MakeNoiseThreshold<MapDomain<AtomDomain<TK>, AtomDomain<IBig>>, MI, MO> for RV

Michael Shoemate

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

Proves soundness of the implementation of `MakeNoiseThreshold` for `RV` over hashmaps of big integers in `mod.rs` at commit f5bb719 (outdated[1]).

This is the core implementation of all variations of the thresholded gaussian or laplace mechanism.

1 Hoare Triple

Precondition

Compiler-Verified

`MakeNoise` is parameterized as follows:

- `DI` is `MapDomain<AtomDomain<TK>, AtomDomain<IBig>>`
- `MI` is `LOPInfDistance<P, AbsoluteDistance<UBig>>`
- `MO` is `MO`

The following trait bounds are required:

- Generic `TK` implements trait `Hashable`
- Const-generic `P` is of type `usize`
- Generic `MO` implements trait `Measure`
- Type `ZExpFamily<P>` implements trait `NoiseThresholdPrivacyMap<LOPInfDistance<P, AbsoluteDistance<UBig>>, MO>`

User-Verified

None

[1] See new changes with `git diff f5bb719..32a450b rust/src/measurements/noise_threshold/mod.rs`

Pseudocode

```
# analogous to impl MakeNoise<VectorDomain<AtomDomain<IBig>>, MI, MO> for RV in Rust
class RV:
    def make_noise_threshold(
        self,
        input_space: tuple[MapDomain[AtomDomain[TK], AtomDomain[IBig]], MI],
        threshold: IBig,
    ) -> Measurement[
        MapDomain[AtomDomain[TK], AtomDomain[IBig]], HashMap[TK, IBig], MI, MO
    ]:
        input_domain, input_metric = input_space
        output_measure = MO.default()
        threshold_magnitude = threshold.into_parts()[1] #
        privacy_map = self.noise_threshold_privacy_map( #
            input_metric, output_measure, threshold_magnitude
        )

        match threshold.sign():
            case Sign.Positive:
                inner = Ordering.Less
            case Sign.Negative:
                inner = Ordering.Greater

        def function(data: HashMap[TK, IBig]) -> HashMap[TK, IBig]:
            out = []
            for k, v in data.items():
                v = self.sample(v) #

                if v.cmp(threshold) != inner:
                    out.append((k, v))
            # shuffle the output to avoid leaking the order of the input
            random.shuffle(out) #
            return dict(out)

        return Measurement.new(
            input_domain,
            Function.new_fallible(function),
            input_metric,
            output_measure,
            privacy_map,
        )
```

Postcondition

Theorem 1.1. For every setting of the input parameters (`self, input_space, threshold, MO, TK, P`) to `make_noise_threshold` such that the given preconditions hold, `make_noise_threshold` raises an error (at compile time or run time) or returns a valid measurement. A valid measurement has the following properties:

1. (Data-independent runtime errors). For every pair of members x and x' in `input_domain`, `invoke`(x) and `invoke`(x') either both return the same error or neither return an error.

2. (Privacy guarantee). For every pair of members x and x' in `input_domain` and for every pair (`d_in`, `d_out`), where `d_in` has the associated type for `input_metric` and `d_out` has the associated type for
`output_measure`, if x, x' are `d_in`-close under `input_metric`, `privacy_map`(`d_in`) does not raise an error, and `privacy_map`(`d_in`) $=$ `d_out`, then `function`(x), `function`(x') are `d_out`-close under `output_measure`.

Proof of data-independent errors. The precondition of `Sample.sample` requires that `self` is a valid distribution. This is satisfied by the postcondition of `NoisePrivacyMap<MI, MO>` on line 13. The postcondition

of Sample.sample guarantees that the function only ever returns an error independently of the data. □

For the proof of the privacy guarantee, start by reviewing the postcondition of NoisePrivacyMap<MI, MO>, which has an associated function noise_privacy_map called on line 13.

Lemma 1.2 (Postcondition of NoisePrivacyMap)**.** Given a distribution self, returns Err(e) if self is not a valid distribution. Otherwise the output is Ok(privacy_map) where privacy_map observes the following:

Define function(x) as a function that updates each pair $(k_i, v_i + Z_i)$, where Z_i are iid samples from self, and discards pairs where $v_i + Z_i$ is further from zero than threshold. The ordering of returned pairs is independent from the input ordering.

For every pair of elements x, x' in VectorDomain<AtomDomain<IBig>>, and for every pair (d_in, d_out), where d_in has the associated type for input_metric and d_out has the associated type for output_measure, if x, x' are d_in-close under input_metric, privacy_map(d_in) does not raise an exception, and privacy_map(d_in) $\leq$ d_out, then function(x), function(x') are d_out-close under output_measure.

Proof of privacy guarantee. Assuming line 13 does not fail, then the returned privacy map is subject to Theorem 1.2. The privacy guarantee applies when the pseudocode matches the algorithm specification, where Z_i are iid samples from self. In this case self describes the noise distribution.

We argue that function is consistent with the function described in Lemma 1.2. Line 26 calls self.sample(x_i) on each element in the input vector. The precondition that self represents a valid distribution is satisfied by the postcondition of Lemma 1.2; the distribution is valid when the construction of the privacy map does not raise an exception. Since the preconditions for Sample.sample are satisfied, the postcondition claims that either returns an error independently of the input v, or v + Z where Z is a sample from the distribution defined by self. The keys are then shuffled on line 31 to ensure that the output is independent of the input ordering. In the Rust implementation, a random hasher is used to ensure that the output ordering is independent of the input ordering. This is consistent with Lemma 1.2.

Therefore, the privacy guarantee from Lemma 1.2 applies to the returned measurement. □