Privacy Proofs for OpenDP: Binary Randomized Response Measurement

Vicki Xu, Hanwen Zhang, Zachary Ratliff

Summer 2022

Contents

1 Algorithm Implementation

1.1 Code in Rust

The current OpenDP library contains the `make_randomized_response_bool` function implementing the binary randomized response measurement. This is defined in lines 28-53 of the file `mod.rs` in the Git repository `https://github.com/opendp/opendp/blob/main/rust/src/meas/randomized_response/mod.rs#L28-L53`

In `make_randomized_response_bool`, which accepts a parameter `prob` of type `Q` and a parameter `constant_time` of type `bool`, the function takes in a boolean data point `arg` and returns the truthful value `arg` with probability `prob` and the untruthful value `!arg` with probability $1 -$ `prob` (in constant time, if the flag is turned on).

1.2 Pseudo Code in Python

We present a simplified Python-like pseudocode of the Rust implementation below. The necessary definitions for the pseudocode can be found at "List of definitions used in the pseudocode".

The use of `code`-style parameters in the preconditions section below (for example, `input_domain`) means that this information should be passed along to the `Measurements` constructor.

Preconditions

To ensure the correctness of the output, we require the following preconditions:

- **User-specified types:**
  - Variable `prob` must be of type `Q`

– Variable `constant_time` must be of type `bool`
– Type `bool` must have trait `SampleBernoulli<Q>`
– Type `Q` must have traits `float`, `ExactIntCast<IntDistance>`, `DistanceConstant<IntDistance>`, `InfSub`, `InfLn`
– Variable `IntDistance` must have trait `InfCast<Q>`

Postconditions

- A `Measurement` is returned (i.e., if a `Measurement` cannot be returned successfully, then an error should be returned).

```
def make_randomized_response_bool(prob : Q, constant_time : bool):
    input_domain = AllDomain(bool)
    output_domain = AllDomain(bool)
    input_metric = DiscreteMetric()
    similarity_metric = MaxDivergence()

    if (prob < 0.5 or prob >= 1):
        raise Exception("probability must be in [0.5, 1)")

    c = inf_ln(inf_div(prob, neg_inf_sub(1, prob)))
    def privacy_map(d_in: u_32) -> u_32:
        return d_in * c;

    def function(arg : bool) -> bool:
        if (sample_bernoulli(prob, constant_time)):
            return arg
        else:
            return !arg

    return Measurement(input_domain, output_domain, function, input_metric,
     similarity_metric, privacy_map)
```

> **Warning 1** (Code is not constant-time)**.** `make_randomized_response_bool` takes in a boolean `constant_time` parameter that protects against timing attacks on the Bernoulli sampling procedure. However, the current implementation does not guard against other types of timing side-channels that can break differential privacy, e.g., non-constant time code execution due to branching.

2 Proof

The necessary definitions for the proof can be found at "List of definitions used in the proofs".

Theorem 2.1. *For every setting of the input parameters `prob`, `constant_time` to `randomized_response` such that the given preconditions hold, `randomized_response` raises an exception (at compile time or runtime) or returns a valid measurement with the following privacy guarantee:*

1. (Domain-metric compatibility.) *The domain `input_domain` matches one of the possible domains listed in the definition of `input_metric`.*

2. (Privacy guarantee.) *Let `d_in` be the associated metric on `input_domain` and has the associated type for `input_metric`, and let D be the similarity measure on probability distributions with the associated type for `similarity_metric`. For every pair of elements v, w in `input_domain` and every `d_in`, if v, w are `d_in`-close under `input_metric`, then `function`(v), `function`(w) are `privacy_map(d_in)`-close with respect to D.*

Proof.

1. **(Domain-metric compatibility.)** For `binary_randomized_response`, this corresponds to showing `AllDomain(bool)` is compatible with `DiscreteMetric`. This follows directly from the definition of `DiscreteMetric`, as stated in the "List of definitions used in the pseudocode".

2. **(Privacy guarantee.)**

> **Note 1** (Proof relies on correctness of Bernoulli sampler). The following proof makes use of the following lemma that asserts the correctness of the Bernoulli sampler function.
>
> **Lemma 2.2.** *`sample_bernoulli(prob, constant_time)`, the Bernoulli sampler function used in `make_randomized_response_bool`, returns `true` with probability (`prob`) and returns `false` with probability (1 - `prob`).*

(vicki) to do: need to relax the epsilon-delta defns.

Let v and w be datasets that are `d_in`-close with respect to `input_metric`. Here, the metric is `DiscreteMetric` which enforces that `d_in` $= 1$ if $v \neq w$ and `d_in` $= 0$ if $v = w$. The case where $v = w$ is trivial so we only consider $v \neq w$ and assume without loss of generality that $v =$ `true` and $w =$ `false`. For shorthand, we let p represent `prob`, the probability that `sample_bernoulli` returns `true`. Observe that $p = [0.5, 1.0)$ otherwise `make_randomized_response_bool` raises an error.

We now consider the max-divergence $D_\infty(Y||Z)$ over the random variables $Y =$ `function`(v) and $Z =$ `function`(w).

$$D_\infty(Y||Z) = \max_{S \subseteq Supp(Y)} [\ln\left(\frac{\Pr[Y \in S]}{\Pr[Z \in S]}\right)]$$

$$= \max(\ln\left(\frac{\Pr[Y = \texttt{true}]}{\Pr[Z = \texttt{true}]}\right), \ln\left(\frac{\Pr[Y = \texttt{false}]}{\Pr[Z = \texttt{false}]}\right))$$

$$= \max(\ln\left(\frac{p}{1-p}\right), \ln\left(\frac{1-p}{p}\right))$$

$$= \ln\left(\frac{p}{1-p}\right)$$

Note that $\ln\left(\frac{p}{1-p}\right)$ is $\leq$ `privacy_map`(`d_in`) when `d_in` $= 1$. Therefore we've shown that for every pair of elements $v, w \in \{\texttt{false}, \texttt{true}\}$ and every $d_{DM}(v, w) \leq$ `d_in` with `d_in` ≤ 1, if v, w are `d_in`-close then `function`(v), `function`$(w) \in \{$`false`, `true`$\}$ are `privacy_map`(`d_in`)-close under `output_metric` (the Max-Divergence). □

Implementation note: c = `inf_ln(inf_div(prob, neg_inf_sub(1, prob)))` rounds upward in the presence of floating point rounding errors. This is because `neg_inf_sub(1, prob))))` appears in the denominator, and to ensure that the bound holds even in the presence of rounding errors, the conservative choice is to round down (so the quantity as a whole is bounded above). Similarly, `inf_div` and `inf_ln` round up.

This does not entirely complete the proof, because we still need to account for failure cases within the code. Going up the chain of failure, there are three cases in which the code raises an exception:

(a) `neg_inf_sub` fails. By the implementation of `neg_inf_sub`, given in the pseudocode definitions doc, the code will raise an exception and terminate here if subtraction overflow occurs.

(b) `inf_div` fails. This step is only reached if `neg_inf_sub` succeeds, which means subtraction overflow did not occur (otherwise the Rust compiler would have thrown an error). As defined in the pseudocode definitions doc, `inf_div` throws an exception if division overflows from a 32-bit integer.

(c) `inf_ln` fails. This step is only reached if `inf_div` succeeds, which means `neg_inf_sub` also had to succeed. Hence, neither subtraction nor division overflow occurred. Given in the pseudocode definitions doc, `inf_ln` throws an exception if the natural log function overflows a 32-bit integer.

□