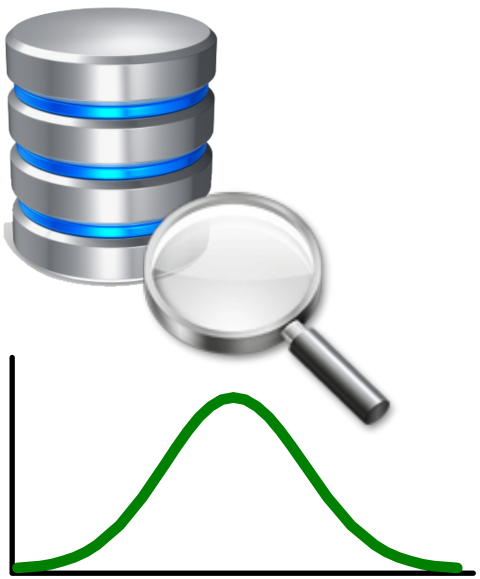
LightDP: Towards Automating Differential Privacy Proofs

Danfeng Zhang   Daniel Kifer
Penn State University

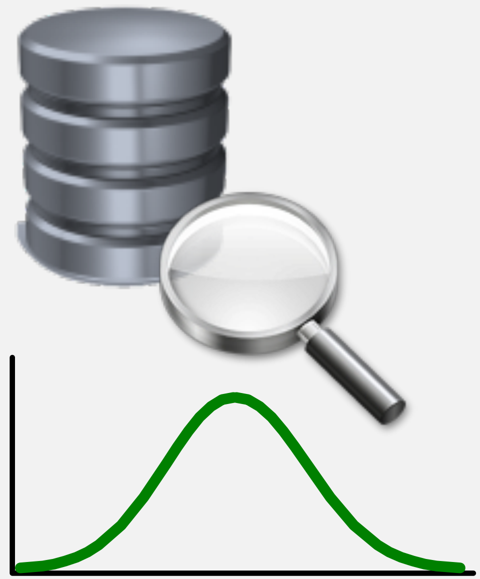
PennState

Database w/
Alice’s data



μ_1

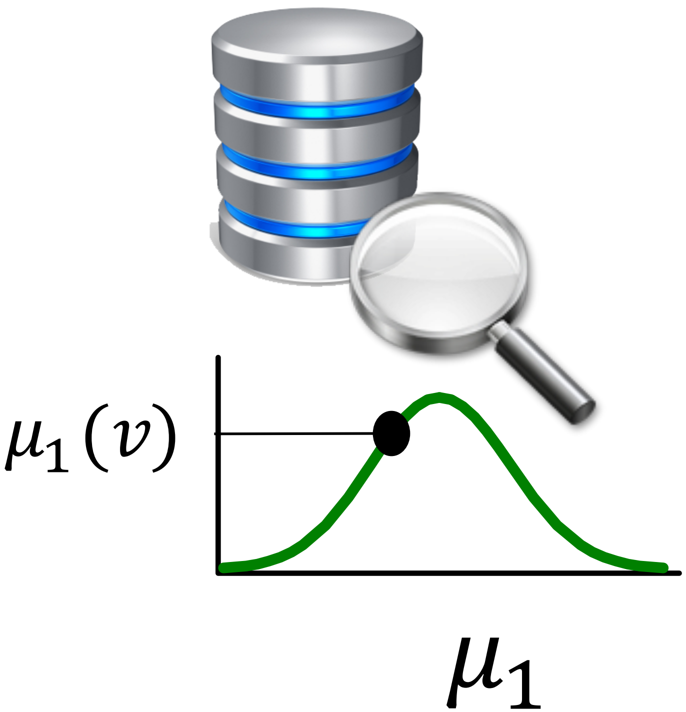
Database ***w/o***
Alice’s data

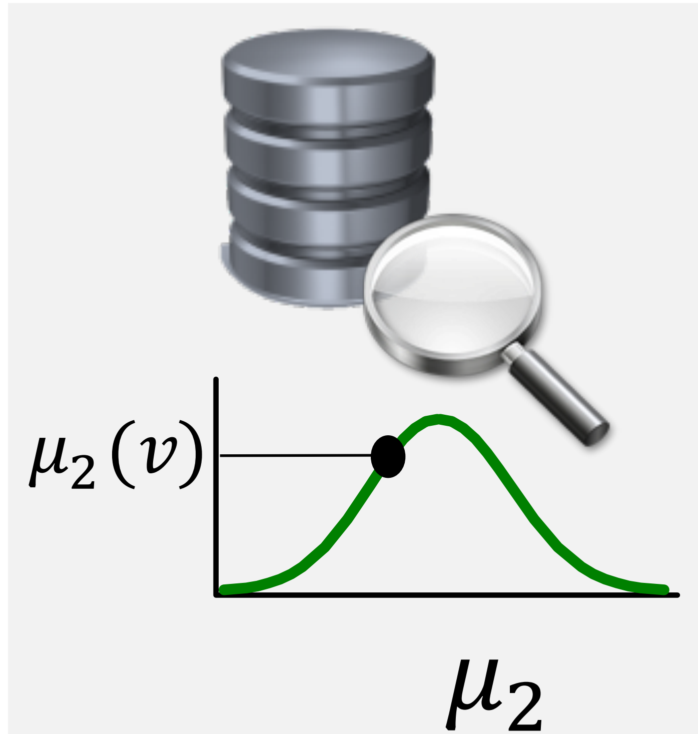


μ_2

Alice’s data remain private if μ_1, μ_2 are *close*

(Pure) Differential Privacy





If for any *adjacent* databases and value v, $\mu_1(v)/\mu_2(v) \leq e^{\epsilon}$ for some constant ϵ, then a computation is ϵ-private

Privacy Cost

Motivation

DP has seen explosive growth since 2006

- U.S. Census Bureau LEHD OnTheMap tool [Machanavajjhala et al. 2008]
- Google Chrome Browser [Erlingsson et al. 2014]
- Apple's new data collection efforts [Greenberg 2016]

But also accompanied with flawed (paper-and-pencil) proofs

- e.g., ones categorized in [Chen&Machanavajjhala'15, Lyu et al.'16]

Rigorous methods are needed for differential privacy proofs

Related Work

DP programming platforms (e.g., PINQ, Airavat)

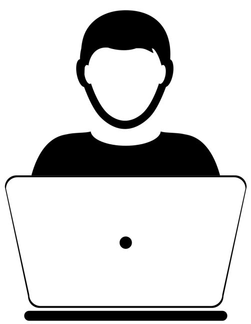
- Use (instead of verify) basic DP mechanisms
- Cannot offer tight bounds for sophisticated algorithms

Methods based on customized logics

- Steep learning curve
- Heavy annotation burden

LightDP offers a better balance between expressiveness and usability

LightDP: Overview



Source Program

```
η1  := Lap (2/ε);
T̃  := T + η1;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  η2 := Lap (4N/ε);
  if (q[i] + η2 ≥ T̃) then
    out:= true::out;
    c1 := c1 + 1;
  else
    out:= false::out;
    c2 := c2 + 1;
  i := i+1;
```

Relational, Dependent Type System

Target Program with distinguished variable $\mathbf{v}_\epsilon$

```
vε := 0;
havoc η1; vε := vε + ε/2;
T̃ := T + η1;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  havoc η2; vε := vε + (q[i] + η2 ≥ T̃ ? 2 : 0) × ε/4N;
  if (q[i] + η2 ≥ T̃) then
    out:= true::out;
    c1 := c1+1;
  else
    out:= false::out;
    c2 := c2+1;
  i := i+1;
```

Main Theorem

Source program type checks + v_ϵ bounded by constant ϵ in the target program

⇒ Source program is ϵ-private

Source Language: Syntax

Random variable

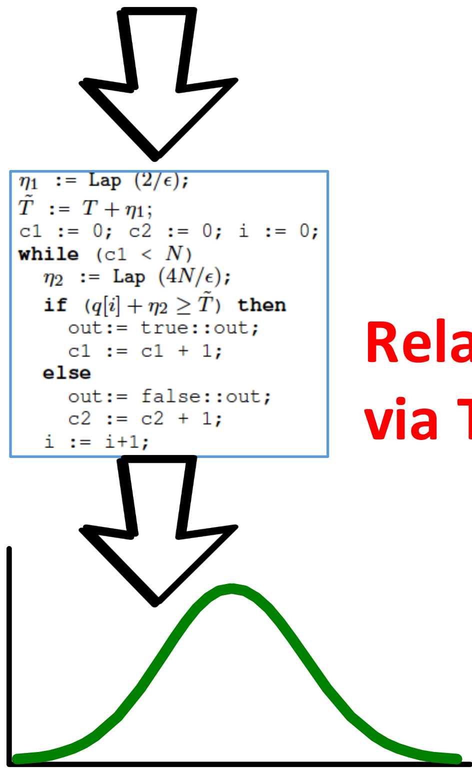
Random Expression (e.g., Laplace dist.)

$$
\begin{array}{llll}
\text{Commands} & c & ::= & \mathtt{skip} \mid x := e \mid \eta := g \mid c_1; c_2 \mid \mathtt{return}\ e \mid \\
& & & \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ e\ \mathtt{do}\ c
\end{array}
$$

Source Language: Semantics

Memory: mapping from variables to values

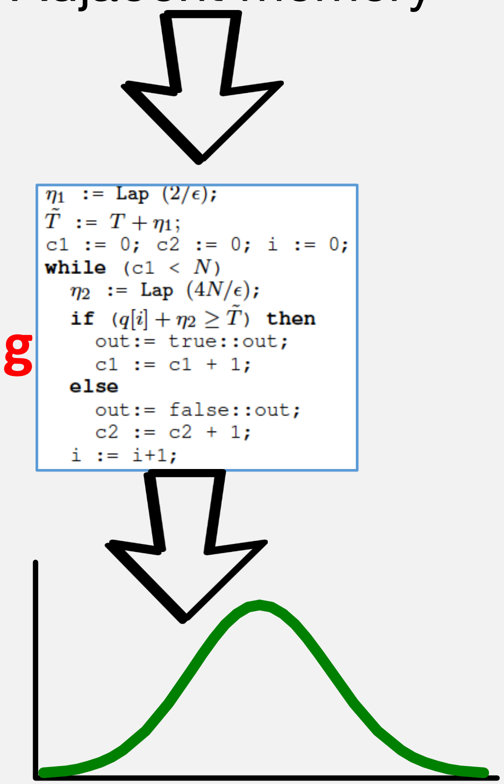
Initial memory



Final memory dist.

Relational Reasoning via Type System

Adjacent memory



Final memory dist.

Relational Types

$$\mathcal{B}_{\mathbb{d}}$$

Base Type

Distance

Example

$\Gamma(x): \mathrm{num}_0$

$\Gamma(y): \mathrm{num}_1$

e.g., int, real

Related Memories

x: u	x: u
y: v	y: v+1

Dependent Types

$\mathcal{B}_{\mathrm{d}}$ Can be a program variable

Example

$\Gamma(x): \mathrm{num}_0$

$\Gamma(y): \mathrm{num}_x$

Related Memories

x: u	x: u
y: v	y: v + u

Dependent Types

$\mathcal{B}_{\Phi}$

Can be a non-prob. expression

Example

$\Gamma(x)\colon \mathrm{num}_0$

$\Gamma(y)\colon \mathrm{num}_{x \geq 1 ? 2 : 0}$

Related Memories

x: u

y: v

x: u

$$y\colon \begin{cases} \mathrm{v}+2, & \mathrm{u} \geq 1 \\ \mathrm{v} & ,\mathrm{u} < 1 \end{cases}$$

Notation

$m_1 \; \Gamma \; m_2$ if m_1 and m_2 are related by Γ

(for the non-probabilistic subset)

Types form an invariant on two related program executions:

If initial memories m_1 Γ m_2

Then after executing
a well-typed program,
final memories m_1' Γ m_2'

Enforced by a type system

Type System

Expression: $\Gamma \vdash e : \mathcal{B}_{\mathbb{d}}$

e.g., $$\frac{\Gamma \vdash e_1 : \mathbf{num}_{\mathbb{d}_1} \quad \Gamma \vdash e_2 : \mathbf{num}_{\mathbb{d}_2}}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{num}_{\mathbb{d}_1 \oplus \mathbb{d}_2}}$$

$+ \mid -$

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 : \mathbf{num}_{\mathbb{d}_1} & (e_1 \odot e_2 \\ \Gamma \vdash e_2 : \mathbf{num}_{\mathbb{d}_2} & \Leftrightarrow (e_1 + \mathbb{d}_1) \odot (e_2 + \mathbb{d}_2)) \end{array}}{\Gamma \vdash e_1 \odot e_2 : \mathtt{bool}_0}$$

$< \mid > \mid = \mid \leq \mid \geq$

Type System

Command: $\Gamma \vdash c$

Distance must be identical

e.g., $$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash x : \mathcal{B}_{\mathbb{d}} \quad \tau = \mathcal{B}_{\mathbb{d}}}{\Gamma \vdash x := e}$$

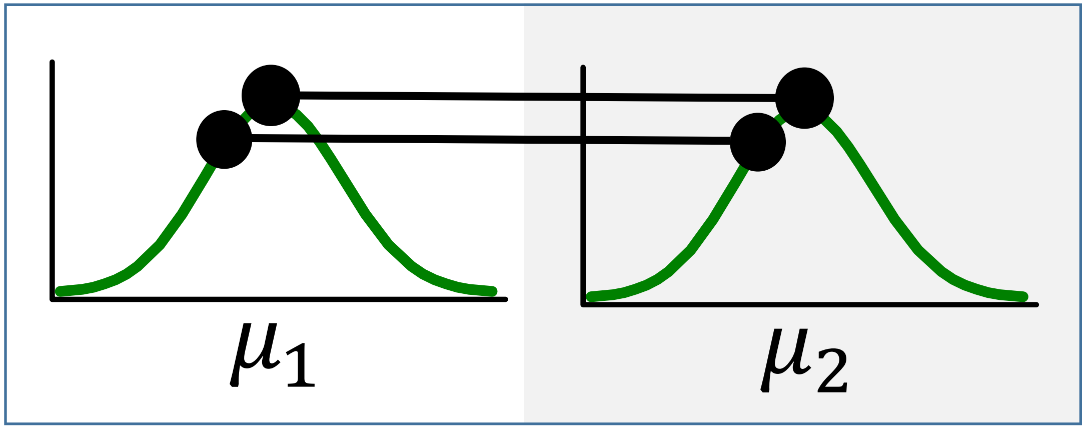
Related executions take same branch

$$\frac{\Gamma \vdash e : \mathtt{bool}_0 \quad \Gamma \vdash c_i \text{ where } i \in \{1, 2\}}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2}$$

Relating Two Distributions

$\mu_1 \; \Gamma \; \mu_2$ ***w.r.t. privacy cost ϵ*** if

$$\forall m.\, \mu_1(m)/\mu_2(\Gamma(m)) \leq e^{\epsilon}$$



Laplace dist. w/ mean 0 and a scale factor r

Program

`η := Lap r`

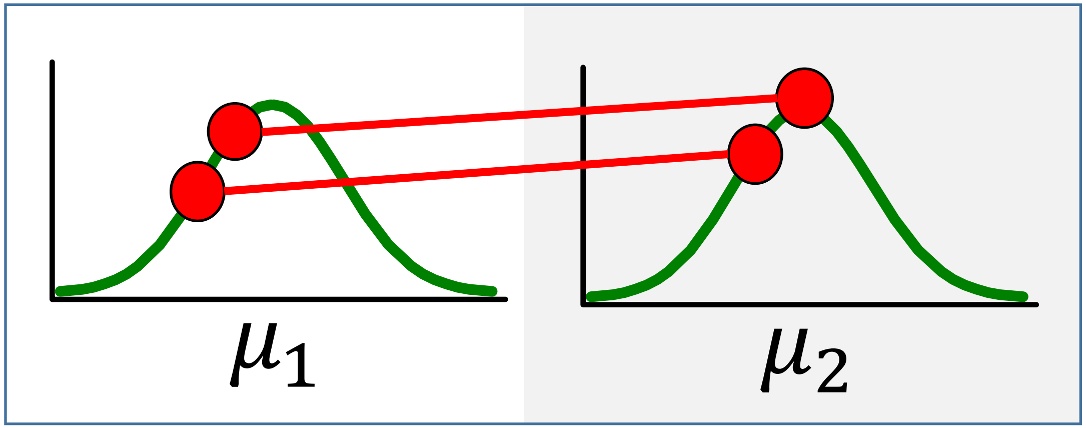
$\Gamma(\eta) = \mathrm{num}_0$

With no cost

Relating Two Distributions

$\mu_1 \, \Gamma \, \mu_2$ w.r.t. privacy cost ϵ if

$$\forall m.\, \mu_1(m)/\mu_2(\Gamma(m)) \leq e^{\epsilon}$$



Laplace dist. w/ mean 0 and a scale factor r

Program

η `:= Lap` r

$\Gamma(\eta) = \text{num}_1$

With cost $1/r$ due to dist. property

Observation

η may have an ***arbitrary distance***, which ***affects the added cost***

Observation

η may have an ***arbitrary distance***, which ***affects the added cost***

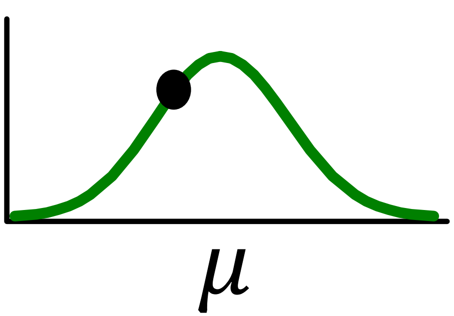
η has a polymorphic type

$$\frac{\Gamma(\eta) = \mathbf{num}_{\mathbb{d}}}{\Gamma \vdash \eta := \mathtt{Lap}\ \ r}$$

Non-deterministic operation

source program

target program, explicitly tracks added privacy cost



Intuitively, target program computes the added cost for one sample from distribution

In General

Source program

```
η1 := Lap (2/ε);
T̃ := T + η1;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  η2 := Lap (4N/ε);
  if (q[i] + η2 ≥ T̃) then
    out:= true::out;
    c1 := c1 + 1;
  else
    out:= false::out;
    c2 := c2 + 1;
  i := i+1;
```

Type System

Target program with distinguished variable $\mathbf{v}_\epsilon$

```
vε := 0;
havoc η1; vε := vε + ε/2;
T̃ := T + η1;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  havoc η2; vε := vε + (q[i] + η2 ≥ T̃ ? 2 : 0) × ε/4N;
  if (q[i] + η2 ≥ T̃) then
    out:= true::out;
    c1 := c1+1;
  else
    out:= false::out;
    c2 := c2+1;
  i := i+1;
```

$$\Gamma \vdash c \rightharpoonup c'$$

source program

target program

Target Language

set x to arbitrary value

$$\text{Commands} \quad c ::= \texttt{skip} \mid x := e \mid \overset{\texttt{havoc}\ x}{\cancel{\eta := g}} \mid c_1; c_2 \mid \texttt{return}\ e \mid$$
$$\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c$$

Verification task in the target language:
Proving $\mathbf{v}_{\epsilon}$ is bounded by some constant ϵ in any execution (in a non-probabilistic program)

A safety property. Can be verified using off-the-shelf tools (e.g., Hoare logic, model checking)

Putting Together

The Sparse Vector Method [Dwork and Roth'14]

Source Program

η_1 := **Lap** $(2/\epsilon)$;
$\tilde{T}$:= $T + \eta_1$;
`c1 := 0; c2 := 0; i := 0;`
while (c1 < N)
  η_2 := **Lap** $(4N/\epsilon)$;
  if $(q[i] + \eta_2 \geq \tilde{T})$ **then**
    `out:= true::out;`
    `c1 := c1 + 1;`
  else
    `out:= false::out;`
    `c2 := c2 + 1;`
  `i := i+1;`

- Correctness proof is subtle
  Incorrect variants categorized in [Chen&Machanavajjhala'15, Lyu et al.'16]

- Formally verified very recently [Barthe et al. 2016]
  with heavy annotation burden

Required Types

$$\mathtt{c1}, \mathtt{c2}, \mathtt{i} : \mathbf{num}_0; \tilde{T}, \eta_1 : \mathbf{num}_1; \eta_2 : \mathbf{num}_{q[i]+\eta_2 \geq \tilde{T} ? 2:0}$$

```
η1 := Lap (2/ε);
T̃ := T + η1;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  η2 := Lap (4N/ε);
  if (q[i] + η2 ≥ T̃) then
    out:= true::out;
    c1 := c1 + 1;
  else
    out:= false::out;
    c2 := c2 + 1;
  i := i+1;
```

Distance depends on the value of ith query answer ($q[i]$)

Type Inference

Types can be inferred by the inference algorithm of LightDP

Target Program

$$\underline{\mathbf{v}_\epsilon := 0;}$$
$$\underline{\texttt{havoc}\ \ \eta_1; \mathbf{v}_\epsilon := \mathbf{v}_\epsilon + \epsilon/2;}$$

```
~~η1 := Lap (2/ε);~~
T̃ := T + η1;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  ~~η2 := Lap (4N/ε);~~
  if (q[i] + η2 ≥ T̃) then
    out:= true::out;
    c1 := c1 + 1;
  else
    out:= false::out;
    c2 := c2 + 1;
  i := i+1;
```

$$\underline{\texttt{havoc}\ \ \eta_2; \mathbf{v}_\epsilon := \mathbf{v}_\epsilon + (q[i] + \eta_2 \geq T?2:0) \times \epsilon/4N;}$$

Completing the Proof

$\underline{\mathbf{v}_\epsilon := 0;}$

$\underline{\texttt{havoc}\ \ \eta_1; \mathbf{v}_\epsilon := \mathbf{v}_\epsilon + \epsilon/2;}$

$\tilde{T} := T + \eta_1;$

```
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
```

$\texttt{Invariant} : \texttt{c1} \leq N \wedge \mathbf{v}_\epsilon = \epsilon/2 + \texttt{c1} \times \frac{\epsilon}{2N}$

$\underline{\texttt{havoc}\ \ \eta_2; \mathbf{v}_\epsilon := \mathbf{v}_\epsilon + (q[i] + \eta_2 \geq \tilde{T}?2:0) \times \epsilon/4N;}$

$\texttt{if}\ \ (q[i] + \eta_2 \geq \tilde{T})\ \ \texttt{then}$

```
    out:= true::out;
    c1 := c1+1;
  else
    out:= false::out;
    c2 := c2+1;
  i := i+1;
```

Loop Invariant

Postcondition: $\mathbf{v}_\epsilon \leq \epsilon$

Main Theorem

Source program type checks
\+ $\mathbf{v}_\epsilon$ bounded by constant ϵ
= source program is ϵ-private

More in the Paper

Type inference algorithm

Searching for proof with minimum cost w/ MaxSMT

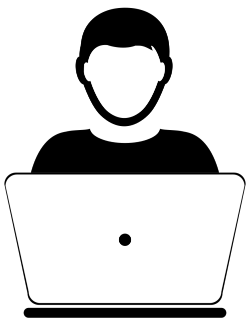
Formal proof for the main theorem

More verified examples (with little manual efforts)

Summary

A safety property (verified by existing tools)

Automated by inference engine



Source Program

```
η1 := Lap (2/ε);
T̃ := T + η1;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  η2 := Lap (4N/ε);
  if (q[i] + η2 ≥ T̃) then
    out:= true::out;
    c1 := c1 + 1;
  else
    out:= false::out;
    c2 := c2 + 1;
  i := i+1;
```

Relational, Dependent Type System

Target Program with distinguished variable $\mathbf{v}_\epsilon$

```
v_ε := 0;
havoc η1; v_ε := v_ε + ε/2;
T̃ := T + η1;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  havoc η2; v_ε := v_ε + (q[i] + η2 ≥ T̃ ? 2 : 0) × ε/4N;
  if (q[i] + η2 ≥ T̃) then
    out:= true::out;
    c1 := c1+1;
  else
    out:= false::out;
    c2 := c2+1;
  i := i+1;
```

Decomposing differential privacy into subtasks substantially simplifies language-based proof