposition, we will restrict attention to pure differential privacy and stable transformations on multisets as in Section 2, but it can all be combined with the more general forms of privacy relations described in Section 4.

5.1 Defining Interactive Measurements

An interactive measurement M is a (possibly randomized) function that takes a private dataset $x \in \text{MultiSets}(\mathcal{X})$ and then "spawns" a (possibly randomized) state machine $Q = M(x)$ called a *queryable.* The queryable consists of an initial private state s and an evaluation function Eval. It then receives a query q_1. Based on q_1 and its current state s_0, it generates a (possibly randomized) answer a_1 and updates its state to s_1. That is, $(a_1, s_1) \leftarrow \text{Eval}(q_1, s_0)$. It then receives a new query q_2, and similarly generates an answer and state update as $(a_2, s_2) \leftarrow \text{Eval}_M(q_2, s_1)$. And so on, arbitrarily long.

To define privacy for interactive measurements, we consider an arbitrary adversarial strategy A interacting with $M(x)$, which selects each query q_i adaptively based on all previous answers $(a_1, \ldots, a_{i-1})$ and any randomness of A. Let $\text{View}(A \leftrightarrow M(x))$ be a random variable denoting the A's *view* of this entire interaction, namely all of A's randomness and the answers to all queries. We say that M is an *ε-DP interactive measurement* if for every adversarial strategy A, and every pair $x \sim x'$ of adjacent datasets in $\text{MultiSets}(\mathcal{X})$, the random variables $Y = \text{View}(A \leftrightarrow M(x))$ and $Y' = \text{View}(A \leftrightarrow M(x'))$

$$D_\infty(Y||Y') \leq \varepsilon.$$

A noninteractive measurement M can be viewed as an interactive measurement M' where $M'(x)$ returns a queryable whose initial state is $s = M(x)$ and whose transition function always returns answer $a = s$ and does not change the state. However, there are more interesting interactive measurements, such the following *adaptive composition measurement.* Here the queryable constructed keeps the dataset x and the remaining privacy-loss budget ε in its state. It takes (noninteractive) measurements as queries, and evaluates them on the dataset if it can do so within the remaining budget, in which case it decrements the privacy loss of the query from the remaining budget.

$\text{AdaptiveComposition}_{\varepsilon,\mathcal{X}}(x)$**:** return the queryable $Q = (s_0, \text{Eval}_{\text{AC}})$ defined as follows.

- $s_0 = (x, \varepsilon)$.
- Query evaluator $\text{Eval}_{\text{AC}}(q, s)$:
    1. Parse s as $s = (x, \varepsilon)$.
    2. If q is not a syntactically valid noninteractive measurement object for data domain $\mathcal{X}$, set $a =$ `'improper query'` and $\varepsilon' = \varepsilon$.
    3. Else if $\text{privacyloss}(q) > \varepsilon$, set $a =$ `'insufficient budget'` and $\varepsilon' = \varepsilon$.
    4. Else set $a \leftarrow q(x)$ and $\varepsilon' = \varepsilon - \text{privacyloss}(q)$.
    5. Set $s' = (x, \varepsilon')$ and return (a, s').

Note that this construction uses the fact that the description of a (noninteractive) measurement includes its privacy-loss parameter (ε) and that this is a publicly exposed attribute.

As we see in the above example, to describe an interactive measurement operator, we need to specify:

1. The domain $\mathcal{X}$ of its data records.
2. The privacy loss ε.
3. The (possibly randomized) function f that generates a queryable from a dataset.

An example implementation of AdaptiveComposition in code is as follows:

```
class InteractiveMeasurement:
  input_domain
  privacy_loss
  function     # --- now the function will output a Queryable ---

class Queryable:
  _state       # --- underscore indicates that the state should be private ---
  eval
  def query(q):
    (a, _state)=eval(q,_state)
    return a

def AdaptiveComposition(dom,epsilon:float):
  input_domain = dom
  privacy_loss = epsilon
  def function(data):
    initial_state=(data,epsilon)
    def eval(query: Measurement, state):
      (st_data, eps) = state
      if query.input_domain!=dom: return ('domain mismatch',eps)
      elif query.privacy_loss > eps: return ('insufficient budget',eps)
      else return (query.function(st_data),eps-query.privacy_loss)
    return Queryable(initial_state,eval)

  return InteractiveMeasurement(input_domain,privacy_loss,spawn)

# Example
queryable_obj=AdaptiveComposition(float,2).function(dataset)
res1=queryable_obj.query(NoisySum)
res2=queryable_obj.query(NoisyCount)
```

Figure 10: Adaptive Composition

In the example of AdaptiveComposition, the only portion of the state $s = (x, \varepsilon)$ that needs to be kept secret is the dataset x, and we can safely augment Eval with additional queries that allow asking for remaining privacy-loss budget ε. However, other interactive queryables do have additional state that needs to be kept secret for their privacy properties (e.g. the noisy threshold in the Sparse Vector Technique).

5.2 Post-processing

Recall that standard post-processing refers to performing an arbitrary computation with the privacy-protected output of a noninteractive measurement operator, For interactive measurements, we think of the queryable itself as the analogue of the "privacy-protected output" of the measurement operator — no matter how one computes with the queryable (as a black box, without examining its internal state!), privacy is maintained. This gives rise to a post-processing principle for interactive measurements. Specifically we can apply a queryable mapping function P that takes the queryable $Q = M(x)$ produced by M and produces a new queryable $Q' = P(Q)$. Importantly, P does not get to examine the internals of the queryable Q, only interact with it as a (stateful) black box, issuing queries and receiving answers. P can also embed the queryable Q *inside* the queryable Q', so that whenever Q' receives a query, it can issue some queries to Q to help compute an answer. Note that Q continually updates its state as P and then Q' issues queries to it. Postprocess(M, P) is the interactive measurement $M'(x) = P(M(x))$ that outputs Q'.

The reason privacy is preserved under this interactive form of post-processing is that for every adversary A interacting with with Postprocess(M, P), there is an adversary A^{in} interacting with M and a function f^{out} such that for every dataset x,

$$\text{View}(A \leftrightarrow \text{Postprocess}(M, P)(x)) = f^{\text{out}}(\text{View}(A^{\text{in}} \leftrightarrow M(x))).$$

That is, views of an adversary interacting with the post-processed mechanism can be obtained by applying a function to the view of an adversary interacting with the original mechanism, and thus differentially privacy of the latter implies differential privacy of the former.

We illustrate this interactive post-processing with two examples. First, we will show how to obtain an instantiation of a differentially private Statistical Query (SQ) Model as a post-processing of AdaptiveComposition, and then we will see how to obtain an instantiation of noisy gradient descent as a post-processing of the SQ model.

Statistical Query Model. In the SQ Model, we are given a dataset $x \in \text{MultiSets}(\mathcal{X})$, and an analyst can issue up to T queries that can be issued are bounded functions $f : \mathcal{X} \to [-B, B]$ and obtain noisy estimates of the average $\text{E}_{z \leftarrow x}[f(z)] = (\sum_{z \in x} f(z))/|x|$. We will implement this using $T + 1$ queries to an AdaptiveComposition$_{\mathcal{X},\varepsilon}$ queryable, spending half the budget on an initial query to estimate the size of the dataset, and then dividing the remaining budget evenly over the T remaining queries to estimate the sum $\sum_{z \in x} f(x)$. In our implementation, we won't require that the queries f are bounded as specified, but will rather enforce it by evaluating the sums using NoisyClampedSum.

To this end, we will assume that for any function $f : \mathcal{X} \to \mathcal{Y}$, we can construct the stability 1 transformation RowTransform$_f$: MultiSets$(\mathcal{X}) \to$ MultiSets$(\mathcal{Y})$ defined as

$$\text{RowTransform}_{\mathcal{X},\mathcal{Y},f}(z) = \{f(w) : w \in z\} \text{ (with multiplicity).}$$

When $\mathcal{Y} = \mathbb{R}$, we can then use chaining to implement the ε-DP mechanism

$$\text{NoisySumFunction}_{f,L,U,\varepsilon} = \text{NoisyClampedSum}_{L,U,\varepsilon} \circ \text{RowTransform}_{\mathcal{X},\mathbb{R},f} : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \mathbb{R}.$$

(We note that allowing arbitrary user-provided code to specify functions f can lead to severe side-channel attacks. We discuss how to address such attacks in Section 7.)

In code, we have:

```
def Postprocess(intMeas: InteractiveMeasurement,queryable_map):
  input_domain = intMeas.input_domain
  privacy_loss = intMeas.privacy_loss
  def function(data):
    queryable_inner=intMeas.function(data)
    return queryable_map(queryable_inner)
  return InteractiveMeasurement(input_domain,privacy_loss,function)

# Example

def MakeRowTransform(in_dom, out_dom, f):
  ...

def MakeNoisySumFunction(in_dom,f,L,U,epsilon):
  return(ChainingMT(MakeNoisyClampedSum(L,U,epsilon),
                    MakeRowTransform(in_dom, float, f)))


def MakeSQmodel(in_dom,T,B,epsilon):
  def queryable_map(AC_queryable):
    eps=epsilon/2
    def sum_query(x): return 1
    n=AC_queryable.query(MakeNoisySumFunction(in_dom,sum_query,-1,1,eps))
    initial_state=T
    def eval(query, state):
      if state>0:
        answer=AC_queryable.query(MakeNoisySumFunction(in_dom,query,-B,B,eps/T))/n
      else
        answer='no more queries'
      fi
      return (answer,state-1)
    return Queryable(initial_state,eval)
  return Postprocess(AdaptiveComposition(in_dom,epsilon),queryable_map)
```

Figure 11: Differentially Private SQ Model

Now we use our implementation of the SQ Model to obtain an instantiation of noisy gradient descent via post-processing. Suppose we have a dataset $z \in \text{MultiSets}(\mathbb{R} \times \mathbb{R})$ and our goal is to find a differentially private slope θ minimizing $L_z(\theta) = \text{E}_{(x,y)\in z}[\ell_{(x,y)}(\theta)]$, where $\ell_{x,y}(\theta) = (\theta x - y)^2$. In each iteration of the algorithm, we will take our current estimate θ, compute a differentially private estimate a of the derivative $L'_z(\theta) = \text{E}_{(x,y)\in z}[\ell'_{x,y}(\theta)]$, where $\ell'_{x,y}(\theta) = 2\theta \cdot (\theta x - y)$, and update to $a \leftarrow \theta - \eta a$, for a "learning rate" parameter η. After a given number T of iterations, we simply output the current value of θ, embedded in the state of a "noninteractive queryable" that will return θ on any query. Notice that in each iteration, we are simply computing an average over the dataset, which we can estimate using a query to an SQ model queryable. We will also need to specify a bound B to be enforced by the SQ model.

In code:

```
def NoninteractiveQueryable(initial_state):
    def eval(q,s): return (s,s)  # always return the state and never change it
    return Queryable(initial_state,eval)

# Example

def MakeNoisyGD(T,B,eta,epsilon):
  def queryable_map(SQ_queryable):
    theta=0
    for i in range(T):
      def derivative(z : float*float): return (2*theta*(theta*z[0]-z[1]))
      a=SQ_queryable.query(derivative)
      theta=theta-eta*a
    return NoninteractiveQueryable(theta)
  return Postprocess(MakeSQmodel(float*float,T,B,epsilon),queryable_map)
```

Figure 12: Noisy Gradient Descent

The examples of AdaptiveComposition and NoisyGD show that it is useful to build interactive measurements from noninteractive ones and vice-versa. Thus, we propose that noninteractive measurements are implemented as a special case of interactive measurements, to avoid duplicating operators (composition, chaining, post-processing, etc.) that can simultaneously deal with both kinds of measurements.

5.3 Chaining and Composition of Interactive Measurements

Chaining generalizes in a straightforward way to interactive measurements. If $T : \text{MultiSets}(\mathcal{X}) \to \text{MultiSets}(\mathcal{Y})$ is a c-stable transformation and M is an ε-DP interactive measurement for data domain $\mathcal{Y}$, then their chaining $M \circ T$ is a $c\varepsilon$-DP interactive measurement for data domain $\mathcal{X}$.

As far as composition, it is natural to extend the AdaptiveComposition procedure described above to allow queries that can be *interactive* measurement operators themselves. For example, we should be able to issue a query q_i describing an interactive measurement operator M_{q_i} that spawns an "inner queryable" $M_{q_i}(x)$ within the Adaptive Composition queryable, to which we can issue subsequent queries. We can then choose to allow for either:

1. *Sequential Composition:* all of the queries to the first inner queryable must be completed before another inner queryable is spawned.

2. *Concurrent Composition:* multiple inner queryables can be spawned and be simultaneously active, with queries to them arbitrarily interleaved.

The basic, additive composition of privacy loss for pure differential privacy applies for both sequential composition and concurrent composition; indeed, PINQ allows for concurrent composition and a formal proof of its correctness is given in [3].

Another restriction of AdaptiveComposition as described above is that when specifying the i'th query $q_i = M_i$, the privacy-loss $\varepsilon_i = \text{privacyloss}(q_i)$ is also specified, and is effectively "consumed" immediately when $\text{Spawn}_{q_i}(x)$ is executed. In contrast, in PINQ, each "inner queryable" does not have a preallocated budget but separately tracks its accumulated privacy loss, and these are summed in order to track the overall privacy loss. To support this in our proposed framework, we would need to implement a theory of *privacy odometers* [12], which are variants of interactive measurements that track accumulated privacy loss rather than specify a total privacy loss at the beginning. It should be relatively straightforward to modify the programming framework to implement composition, chaining, and post-processing for pure-DP odometers, as well as conversion of odometers to standard interactive measurements (by not answering a query if it will make the accumulated privacy loss exceed the budget).

However, we remark that the aforementioned generalizations of composition appear to be more complex for variants of differential privacy (such as approximate differential privacy). Indeed, the Advanced and Optimal Composition Theorems for approximate differential privacy do not hold as is for approximate-DP odometers, and in fact they require the entire sequence of privacy-loss parameters $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_T$ to be specified before any queries are made. (Specifying the sequence of privacy-loss parameters in advance is often a good thing in any case, as it allows for verifying that the budget won't inadvertently be consumed before an analysis is complete.) As far as we know, there has also been no rigorous analysis of concurrent composition for approximate differential privacy or other variants of differential privacy; this also seems to be an important direction for future work. (The cryptography literature indicates that concurrent composition is often much more subtle than sequential composition.)

6 Putting it Together

In Section 2, we described a basic framework of pure DP measurements and stable transformations on multisets. In Sections 3, 4, and 5, we described three orthogonal extensions to the basic framework:

1. A framework for ensuring that measurements and transformations have their claimed privacy or stability properties, by only allowing procedures that have been vetted (e.g. through the OpenDP editorial board reviewing a written proof) the right to directly construct measurements and transformations.
2. Allowing for many sensitive datatypes (not just multisets), many measures of distance between sensitive data items (not symmetric difference), and many measures of closeness between probability distributions (not just the multiplicative notion used to define pure differential privacy). With this generalization, stability and privacy are no longer measured by single parameters, but are given by *relations* that tell us whether a given kind of "closeness" of inputs implies a given kind of "closeness" of outputs.
3. Generalizing measurements to spawn interactive query procedures (randomized state machines called *queryables*). This allows for modelling adaptive composition theorems as measurements, as well as more sophisticated interactive DP algorithms. It also provides for in-library (and thus vetted) mechanisms for managing privacy budgets, rather than delegating this to external systems.

Although we described these extensions separately for sake of exposition, we propose that all three be combined in the OpenDP Programming Framework. They combine with each other in natural ways:

- Our definition of privacy for interactive measurements readily extends to other privacy notions, in particular by simply asking that the views of the adversary on two input data items (that are close according to some distance measure) are close according to any desired notion of closeness between probability distributions (capturing, for example, approximate differential privacy or concentrated differential privacy). We can replace the Adaptive Composition interactive measurement for pure DP with other interactive composition procedures that can be analyzed using other privacy notions, and the resulting bounds would be encoded by the privacy relation.
- The verification principle from Section 3 can and should be applied to more general measurements and transformations from Sections 4 and 5. In particular, now a measurement or transformation is considered *valid* if its privacy or stability relation is sound: it should never claim that a false privacy or stability bound holds.

7 Ensuring Privacy in Implementation

Like with any other implementation of security or privacy critical code, we need to beware of places where the mathematical abstraction used in the privacy analysis may deviate from the implementation.

For example, in the framework above we abstract transformations as functions $T : \mathcal{X} \to \mathcal{Y}$ and measurements as randomized functions $M : \mathcal{X} \rightsquigarrow \mathcal{Y}$. Implicit in this modelling is that when executing $T(x)$, the only private information it has access to is x, and that it does not produce any observable output other than $T(x)$. That is, we need to prevent leakage of information through *side channels*.

One known side channel of concern is the *timing channel*, where the amount of time to execute a function reveals information about its input. Standard mitigations against the timing channel include introducing delays or truncating long executions to make a function essentially constant time. This problem and side channels in general are exacerbated when a transformation or measurement is built using a user-defined function f (which may come from an interactive query issued by an adversary). For example, $\mathrm{RowTransform}_f$ can cause privacy leakage if the function f can write to an I/O device observable to the adversary or if the function f intentionally changes its execution time based on the data record it is fed. Side-channel attacks using user-defined functions can be mitigated by some programming language support. (See Section 8.)

Another concern comes from the implementation of arithmetic. Like in much of statistics, in the mathematical analysis of differential privacy, we often model data using real numbers, use continuous noise distributions, and allow probabilities that are arbitrary real numbers in $[0, 1]$. Indeed, we followed this tradition above, starting from the beginning of Section 2 when we presented NoisySum as taking input datasets whose records are elements of the real interval $[L, U]$ and as adding continuous Laplace noise to the output. In the code examples, however, we used floating-point numbers as a proxy for real numbers. Unfortunately, as shown by Mironov [9] the discrepancy between floating-point numbers and true real numbers leads to severe violations of differential privacy. For this reason, following [1], we advocate the use of *fixed-point* numbers and integer arithmetic as much as possible in the OpenDP library. For code that is in the fully certified library, any deviations from standard arithmetic (e.g. overflow of floating-point arithmetic) should be explicitly modelled in the privacy analysis. (There may be versions of the library that are subject to less rigorous constraints for the purpose of prototyping and experimentation.)

Differentially private algorithms also rely crucially on high-quality randomness. To avoid this being a source of vulnerability, the OpenDP library should use a cryptographic-strength pseudorandom generator for its random bits.

8 Discussion of Implementation Language(s)

The choice of programming language, or languages, for the OpenDP library implementation is an important one. It has implications on performance, reliability and also impacts the degree to which the library will achieve each of the goals identified in the introduction. This section identifies some features of the programming language that might be particularly beneficial and offers a concrete proposal for OpenDP. It is likely that no single language meets all criteria and that different components of the library will be implemented using different languages. Our initial prototyping was done using Python with a mix of object-oriented and functional paradigms, as reflected in the code examples above, and many of the features identified below are a reflection of that experience.

Section 1 outlines high-level goals including *usability* and *programmability*. To support the goal of usability, it will be important that the library offer APIs in languages that are commonly used in OpenDP's target use cases. The most common (open-source) languages for statistical/data analysis include Python and R. By providing APIs in one or both of these languages, programmers will be able to integrate the library into the conventional data workflows. There will also be a need to integrate with a variety of back-end data access layers, which may necessitate re-implementing some core functionality in a back-end compatible language such as as SQL. It is anticipated that this re-implementation would be limited to data-intensive transformations, though prior work has also explored implementing measurements in the data layer [6]. To support the goal of programmability, it may be preferable to select a language in the more familiar imperative paradigm rather than adopt a purely functional language, though as noted below, choosing a language that has support for some functional features may be desirable.

8.1 Desired language support

Below are some specific programming language features that are desired.

Memory safety, encapsulation and immutability. Given the library's principal goal of mediating access to private data, securing the data against unauthorized access is a paramount concern. While systems hardening will likely happen at many layers, some features at the programming language level can help avoid inadvertent data leakage and reduce the verification effort for new contributions. These features include memory safety and state encapsulation (in the object-oriented sense). For example, it should be impossible for the user to access the internal state of a queryable. In addition, the ability to control mutability may be desirable. For instance, if a query is implemented as a mutable object, then the programmer could potentially manipulate the state of the query object *while it is being evaluated by the queryable.* Immutable structures may help avoid programmer mistakes and make it easier to automatically analyze a program for its safety.

Abstract (parameterized) data types. Many of the code examples presented earlier make use of specialized data types. In many cases these types are similar to conventional types but with additional constraints on allowable values (e.g., a float that is bounded). Being able to precisely constrain the data type has tangible benefits in the context of a DP library: it often simplifies privacy semantics for individual functions and invites more modular design (upstream transformations often refine the type to make downstream processing easier). It also reduces the burden for verifying contributed functions: since the privacy promise is conditional on type, narrower types reduce the scope of the claim to be verified. But to do all of this, the library needs support from the programming language to enable the creation of specialized data types. Further, these types are often parameterized (e.g., the bounds of a bounded float may not be specified until runtime).

Type safety. Strictly speaking, the library provides its own layer of type enforcement (e.g., chaining verifies type compatibility between chained functions). However, it may be beneficial to use a language that helps programmers write type-safe code, so languages that are strongly and statically typed may be preferable.

Functional features and "pure" functions. A number of functional paradigm features proved quite useful in prototyping, including: higher-order functions, function composition (chaining), and partial function evaluation. Some transformations, such as RowTransform, take an arbitrary, user-defined function and apply it to the data. To ensure privacy, the library must enforce that this function is free of side effects, as discussed in Section 7. This can be achieved by purifying user-defined functions (as is done in PINQ [8]) or requiring users to express computations in a restricted domain-specific language (as is done in Psi [4]).

Generics. We would like to allow the programmer to construct general-purpose transformations (or measurements) that are agnostic to the data type to which they are applied. An example where this would be useful is RowTransform: ideally one implements a single generic routine that applies the user-defined function to map a multiset of one type to a multiset of another type, where the types a generic and can be specified at compile/run time. This kind of behavior can be supported even in statically typed languages through the use of generics (parametric polymorphism).

Structures or object-orientation. In our implementation, we took advantage of the object-oriented support of Python to define classes that allowed us to link together related functionality (e.g., a function and its privacy relation) into a single object that can be treated as a unit. But similar functionality is present in languages that are not considered object-oriented (e.g., Rust).

8.2 Proposal: Rust + Python

We and the OpenDP Design Committee propose to implement the core library using Rust. Rust is a fast-growing, imperative functional language. It is both high performance and reliable with good memory safety and type safety. It also meets many of the desiderata identified previously: structures, generics, higher-order functions, and values are immutable by default. It has a strong ownership model that gives memory safety without the necessity of garbage collection. Collectively, this makes Rust code much easier to reason about concretely. For example, there are no dangling pointers, there are no memory races as there are in C++.[3]

Where Rust may fall short is in the programmability criteria, as it is not a widely used language (though interest in it is growing, and it has been ranked the "most loved" language on Stackoverflow, edging out Python for the last four years) and it can be challenging to program (at least in part because of the built-in safety features).

The OpenDP project has already been developing an end-to-end differential privacy system as a joint project with Microsoft, and for this project required a library to stand in while the core OpenDP library is not yet ready. The development team built a prototype library in Rust, using ideas from the PSI library [4] and some advances in the direction of those proposed in this paper. All of the developers were new to Rust, and all picked it up straightforwardly. Also, one graduate student who wished to contribute code from a new research paper was able to quickly port their code to Rust with some small direction.

Our proposal is that Rust should be the language for the robust, polished, production code that enters the maximally trusted OpenDP library. However, to encourage contributions from researchers and other developers, as well as to encourage experimentation and sandboxing of performance, the library should be able to work with Python and R code components. We propose to enable this through Python and R bindings (through a Protocol Buffer API described next). Initial code could be contributed in Python (and R), with the goal that final, vetted and performant code be ported to Rust, either by contributors when possible, or by the core OpenDP development team, or in collaboration.

APIs. Rust uses the same Foreign Function Interfaces as C++ for bindings so the bindings compiled from either Rust or C++ are identical. However, in the OpenDP project with Microsoft, the code to write these bindings has been found to be clearer than in C++. That project has created Python bindings to the prototype differential privacy library that have been heavily tested and work well. R bindings are in construction and have been found to be straightforward too.

In that project, an API to the library was built using Protocol Buffers[4], a serialized data language, like JSON or XML, but heavily typed and very compact. This allowed for defining any sequence of calls to the library as a plan, in the style of εktelo or more specifically, a computational graph of library components. The protobuf API calls Rust components for the runtime. The Python and R bindings also call the API which calls the Rust library. In the case of Python, these bindings can be distributed on PyPI and pip installed, and executed in a Python notebook seemingly natively, although the actual execution is in Rust. The OpenDP Design Committee proposes to release a Protocol Buffer API for the core library as part of the OpenDP Commons, by which it will also offer Python and R bindings. This API would facilitate other possible bindings, as well a way to call the library in more complicated system architectures.

9 Using the Library

The programming framework proposed in this paper is meant to capture all the mathematical reasoning about privacy needed for use of differential privacy. However, we do not expect it to be directly used by end users; instead it is meant as a substrate to build larger, user-friendly end-to-end differential privacy systems. Here we discuss some aspects of how such a system should use the library envisioned here.

Calling the Library from a DP System.

1. The system should first determine, or elicit from its user, the following:

    (a) The sensitive dataset on which differential privacy is going to be applied.

[3]The Microsoft Security Response Center (MSRC) released reports stating that 70% of security vulnerabilities addressed through Microsoft security patches are memory safety issues that would have been avoided in Rust,`https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/`, and explicitly suggests moving away from C++ to Rust `https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/`.

[4]`https://developers.google.com/protocol-buffers/`

(b) The type of the dataset (e.g. is it a multiset of records, or social network graph, or a collection of tables). Do the records consist of a known set of attributes?

(c) The granularity of privacy protection. For example, if we want to protect privacy at the level of individual human beings, we need to know whether every individual human corresponds to at most one record in the multiset, at most one node in a social network graph, or at most one key in a collection of tables.

(d) The privacy measure to be used (e.g. pure, approximate, or concentrated DP), and the desired privacy-loss parameters.

Regarding the type of the dataset, it is best for the system to make as few assumptions as possible about the data type, as long as the granularity of privacy can be well-specified, to reduce the chance that an incorrect assumption leads to a violation of privacy. For example, it is safer to assume that the dataset is a multiset of records of arbitrary type or a multiset of strings, rather than assuming it is a multiset of records that each consist of 10 floating-point numbers in the interval $[0, 1]$. Additional public, non-sensitive information about the dataset (e.g. a full schema) can typically be exploited later for greater utility without being used for the privacy calculus. Note that specifying the granularity of privacy requires keeping information in the datasets that one might be tempted to remove beforehand — in order to enforce privacy at the level of individuals in a database where an individual might contribute to multiple records, we need the data records to have a key that corresponds to distinct individuals.

2. Then the system should select an interactive measurement family from the library (typically some form of adaptive composition) that is compatible with the data type, granularity of privacy, and privacy measure, and use it to spawn a queryable on the sensitive dataset that will mediate *all* future access to the dataset.

3. If additional type information about the dataset is known (e.g. about the individual records, or even a more complex property like being a social network of bounded degree), then before applying any queries to the dataset, the library can be used to apply a stable "casting" transformation that ensures that the dataset actually has the specified properties. For example, for multisets we can apply $\mathrm{RowTransform}_f$ for a function f that maps any record that does not have the specified type to a default value with that type.

4. All queries to the dataset should be made through the initial queryable. The queryable automatically ensures that we stay within the initially specified privacy-loss budget. Deciding how to apportion the budget among current and future queries is for the system to decide, but we expect that statistical algorithms in the library will expose their utility in ways that assist a system in making such decisions (e.g. through a priori utility estimates provided analytically or through benchmarking).

Exposing System Features to the Library. To make efficient and safe use of the library, it is also important to expose some system features to the library:

- Exposing the data-access model to the library. What means do the library functions have to read the sensitive data? If, for example, the data is only available in a streaming format, that will limit the choice of methods in the library that can be applied.

- Exposing the capabilities of the back-end data system to the library. Global data transformations, aggregates, sorting, and joins can be performed much more efficiently if they are done by an optimized database engine than if the data needs to be extracted from the database and computed on externally. Thus, there should be a way of identifying which transformations in the library can be done by the database system. On the other hand, this means that the resulting privacy properties also depend on the database engine faithfully and securely implementing the specified transformation, without observable side effects (including timing) that might leak sensitive data. More complex capabilities that the database may have include randomized procedures like subsampling.

- Defining the partition between secure and insecure storage and compute according to the threat model (to what might the privacy adversary have access), so that private data does not cross the "privacy barrier" until after the protections of differential privacy have been applied. Recall that in addition to the original sensitive dataset, the results of transformations, and the state of queryables need to be kept secret from the privacy adversary.

User interfaces. A system should also provide a user-friendly way for data custodians and data analysts without expertise in differential privacy to make effective use of the library. In particular, we envision that many existing DP systems can be built as layers on top of the library, using the certified code in the library for all the privacy analysis. Such interfaces could include:

- A SQL-like interface as in PINQ and εktelo, where queries are specified as plans that are compiled into DP measurements.
- A GUI for selecting from a predefined set of statistics and transformations, as in PSI.
- A Python-like notebook interface for issuing queries to the data, like in the OpenDP System being built in collaboration with Microsoft.

Another important aspect of the user interface is how accuracy and statistical confidence are exposed to users (so that analysts don't draw incorrect conclusions due to the noise for privacy), and it is important that the algorithms in the library provide support for this. We defer discussion of this to the Statistics section of the OpenDP Whitepaper.

10 Contributing to the Library

Building on the discussion on Section 3, we elaborate here on the different types of code contributions we envision to the library.

1. A new measurement or transformation family constructed by combining existing measurement and transformation families in the library using operators that already exist in the library (like composition, chaining, and post-processing).

   This is the easiest kind of contribution, as the certificates of privacy or stability are automatically generated by the built-in components, and we also expect it to be the most frequent contribution once the library has a large enough base of building-block components. Although no proof of privacy or stability is needed for such contributions, they will still need to be reviewed for having demonstrated utility (see the Statistics section of the OpenDP Whitepaper), code quality, and documentation.

2. A new "atomic" measurement or transformation family, where the privacy or stability properties are proven "from scratch."

   For the sake of modularity and ease of verification, such contributions should be broken down into as small and simple sub-measurements or sub-transformations as possible, each of which has a proof from scratch. Together with each such a contribution, one must also provide a mathematical proof that it actually achieves its claimed privacy or stability properties. One day, we hope that some of these proofs can be formally verified using the rapidly advancing set of tools for formal verification of differential privacy. However, in many cases, we expect that what will be provided is a written proof together with a pseudocode description of the algorithm and its claimed privacy properties. The DP researchers on the OpenDP editorial board will be responsible for verifying that the proof is correct for the pseudocode. The OpenDP Committers, with expertise on the programming framework and its implementation, will be responsible for verifying that the actual code is a faithful implementation of the pseudocode.

Sometimes the privacy or stability proof of a new contribution will be based on some property of an existing component in the library that is not captured by the privacy or stability properties of that component. For example, "stability methods" for approximate differential privacy (like "propose–test–release") often rely on the *accuracy* of other DP mechanism (e.g. that with probability at least $1 - \delta$, the Laplace mechanism produces an estimate that is correct to within $c \log(1/\delta)/\varepsilon$). Whenever this occurs, there should also be a proof provided that the said component has the given property, this proof should be attached as a certified assertion to the implementation of that component, and the privacy or stability relation for the new contribution should check the certificate. This way, if the component later changes and the assertion has to be removed, the soundness of the contribution relying on it is preserved.

3. A new combining primitive for measurements or transformations.

   This can be a new form or analysis of composition or chaining, or something more sophisticated like privacy amplification by subsampling (viewed as an operator on measurements). In the proposed framework, these are also measurement or transformation families and the process for contributing them and having them vetted and accepted is the same as above.

4. Adding a new private data type, distance measure between private data types, or privacy notion (distance measure between distributions).

   There should be a standardized format for adding such elements to the library, and what information needs to be provided and checked (again by a combination of the editorial board and committers). For example, for a new distance measure, one may provide a statement of the data types to which it applies, specify whether it satisfies properties like monotonicity and symmetry, give conversions between it and other distance measures, etc. For a new privacy notion, it needs to be verified that it satisfies the post-processing property.

5. Adding a new type of privacy or stability calculus that is not supported by the existing framework.

   Examples include introducing frameworks for privacy odometers, local sensitivity and variants (like smooth sensitivity), or per-analyst privacy budgets. These will require a more holistic architectural review before being incorporated.

11 The Scope of the Framework

The programming framework presented in Sections 2–6 is meant to be very general and flexible, and we believe it can support a great deal of the privacy calculus in the differential privacy literature. However, it does not support all important ways of reasoning about differential privacy. Thus, below we list some concepts that we believe can be easily supported and others that would require an expansion of the framework.

11.1 Within the Current Framework

Many different private dataset types and granularities of privacy. This includes unbounded DP (multisets, adjacency by adding or removing a record), bounded DP (datasets consisting of a known number of records, adjacency by changing a record), networks with edge-level or node-level privacy, a data stream with person-level or event-level privacy, multirelational databases with person-level privacy, DP under continual observation.

Many different measures of privacy loss. This includes pure DP, approximate DP, zero-concentrated DP, Rényi, and Gaussian DP. (We require the privacy-loss measure to be closed under postprocessing, so the original formulation of concentrated DP would not be supported.) Group privacy is also captured by the proposed privacy relations, as it allows for interrogating about the privacy loss at inputs of different distances.

Combining primitives to build complex mechanisms from simpler ones. This includes many of the sophisticated composition theorems for differential privacy (such as the advanced and optimal composition theorems for approximate differential privacy, the composition of zCDP, and more), "parallel composition" via partitioning, privacy amplification by subsampling, the sample-and-aggregate framework.

Global-sensitivity-based mechanism families. The exponential mechanism, the geometric mechanism, the (discrete) Gaussian mechanism, the Matrix Mechanism, and the Sparse Vector Technique.

Common database transformations. Subsetting according to a user-defined predicate, partitioning according to a user-defined binning, clamping into a given range/ball, vectorizing into a histogram, join (or variant) on a multi-relational dataset, user-defined row-by-row transformations, and summing over a dataset.

Restricted sensitivity and Lipschitz projections. Both of these refer to (global) stability properties of transformations with respect to rich data types, which the framework supports.

11.2 Outside the Current Framework

Below are some general concepts in differential privacy that the proposed framework does not currently support reasoning about. When these concepts are used only as an intermediate step to proving a standard privacy property, then the entire mechanism can be added to the library as a measurement. But the framework would not support reasoning directly these concepts without further augmentation.

Local sensitivity and mechanisms based on it. We envision that the local sensitivity of a transformation could be captured by adding a "local stability" relation to the attributes of a transformation, which takes a dataset as an additional argument. Such a relation would also need to be verified similarly to the already-proposed stability relations. By appropriate augmentations like this to the framework, we hope it will be possible to support reasoning about differentially private mechanisms based on variants of local sensitivity, such as smooth sensitivity, stability-based mechanisms, and propose-test release.

Privacy odometers. These are variants of interactive measurements that track accumulated privacy loss rather than specify a total privacy loss at the beginning [12]. As discussed in Section 5, it should be relatively straightforward to modify the programming framework to incorporate at least pure-DP odometers.

Complex adversary models. These include pan-privacy with a limited number of intrusions, computational differential privacy, adversaries with limited collusion (e.g. when we give different analysts budgets of their own)

Randomized transformations. It is possible and sometimes useful to reason about stability properties of randomized transformations, often via couplings. For example, we can call a randomized transformation $T : \text{MultiSets}(\mathcal{X}) \rightsquigarrow \text{MultiSets}(\mathcal{Y})$ *c-stable* if for every two adjacent datasets $x \sim x'$, there is a coupling (Y, Y') of the random variables $Y = T(x)$ and $Y' = T(x')$ such that it always holds that $d_{\text{Sym}}(Y, Y') \leq c$. The framework should be easily extendable to this way of capturing the stability properties of randomized transformations, but more complex coupling properties may be more challenging to capture.

Interactive transformations. In some DP systems, like PINQ, transformations can also be performed in an interactive and adaptive manner, creating a multitude of derived datasets (behind the "privacy barrier") on which queries can subsequently be issued. The framework proposed so far only allows for measurements to be interactive, which may mean that the same derived dataset is recomputed multiple times if it is an intermediate dataset in multiple measurements formed by chaining. This can be potentially remedied by modifying the execution engine to retain previously derived datasets and recognize when they are being recreated. However, it may eventually be useful to extend the programming framework to directly work with some form of interactive transformations, just as we allow interactive measurements. (But a general theory of interactive transformations and how they may be combined with measurements may need to be developed first.)

References

[1] V. Balcer and S. Vadhan. Differential Privacy on Finite Computers. *Journal of Privacy and Confidentiality*, 9(2), September 2019. Special Issue on TPDP 2017. Preliminary versions in ITCS 2018 and posted as arXiv:1709.05396 [cs.DS].

[2] M. Bun and T. Steinke. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In M. Hirt and A. D. Smith, editors, *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 635–658, 2016.

[3] H. Ebadi and D. Sands. Featherweight pinq. *Journal of Privacy and Confidentiality*, 7(2), 2016.

[4] M. Gaboardi, J. Honaker, G. King, J. Murtagh, K. Nissim, J. Ullman, and S. Vadhan. Psi ({\Psi}): a private data sharing interface. *arXiv preprint arXiv:1609.04340*, 2016.

[5] N. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment*, 11(5):526–539, 2018.

[6] N. M. Johnson, J. P. Near, J. M. Hellerstein, and D. Song. Chorus: Differential privacy via query rewriting. *CoRR*, abs/1809.07750, 2018.

[7] I. Kotsogiannis, Y. Tao, X. He, M. Fanaeepour, A. Machanavajjhala, M. Hay, and G. Miklau. Privatesql: a differentially private sql query engine. *Proceedings of the VLDB Endowment*, 12(11):1371–1384, 2019.

[8] F. D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30. ACM, 2009.

[9] I. Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 650–661, 2012.

[10] C. Palamidessi and M. Stronati. Differential privacy for relational algebra: Improving the sensitivity bounds via constraint systems. *Electronic Proceedings in Theoretical Computer Science*, 85:92–105, Jul 2012.

[11] J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In P. Hudak and S. Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 157–168. ACM, 2010.

[12] R. Rogers, A. Roth, J. Ullman, and S. Vadhan. Privacy odometers and filters: Pay-as-you-go composition. In *Advances in Neural Information Processing Systems 29 (NIPS '16)*, pages 1921–1929, December 2016. Full version posted as arXiv:1605.08294 [cs.CR].

[13] D. Zhang, R. McKenna, I. Kotsogiannis, M. Hay, A. Machanavajjhala, and G. Miklau. Ektelo: A framework for defining differentially-private computations. In *Proceedings of the 2018 International Conference on Management of Data*, pages 115–130. ACM, 2018.