A Programming Framework for OpenDP (extended abstract)*

Marco Gaboardi† Michael Hay‡ Salil Vadhan§

In this paper, we propose a programming framework for the library of differentially private algorithms that will be at the core of the new OpenDP open-source software project (`http://opendp.io/`). 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.

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.

Verifiability External code contributions need to be verified to actually provide the differential privacy properties they promise.

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.

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).

Programming Framework Components

We follow the tradition of systems such as PINQ [2] and Ektelo [3], and we identify *measurements* and *transformations* as the two main kinds of operators.

Measurements A *measurement* M is a randomized mapping from datasets to outputs of an arbitrary type. A measurement operator M and its privacy properties are specified by 5 attributes:

1. `input_domain`, describing the kind of data the measurement expects as input.
2. `input_metric`, describing the metric (or adjacency relation) that is used for inputs.
3. `output_measure`, the similarity measure that the mechanism guarantees on the output distributions.
4. `privacy_relation`, the relation asserting the privacy properties of the measurement
5. `function`, the randomized function implemented by the measurement.

The structure of a measurement operator favors generality. By changing the different attributes we expect to be able to express most of the useful current and future privacy mechanisms that satisfy variants of differential privacy. One key design choice here is that we assert privacy through a *privacy relation* between an input distance d_i and an output distance d_o and certifying that "for all x, x' in the input domain, if x is d_i-close to x' under the input metric, then the outputs of running the function on x and running it on x' are d_o-close under the similarity output measure".

A crucial assumption about the output similarity measures is that they satisfy *post-processing*. Beyond post-processing, the proposed framework does not need to assume anything else about the output measures.

Transformations A *transformation* is a (deterministic) mapping from datasets to datasets. A transformation T and its stability properties are specified by six attributes:

1. `input_domain`, describing the kind of data the transformation expects as input.
2. `input_metric`, describing the metric (or adjacency relation) that is used for inputs.
3. `output_domain`, describing the kind of data the transformation produces as output.
4. `output_metric`, describing the metric (or adjacency relation) that is used for outputs.
5. `stability_relation`, the relation asserting the stability properties of the transformation.
6. `function`, the function implemented by the transformation.

Similarly to measurements, the structure of a transformation operator favors generality. By changing the different attribute we expect to be able to express most of the current and future data transformations used when constructing or applying differentially private mechanisms. To assert the properties of a transformation we have a *stability relation* between an input distance d_i and an output distance d_o and certifying that "for all x, x' in the input domain, if x is d_i-close to x' under the input metric, then the outputs of running the function on x and running it on x' are d_o-close under the output metric".

*A full version of this paper is available at `http://opendp.io/`

†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.

Chaining and composition From transformations and measurements defined as above, we can build up more complex transformations and measurements through various operators that combine them.

Chaining For combining measurements and transformations we use *chaining*, which is simply function composition. We can chain two transformations to produce a new transformation, or chain a transformation and a measurement to produce a new measurement.

Composition For combining multiple measurements we can use *composition* functions, which take multiple measurements and return a new measurement with a privacy relation built out of the privacy relations of the input measurements accordingly to some composition theorem of the underlying privacy measure.

This form of composition is "non-adaptive." 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.) We will discuss how we manage this form of adaptivity in the next section.

Interactive Measurements The framework described before assumes that measurements are one-shot randomized functions. 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).

An interactive measurement M is a (possibly randomized) function that takes a private dataset and then "spawns" a (possibly randomized) state machine 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.

An interactive measurement operator is specified with same attributes as noninteractive measurements, as described above, except that the `function` is now the (possibly randomized) function that generates a queryable from the input dataset.

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 *private with respect to the output measure* if for every adversarial strategy A, and for all x, x' in the input domain, if x is d_i-close to x' under the input metric, then the random variables $Y = \text{View}(A \leftrightarrow M(x))$ and $Y' = \text{View}(A \leftrightarrow M(x'))$ are d_o-close under the similarity output measure.

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.

Post-processing An important property of differential privacy is that it is closed under post-processing. 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. $\text{Postprocess}(M, P)$ 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 $\text{Postprocess}(M, P)$, 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.

A concrete example We illustrate the different components with an example: an instantiation of the differentially private Statistical Query (SQ) Model. This can be implemented as a post-processing of an interactive measurement AdaptiveComposition.

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 can implement this using $T + 1$ queries to an $\text{AdaptiveComposition}_{\mathcal{X},\varepsilon}$ 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 don't require that the queries f are bounded as specified, but will rather enforce it by evaluating the sums using a NoisyClampedSum measurement.

Due to space constraints, we present this example in a simplified form of the framework, where the private data is always an element of $\text{MultiSets}(\mathcal{X})$ for some record domain $\mathcal{X}$, the metrics on the private data are always symmetric difference of multisets, the output measure for measurements is always pure differential privacy, and the privacy and stability relations are replaced by constants representing the pure-DP parameter ε or the stability/Lipschitz constant.

```
class InteractiveMeasurement:
  input_domain
  privacy_loss
  function

class Queryable:
  _state       # --- the state need to be private ---
  eval
  def query(q):
    (a, _state)=eval(q,_state)
    return a

def MakeAdaptiveComposition(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,function)

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

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'
      return (answer,state-1)
    return Queryable(initial_state,eval)
  return Postprocess(MakeAdaptiveComposition(in_dom,epsilon),queryable_map)
```

Figure 1: Differentially Private SQ Model

Chaining and Composition of Interactive Measurements Chaining generalizes in a straightforward way to interactive measurements. 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 [1]. 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, or for general privacy odometers; this is an important direction for future work.

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).

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.

Other considerations *Ensuring privacy in implementations* The design we outlined above guarantees private data to be accessed only by means of valid measurement and transformations. In addition, we will need to prevent leakages potentially caused by:

- Timing channels
- Implementation of arithmetic
- Use of pseudorandomness

Using the Library Calling the library from a DP system requires:

- determining the dataset and its type, determining the privacy notion one wants to use and its granularity,
- selecting an interactive measurement from the library,
- presenting all the queries to the queryable created by the measurement.

To do this, the library needs information about the runtime system, e.g. data access model, capabilities of the backend, partition between secure and insecure storage, that need also to be provided before the execution. These can be exposed directly or through user interfaces built on top of the library, e.g. SQL-like, GUI, Python notebook, etc.
Contributing to the Library We envision different kinds of code contributions:

- New measurements or transformations combining existing library components.
- New private data types, distance measures, or privacy notions.
- New primitives to combine measurements and transformations.
- New "atomic" measurements or transformations with proof of correctness.
- New types of privacy or stability calculus.

The Scope of the Framework What is supported within the current framework:

- Many different dataset types, privacy measures and granularities, ways to combine different primitives to build more complex mechanisms.
- Common database transformations, mechanisms based on global sensitivity or restricted sensitivity

What will require extensions of the framework:

- Mechanisms based on local sensitivity, privacy odometers, privacy with explicit adversary models, randomized or interactive transformations.

References

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

[2] 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.

[3] 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.