

# fn make\_fully\_adaptive\_composition\_queryable

Michael Shoemate

This proof resides in “**contrib**” because it has not completed the vetting process.

Proves soundness of fn make\_fully\_adaptive\_composition\_queryable.

## 1 Hoare Triple

### Precondition

#### Compiler-verified

Types matching those in the pseudocode.

- Generic DI implements **Domain**.
- Generic MI implements **Metric**.
- Generic MO implements **CompositionMeasure**.
- (DI, MI) implements **MetricSpace**.

#### User-verified

data is a member of input\_domain.

### Pseudocode

```
1 def new_fully_adaptive_composition_queryable(  
2     input_domain: DI,  
3     input_metric: MI,  
4     output_measure: MO,  
5     data: DI_Carrier,  
6 ) -> OdometerQueryable[Measurement[DI, TO, MI, MO], TO, MO_Distance]:  
7  
8     is_sequential = matches(  
9         output_measure.theorem(Adaptivity.FullyAdaptive),  
10        Sequentiality.Sequential  
11    )  
12  
13    privacy_maps = [] # Vec<MO_Distance>  
14  
15    def transition( #  
16        self_: OdometerQueryable[Measurement[DI, TO, MI, MO], TO, MO_Distance],  
17        query: Query[OdometerQuery[Measurement[DI, TO, MI, MO]]]  
18    ):  
19        # this queryable and wrapped children communicate via an AskPermission query  
20        # defined here, where no-one else can access the type  
21        @dataclass
```

```

22     class AskPermission:
23         id: usize
24
25     match query:
26         # evaluate external invoke query
27         case Query.External(OdometerQuery.Invoke(measurement)): #
28             assert_components_match( #
29                 DomainMismatch,
30                 input_domain,
31                 measurement.input_domain
32             )
33
34             assert_components_match( #
35                 MetricMismatch,
36                 input_metric,
37                 measurement.input_metric
38             )
39
40             assert_components_match( #
41                 MeasureMismatch,
42                 output_measure,
43                 measurement.output_measure
44             )
45
46         if is_sequential:
47             # when the output measure doesn't allow concurrent composition,
48             # wrap any interactive queryables spawned.
49             # This way, when the child gets a query it sends an AskPermission query
50             # to this parent queryable, giving this sequential odometer queryable
51             # a chance to deny the child permission to execute
52             child_id = privacy_maps.len()
53
54             seq_wrapper = Wrapper.new_recursive_pre_hook( #
55                 lambda: self.eval_internal(AskPermission(child_id))
56             )
57         else:
58             seq_wrapper = None
59
60         answer = measurement.invoke_wrap(data, seq_wrapper) #
61
62         # We've now increased our privacy spend.
63         # This is our only state modification
64         privacy_maps.push(measurement.privacy_map) #
65
66         return Answer.External(OdometerAnswer.Invoke(answer))
67
68     # evaluate external privacy loss query
69     case Query.External(OdometerQuery.PrivacyLoss()):
70         d_mids = [m.eval(d_in) for m in privacy_maps]
71         d_out = output_measure.compose(d_mids)
72         return Answer.External(OdometerAnswer.Map(d_out))
73
74     case Query.Internal(query):
75         # Check if the query is from a child queryable
76         # who is asking for permission to execute
77         if isinstance(query, AskPermission): #
78             # deny permission if the sequential odometer has moved on
79             if query.id + 1 != privacy_maps.len():
80                 raise ValueError("sequential odometer has received a new query")
81
82             # otherwise, return Ok to approve the change
83             return Answer.internal(())
84
85     # handler to see privacy usage after running a query.

```

```

86         # Someone is passing in an OdometerQuery internally,
87         # so return the potential privacy loss of this odometer after running this
query
88         if isinstance(query, OdometerQuery): #
89             match query:
90                 case OdometerQuery.Invoke(meas):
91                     pending_d_mids = [*d_mids, meas.map(d_in)] #
92                     pending_d_out = output_measure.compose(pending_d_mids)
93                     return Answer.internal(PendingLoss.New(pending_d_out))
94
95                 case OdometerQuery.PrivacyLoss():
96                     return Answer.internal(PendingLoss.Same()) #
97
98             raise ValueError("query not recognized")
99
100     return Queryable.new(transition) #

```

## Postconditions

**Theorem 1.1.** For every setting of the input parameters ( $\text{input\_domain}$ ,  $\text{input\_metric}$ ,  $\text{output\_measure}$ ,  $\text{data}$ ,  $\text{DI}$ ,  $\text{TO}$ ,  $\text{MI}$ ,  $\text{MO}$ ) to  $\text{new\_fully\_adaptive\_composition\_queryable}$  such that the given preconditions hold,  $\text{new\_fully\_adaptive\_composition\_queryable}$  raises a data-independent error or returns a valid odometer queryable (queryable). A valid odometer queryable is an **OdometerQueryable** with the following properties:

1. (Data-independent errors). For every pair of members  $x$  and  $x'$  in  $\text{input\_domain}$ , and every query  $q$ ,  $\text{queryable.invoke}_x(q)$  and  $\text{queryable.invoke}_{x'}(q)$  either both return the same error or neither return an error.
2. (Privacy Guarantee). For every pair of members  $x$  and  $x'$  in  $\text{input\_domain}$ , every adversary  $\mathcal{A}$ , and for every pair  $(d\_in, d\_out)$ , where  $d\_in$  has the associated type for  $\text{input\_metric}$  and  $d\_out$  has the associated type for  $\text{output\_measure}$ , if  $x$  and  $x'$  are  $d\_in$ -close under  $\text{input\_metric}$ ,  $\text{queryable.eval}(\text{privacy\_loss})$  does not return an error, and  $\text{queryable.eval}(\text{privacy\_loss}) = d\_out$ , then  $\text{View}(\mathcal{A} \leftrightarrow \text{queryable}_x), \text{View}(\mathcal{A} \leftrightarrow \text{queryable}_{x'})$  are  $d\_out$ -close under  $\text{output\_measure}$ , where  $\text{queryable}_x$  denotes all messages received by  $\mathcal{A}$  under  $x$ .
3. (Pending Privacy Guarantee). For every pair of members  $x$  and  $x'$  in  $\text{input\_domain}$ , every adversary  $\mathcal{A}$ , every **OdometerQuery**  $q$ , and for every pair  $(d\_in, d\_out)$ , where  $d\_in$  has the associated type for  $\text{input\_metric}$  and  $d\_out$  has the associated type for  $\text{output\_measure}$ , if  $x$  and  $x'$  are  $d\_in$ -close under  $\text{input\_metric}$ ,  $\text{queryable.eval\_internal}(q)$  does not return an error, and  $\text{queryable.eval\_internal}(q) = d\_out$ , then the privacy loss remains the same if  $d\_out = \text{PendingLoss.Same}$ , otherwise  $\text{View}(\mathcal{A} \leftrightarrow \text{queryable}'_x), \text{View}(\mathcal{A} \leftrightarrow \text{queryable}'_{x'})$  are  $d\_out$ -close under  $\text{output\_measure}$ , where  $\text{queryable}'_x$  denotes all messages received by  $\mathcal{A}$  under  $x$ , as well as  $\text{queryable.eval}(q)$ .

*Proof of data-independent errors.* The only interaction with the data is on line 60. By the postcondition of a valid measurement, errors are data-independent.  $\square$

*Proof of privacy guarantee.* The transition function follows the  $\mathcal{G}$ -OdomCon(IM) procedure described in Algorithm 6 of [HST<sup>+</sup>23], with some adjustments that do not materially affect the privacy analysis.

- When an odometer invokes  $\mathcal{M}_{k+1}$ , the OpenDP Library implementation eagerly accounts for all future privacy loss of the child mechanism immediately (on line 64), even if the analyst never interacts with the spawned interactive mechanism.
- The OpenDP Library represents interactive mechanisms via queryables, where the state  $s$  is hidden from the adversary. When invoking  $\mathcal{M}_k$ , no initial state  $\lambda$  is passed, and the response is a queryable with hidden state. By the definition of a valid interactive measurement, the view an adversary gains from this queryable has privacy loss  $d_k$  for some choice of  $d\_in$ .

- The queryable for child  $k$  can be viewed as  $\mathcal{M}_k$  together with  $s_k$ . Since the queryable itself satisfies  $d_k$ -DP, the queryable can be returned to the user to provide query access, child queryables ( $\mathcal{M}_k$  and  $s_k$ ) can be removed from the odometer state, and the handling of child update queries  $m = (j, q)$  can be removed while preserving equivalency with Algorithm 6. This equivalently reduces the odometer state to  $(s, [d_1, \dots, d_k])$ .
- In addition to storing the dataset  $x$ , the state also contains the input domain, input metric and privacy measure. Each incoming mechanism must share these same supporting elements. This is an implicit requirement of the paper made explicit in the implementation.

We now discuss how the pseudocode implements this equivalent algorithm. The input domain, input metric, privacy measure and data are moved into the transition function, as well as a mutable list of privacy losses from line 13, to form the state.

First consider the case where the privacy measure satisfies concurrent composition. Lines 34 and 40 are necessary to guarantee that the constructed odometer queryable gives valid privacy loss guarantees with respect to `input_metric` and `output_measure`. On line 60, the user-verified preconditions for invoking the measurement are met, by the precondition that the `data` is a member of `input_domain` and the check that the measurement’s input domain matches `input_domain` on line 28. Therefore by the definition of a valid measurement, `answer` satisfies `measurement.privacy_map(d_in)`-DP, with respect to `output_measure`, for some choice of `d_in`.

Now consider the case where the privacy measure does not satisfy concurrent composition. Then `seq_wrapper` on line 54 is a wrapper that will cause wrapped queryables to send an `AskPermission` query with their self-reported id to this queryable before answering any query. By line 77, permission is only granted if the most recently spawned child is asking for permission to execute a query. This ensures that an adversary may only interact with the most recently spawned child mechanism.

Now that we’ve shown a correspondence between the pseudocode and Algorithm 6 of [HST<sup>+</sup>23], the proof from Appendix B.1 shows that the pseudocode implements a valid  $\mathcal{D}$  DP privacy loss accumulator for interactive mechanisms.

The transition function is then used to construct a queryable on line 100. □

*Proof of pending privacy guarantee.* The relevant handler is on line 88. The proof follows similarly to the proof of the privacy guarantee, but with the addition of the privacy loss of the pending query on line 91.

In the case of a pending privacy loss query, since the privacy loss query does not change the privacy loss of the odometer, `PendingLoss.Same` is returned on line 96. □

## References

- [HST<sup>+</sup>23] Samuel Haney, Michael Shoemate, Grace Tian, Salil Vadhan, Andrew Vyrros, Vicki Xu, and Wanrong Zhang. Concurrent composition for interactive differential privacy with adaptive privacy-loss parameters, 2023.