DPGen: Automated Program Synthesis for Differential Privacy

Yuxin Wang
Pennsylvania State University
University Park, PA, USA
yxwang@psu.edu

Zeyu Ding
Pennsylvania State University
University Park, PA, USA
zyding@psu.edu

Yingtai Xiao
Pennsylvania State University
University Park, PA, USA
yxx5224@psu.edu

Daniel Kifer
Pennsylvania State University
University Park, PA, USA
dkifer@cse.psu.edu

Danfeng Zhang
Pennsylvania State University
University Park, PA, USA
zhang@cse.psu.edu

ABSTRACT

Differential privacy has become a de facto standard for releasing data in a privacy-preserving way. Creating a differentially private algorithm is a process that often starts with a noise-free (non-private) algorithm. The designer then decides where to add noise, and how much of it to add. This can be a non-trivial process – if not done carefully, the algorithm might either violate differential privacy or have low utility.

In this paper, we present DPGen, a program synthesizer that takes in non-private code (without any noise) and automatically synthesizes its differentially private version (with carefully calibrated noise). Under the hood, DPGen uses novel algorithms to automatically generate a sketch program with candidate locations for noise, and then optimize privacy proof and noise scales simultaneously on the sketch program. Moreover, DPGen can synthesize sophisticated mechanisms that adaptively process queries until a specified privacy budget is exhausted. When evaluated on standard benchmarks, DPGen is able to generate differentially private mechanisms that optimize simple utility functions within 120 seconds. It is also powerful enough to synthesize adaptive privacy mechanisms.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; • **Theory of computation → Program analysis**.

KEYWORDS

Differential privacy; program synthesis

ACM Reference Format:
Yuxin Wang, Zeyu Ding, Yingtai Xiao, Daniel Kifer, and Danfeng Zhang. 2021. DPGen: Automated Program Synthesis for Differential Privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3460120.3484781

1 INTRODUCTION

Differential privacy [23] has become a de facto standard for releasing data in a privacy-preserving way. It has been increasingly adopted in industry [14, 19, 25, 34, 45] and the public sector [1, 15, 32, 37]. Crucial to any differentially private system is a set of *privacy mechanisms*, the building blocks of larger privacy-preserving algorithms. Privacy mechanisms inject randomness into non-private computations in order to ensure privacy protections. However, developing such mechanisms is a daunting task for several reasons.

- From the privacy perspective, one must carefully choose exactly where the noise must be injected and how much noise to use. Such decisions are notoriously tricky. For example, the Sparse Vector Technique (SVT) [24] is designed to return the identities of N queries whose answers are likely to be larger than a public threshold T. Lyu et al. [36] catalog several peer-reviewed yet incorrect variants of SVT. At the source code level, these incorrect variants are very similar to the correct ones, but the tiny differences broke their privacy properties.
- From the utility perspective, there are many ways of converting a non-private program into a differentially private one – each such option could have wildly different utility properties. For instance, in the aforementioned SVT, injecting noise in one specific place (the threshold) allows the mechanism to use much less noise everywhere else (while processing more queries). Aside from different valid choices in noise locations, the mechanism must also allocate its privacy budget among different code fragments. This results in a trade-off where fragments with a larger share of the privacy budget use smaller amounts of noise. Picking the optimal (in terms of utility) way of adding randomness and allocating the privacy budget is also a non-trivial task (this is especially true for SVT [36]).

Most current tools focus on checking the privacy properties of algorithms. For example, verification tools have been developed to mechanically (and sometimes, automatically) prove that a (correct) privacy mechanism satisfies differential privacy [3, 5–9, 48, 49]. Counterexample detectors for differential privacy [12, 13, 20, 29] can find evidence that an (incorrect) privacy mechanism fails to satisfy its claimed privacy levels. Moreover, a few tools can combine both functionalities: either proving a mechanism is correct or finding a counterexample [4, 26, 47]. While these tools are invaluable for ensuring correctness of privacy mechanisms, they all require a putative differentially private algorithm as a starting point.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484781

Recently, Roy et al. [41] took a step further: they proposed a tool called KOLAHAL for automatically learning an accurate and differentially private mechanism *given a mechanism sketch provided by a domain expert.* In other words, their approach synthesizes how much noise should be added in pre-specified locations. Note that it does not determine *where* the noise should be added. It also cannot synthesize mechanisms that use their privacy budget adaptively. An example of such a mechanism is a recently proposed variant of SVT, called Adaptive Sparse Vector with Gap [21]. This mechanism has extra flexibility for saving privacy budget on some queries, allowing it to keep iterating until its privacy budget is exhausted.

In this paper, we present DPGen, the first *fully automated* approach that can synthesize an accurate and differentially private program from a given non-private (noiseless) program. Significantly, DPGen employs a novel inference algorithm to automatically generate a mechanism sketch from a non-private program. We formalize the synthesis problem as a constrained optimization problem: maximizing *utility* while simultaneously satisfying *privacy* constraints in a transformed version of the mechanism sketch. DPGen then uses a counterexample-guided synthesis (CEGIS) loop [44] and an optimizer like Particle Swarm Optimization (PSO) [35] to synthesize and optimize the mechanism. Compared with KOLAHAL [41], the new optimization approach is shown to be more efficient. In some cases, KOLAHAL can take 900 to 5460 seconds to synthesize a mechanism, while DPGen can successfully synthesize an equivalent or more accurate version in 10 to 120 seconds.

Moreover, DPGen is equipped with a novel feature called a *while-private* loop, written as **while-priv** e **do** c. Semantically, the while-private loop (after synthesis) executes c whenever e evaluates to `true` and *as long as the dynamically tracked privacy budget has not been depleted.* Notably, this feature allows DPGen to synthesize sophisticated mechanisms such as Adaptive Sparse Vector with Gap [21] that try to minimize the amount of privacy budget spent in each loop iteration, and hence keep iterating until the privacy budget has been depleted. To the best of our knowledge, DPGen is the first program synthesizer that can automatically generate such sophisticated mechanisms.[1]

We evaluated DPGen on standard benchmarks that consist of various privacy mechanisms. For each privacy mechanism, we removed the randomness in it and asked DPGen to automatically synthesize a differentially private version. In all cases, DPGen was able to synthesize an equivalent or even more accurate version compared with the baseline. For adaptive mechanism that uses while-private loop, program synthesis is more complicated. But DPGen was still able to synthesize private and accurate mechanisms.

In summary, this paper makes the following contributions:

(1) DPGen, the first fully automated tool that can synthesize an accurate and differentially private mechanism from a noiseless non-private program.
(2) A novel inference algorithm that automatically generates a mechanism sketch (i.e., code with noise of unknown scales added to automatically selected program locations) from a non-private program (Section 4).
(3) A customized CEGIS loop that incrementally optimizes the tentative mechanism while generating its privacy proof (Section 5.2).
(4) A novel *while-private* feature that allows DPGen to synthesize adaptive privacy mechanisms (Section 5.3).
(5) Case studies and experimental comparisons between DPGen and KOLAHAL [41]. In addition to being able to synthesize more programs, DPGen also shows improvements on mechanims that both approaches can synthesize. In the benchamrks, DPGen generated identical or more accurate mechanisms within a considerably shorter amount of time (Section 6).

2 BACKGROUND

2.1 Differential Privacy

In this paper, we focus on *pure* differential privacy [23]. Intuitively, a data analysis A satisfies differential privacy if and only if for any dataset D, adding, removing, or changing a record in D has little impact on the analysis result. Therefore, a differentially private analysis reveals little about any data record being analyzed. Each analysis is built out of atomic components called differentially private mechanisms (privacy mechanisms for short). These components themselves satisfy differential privacy.[2]

More formally, we say that two datasets $D, D' \in \mathcal{D}$ are *adjacent*, written $D \sim D'$, when they only differ on one record. To offer privacy, a differentially private mechanism (or analysis), say $M : \mathcal{D} \to \mathcal{O}$, injects carefully calibrated random noise during its computation. We call the execution of M on D, written $M(D)$, the *original execution* and its execution on (neighboring) dataset D', written $M(D')$, the *related execution.* Intuitively, M (or A) is ϵ-differentially private for some constant ϵ if for any possible output $o \in \mathcal{O}$, the ratio between the probabilities of producing o on D and D' is bounded by e^{ϵ}:

Definition 2.1 (Pure Differential Privacy [22]). Let $\epsilon \geq 0$. A probabilistic computation $M : \mathcal{D} \to \mathcal{O}$ is ϵ-differentially private if $\forall D \sim D'$ (where $D, D' \in \mathcal{D}$) and $\forall o \in \mathcal{O}$, we have

$$\mathbb{P}[M(D) = o] \leq e^{\epsilon}\, \mathbb{P}[M(D') = o]$$

A differentially private analysis A interacts with a dataset through one or multiple privacy mechanisms that take a list of queries and their exact answers as input, and produce a differentially private (noisy) aggregation of them. An important factor to determine the amount of noise needed for privacy is the *sensitivity* of queries, which intuitively quantifies the maximum difference of the query results on adjacent databases. We use a vector $(q_1, q_2, \ldots)$ to denote the exact query answers from running a sequence of queries on a dataset and say that each query answer q_i has a *sensitivity* of Δ_i if its corresponding query has a *global sensitivity* of Δ_i:

Definition 2.2 (Global Sensitivity [24]). The global sensitivity Δ_f of a query f is $\sup_{D \sim D'} \left| f(D) - f(D') \right|$.

Similar to dataset adjacency, we say two vectors of query answers are *adjacent*, written $(q_1, q_2, \ldots) \sim (q'_1, q'_2, \ldots)$, when $\forall i.\ |q_i - q'_i| \leq \Delta_i$. Moreover, a privacy mechanism M satisfies ϵ-differential privacy

[1] Note that *while-private* is a programmer hint that the while loop should be executed in a best-effort way (a hallmark of the sparse vector family of privacy mechanisms) rather than exactly as many times as the non-private version would execute.

[2] In general, the privacy parameter of the analysis is upper bounded by the sum of the individual privacy parameters of the mechanisms [38].

if for all pairs of adjacent query answers $(q_1, q_2, \ldots) \sim (q'_1, q'_2, \ldots)$ and all outputs $o \in O$, we have $\mathbb{P}[M(q_1, q_2, \ldots, \text{params}) = o] \leq e^{\epsilon}\, \mathbb{P}[M(q'_1, q'_2, \ldots, \text{params}) = o]$, where params represent data-independent parameters (e.g., the value of ϵ) to M. As the goal of this paper is to synthesize privacy mechanisms, we assume that the sensitivity of inputs are either manually specified or computed by sensitivity analysis tools (e.g., [28, 40]).

One popular privacy mechanism is the Laplace Mechanism [23], which adds Laplace noise to query answers.

Theorem 2.3 (Laplace Mechanism [23]). *Let* $\texttt{Lap}\ (n)$ *be a sample from the Laplace distribution with mean 0 and scale n. The Laplace Mechanism takes as input a query answer q with sensitivity* Δ_q*, and a privacy parameter* ϵ*. It outputs* $q + \texttt{Lap}\ (\Delta_q/\epsilon)$ *and it satisfies* ϵ*-differential privacy.*

In this paper, we will use Laplace noise to also synthesize more sophisticated privacy mechanisms.

2.2 Randomness Alignment

To synthesize a privacy mechanism, we need to reason about its correctness (i.e., it must satisfy pure differential privacy with a given privacy parameter ϵ). To mechanize the correctness reasoning, we adopt the *Randomness Alignment* technique, a simple yet powerful proof technique that enables various verification tools and counterexample detectors [47–49].

Consider a privacy mechanism M and an arbitrary pair of adjacent vectors of query answers $(q_1, q_2, \ldots) \sim (q'_1, q'_2, \ldots)$. A randomness alignment is a function $\phi : \mathbb{R}^{\infty} \to \mathbb{R}^{\infty}$ that maps random samples used by an execution of M on $(q_1, q_2, \ldots)$ to random samples used by the adjacent execution of M on $(q'_1, q'_2, \ldots)$ such that both executions produce the same output.

For example, consider the mechanism $M(x) = x + \texttt{Lap}\ (\epsilon)$ that adds Laplace noise to a query answer x of sensitivity 1. Then, given any pair of adjacent query answers $q \sim q'$, the function $\phi(r) = r + q - q'$ is an alignment. The reason is that for any possible Laplace random sample η generated by $M(q)$, we have $q + \eta = q' + (\eta + q - q') = q' + \phi(\eta)$ (i.e., $M(q')$ produces the same result when its Laplace sample is $\phi(\eta)$).

To finish the privacy proof, we note that for Laplace distribution $\texttt{Lap}\ (\epsilon)$, the ratio of the probabilities of sampling η and $\phi(\eta)$ is bounded by $e^{\epsilon \max_{r \in \mathbb{R}} |\phi(r) - r|} = e^{\epsilon \max_{r \in \mathbb{R}} |q - q'|} \leq e^{\epsilon}$. Hence, the *privacy cost*, the natural log of this ratio, is bounded by ϵ.

In general, it is useful to treat the privacy cost as a function of the alignment needed for each sampling instruction. For each sampling instruction $\eta = \texttt{Lap}\ (r)$, we define the *distance* of η, written as $\widehat{\eta}$, as $\phi(\eta) - \eta$[3]. Then, the privacy cost of aligning the sample η is bounded by $\frac{|\widehat{\eta}|}{r}$. To find the overall privacy cost (i.e., the ϵ in pure differential privacy), we then take the summation of privacy cost of each sample generated in program execution, due to the Composition Theorem of pure differential privacy [24]. We note that since we can align each sample individually, randomness alignment is also applicable to sophisticated mechanisms where the composition theorem falls short [48, 49]. This is a key to automated synthesis of a variety of mechanisms studied in this paper.

2.3 Particle Swarm Optimization (PSO)

Prior tools using the Randomness Alignment technique (e.g., [47–49]) focus on *privacy* only; they model privacy proof as a constraint-solving problem which is solved by an external SMT solver. However, synthesizing DP mechanism is better described as a *constrained optimization* problem: maximizing *utility* among various candidates that have the same overall differential privacy parameter ϵ.

In this paper, we use Particle Swarm Optimization (PSO) [35] to help with the synthesis. PSO is a meta-heuristic optimization algorithm that is inspired by swarm behaviors such as birds in nature. It deploys a large population of candidate solutions ("particles") in the search space and the particles move around iteratively to find the best location. For each iteration, each particle updates its position and velocity according to a mathematical formula consisting of its own local best position, the swarms' best position and its previous velocity. By adopting this strategy, the entire swarm is guided towards the best solutions. PSO makes no assumption about the problem being optimized and is suitable for very large search spaces. This is well suited for our complex, non-differentiable optimization problem, which makes other gradient-based optimization methods inapplicable. Specifically for the synthesis task, each candidate mechanism in the search space corresponds to a particle in PSO, and the instantiations of the sketch mechanism serves as its position. For each iteration, every candidate explores the search space by changing itself slightly according to the current global best candidate (with the best utility), its own local best in history and the amount of changes from previous iterations. The global best solution is returned after a number of iterations.

2.4 Sparse Vector Technique (SVT)

In this paper, we use Sparse Vector Technique (SVT) [24] and its variants as running examples. Given a sequence of queries, SVT tries to find the first N queries whose query answers are likely[4] to be above a publicly known threshold T. When privacy is not a concern, the pseudo code of SVT's basic functionality is shown in Figure 1 (we call it `SVTBase`). For now, we can safely ignore the function signature. `SVTBase` checks each exact query answer: it outputs `true` (resp. `false`) if the query answer is above (resp. below) the threshold until N `true` outputs are produced.

To enforce differential privacy, SVT adds *carefully calibrated* independent Laplace noise both to the threshold (T) and each query answer ($q[i]$). The pseudo code is shown in Figure 1 (we call it `SVT`), where the changes are highlighted. The sampling instruction $\texttt{Lap}\ (r)$ draws one sample from the Laplace distribution with mean 0 and scale factor of $r \in \mathbb{R}$. For each query, the mechanism outputs `true` if the *noisy* query answer is above the *noisy* threshold; otherwise it outputs `false`. It is well-known that SVT satisfies ϵ differential privacy [24].

3 OVERVIEW

3.1 Challenges

The goal of this paper is to *automatically synthesize* a differentially private program (e.g., function `SVT`) from a base program that is not necessarily differentially private (e.g., function `SVTBase`). Like

[3] Here we abuse notation slightly by applying ϕ point-wise, letting $\phi(\eta)$ be the random sample M should use in place of η in the adjacent execution.

[4] The uncertainty is introduced by privacy requirements.

function SVTBase (T,N,size: num, q: list num$^\bullet$)
returns (out: list num), **bound**(ϵ)
precondition $\forall$ i. $-1 \le (\widehat{\mathtt{q}}[\mathtt{i}]) \le 1 \land$ N < size / 5

```
i := 0; count := 0;
while (i < size ∧ count < N)
  if (q[i] ≥ $T$) then
    out := true::out;
    count := count + 1;
  else
    out := false::out;
  i := i + 1;
```

function SVT (T,N,size: num, q: list num)
returns (out: list num)

```
1   i := 0; count := 0;
2   $\eta_1$ := Lap ($3/\epsilon$)
3   $T_\star := T + \eta_1$;
4   while (i < size ∧ count < N)
5     $\eta_2$ := Lap ($3N/\epsilon$);
6     if (q[i] $+\eta_2$ ≥ $T_\star$) then
7       out := true::out;
8       count := count + 1;
9     else
10      out := false::out;
11    i := i + 1;
```

Figure 1: Sparse Vector Technique.

function SVT-ALT (T,N,size: num, q: list num)
returns (out: list num)

```
i := 0; count := 0;
while (i < size ∧ count < N)
  $\eta_2$ := Lap ($size/\epsilon$);
  if (q[i] $+\eta_2$ ≥ $T$) then
    out := true::out;
    count := count + 1;
  else
    out := false::out;
  i := i + 1;
```

Figure 2: An alternative way of making SVTBase ϵ-private.

other program synthesis techniques [31, 44], the synthesized program must implement similar functionality to the original program / specification. Since a privacy mechanism injects noise to offer privacy, this can be more precisely stated as: any output of the original program is still possible for the synthesized program.

What makes DPGen distinguished from other program synthesizers is its capability of synthesizing a *private* and *useful* counterpart of the original program:

- Privacy: the synthesized program needs to inject sufficient noise in the right places to satisfy pure differential privacy, as formally defined in Definition 2.1.
- Utility: the synthesized program needs to carefully calibrate the injected noise to make the randomized outputs useful (i.e., to make the outputs "close" to the ones from the original program). This involves choosing the correct noise scales (including using no noise wherever it is safe to do so).

Next, we highlight the main challenges in both aspects.

Privacy. Developing differentially private mechanisms is a non-trivial task: injecting sufficient amount of noise in the right places and then proving correctness is notoriously tricky. For instance, Lyu et al. [36] catalog several incorrect variants of SVT, where each variant slightly modifies the functionality and/or injected noise of function SVT in Figure 1 (for now, safely ignore the annotations in the function signature). While the changes are minimal, the incorrect variants fail to meet their claimed differential privacy guarantees. For example, one variant tweaks the mechanism to output the noisy query answer when it is above the threshold. That is, it changes Line 7 of SVT by replacing true with $q[i] + \eta_2$. As a result, it fails to satisfy ϵ-differential privacy for any value of ϵ [36].

Utility. What makes synthesizing differentially private mechanisms even more challenging is that we also need to add as little noise as possible while maintaining the desired privacy levels (otherwise the noisy outputs may not be useful). For example, in the simplest case, if we increase the scale of noise injected at Lines 2 and 5 in SVT (Figure 1), the mechanism is still ϵ-differentially private. However, the extra randomness reduces the accuracy of SVT. Furthermore, utility is also affected by *where* the noise is added. For example, an alternative way of making function SVTBase ϵ-private is shown in Figure 2. Compared with SVT, SVT-ALT does not add any noise to the threshold T; instead, it injects Laplace noise Lap ($size/\epsilon$) (rather than Lap ($3N/\epsilon$)) to each query answer. This provides the same privacy guarantees (SVT and SVT-ALT both satisfy ϵ-differential privacy for the same value of ϵ). However, since N is typically much smaller than size (the total number of queries), SVT-ALT injects significantly more noise into its computation.

Handling these kinds of decisions during the synthesis process is a highly non-trivial task and requires deep understanding of the privacy cost introduced by each sampling instruction. For example, SVT and its correct variants [17, 21, 24, 36] have the interesting property that outputting false *does not* incur any privacy cost (i.e., the costs[5] are only incurred for making the threshold noisy and for outputting true). On the other hand SVT-ALT is too naive and incurs a privacy cost of $\epsilon/$size for each iteration of the while loop (for a total cost of ϵ).

Finally, in many mechanisms (including SVT) and its variants, one needs to decide how to divide up a total privacy budget ϵ among different parts of the mechanism (i.e., what should the privacy cost of each part of the mechanism be). In the case of SVT, a synthesizer would decide how much of the budget should be consumed by adding noise to threshold T and how much should be consumed

[5] The privacy cost of the threshold is $\epsilon/3$ and each of the N true outputs incurs a privacy cost of $2\epsilon/(3N)$.

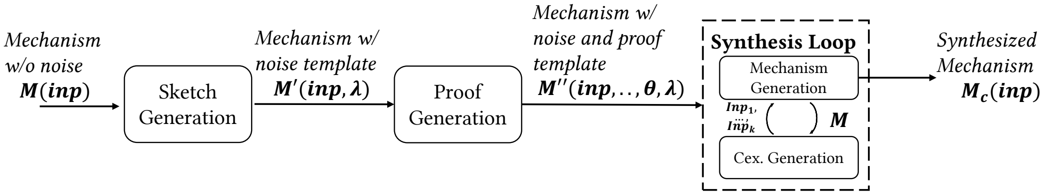

Figure 3: Overview of DPGen.

by the while loop. This is equivalent to deciding how much noise should be used for the threshold and how much should be used for the noisy query answers. In Figure 1, the noise scale for the threshold is $\sigma_1 = 3/\epsilon$ while the noise scale for each query answer is $\sigma_2 = 3N/\epsilon$. However, any choice of σ_1, σ_2 that satisfies $1/\sigma_1 + 2N/\sigma_2 = \epsilon$ will result in ϵ-differential privacy [36]. As shown by Lyu et al. [36], an approximately optimal ratio of $\sigma_1 : \sigma_2$ is $1 : (2N)^{2/3}$.

3.2 Approach Overview

To synthesize a privacy mechanism, DPGen adds proper amount of noise to the original program. This naturally involves two tasks: (1) finding program locations to add random noise to, and (2) finding the amount (scale) of each noise. Accordingly, DPGen synthesizes a privacy mechanism as shown in Figure 3.

Phase 1: Sketch Generation (Section 4). In Phase 1, DPGen generates a *sketch mechanism* with candidate locations for noise. The sketch mechanism might contain more locations for noise than needed, as the unnecessary ones will eventually be optimized away in Phase 2. Moreover, each noise location η_i is paired with a scale template $\mathcal{S}_i$ which consists of a set of unknown scale holes λ to be synthesized in Phase 2. We use $M'(inp, \lambda)$ to denote such a sketch mechanism with unknown scale holes.

Phase 2: Synthesis Loop (Section 5). Due to the tension between privacy and utility, mechanism synthesis cannot proceed without privacy in mind. Hence, DPGen next generates a transformed relational program with both scale templates $\mathcal{S}$ containing holes λ, and proof templates (in the form of alignments) $\mathcal{A}$ containing holes θ to be synthesized. Next, DPGen employs a customized CEGIS loop that iteratively refines a candidate mechanism (i.e., an instantiation of θ and λ) by generating more and more counterexamples (i.e., inputs that violates privacy constraints).

The CEGIS loop consists of two components. The counterexample generation component starts with a null mechanism (with $\theta = \vec{0}$ and $\lambda = \vec{1}$) and first searches for a counterexample (i.e., inputs) that *maximizes* the total number of privacy violations. The reason behind the optimization goal is the following: CEGIS benefits greatly from a good set of counterexamples; intuitively, a counterexample that violates maximum number of privacy constraints serves as better guides than others.

With a set of counterexamples, the mechanism generation component searches for a mechanism (i.e., an instantiation of the mechanism template) that *maximizes* utility while still being private. More specifically, the utility is defined both for privacy and accuracy:

- **Privacy.** A mechanism must be private for all previously seen counterexamples. Hence, any mechanism that is deemed as non-private on counterexamples has a negative utility score.
- **Accuracy.** DPGen is parameterized by either a default utility function (sum of variances), or a user-provided one. The utility function is used as the quality metric of each private candidate.

Once DPGen finds a mechanism where no counterexamples can be found, the CEGIS loop terminates and DPGen sends the mechanism to a verifier (we use CPAChecker [11]). Note that although we did not encounter any incorrect synthesized mechanism in our experiments, verification is needed in general as an optimizer might miss a solution when one exists.

Syntax of Source Language

Reals	r	$\in$	$\mathbb{R}$
Booleans	b	$\in$	$\{\mathtt{true}, \mathtt{false}\}$
Vars	x	$\in$	V
Linear Ops	$\oplus$	$::=$	$+ \mid -$
Other Ops	$\otimes$	$::=$	$\times \mid /$
Comparators	$\odot$	$::=$	$< \mid > \mid = \mid \leq \mid \geq$
Bool Exprs	$\mathbb{b}$	$::=$	$\mathtt{true} \mid \mathtt{false} \mid x \mid \neg \mathbb{b} \mid \mathbb{n}_1 \odot \mathbb{n}_2$
Num Exprs	$\mathbb{n}$	$::=$	$r \mid x \mid \mathbb{n}_1 \oplus \mathbb{n}_2 \mid \mathbb{n}_1 \otimes \mathbb{n}_2 \mid \mathbb{b}\, ?\, \mathbb{n}_1 : \mathbb{n}_2$
Expressions	e	$::=$	$\mathbb{n} \mid \mathbb{b} \mid e_1 :: e_2 \mid e_1[e_2]$
Commands	c	$::=$	$\mathbf{skip} \mid x := e \mid c_1; c_2 \mid \mathbf{return}\ e \mid$ $\mathbf{if}\ e\ \mathbf{then}\ (c_1)\ \mathbf{else}\ (c_2) \mid$ $\mathbf{while}\ e\ \mathbf{do}\ (c) \mid \mathbf{while\text{-}priv}\ e\ \mathbf{do}\ (c)$

Syntax of Target Language

Rand Vars	η	$\in$	H
Num Exprs	$\mathbb{n}$	$::=$	$\cdots \mid \eta$
Commands	$c^\bullet$	$::=$	$\mathbf{skip} \mid x := e \mid c_1^\bullet; c_2^\bullet \mid \mathbf{return}\ e \mid$ $\mathbf{if}\ e\ \mathbf{then}\ (c_1^\bullet)\ \mathbf{else}\ (c_2^\bullet) \mid$ $\mathbf{while}\ e\ \mathbf{do}\ (c^\bullet) \mid \eta := \mathsf{Lap}\ (\mathbb{n})$

Figure 4: DPGen: source and target language syntax.

4 SKETCH GENERATION

As discussed in Section 3, DPGen synthesizes a DP mechanism in two phases. In this section, we first show the syntax of its source and target languages. Then, we propose novel algorithms to identify potential violations of privacy in the source code, and then, to inject noise at proper locations to form a program sketch to be further analyzed in Phase 2 (Section 5).

4.1 Syntax of Source and Target Program

Source Language. The syntax of DPGen source code is listed in Figure 4. The source language models an expressive imperative language with the following standard features:

- Values of real numbers, Booleans and operations on them;
- Ternary expressions $\mathbb{b}$? $\mathbb{n}_1$: $\mathbb{n}_2$, which returns $\mathbb{n}_1$ (resp. $\mathbb{n}_2$) when $\mathbb{b}$ evaluates to `true` (resp. `false`);
- List of values as well as append (::) and projection ([]) operations on lists. Note that all lists are initialized to be empty.
- No-op commands (**`skip`**), assignments, sequential commands ($c_1; c_2$), return commands, if branches and while loops.

One novel feature of the source language is a while-private loop written as **`while-priv`** e **`do`** c; it requests the synthesizer to synthesize an *adaptive* privacy mechanism (e.g., Adaptive Sparse Vector with Gap [21]) that runs **`while`** e **`do`** c until the privacy budget is exhausted. This powerful feature allows the synthesized privacy mechanism to adaptively control the number of outputs based on the remaining privacy budget, in order to increase the amount of queries that they can process. We show how to synthesize the Adaptive Sparse Vector with Gap mechanism in Section 5.3.

Finally, the source language requires a few user-provided privacy specifications that the synthesizer should obey, including private inputs and their sensitivity[6], the desired privacy bound (i.e., ϵ in ϵ-differential privacy), as well as assumptions on the query answers. While we do not formalize the syntax of such specification, we use `type`$^\bullet$ to denote private input of some `type`, **bound**(ϵ) to denote the privacy budget, and specify sensitivity on private inputs ($\widehat{x}$ represents the sensitivity of x) and other assumptions on inputs as program precondition. For example, the source program `SVTBase` in Figure 1 specifies that query answers q are the only private inputs and their sensitivity is 1. Moreover, the mechanism assumes that N is much smaller than $size$, and the goal is to synthesize an ϵ-differentially private mechanism.

Target Language. The goal of DPGen is to synthesize a randomized mechanism that both preserves the source program's semantics and offers ϵ-differential privacy (where ϵ is annotated in the source program). Hence, the target language (shown in Figure 4) is similar to the source language, with a few important changes:

- The target language is probabilistic: it extends the (deterministic) source language with random variables η and sampling commands, written as $\eta := \mathsf{Lap}\ (\mathbb{n})$.
- The target language excludes the (non-executable) while-private loops; such loops in the source code are replaced by fully synthesized standard loops that terminate the loop whenever the privacy budget is exhausted.

Consider Figure 1. Function `SVT` is the target program synthesized from the source program `SVT-Base`. Note that they are very similar, but function `SVT` properly injects noise at various locations to satisfy ϵ-differential privacy.

4.2 Adding Noise Locations to Source Code

The first step of DPGen is to find a *set of program locations* in the source program where extra noise is needed. In this step, the primary concern is *privacy*; in other words, the lack of randomness in the source program violates differential privacy. Hence, we use static program analysis to (1) identify *where* privacy is violated in the source code, (2) infer a set of variables that might require randomness, and (3) instrument the source code to inject noise to the identified variables.

4.2.1 Identify Violations of Differential Privacy. Recall that DPGen is built on the Randomness Alignment technique (Section 2.2) to reason about privacy. Hence, instead of analyzing properties on distributions directly, as stated in Definition 2.1, we over-approximate "Violations of Differential Privacy" as "Violations of Alignment Requirements". Recall that randomness alignment requires that when running on a pair of adjacent private inputs, a program will produce identical outputs. Since the source code has no randomness, this requirement can be formalized as the standard non-interference property [30]. Hence, we use a static taint analysis (e.g., [33, 42, 46]) to identify violations in the source code:

- Initially, only the private inputs are tainted.
- The analysis tracks all *explicit flows* in the program.
- The analysis does not track, but reports all *implicit flows*, where a tainted value is used in a branch condition.
- The analysis reports all outputs with a tainted value.

For example, since query answers q are the only tainted inputs in `SVTBase` (Figure 1), the taint analysis finds one violation of privacy at Line 3, where the branch condition uses a tainted value $q[i]$. Since the taint analysis is standard, we omit the details here.

4.2.2 Identify Offending Variables. The static taint analysis returns a set of offending assignments $x := e$ and offending branches **`if`** e **`then`** c_1 **`else`** c_2, where e is tainted. We use $\mathbb{E}$ to represent the set of expressions that are either on the RHS of offending assignments, or in the branch condition of offending branches. Next, we need to infer a set of variables, that when randomized, will allow randomness alignment to exist on the randomized code. We call such a set of variables *offending variables.*

Consider the offending branch in our running example:

```
if q[i] ≥ T then ... else ...
```

where $q[i]$ is tainted while T is not. To make the branch outcome identical on two adjacent inputs $q[i] \sim q'[i]$, we can either inject noise to $q[i]$, or to T, or to both. While all options can allow the offending branch to be aligned, the difference will show up when we analyze their corresponding utility. For example, adding noise to T is crucial to make SVT useful; intuitively, it allows the noisy T to be reused across different loop iterations, which results in a less noisy program. We defer the discussion on utility to Section 5.2.2.

Based on the insight above, we define all variables used in any $e \in \mathbb{E}$ as offending variables. Note that by definition, the set of tainted variables is always a subset of offending variables.

4.2.3 Instrument Source Code with Extra Noise. Finally, DPGen injects noise with *unknown scales* (to be synthesized in later stages) to the source code. In particular, it injects Laplacian noise both at the definition of an offending variable, as well as right before its corresponding uses in an offending command. While adding noise to both locations might seem unnecessary at this point, DPGen eventually uses a utility optimizer (Section 5.2.2) to remove unnecessary noise in the code sketch.

[6] Determining the sensitivity of queries is crucial to produce an appropriate noise scale. Here, we assume that this information is provided by the user, as the sensitivities of simple queries, such as sum, mean and median, are fairly easy to compute as demonstrated in [24]. For more complex queries, users can either derive manually or use sensitivity analysis tools (e.g., [25, 37]) to calculate sensitivity.

```
function SVT-Sketch ($\epsilon$, T, N, size:num, q:list num, $\lambda$ : list num)
returns (out: list num)

1   $\eta_1 := \text{Lap}\ ((\lambda_0 + \lambda_1 \times N + \lambda_2 \times T + \lambda_3 \times size)/\epsilon)$
2   $T_\diamond := T + \eta_1$;
3   i := 0; count := 0;
4   while (i < size ∧ count < N)
5     $\eta_3 := \text{Lap}\ ((\lambda_4 + \lambda_5 \times N + \lambda_6 \times T + \lambda_7 \times size)/\epsilon)$;
6     $T_\diamond := T_\diamond + \eta_3$;
7     $\eta_2 := \text{Lap}\ ((\lambda_8 + \lambda_9 \times N + \lambda_{10} \times T + \lambda_{11} \times size)/\epsilon)$;
8     $q_\diamond := q[i] + \eta_2$;
9     if ( $q_\diamond \geq T_\diamond$ ) then
10      out := true::out;
11      count := count + 1;
12    else
13      out := false::out;
14    i := i + 1;
```

Figure 5: Sketch of SVT-Base with extra noise.

Moreover, as the scale of each Laplacian noise is unknown at this point, we replace them with *scale templates* as follows:

$$(\lambda_0 + \sum_{v_i \in \mathbb{V}} \lambda_i \times v_i)/\epsilon \text{ with fresh } \lambda_i$$

where $\mathbb{V}$ contains all *non-private* function parameters (as making scale private could violate privacy directly by revealing distribution statistics). Return to our running example of SVT, the code sketch with extra noise is shown in Figure 5 where all changes are highlighted. Notably, the sketched function explicitly adds scale parameters λ (we use λ_i instead of $\lambda[i]$ for better readability) as extra inputs to be optimized later. No noise is injected at Line 2 for $q[i]$, essentially an iterator of q, as it is not in scope at that point.

Hereafter, we use $M(inp)$ and $M'(inp, \lambda)$ to represent the original program with inputs *inp* and mechanism sketch with scale parameters λ respectively.

5 SYNTHESIS AND OPTIMIZATION

In Phase 2, DPGen completes program synthesis with two sub-goals:

- It synthesizes and optimizes the randomness alignment of each sampling instruction; a sampling instruction with alignment 0 implies that the instruction can be removed without violating differential privacy.
- It synthesizes and optimizes the scales λ in the sketch code from Phase 1 to offer good utility.

The main challenge is that instead of synthesizing *some* privacy proof (as done in prior work with proof synthesis [3, 47]) or *optimize* scales with given randomness locations (as done in [41]), our goal is to synthesize and optimize both the proof (with fewest randomness locations) and scales.

We first introduce the optimization problem without any while-private loop in source code and assume a default utility function that minimizes sum of variances. Then, we propose a synthesis loop to optimize alignments and scales simultaneously. Finally, we generalize the approach to optimize sketch code with while-private loops and customized utility functions.

5.1 Mechanism Synthesis Problem

Reasoning about Privacy. To reason about privacy, DPGen uses a syntax-directed transformation from the sketch program to non-probabilistic relational code with explicit alignments and proof obligations (i.e., assertions to ensure privacy). For commands, each transformation rule has the following format:

$$\vdash \Gamma \ \{c \rightharpoonup c'\} \ \Gamma'$$

where a typing environment Γ tracks for each program variable x its data type with its *distance* written as $\widehat{x}$. Recall that in the Randomness Alignment technique, the distance of a variable is defined as its value difference across two executions on adjacent query answers (Section 2.2). Moreover, c and c' are the sketch code and relational code respectively, and the flow-sensitive type system also updates typing environment to Γ' after command c.

Most importantly, the transformation inserts assertions to ensure the following (informal) soundness property:

$$\text{if } M'(inp, \lambda) \text{ is transformed to } M''(inp, \widehat{inp}, sample, \theta, \lambda), \text{ then}$$
$$\exists \theta, \lambda.\ \forall inp, \widehat{inp}, sample.\ \textit{all assertions in } M'' \textit{ pass}$$
$$\implies M'(inp, \lambda) \text{ is differentially private}$$

The most interesting transformation rule is for the sampling commands in the sketch code, which is shown in Figure 6. It performs the following important tasks:

(1) Each sampling command is replaced by a non-probabilistic counterpart ($\eta := sample[idx]; idx := idx + 1;$) that reads a sample from the instrumented function input *sample*.

(2) An alignment template (i.e., $\mathcal{A}$) is generated for each sampling command; each template contains a few holes, i.e., θ, which is also instrumented as function input. Here, we reuse the GenerateTemplate function proposed by CheckDP [47]. Intuitively, the alignments serve as a way to satisfy all inserted assertions in the transformed program. To do so, each alignment template $\mathcal{A}_i$ for random variable η_i contains distance variables of program variables that (1) appear in assertions, and (2) depend on η_i. Hence, GenerateTemplate takes the typing environment at the sampling command and all assertions as input, and properly calculates an alignment template, a linear function on a set of relevant distance variables as stated above. Since the GenerateTemplate function is identical to the one used in CheckDP [47], we only provide its pseudo-code in the Appendix. We refer interested readers to [47] for a more detailed discussion.

(3) The transformed code uses a distinguished variable $\mathbf{v}_\epsilon$ to track the overall privacy cost. Moreover, $\mathbf{v}_\epsilon$ is updated to $\mathbf{v}_\epsilon + |\mathcal{A}|/\mathcal{S}$, where $\mathcal{S}$ is the scale template instrumented in Phase 1. As discussed in Section 2.2, the update soundly accounts for the privacy cost of aligning the Laplace noise with alignment $\mathcal{A}$ and scale $\mathcal{S}$.

(4) Assertions are inserted in the transformed code to ensure the (informal) soundness property stated above. In particular, it inserts an assertion c_a that checks if the alignment function $\phi(r) =$

$$\frac{\mathcal{A} = \texttt{GenerateTemplate}(\Gamma, \text{All Assertions}) \quad c_a = \textbf{assert}\ (((\eta + \mathcal{A})\{\eta_1/\eta\} = (\eta + \mathcal{A})\{\eta_2/\eta\} \Rightarrow \eta_1 = \eta_2))}{\vdash \Gamma\ \{\eta := \mathsf{Lap}\ \mathcal{S} \rightharpoonup c_a; \eta := sample[idx]; idx := idx + 1; \mathrm{v}_\epsilon := \mathrm{v}_\epsilon + |\mathcal{A}|/\mathcal{S}; \widehat{\eta} := \mathcal{A};\}\ \Gamma[\eta \mapsto \mathsf{num}_*]}\ \text{(T-Laplace)}$$

Figure 6: A snippet of program transformation rules. $\mathcal{S}$ represents the scale template instrumented in Phase 1. Distinguished variable v_ϵ and assertions are added to ensure differential privacy when all assertions are satisfied. The complete transformation rules are available in the Appendix.

$r + \mathcal{A}$ is injective (i.e., $\forall r_1, r_2 . \phi(r_1) = \phi(r_2) \implies r_1 = r_2$). This a fundamental requirement of alignment-based proof [49].

For example, the transformed program of the sketch mechanism in Figure 5 is shown in Figure 7 with the instrumented code highlighted. Here, each random variable η_i is paired with a corresponding alignment template $\mathcal{A}_i$ computed by GenerateTemplate:

$$\mathcal{A}_1 : \theta_0$$

$$\mathcal{A}_2 : (\Omega\ ?\ \theta_1 + \theta_2 \times \widehat{T_\diamond} + \theta_3 \times \widehat{q}[i] : \theta_4 + \theta_5 \times \widehat{T_\diamond} + \theta_6 \times \widehat{q}[i])$$

$$\mathcal{A}_3 : (\Omega\ ?\ \theta_7 + \theta_8 \times \widehat{T_\diamond} + \theta_9 \times \widehat{q}[i] : \theta_{10} + \theta_{11} \times \widehat{T_\diamond} + \theta_{12} \times \widehat{q}[i])$$

where Ω represents the branch condition at Line 13. Note that the privacy cost of each alignment is soundly tracked at Lines 3, 8 and 10. Moreover, the distances of variables (e.g., $T_\diamond$ and $q_\diamond$) are properly updated after each assignment. Finally, the transformed code contains assertions to ensure that (1) two related execution of the sketch mechanism will follow the same control flow (e.g., Lines 14 and 18); (2) The distances of output expressions must be zero (not present in Figure 7 since the output values are already zero-distance literals; and (3) the overall privacy cost of the program does not exceed the privacy budget (e.g., Line 21) .

Since the other transformation rules are mostly identical to those introduced in CheckDP [47] and the soundness property is a direct implication of Theorem 3 in [47], we include the full transformation rules in the Appendix for completeness, and omit the formal statement of the soundness property and its proof in this paper.

Reasoning about Utility. Note that utility is a property of an instantiation of the mechanism sketch (i.e., fully synthesized program with concrete scales). Hence, reasoning about utility is relatively easy on the mechanism sketch $M'(inp, \lambda)$. The only interesting part is that utility computation should also take into account the alignments θ, as a random variable with $\theta_i = 0$ implies that the variable is unnecessary from the privacy perspective; hence, it will be removed in the final synthesized code.

In general, the particular metrics of utility might be application- and data-specific. DPGen is designed to be modular: users can plug in their customized utility metrics, and even sample data to optimize the utility of the synthesized privacy mechanism. Hence, in general, DPGen is parameterized by a utility function $\texttt{Utility}(M', \theta, \lambda)$, where M' is mechanism sketch and θ, λ are the synthesized alignments and scales respectively. By default, DPGen uses the sum of variances of all random variables as the utility function (note that DPGen currently only supports Laplace noise):[7]

$$\texttt{Utility}(M', \theta, \lambda) = -(\sum_{\{\mathcal{S}_i :\ \mathcal{A}_i \neq 0\}} 2\mathcal{S}_i^2) \qquad (1)$$

function TRANSFORMED SVT (T, N, size, q, $\widehat{\text{q}}$, *sample*, θ, λ)
returns (out)

```
1   $\mathrm{v}_\epsilon$ := 0; idx = 0;
2   $\eta_1$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_1}$ := $\mathcal{A}_1$;
3   $\mathrm{v}_\epsilon$ := $|\mathcal{A}_1|/\mathcal{S}_1$;
4   $T_\diamond$ := $T + \eta_1$; $\widehat{T_\diamond}$ := $\widehat{\eta_1}$;
5   count := 0; i := 0;
6   while (count < N ∧ i < size)
7     $\eta_3$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_3}$ := $\mathcal{A}_3$;
8     $\mathrm{v}_\epsilon$ := $\mathrm{v}_\epsilon + |\mathcal{A}_3|/\mathcal{S}_3$;
9     $\eta_2$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_2}$ := $\mathcal{A}_2$;
10    $\mathrm{v}_\epsilon$ := $\mathrm{v}_\epsilon + |\mathcal{A}_2|/\mathcal{S}_2$;
11    $T_\diamond$ := $T + \eta_3$; $\widehat{T_\diamond}$ := $\widehat{T_\diamond}$ + $\widehat{\eta_3}$;
12    $q_\diamond$ := $q[i] + \eta_2$; $\widehat{q_\diamond}$ := $\widehat{q}$[i] + $\widehat{\eta_2}$;
13    if ($q_\diamond \geq T_\diamond$) then
14      assert ($q_\diamond$ + $\widehat{q_\diamond}$ $\geq$ $T_\diamond + \widehat{T_\diamond}$);
15      out := true::out;
16      count := count + 1;
17    else
18      assert ($\neg$($q_\diamond$ + $\widehat{q_\diamond}$ $\geq$ $T_\diamond + \widehat{T_\diamond}$));
19      out := false::out;
20    i := i + 1;
21  assert ($\mathrm{v}_\epsilon \leq \epsilon$);
```

Figure 7: Transformed mechanism of SVT-Sketch by CheckDP. The instrumented codes are underlined. For better readability, the proof and scale templates are represented by $\mathcal{A}_i$ and $\mathcal{S}_i$, respectively.

where $\mathcal{A}_i$ and $\mathcal{S}_i$ denote the synthesized scale and alignment for random variable η_i. As discussed earlier, we explicitly exclude the ones with 0 alignments, since they are unnecessary.

Note that to compute utility based on the default utility function, there is no need to execute M'. Hence, synthesizing privacy mechanisms with the default utility function is very efficient. Moreover, despite its simplicity, it allows us to synthesize many privacy mechanisms (Section 6). For now, we assume the default utility function is in use; how to synthesize with more complicated utility function is deferred to Section 5.3.

5.2 Mechanism Optimization Problem

Recall that the goal of DPGen is to generate an *accurate* and *private* mechanism. That is, for a search space of alignment holes Θ and scale holes Λ, the constrained optimization problem is defined

[7] This is inspired by Lyu et al. [36] who derived the approximately optimal budget allocation of SVT by minimizing the variance of the branch (Line 3 in Figure 1).

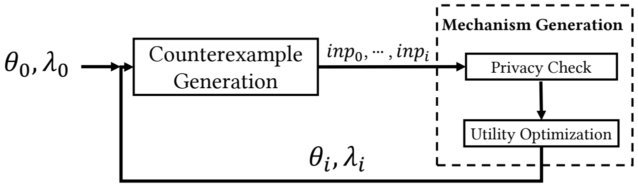

Figure 8: Overview of the search loop.

follows:

$$\begin{aligned} \max_{(\theta,\lambda)\in\Theta\times\Lambda} \quad & \texttt{Utility}(M', \theta, \lambda) \\ \text{s.t.} \quad & \forall inp.\text{all assertions in } M'' \text{ pass} \end{aligned}$$

To find alignment holes (θ) and scale holes (λ) according to the optimization problem above, DPGen uses a customized Counterexample-Guided Inductive Synthesis (CEGIS) [44] loop, as illustrated in Figure 8. Each synthesis iteration contains two steps:

- With a candidate mechanism (initialized with null mechanism of $\theta_0 = \vec{0}, \lambda_0 = \vec{1}$), the "counterexample generation" component tries to find inputs *inp* that "break" the privacy requirements (i.e., assertion violations in M'').
- With a set of counterexamples seen so far, the "mechanism generation" component synthesizes a privacy mechanism by optimizing the utility objective function (we use PSO as a black-box optimization technique in this paper) while satisfying all previously-generated counterexamples.

The CEGIS loop terminates when no counterexamples can be generated; then, the final privacy mechanism is returned.

Compared with the "bi-directional" search loop of CheckDP [47] that improves both privacy proof and counterexamples simultaneously, the CEGIS loop in Figure 8 is more standard, as there is no need to improve counterexamples for DPGen. Hence, the use of "bi-directional" CEGIS loop is not necessary.

Discussion on Soundness. Note that since most optimizers (including PSO [35] that DPGen uses) are unsound (i.e., they might miss a solution when one exists), the synthesized privacy program might be (in rare cases) non-private. To ensure soundness, the synthesized mechanism can be further verified by sound tools like CheckDP [47]. If verification fails, the counterexamples generated from CheckDP can be passed back to the CEGIS loop to continue the search. In practice, we did not experience any such unsound cases by running separate verification passes in CheckDP; we leave the integration of DPGen and CheckDP as future work.

5.2.1 Counterexample Generation. Given a candidate mechanism instantiated with some θ, λ, as well as a transformed mechanism with explicit alignments $M''(inp, \widehat{inp}, sample, \theta, \lambda)$, a counterexample C is defined as a solution of the following term:

$$\exists inp, \widehat{inp}, sample.\ \text{some assertions in } M''(inp, \widehat{inp}, sample, \theta, \lambda) \text{ fail.}$$

We note that this naive definition treats all counterexamples equally: two distinct counterexamples which violate 1 and 100 assertions respectively are both acceptable. To quantify and optimize the qualities of counterexamples (for better performance), we slightly modify the mechanism M'' to return the total number of assertion violations and use an optimizer to find a counterexample according to the following metric:

$$\max_{inp,\widehat{inp},sample} \quad M''(inp, \widehat{inp}, sample, \theta, \lambda)$$

Consider the transformed program of our running example in Figure 7 with a null mechanism ($\theta = \vec{0}, \lambda = \vec{1}$) for bootstrapping the process. The optimizer tries to find a counterexample that fails as many assertions as possible. Since no alignments are set to offset $\widehat{\mathtt{q}}[i]$ (the differences introduced by the query variable $q[i]$) in the assertions, a counterexample is found by making all queries fall in the `true` branch (i.e., query answers $q[i]$ are all above the threshold T). Suppose later, an improved alignment, which properly aligns the branch by $-\widehat{\mathtt{q}}[i]$, is fed in, which makes the false branch also incur a privacy loss. Therefore a counterexample will then be generated with query answers below the threshold, to make privacy cost exceed the total privacy budget (the last assertion in code).

5.2.2 Mechanism Generation. In general, mechanism generation runs on both the transformed program M'' and the sketch mechanism M' as follows:

- For any candidate solution (of θ, λ) that fails to satisfy any privacy constraint in M'' given any previously-generated counterexample, we assign a negative utility score to the solution.
- Otherwise, we use the utility function $\texttt{Utility}(M', \theta, \lambda)$ as its utility score.

Based on the utility scores defined above, DPGen uses an optimizer to find a privacy mechanism that optimizes the utility function while remaining differentially private.

Returning to our running example. The initial few discovered counterexamples likely include ones that go to different branches to cover all code paths. They can serve as good guides to lead the optimizer towards finding a more general solution, by aligning true and false branch differently, using a conditional alignment in the form of $\Omega\ ?\ \bullet : \bullet$, as other solutions will result in a negative utility score since they violate privacy.

Among the solutions that do satisfy all privacy constraints, the mechanism generation component ranks them based on their utility scores. Here, a solution that assigns a large noise (e.g., $size/\epsilon$) to the queries, although private, will have smaller utility scores than one which assigns $3N/\epsilon$ (since $N <$ `size`$/5$ in precondition). Moreover, a solution that assigns three random variables (two for the threshold, and one for the queries) will be less favorable due to larger sum of variances. This shows the power of our utility metric function in selecting good candidate solutions.

5.3 Handling While-Private Loop and User-Provided Utility Function

Next, we explore the full-fledged version of DPGen, with advanced features of while-private loop and user-provided utility function. We use a recently proposed variant of SVT that we call `AdaptiveSVT` (i.e., Adaptive Sparse Vector with Gap in [21]) as an example; its pseudo-code without noise is shown in Figure 9. Compared with SVT, there are three major changes:

- The mechanism uses while-private loop (Line 2) to request the synthesizer to adaptively answer as many quires as possible

```
function AdaptiveSVT-Base (T,N,size,$\sigma$: num, q: list num$^\bullet$)
returns (out: list num), bound($\epsilon$)
precondition $\forall$ i. $-1 \le (\widehat{q}[i]) \le 1$

i := 0;
while-priv (i < size)
  if (q[i] $- T \ge \sigma$) then
    out := (q[i] - T)::out;
  else
    if (q[i] $- T \ge 0$) then
      out := (q[i] - T)::out;
    else
      out := 0::out;
  i := i + 1;
```

Figure 9: AdaptiveSVT-Base, while-private feature is used to enable the synthesis of adaptive mechanisms.

(the input N specifies the *minimum* number of above-threshold queries that the algorithm should output).

- The mechanism partitions query answers into three ranges: $(-\infty, T)$, $[T, T+\sigma)$ and $[T+\sigma, +\infty)$ and requests DPGen to automatically allocate the total privacy budget among quires in each range.
- When $q[i] \ge T$, the mechanism releases the gap between $q[i]$ and T, instead of a constant.

Overall, the mechanism improves over SVT since it can use less privacy budget (i.e., add more noise) for queries that are much larger than the threshold T (i.e., in range $[T+\sigma, +\infty)$), in order to increase the amount of queries that it can process. Moreover, it is shown that the gap information can be released for free [21].

From program synthesis perspective, it poses two challenges for DPGen: (1) to synthesize executable code for while-private loop, and (2) to adopt a user-specified utility function.

Synthesizing while-private Loop. Recall that in the transformed program M'', there is an distinguished variable $\mathbf{v}_\epsilon$ that tracks the consumed privacy cost at each program point. The transformation of while-private loop (Figure 11) uses $\mathbf{v}_\epsilon$ to ensure that the loop terminates if $\mathbf{v}_\epsilon$ might exceed ϵ after one more iteration: it inserts an unknown bound on the privacy cost of running one iteration (◯) and ensures that the actual cost of *each iteration* never exceeds the bound with the assertion inserted at the end. We note that while-private (`while-priv`) is a new feature of DPGen; it enables DPGen to automatically infer and even optimize the loop termination conditions that are previous manually annotated in CheckDP [47].

Discussion on the Soundness of `while-priv`. Although `while-priv` is a new feature of DPGen, we note that this feature is transformed to a normal `while` loop by the transformation rule in Figure 11. By construction, the unknown bound on the privacy cost of each loop iteration (◯) is sound. Moreover, as a synthesized mechanism only contains normal `while` loops, a synthesized mechanism can further be verified by tools like CheckDP.

```
function AdaptiveSVT (T,N,size,$\sigma$: num,q: list num$_*$)
returns (out:list num)
precondition $\forall$ i. $-1 \le (\widehat{q}[i]) \le 1$

1   i := 0;
2   $\eta_1$ := Lap ($4/\epsilon$);
3   $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon + 4/\epsilon$;
4   $T_{\diamond 1}$ := $T + \eta_1$;
5   while (i < size $\wedge$ $\mathbf{v}_\epsilon \le \epsilon - 2\epsilon/(2N+3)$)
6     $\eta_2$ := Lap ($(4N+6)/\epsilon$);
7     $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon + (\Omega_{Top}\ ?\ 2 : 0) \times \epsilon/(4N+6)$;
8     $q_{\diamond 1}$ := q[i] + $\eta_2$;
9     if ($q_{\diamond 1}$ - $T_{\diamond 1} \ge \sigma$) then
10      out := ($q_{\diamond 1}$ - $T_{\diamond 1}$)::out;
11    else
12      $\eta_3$ := Lap ($(2N+3)/\epsilon$);
13      $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon + (\Omega_{Middle}\ ?\ 2 : 0) \times \epsilon/(2N+3)$;
14      $q_{\diamond 2}$ := q[i] + $\eta_3$;
15      if ($q_{\diamond 2} - T_{\diamond 1} \ge 0$) then
16        out := ($q_{\diamond 2} - T_{\diamond 1}$)::out;
17      else
18        out := 0::out;
19    i := i + 1;
```

Figure 10: Synthesized AdaptiveSVT based on AdaptiveSVT-Base. Ω_{Top} and Ω_{Middle} stands for the branch condition at Line 9 and Line 15, respectively.

User-Specified Utility Function. Consider the default utility function that minimizes the sum of variances of all random variables (Equation 1). A solution that outputs no queries at all always beats other solutions since it injects no noise (`Utility` $= \infty$). However, the solution fails the requirement of outputting at least N queries in total, where N is a parameter of the mechanism. Therefore, a more informative utility function is required for Adaptive SVT.

Recall that the family of SVTs are designed to report whether a query answer is above a certain threshold or not. Hence, a natural utility measurement is the number of true positives and false positives of the above-threshold queries. Moreover, the design of Adaptive SVT assumes that many queries are well-above the threshold; this allows mechanism to add relatively large noise to the outliers without impacting number of false positives. Finally, by definition, the synthesized privacy mechanism should output at least N queries in total, where N is a parameter of the mechanism.

Hence, we use a sample input inp_{ex} where many queries are well-above the threshold, create a modified sketch mechanism M'_θ that removes η_i from M' whose alignment is 0, and returns the number of true positives ($\#tp$) and false positives ($\#fp$). Hence, the user-specified utility function is defined as follows:

$$\texttt{Utility}(M', \theta, \lambda) = (\#tp - \#fp) - p \times \max(N - (\#tp + \#fp), 0) \quad (2)$$

where p is the penalty of outputting less than N outputs, which we set as 1 to guide the search to favor a solution that answers at least N above-threshold queries.

$$\frac{\cdots}{\vdash \Gamma\ \{\texttt{while-priv}\ e\ \texttt{do}\ c \rightharpoonup \cdots ; (\texttt{while}\ (e \wedge \mathrm{v}_\epsilon \le \epsilon - \bigcirc)\ \texttt{do}\ (\mathrm{v}_t = \mathrm{v}_\epsilon; \cdots ; \texttt{assert}\ (\mathrm{v}_\epsilon - \mathrm{v}_t \le \bigcirc)))\}\ \Gamma \sqcup \Gamma_f}\ \text{(T-While-Priv)}$$

Figure 11: The transformation rule of while-private loop; the parts identical to a standard while-loop are omitted for readability. The complete rule is available in the Appendix.

Choice of Utility Functions. The quality of the synthesized mechanism is dependent on the quality of the utility function, as the latter defines "utility" in the search. In general, a proper utility function of a privacy mechanism might be both data- and application-specific, such as the data- and application-specific utility function that we derived for Adaptive SVT. Nevertheless, for a variety of mechanisms, as showcased in our evaluation, the default utility function (i.e., the sum of variances of all random variables) already allows DPGen to synthesize high quality privacy mechanisms.

6 IMPLEMENTATION AND EVALUATION

We implemented a prototype[8] of DPGen in Python. The prototype uses the pyswarms package [39] for PSO optimization. For each component in the CEGIS loop (Figure 8), we run the optimization for 500 iterations. To speedup searching, DPGen stops early if the best value stays within tolerance $t = 1$ for 50 iterations. By default, the search space for each hole in the alignments and scales is set to $[-10, 10]$ and $[0, 10]$ respectively. This is chosen based on the typical values of those parameters in correct privacy mechanisms. Moreover, the number of query answers is set to 100. DPGen automatically expands the search space for the holes and the number of query answers until a mechanism is successfully generated.

We note that the use of the optimizer is to *discover* a solution; the generated mechanism is eventually verified by an off-the-shelf sound verifier CPAChecker [11] with arbitrary array lengths. Moreover, to speed up the synthesis of adaptive mechanisms, we split the mechanism sketch into multiple sketches each with a unique combinations of different random variable locations, and run all sketches in parallel. To make the generated mechanism easier to read and more friendly for off-the-shelf verifier, we round up the scales of generated mechanism to nearest integer. However, this can be switched off if the user wants a more refined mechanism.

We evaluate DPGen on a Intel® Xeon® E5-2620 v4 CPU machine with 64 GB memory. Table 1 lists the synthesized scales, alignments and synthesis time for each mechanism that we introduce next.

6.1 Case Studies

To illustrate the expressiveness of DPGen and its capability of synthesizing privacy mechanisms of different characteristics, we used a standard benchmark as seen in prior works [3, 12, 20, 47, 48], including SVT under different conditions, other variants of SVT such as NumSVT and GapSVT, the Report Noisy Max mechanism [24], Partial Sum and Smart Sum [16]. The psudo-code and transformed program of each case study can be found in the Appendix. All of the mechanisms we synthesize are proved to satisfy ϵ-differential privacy. Here we focus on the most interesting mechanisms; the rest can be found in the Appendix.

SVT under Different Conditions. As discussed earlier (Section 3.1), the SVT-Base program can be made private in multiple ways, and its utility might depend on the characteristics of the data being analyzed as well.

For example, the standard SVT mechanism makes the use of the fact that the number of above-threshold queries to answer (N) is relatively small (hence the name "sparse vector"). This is specified by the precondition $N < size/5$ in the function signature. Given this assumption on data, DPGen successfully synthesizes the privacy mechanism shown in Figure 1, which is the standard SVT mechanism.

In the SVT-All case, we change the assumption to be most queries answers are above the threshold. Under this assumption, the standard SVT is no longer preferred, as intuitively, the privacy cost paid for the threshold can no longer be offset by its gain from paying no cost for the below-threshold queries. As expected, DPGen synthesizes a privacy mechanism that only injects noise to query answers but not to the threshold, which is the same as the mechanism shown in Figure 2.

In the SVT-Inverse case, we flip SVT to answer at most N *below-threshold* queries, rather than to answer at most N above-threshold queries. Accordingly, the same precondition $N < size/5$ in the function signature now specifies that the number of *below-threshold* queries to answer is relatively small. Not surprisingly, DPGen successfully synthesizes the dual of standard SVT, with flipped alignments on the `true` and `false` branches but the same scales and random variables.

Finding Approximately Optimal Budget Allocation For SVT. As shown in [36], the approximately optimal budget allocation between the threshold and queries for SVT is $1 : (1+(2N)^{\frac{2}{3}})$. Although DPGen currently lacks the ability to solve the optimization problem with a symbolic N, we analyze a case where input N of SVT is fixed to 1. Also, we disabled integer rounding for synthesizing the approximately optimal allocation for this particular instance of SVT. DPGen is able to synthesize a solution with scale $2.587430/\epsilon$ on η_1, the noise added to the threshold, and scale $3.259844/\epsilon$ on η_2, the noise added to each query answer; while the approximately optimal ones are $2.587401/\epsilon$ and $3.259960/\epsilon$ respectively when $N = 1$.

Variants of SVT Using while-private Loop. To showcase the power of while-private loop and user-provided utility function, we evaluate on two mechanisms that use these features. The first is Adaptive SVT, which is already introduced in Section 5.3. The second, called SVT-WhilePriv, is a modified version of SVT where the user simply uses a while-private loop and asks the synthesizer to adaptively adjust the privacy cost paid to the above and below threshold answers respectively:

```
i := 0;
while-priv (i < size)
  if (q[i] ≥ T) then
```

[8] Publicly available at https://github.com/cmla-psu/dpgen.

Table 1: Synthesized random variables with corresponding alignment proof. Ω_* stands for the branch condition in each mechanism, where $\Omega_{NM} = q_\diamond > bq \vee i = 0$, $\Omega_{SVT} = q_\diamond \geq T_\diamond$, $\Omega_{Top} = q_{\diamond 1} - T_{\diamond 1} \geq \sigma$, $\Omega_{Middle} = q_{\diamond 2} - T_{\diamond 1} \geq 0$. Unnecessary random variables that are removed in the optimization are omitted.

Mechanism	Random Variables						Time (s)	KOLAHAL [41]
	η_1		η_2		η_3			
	Scale	Alignment	Scale	Alignment	Scale	Alignment		
ReportNoisyMax	$2/\epsilon$	$\Omega_{NM}\ ?\ 1 - \widehat{q}[i] : 0$	N/A	N/A	N/A	N/A	120	1920
PartialSum	$1/\epsilon$	$-\widehat{\text{sum}}$	N/A	N/A	N/A	N/A	10	900
SmartSum	$2/\epsilon$	$-\widehat{sum} - \widehat{q}[i]$	$2/\epsilon$	$-\widehat{q}[i]$	N/A	N/A	25	5460*
SVT	$3/\epsilon$	1	$3N/\epsilon$	$\Omega_{SVT}\ ?\ 1 - \widehat{\text{q}}[i] : 0$	N/A	N/A	29	2640
SVT-Inverse	$3/\epsilon$	-1	$3N/\epsilon$	$\Omega_{SVT}\ ?\ 0 : -2$	N/A	N/A	28	N/A
SVT-All	N/A	N/A	$\texttt{size}/\epsilon$	$\Omega_{SVT}\ ?\ 1 : -1$	N/A	N/A	38	N/A
SVT (N = 1)	$2.587430/\epsilon$	1	$3.259844/\epsilon$	$\Omega_{SVT}\ ?\ 1 - \widehat{\text{q}}[i] : 0$	N/A	N/A	16	N/A
GapSVT	$3/\epsilon$	1	$3N/\epsilon$	$\Omega_{SVT}\ ?\ 1 - \widehat{\text{q}}[i] : 0$	N/A	N/A	25	N/A
NumSVT	$4/\epsilon$	1	$4N/\epsilon$	$\Omega_{SVT}\ ?\ 2 : 0$	$4N/\epsilon$	$-\widehat{q}[i]$	35	N/A
SVT-WhilePriv	$3/\epsilon$	1	$3N/\epsilon$	$\Omega_{SVT}\ ?\ 2 : 0$	N/A	N/A	617	N/A
AdaptiveSVT	$4/\epsilon$	1	$(4N+6)/\epsilon$	$\Omega_{Top}\ ?\ 1 - \widehat{q}[i] : 0$	$(2N+3)/\epsilon$	$\Omega_{Middle}\ ?\ 1 - \widehat{q}[i] : 0$	3026	N/A

* The ideal solution was ranked 4th among the candidates generated by KOLAHAL.

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

In both cases, we use the user-provided utility function of Equation 2 (Section 5.3). The utility function requires the user provide an example input for evaluation. To capture the characteristics of a typical usage of SVT, where the amount of above-threshold query answers is small, we designed an input as follows: we use a sample set of 100 query answers where 75 are well below the threshold ($\leq T - 1000$), 10 are well above the threshold ($\geq T + 1000$) and 15 close to the threshold ($= T + 50$). In both cases, the input N (the minimum number of above-threshold queries to answer) is set to 20 in order to avoid answering queries that are well-below the threshold.

Moreover, due to the nature of the utility function, which computes utility based on true positives and false positives, we need to run the sketch mechanism (with randomness) many iterations for a good estimation of the utility. In the evaluation, we set the number of iterations to 2500.

For SVT-WhilePriv, DPGen successfully synthesizes a privacy mechanism that is identical to standard SVT: the synthesized mechanism adds noise with scale $3/\epsilon$ (resp. $3N/\epsilon$) to the threshold (resp. each query answer). The synthesized while condition is **while** $(i < size \wedge \mathbf{v}_\epsilon \leq \epsilon - 2\epsilon/(3N))$. The synthesized program increments $\mathbf{v}_\epsilon$ by $\epsilon/3$ before the branch, increments it by $2\epsilon/(3N)$ in the true branch and leaves it unchanged in the false branch (as the alignment in that case is 0). Note that although the synthesized code is syntactically different from standard SVT, they have exactly the same semantics.

For AdaptiveSVT, DPGen synthesizes a version (last row in Table 1) that is different from the one proposed in [21]. However, we confirmed that the average utility score for the synthesized mechanism across 2500 iterations is 24.4, meaning that it almost answers all above-threshold queries in an accurate way, with no false positives. In this case, DPGen has successfully synthesized a *private* solution that offers better utility (as measured by the user-provided utility function) on the sample data compared with the original mechanism in [21]. In practice, a user could provide more sample data to avoid over-fitting, with the cost of a longer synthesis time.

Report Noisy Max. Another well-known privacy mechanism is Report Noisy Max: it finds the *identity* of the item with the maximum score in the database. We present this mechanism in a simplified manner: given a series of query answers as inputs, the mechanism returns the index of the query with maximum answer.

The synthesis of Report Noisy Max requires an extension of the alignment-based proof technique called Shadow Execution [48], which is also supported by DPGen. While the synthesis time for Report Noisy Max is slightly longer than other mechanisms without while-private loop, DPGen synthesizes a private mechanism that is the same as the standard Report Noisy Max.

Partial Sum and Smart Sum. These two mechanisms release aggregate statistics. Partial Sum simply sums up all query answers and directly release the final sum. A more advanced mechanism [16] is proposed by Chen et al. to release the prefix sum of a series query answers: $q[0], q[0] + q[1], \cdots, \sum_{i=0}^{T} q[i]$. The details of this mechanism can be found in the Appendix. Notably, these two mechanisms rely on a slightly different adjanceny definitions: at most one of the query answers can differ by at most 1:

$$\forall \texttt{i}.\ -1 \leq (\widehat{\texttt{q}}\texttt{[i]}) \leq 1 \wedge (\forall \texttt{i}.\ (\widehat{\texttt{q}}\texttt{[i]}) \neq 0 \Rightarrow (\forall \texttt{j}.\ \widehat{\texttt{q}}\texttt{[j] = 0}))$$

Despite such difference, DPGen is able to synthesizes both Partial Sum and Smart Sum.

Comparison with KOLAHAL [41]. As the implementation of KOLAHAL is not publicly available, we were unable to make a direct comparison with KOLAHAL on all benchmarks. The last column of Table 1 shows data that we collect from [41] when the corresponding mechanism is also evaluated on KOLAHAL (N/A is listed if the mechanism is not part of the experiments in [41]). We note that one difference between DPGen and KOLAHAL is that the latter synthesizes a set of candidate solutions instead of one; sometimes, the ideal solution might not be a top candidate: for example, the ideal solution for Smart Sum is ranked as the 4th one. Moreover, KOLAHAL requires manually-provided mechanism sketches where noise locations are annotated, while DPGen automatically generates sketches as discussed in Section 4.

6.2 Performance

We note that the synthesis time with the default utility function is significantly smaller than the one with a user-provided utility function (used for SVT-WhilePriv and AdaptiveSVT). The reason is that the default utility function does not need to execute the sketch mechanism at all. Moreover, among the ones using the default utility function, Report Noisy Max takes longer to synthesis, since it needs to use the Shadow Execution [48] feature of DPGen.

Comparison with KOLAHAL. We note that for the same mechanisms, the synthesis time of DPGen is considerably smaller than that of KOLAHAL. While this is not an apple-to-apple comparison, we contribute the efficiency to the reduced search space of our sketch generation algorithm and the qualities of the counterexamples generated in the search loop.

7 RELATED WORK

Synthesizing Differentially Private Algorithms. Closest to our work is the synthesizer KOLAHAL recently proposed by Roy et al. [41]. KOLAHAL takes, as inputs, a sketch mechanism with noise expressions in known locations as holes and a finite grammar for noise expressions, and leverages counterexamples generated by StatDP [20] and continuous optimization approximation to guide the optimization of noise functions. It supports multiple noise distributions (Laplace, Exponential) and is the first tool capable of synthesizing complex differential privacy mechanisms including NoisyMax, SVT and SmartSum. Compared with KOLAHAL, DPGen (1) automatically generates the locations of randoms variables, (2) is more efficient in synthesizing non-adaptive mechanisms due to reduced search space of the templates, and (3) is able to synthesize sophisticated mechanisms such as AdaptiveSVT. An earlier synthesizer [43] relies on user supplied examples and uses a sensitivity-directed program synthesis technique based on DFuzz [28]. However, it can only synthesize simple mechanisms where the privacy analysis follows directly from the composition theorem.

Proving and Disproving Differential Privacy. Differential privacy has been a fruitful target for formal verification due to its compositional property. Fuzz [40] and DFuzz [28] use linear dependent type systems to analyse program sensitivity and prove (pure) differential privacy properties. Amorim et al. [18] extend such systems to work under approximate differential privacy. Barthe et al. [5–9] developed several customized relational logics based on probabilistic couplings for reasoning about differential privacy. Zhang and Kifer [49] introduced the Randomness Alignment technique as a simpler but more restricted alternative of probabilistic coupling. Wang et al. [48] extend the type system in [49] to allow more complicated Randomness Alignment functions to be used for sophisticated mechanisms. Albarghouthi and Hsu [3] synthesize probabilistic couplings and randomness alignment into coupling strategies, creating the first fully automated tool capable of generating coupling proofs for complex mechanisms.

A complementary line of work [12, 13, 20] is concerned with developing automated techniques to search for counterexamples that witness violations of differential privacy. StatDP [20] uses statistical hypothesis testing to demonstrate high probability of privacy violations. DP-Finder [12] uses symbolic differentiation and gradient descent to search for counterexamples. More recently, DP-Sniper [13] trains a classifier – a parametric family of posterior probability distributions to predict if an observed output is likely generated from one of two possible inputs, and use this classifier to select a set of outputs that can best distinguish these two inputs. All these methods rely on *sampling* – running an algorithm hundreds of thousands of times to estimate the output distribution of mechanisms and generate counterexample candidates/training data.

Recent work [4, 26, 47] targets both proving and disproving differential privacy. CheckDP [47] also relies on the Randomness Alignment technique. It reduces the search space of proofs to templates with holes. Moreover, it embeds a novel bi-directional CEGIS loop to improve proof and counterexample simultaneously. Barthe et al. [4] identify a non-trivial class of programs where checking (pure and approximate) differential privacy is decidable. However, these programs only allow a bounded number of samples from the Laplace distribution, and their inputs and outputs are from a finite domain. Farina [26] builds a relational symbolic execution framework, which when combined with probabilistic couplings, is able to prove differential privacy for SVT or generate failing traces for its two incorrect variants.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we present DPGen, an automated differential privacy mechanism synthesizer that is able to synthesize sophisticated DP mechanisms such as adaptive mechanisms. DPGen employs a novel approach to automatically generate sketch mechanisms with potential random variables, and uses an enhanced CEGIS loop to fill the holes in the sketch according to customizable utility functions. Compared with recent synthesis work, DPGen is reasonably faster in synthesizing non-adaptive mechanisms, and is the only tool that is powerful enough to synthesize sophisticated adaptive ones. Evaluations show DPGen synthesizes a variety of non-adaptive mechanisms within minutes and adaptive ones within an hour.

Future work includes exploring more utility metrics for optimizing mechanism, as well as extending DPGen to support solving the optimization problem symbolically, which provides more general forms of budget allocations to different random variables in the mechanism. Another possibility is to extend the underlying proof technique (i.e., randomness alignment) to support more complex mechanisms such as PrivTree, where the intermediate results depend on the data, but the aggregate result does not. Moreover, we focus on Laplace distribution due to its adoption in a variety of mechanisms as shown in our benchmark. In general, new random distributions can be added to alignment-based proofs in a modular way via extra typing rules, as showcased in [49]. Extending DPGen for other random distributions is another potential future direction.

ACKNOWLEDGMENTS

We thank the anonymous CCS reviewers for their insightful feedbacks. This work was supported by NSF Awards CNS-1702760, CNS-1931686 and a gift from Facebook.

REFERENCES

[1] John M. Abowd. 2018. The U.S. Census Bureau Adopts Differential Privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (London, United Kingdom) *(KDD '18)*. ACM, New York, NY, USA, 2867–2867.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[3] Aws Albarghouthi and Justin Hsu. 2017. Synthesizing Coupling Proofs of Differential Privacy. *Proceedings of ACM Programming Languages* 2, POPL, Article 58 (dec 2017), 30 pages.

[4] Gilles Barthe, Rohit Chadha, Vishal Jagannath, A. Prasad Sistla, and Mahesh Viswanathan. 2020. Deciding Differential Privacy for Programs with Finite Inputs and Outputs. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 141–154. https://doi.org/10.1145/3373718.3394796

[5] Gilles Barthe, George Danezis, Benjamin Gregoire, Cesar Kunz, and Santiago Zanella-Beguelin. 2013. Verified Computational Differential Privacy with Applications to Smart Metering. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium (CSF '13)*. IEEE Computer Society, Washington, DC, USA, 287–301.

[6] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Advanced Probabilistic Couplings for Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 55–67.

[7] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. ACM, New York, NY, USA, 749–758.

[8] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 97–110.

[9] Gilles Barthe and Federico Olmedo. 2013. Beyond Differential Privacy: Composition Theorems and Relational Logic for f-divergences Between Probabilistic Programs. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II* (Riga, Latvia) *(ICALP'13)*. Springer-Verlag, Berlin, Heidelberg, 49–60.

[10] Jean-Francois Bergeretti and Bernard A. Carré. 1985. Information-flow and Data-flow Analysis of While-programs. *ACM Trans. Program. Lang. Syst.* 7, 1 (Jan. 1985), 37–61. https://doi.org/10.1145/2363.2366

[11] Dirk Beyer and M. Erkan Keremoglu. 2011. CPACHECKER: A Tool for Configurable Software Verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 184–190.

[12] Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. 2018. DP-Finder: Finding Differential Privacy Violations by Sampling and Optimization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 508–524.

[13] B. Bichsel, S. Steffen, I. Bogunovic, and M. Vechev. 2021. DP-Sniper: Black-Box Discovery of Differential Privacy Violations using Classifiers. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 391–409. https://doi.org/10.1109/SP40001.2021.00081

[14] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. 2017. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 441–459. https://doi.org/10.1145/3132747.3132769

[15] U. S. Census Bureau. 2019. On The Map: Longitudinal Employer-Household Dynamics. https://lehd.ces.census.gov/applications/help/onthemap.html#!confidentiality_protection.

[16] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. 2011. Private and Continual Release of Statistics. *ACM Trans. Inf. Syst. Secur.* 14, 3, Article 26 (Nov. 2011), 24 pages.

[17] Yan Chen and Ashwin Machanavajjhala. 2015. On the Privacy Properties of Variants on the Sparse Vector Technique. http://arxiv.org/abs/1508.07306.

[18] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. *Probabilistic Relational Reasoning via Metrics.* IEEE Press.

[19] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. 2017. Collecting Telemetry Data Privately. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., USA, 3574–3583. http://dl.acm.org/citation.cfm?id=3294996.3295115

[20] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting Violations of Differential Privacy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS 1́8)*. ACM, New York, NY, USA, 475–489.

[21] Zeyu Ding, Yuxin Wang, Danfeng Zhang, and Daniel Kifer. 2019. Free Gap Information from the Differentially Private Sparse Vector and Noisy Max Mechanisms. *PVLDB* 13, 3 (2019), 293–306. https://doi.org/10.14778/3368289.3368295

[22] Cynthia Dwork. 2006. Differential Privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II* (Venice, Italy) *(ICALP'06)*. Springer-Verlag, Berlin, Heidelberg, 1–12. https://doi.org/10.1007/11787006_1

[23] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography*, Shai Halevi and Tal Rabin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–284.

[24] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[25] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. ACM, New York, NY, USA, 1054–1067.

[26] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2021. Coupled Relational Symbolic Execution for Differential Privacy. *Programming Languages and Systems* 12648 (2021), 207.

[27] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[28] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 357–370. https://doi.org/10.1145/2429069.2429113

[29] Anna Gilbert and Audra McMillan. 2018. Property Testing for Differential Privacy. arXiv:1806.06427 [cs.CR]

[30] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, USA, 11–11. https://doi.org/10.1109/SP.1982.10014

[31] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* 46, 6 (2011), 62–73.

[32] Samuel Haney, Ashwin Machanavajjhala, John M. Abowd, Matthew Graham, Mark Kutzbach, and Lars Vilhuber. 2017. Utility Cost of Formal Privacy for Releasing National Employer-Employee Statistics. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 1339–1354. https://doi.org/10.1145/3035918.3035940

[33] Sebastian Hunt and David Sands. 2006. On Flow-sensitive Security Types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '06)*. ACM, New York, NY, USA, 79–90.

[34] Noah Johnson, Joseph P Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (2018), 526–539.

[35] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4. IEEE, 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

[36] Min Lyu, Dong Su, and Ninghui Li. 2017. Understanding the sparse vector technique for differential privacy. *Proceedings of the VLDB Endowment* 10, 6 (2017), 637–648.

[37] A. Machanavajjhala, D. Kifer, J. Abowd, J. Gehrke, and L. Vilhuber. 2008. Privacy: Theory meets Practice on the Map. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, Piscataway, NJ, USA, 277–286. https://doi.org/10.1109/ICDE.2008.4497436

[38] Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. ACM, New York, NY, USA, 19–30.

[39] Lester James V. Miranda. 2018. PySwarms, a research-toolkit for Particle Swarm Optimization in Python. *Journal of Open Source Software* 3 (2018). Issue 21. https://doi.org/10.21105/joss.00433

[40] Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/1863543.1863568

[41] S. Roy, J. Hsu, and A. Albarghouthi. 2021. Learning Differentially Private Mechanisms. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1033–1046. https://doi.org/10.1109/SP40001.2021.00060

[42] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003),

5–19.

[43] Calvin Smith and Aws Albarghouthi. 2019. Synthesizing Differentially Private Programs. *Proc. ACM Program. Lang.* 3, ICFP, Article 94 (July 2019), 29 pages. https://doi.org/10.1145/3341698

[44] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[45] Apple Differential Privacy Team. 2017. Learning with Privacy at Scale. https://machinelearning.apple.com/2017/12/06/learning-with-privacy-at-scale.html

[46] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 3 (1996), 167–187.

[47] Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng Zhang. 2020. CheckDP: An Automated and Integrated Approach for Proving Differential Privacy or Finding Precise Counterexamples. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 919–938. https://doi.org/10.1145/3372297.3417282

[48] Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving Differential Privacy with Shadow Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 655–669. https://doi.org/10.1145/3314221.3314619

[49] Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards Automating Differential Privacy Proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 888–901.

A FULL CASE STUDIES

In this section we list the examples we studied in this paper. For each mechanism we show the original mechanism (the user's input), and the transformed mechanism for the synthesis loop.

function NoisyMax (size: num, q: list num$^\bullet$)
returns max: num
precondition $\forall$ i. $-1 \leq \widehat{\mathtt{q}}$[i] ≤ 1

```
i := 0; bq := 0; max := 0;
while (i < size)
  if (q[i] > bq ∨ i = 0)
   max := i;
   bq := q[i];
  i := i + 1;
```

function Transformed NoisyMax (size, q, $\widehat{\mathtt{q}}$, *sample*, θ, λ)

```
$\mathbf{v}_\epsilon$ := 0; idx := 0;
i := 0; bq := 0; max := 0;
$\eta_2$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_2}$ := $\mathcal{A}_2$;
$\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon$ + $|\mathcal{A}_2|/\mathcal{S}_2$;
$bq_\diamond$ := bq + $\eta_2$;
$\widehat{bq_\diamond}$ := $\widehat{\eta_2}$;
$\widehat{\mathtt{bq}}^\circ$ := 0; $\widehat{\mathtt{bq}}^\dagger$ := 0; $\widehat{\mathtt{max}}^\circ$ := 0; $\widehat{\mathtt{max}}^\dagger$ := 0;
while (i < size)
  $\eta_1$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_1}$ := $\mathcal{A}_1$;
  $\mathbf{v}_\epsilon$ := $(\mathcal{L}_1 \,?\, \mathbf{v}_\epsilon : 0)$ + $|\mathcal{A}_1|/\mathcal{S}_1$;
  $q_\diamond$ := q[i] + $\eta_1$;
  $\widehat{q_\diamond}$ := $\widehat{\eta_1}$;
  $\eta_3$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_3}$ := $\mathcal{A}_3$;
  $\mathbf{v}_\epsilon$ := $(\mathcal{L}_2 \,?\, \mathbf{v}_\epsilon : 0)$ + $|\mathcal{A}_3|/\mathcal{S}_3$;
  $bq_\diamond$ := bq + $\eta_3$;
  $\widehat{bq_\diamond}$ := $\widehat{bq_\diamond}$ + $\widehat{\eta_3}$;
  if ($\mathcal{L}_1$) $\widehat{\mathtt{bq}}^\circ$ := $\widehat{\mathtt{bq}}^\dagger$; $\widehat{\mathtt{max}}^\circ$ := $\widehat{\mathtt{max}}^\dagger$;
  if ($\mathcal{L}_2$) $\widehat{\mathtt{bq}}^\circ$ := $\widehat{\mathtt{bq}}^\dagger$; $\widehat{\mathtt{max}}^\circ$ := $\widehat{\mathtt{max}}^\dagger$;
  if ($q_\diamond$ > $bq_\diamond$ ∨ i = 0)
    assert $(\mathtt{q[i]} + \widehat{\mathtt{q}}\mathtt{[i]} + \eta + \eta^\circ > \mathtt{bq} + \mathtt{bq}^\circ \vee \mathtt{i} = 0)$;
    max := i;
    $\mathtt{max}^\circ$ := 0;
    $\widehat{\mathtt{bq}}^\dagger$ := bq + $\widehat{\mathtt{bq}}^\dagger$ - (q[i] + $\eta$);
    bq := q[i] + $\eta$;
    $\widehat{\mathtt{bq}}^\circ$ := $\widehat{\mathtt{q}}^\circ$[i] + $\widehat{\eta}^\circ$;
  else
    assert $(\neg(\mathtt{q[i]} + \widehat{\mathtt{q}}\mathtt{[i]} + \eta + \eta^\circ > \mathtt{bq} + \mathtt{bq}^\circ \vee \mathtt{i} = 0))$;
  // shadow execution
  if $(\mathtt{q[i]} + \widehat{\mathtt{q}}^\dagger\mathtt{[i]} + \eta > \mathtt{bq} + \widehat{\mathtt{bq}}^\dagger \vee \mathtt{i} = 0)$
    $\widehat{\mathtt{bq}}^\dagger$ := q[i] + $\widehat{\mathtt{q}}^\dagger$[i] $+ \eta -$ bq;
    $\widehat{\mathtt{max}}^\dagger$ := i - max;
  i := i + 1;
assert $(\mathbf{v}_\epsilon \leq \epsilon)$;
```

Figure 12: Report Noisy Max and its transformed code.$\mathcal{L}_i$ stands for shadow execution selectors.

function NumSVT (T, N, size: num, q: list num$^\bullet$)
returns (out: list num), **bound**(ϵ)
precondition $\forall$ i. $-1 \leq$ ($\widehat{\mathtt{q}}$[i]) $\leq 1 \wedge$ N < size / 5

```
count := 0; i := 0;
while (count < N ∧ i < size)
  if (q[i] ≥ $T$) then
    out := (q[i])::out;
    count := count + 1;
  else
    out := false::out;
  i := i + 1;
```

function Transformed NumSVT (T, N, size, q, $\widehat{\mathtt{q}}$, *sample*, θ, λ)
returns (out)

```
$\mathbf{v}_\epsilon$ := 0; idx = 0;
$\eta_1$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_1}$ := $\mathcal{A}_1$;
$\mathbf{v}_\epsilon$ := $|\mathcal{A}_1|/\mathcal{S}_1$;
$T_\diamond$ := $T + \eta_1$; $\widehat{T_\diamond}$ := $\widehat{\eta_1}$;
count := 0; i := 0;
while (count < N ∧ i < size)
  $\eta_4$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_4}$ := $\mathcal{A}_4$;
  $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon + |\mathcal{A}_4|/\mathcal{S}_4$;
  $T_\diamond$ := $T + \eta_4$; $\widehat{T_\diamond}$ := $\widehat{T_\diamond}$ + $\widehat{\eta_4}$;
  $\eta_2$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_2}$ := $\mathcal{A}_2$;
  $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon + |\mathcal{A}_2|/\mathcal{S}_2$;
  $q_\diamond$ := $q[i] + \eta_2$; $\widehat{q_\diamond}$ := $\widehat{q}$[i] + $\widehat{\eta_2}$;
  if ($q_\diamond \geq T_\diamond$) then
    $\eta_3$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_3}$ := $\mathcal{A}_3$;
    $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon + |\mathcal{A}_3|/\mathcal{S}_3$;
    $q_\diamond$ := $q[i] + \eta_3$; $\widehat{q_\diamond}$ := $\widehat{q}$[i] + $\widehat{\eta_3}$;
    assert ($q_\diamond$ + $\widehat{q_\diamond}$ ≥ $T_\diamond + \widehat{T_\diamond}$);
    out := ($q_\diamond$)::out;
    count := count + 1;
  else
    assert (¬($q_\diamond$ + $\widehat{q_\diamond}$ ≥ $T_\diamond + \widehat{T_\diamond}$));
    out := false::out;
  i := i + 1;
assert $(\mathbf{v}_\epsilon \leq \epsilon)$;
```

Figure 13: Numerical Sparse Vector Technique and its transformed code.

```
function GapSVT-Base (T,N,size: num, q: list num$^\bullet$)
returns (out: list num), bound($\epsilon$)
precondition ∀ i. $-1 \le (\widehat{q}[i]) \le 1$ ∧ N < size / 5

i := 0; count := 0;
while (i < size ∧ count < N)
  if (q[i] ≥ $T$) then
    out := (q[i] - T)::out;
  else
    out := false::out;
    count := count + 1;
  i := i + 1;
```

```
function Transformed GapSVT (T,N,size,q, $\underline{\widehat{q}, \mathit{sample}, \theta, \lambda}$)
returns (out)

$\underline{\mathbf{v}_\epsilon \texttt{ := 0; idx = 0;}}$
$\underline{\eta_1 \texttt{ := } \mathit{sample}\texttt{[idx]; idx := idx + 1; } \widehat{\eta_1} \texttt{ := } \mathcal{A}_1\texttt{;}}$
$\underline{\mathbf{v}_\epsilon \texttt{ := } |\mathcal{A}_1|/\mathcal{S}_1\texttt{;}}$
$T_\diamond$ := $T + \eta_1$; $\underline{\widehat{T_\diamond} \texttt{ := } \widehat{\eta_1}\texttt{;}}$
count := 0; i := 0;
while (count < N ∧ i < size)
  $\underline{\eta_2 \texttt{ := } \mathit{sample}\texttt{[idx]; idx := idx + 1; } \widehat{\eta_2} \texttt{ := } \mathcal{A}_2\texttt{;}}$
  $\underline{\mathbf{v}_\epsilon \texttt{ := } \mathbf{v}_\epsilon + |\mathcal{A}_2|/\mathcal{S}_2\texttt{;}}$
  $\underline{\eta_3 \texttt{ := } \mathit{sample}\texttt{[idx]; idx := idx + 1; } \widehat{\eta_3} \texttt{ := } \mathcal{A}_3\texttt{;}}$
  $\underline{\mathbf{v}_\epsilon \texttt{ := } \mathbf{v}_\epsilon + |\mathcal{A}_3|/\mathcal{S}_3\texttt{;}}$
  $T_\diamond$ := $T + \eta_2$; $\underline{\widehat{T_\diamond} \texttt{ := } \widehat{T_\diamond} + \widehat{\eta_2}\texttt{;}}$
  $q_\diamond$ := $q[i] + \eta_3$; $\underline{\widehat{q_\diamond} \texttt{ := } \widehat{q}\texttt{[i]} + \widehat{\eta_3}\texttt{;}}$
  if ($q_\diamond \ge T_\diamond$) then
    $\underline{\textbf{assert } (q_\diamond + \widehat{q_\diamond} \ge T_\diamond + \widehat{T_\diamond})\texttt{;}}$
    $\underline{\textbf{assert } (\widehat{q_\diamond} - \widehat{T_\diamond} = 0)\texttt{;}}$
    out := ($q_\diamond - T_\diamond$)::out;
  else
    $\underline{\textbf{assert } (\neg(q_\diamond + \widehat{q_\diamond} \ge T_\diamond + \widehat{T_\diamond}))\texttt{;}}$
    out := false::out;
    count := count + 1;
  i := i + 1;
$\underline{\textbf{assert } (\mathbf{v}_\epsilon \le \epsilon)\texttt{;}}$
```

Figure 14: GapSVT and its transformed code.

```
function SVTBase-Inverse (T,N,size: num, q: list num$^\bullet$)
returns (out: list num), bound($\epsilon$)
precondition ∀ i. $-1 \le (\widehat{q}[i]) \le 1$ ∧ N < size / 5

i := 0; count := 0;
while (i < size ∧ count < N)
  if (q[i] ≥ $T$) then
    out := true::out;
  else
    out := false::out;
    count := count + 1;
  i := i + 1;
```

```
function Transformed SVT (T,N,size,q, $\underline{\widehat{q}, \mathit{sample}, \theta, \lambda}$)
returns (out)

$\underline{\mathbf{v}_\epsilon \texttt{ := 0; idx = 0;}}$
$\underline{\eta_1 \texttt{ := } \mathit{sample}\texttt{[idx]; idx := idx + 1; } \widehat{\eta_1} \texttt{ := } \mathcal{A}_1\texttt{;}}$
$\underline{\mathbf{v}_\epsilon \texttt{ := } |\mathcal{A}_1|/\mathcal{S}_1\texttt{;}}$
$T_\diamond$ := $T + \eta_1$; $\underline{\widehat{T_\diamond} \texttt{ := } \widehat{\eta_1}\texttt{;}}$
count := 0; i := 0;
while (count < N ∧ i < size)
  $\underline{\eta_2 \texttt{ := } \mathit{sample}\texttt{[idx]; idx := idx + 1; } \widehat{\eta_2} \texttt{ := } \mathcal{A}_2\texttt{;}}$
  $\underline{\mathbf{v}_\epsilon \texttt{ := } \mathbf{v}_\epsilon + |\mathcal{A}_2|/\mathcal{S}_2\texttt{;}}$
  $\underline{\eta_3 \texttt{ := } \mathit{sample}\texttt{[idx]; idx := idx + 1; } \widehat{\eta_3} \texttt{ := } \mathcal{A}_3\texttt{;}}$
  $\underline{\mathbf{v}_\epsilon \texttt{ := } \mathbf{v}_\epsilon + |\mathcal{A}_3|/\mathcal{S}_3\texttt{;}}$
  $T_\diamond$ := $T + \eta_2$; $\underline{\widehat{T_\diamond} \texttt{ := } \widehat{T_\diamond} + \widehat{\eta_2}\texttt{;}}$
  $q_\diamond$ := $q[i] + \eta_3$; $\underline{\widehat{q_\diamond} \texttt{ := } \widehat{q}\texttt{[i]} + \widehat{\eta_3}\texttt{;}}$
  if ($q_\diamond \ge T_\diamond$) then
    $\underline{\textbf{assert } (q_\diamond + \widehat{q_\diamond} \ge T_\diamond + \widehat{T_\diamond})\texttt{;}}$
    out := q[i]::out;
  else
    $\underline{\textbf{assert } (\neg(q_\diamond + \widehat{q_\diamond} \ge T_\diamond + \widehat{T_\diamond}))\texttt{;}}$
    out := false::out;
    count := count + 1;
  i := i + 1;
$\underline{\textbf{assert } (\mathbf{v}_\epsilon \le \epsilon)\texttt{;}}$
```

Figure 15: SVT-Inverse and its transformed code.

function PartialSum (size: num, q: list num$^\bullet$)
returns (out:num), **bound**(ϵ)
precondition
$\forall$i. $-1 \le (\widehat{\mathtt{q}}$[i]$) \le 1 \land (\forall$i. $(\widehat{\mathtt{q}}$[i]$) \neq 0 \Rightarrow (\forall$j. $\widehat{\mathtt{q}}$[j] = 0))

```
sum := 0; i := 0;
while (i < size)
  sum := sum + q[i];
  i := i + 1;
out := sum;
```

function Transformed PartialSum (size, q, $\widehat{\mathtt{q}}$, *sample*, θ, λ)
returns (out)

```
$\mathbf{v}_\epsilon$ := 0;
$sum_\diamond$ := 0; i := 0;
$\widehat{sum_\diamond}$ := 0;
while (i < size)
  $sum_\diamond$ := $sum_\diamond$ + q[i];
  $\widehat{sum_\diamond}$ := $\widehat{sum_\diamond}$ + $\widehat{q}$[i];
  i := i + 1;
$\eta_1$ := $sample$[idx]; idx := idx + 1; $\widehat{\eta_1}$ := $\mathcal{A}_1$;
$\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon$ + $|\mathcal{A}_1|/\mathcal{S}_1$;
$sum_\diamond$ := $sum_\diamond$ + $\eta_1$; $\widehat{sum_\diamond}$ := $\widehat{sum_\diamond}$ + $\widehat{\eta_1}$;
assert ($\widehat{sum_\diamond}$ == 0);
out := $sum_\diamond$;
assert ($\mathbf{v}_\epsilon \le \epsilon$);
```

Figure 16: PartialSum and its transformed code.

B PSEUDO-CODE FOR GENERATETEMPLATE

Here for completeness, we include the pseudo-code of the helper function GenerateTemplate proposed by [47]. Note that Depends is a variable dependence checking oracle which returns true if the expression e depends on the variable η. This oracle can be implemented as standard program dependency analysis [2, 27] or information flow analysis [10].

Algorithm 1: Template generation for $\eta := \mathsf{Lap}\ r$

```
input: $\Gamma_s$: typing environment at sampling command
       $A$: set of the generated assertions in the program
function GenerateTemplate($\Gamma_s$, $A$):
    $\mathbb{E} \leftarrow \emptyset, \mathbb{V} \leftarrow \emptyset$
    foreach assert ($e$) $\in A$ do
        if Depends($e, \eta$) then
            if assert ($e$) is generated by (T-If) then
                $e'$ $\leftarrow$ the branch condition of if
                $\mathbb{E} \leftarrow \mathbb{E} \cup \{e'\}$
            foreach $v \in Vars \cup \{e_1[e_2] | e_1[e_2] \in e\}$ do
                if $\Gamma_s \nvdash v : \mathcal{B}_0 \land$ Depends($e, v$) then
                    $\mathbb{V} \leftarrow \mathbb{V} \cup \{v\}$
    foreach $e \in \mathbb{E} \cup \mathbb{V}$ do
        remove $e$ from $\mathbb{E}$ and $\mathbb{V}$ if not in scope
    return $\mathbb{E}$,$\mathbb{V}$;
```

function SmartSum (M, T, size: num, q: list num$^\bullet$)
returns (out:list num), **bound**(ϵ)
precondition
$\forall$i. $-1 \le (\widehat{\mathtt{q}}$[i]$) \le 1 \land (\forall$i. $(\widehat{\mathtt{q}}$[i]$) \neq 0 \Rightarrow (\forall$j. $\widehat{\mathtt{q}}$[j] = 0))

```
i := 0; next := 0; sum := 0;
while (i < size ∧ i ≤ T)
  if ((i + 1) mod M = 0) then
    next := sum + q[i];
    sum := 0;
    out := next::out;
  else
    next:= next + q[i];
    sum := sum + q[i];
    out := next::out;
  i := i + 1;
```

function Transformed SmartSum (M, T, size, q, $\widehat{\mathtt{q}}$, *sample*, θ, λ)
returns (out)

```
$\mathbf{v}_\epsilon$ := 0; idx := 0;
i := 0; $next_\diamond$ := 0; $sum_\diamond$ := 0;
$\widehat{sum_\diamond}$ := 0; $\widehat{next_\diamond}$ := 0;
while (i < size ∧ i ≤ T)
  if ((i + 1) mod M = 0) then
    $\eta_1$ := $sample$[idx]; idx := idx + 1;
    $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon$ + $|\mathcal{A}_1|$ / $\mathcal{S}_1$; $\widehat{\eta_1}$ := $\mathcal{A}_1$;
    $next_\diamond$ := $sum_\diamond$ + q[i] + $\eta_1$;
    $\widehat{next_\diamond}$ := $\widehat{sum_\diamond}$ + $\widehat{q}$[i] + $\widehat{\eta_1}$;
    $sum_\diamond$ := 0; $\widehat{sum_\diamond}$ := 0;
    assert ($\widehat{next_\diamond}$ = 0);
    out := next::out;
  else
    $\eta_2$ := $sample$[idx]; idx := idx + 1;
    $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon$ + $|\mathcal{A}_2|$ / $\mathcal{S}_2$; $\widehat{\eta_2}$ := $\mathcal{A}_2$;
    $next_\diamond$ := $next_\diamond$ + q[i] + $\eta_2$;
    $\widehat{next_\diamond}$ := $\widehat{next_\diamond}$ + $\widehat{q}$[i] + $\widehat{\eta_2}$;
    $\eta_3$ := $sample$[idx]; idx := idx + 1;
    $\mathbf{v}_\epsilon$ := $\mathbf{v}_\epsilon$ + $|\mathcal{A}_3|$ / $\mathcal{S}_3$; $\widehat{\eta_3}$ := $\mathcal{A}_3$;
    $sum_\diamond$ := $sum_\diamond$ + q[i] + $\eta_3$;
    $\widehat{sum_\diamond}$ := $\widehat{sum_\diamond}$ + $\widehat{q}$[i] + $\widehat{\eta_3}$;
    assert ($\widehat{next_\diamond}$ = 0);
    out := next::out;
  i := i + 1;
assert ($\mathbf{v}_\epsilon \le \epsilon$);
```

Figure 17: SmartSum and its transformed code.

C COMPLETE TRANSFORMATION RULES

In this section we list the transformation rules in Figure 18 for completeness. Note that most rules are identical to the ones in CheckDP [47], with the differences highlighted in gray.

Transformation rules for expressions with form $\Gamma \vdash e : \mathcal{B}_{\mathbb{n}}$

$$\frac{}{\Gamma \vdash r : \mathsf{num}_0 \mid \mathtt{true}}\ (\textsc{T-Num}) \qquad \frac{}{\Gamma \vdash b : \mathsf{bool} \mid \mathtt{true}}\ (\textsc{T-Boolean}) \qquad \frac{}{\Gamma, x : \mathcal{B}_0 \vdash x : \mathcal{B}_0 \mid \mathtt{true}}\ (\textsc{T-VarZero})$$

$$\frac{}{\Gamma, x : \mathcal{B}_* \vdash x : \mathcal{B}_{\widehat{x}} \mid \mathtt{true}}\ (\textsc{T-VarStar}) \qquad \frac{\Gamma \vdash e : \mathsf{bool} \mid C}{\Gamma \vdash \neg e : \mathsf{bool} \mid C}\ (\textsc{T-Neg}) \qquad \frac{\Gamma \vdash e_1 : \mathcal{B}_{\mathbb{n}_1} \mid C_1 \quad \Gamma \vdash e_2 : \mathcal{B}_{\mathbb{n}_2} \mid C_2}{\Gamma \vdash e_1 \oplus e_2 : \mathcal{B}_{\mathbb{n}_1 \oplus \mathbb{n}_2} \mid C_1 \wedge C_2}\ (\textsc{T-OPlus})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{num}_{\mathbb{n}_1} \mid C_1 \quad \Gamma \vdash e_2 : \mathsf{num}_{\mathbb{n}_2} \mid C_2}{\Gamma \vdash e_1 \otimes e_2 : \mathsf{num}_0 \mid C_1 \wedge C_2 \wedge (\mathbb{n}_1 = \mathbb{n}_2 = 0)}\ (\textsc{T-OTimes}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{num}_{\mathbb{n}_1} \mid C_1 \quad \Gamma \vdash e_1 : \mathsf{num}_{\mathbb{n}_2} \mid C_2}{\Gamma \vdash e_1 \odot e_2 : \mathsf{bool} \mid C_1 \wedge C_2 \wedge \begin{matrix}(e_1 \odot e_2) \Leftrightarrow \\ (e_1 + \mathbb{n}_1) \odot (e_2 + \mathbb{n}_2)\end{matrix}}\ (\textsc{T-ODot})$$

$$\frac{\Gamma \vdash e_1 : \mathcal{B}_{\mathbb{n}_1} \mid C_1 \quad \Gamma \vdash e_2 : \mathtt{list}\ \mathcal{B}_{\mathbb{n}_2} \mid C_2}{\Gamma \vdash e_1 :: e_2 : \mathtt{list}\ \mathcal{B}_{\mathbb{n}} \mid C_1 \wedge C_2 \wedge (\mathbb{n}_1 = \mathbb{n}_2 = 0)}\ (\textsc{T-Cons}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{list}\ \tau \mid C_1 \quad \Gamma \vdash e_2 : \mathsf{num}_{\mathbb{n}} \mid C_2}{\Gamma \vdash e_1[e_2] : \tau \mid C_1 \wedge C_2 \wedge (\mathbb{n} = 0)}\ (\textsc{T-Index})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \mid C_1 \quad \Gamma \vdash e_2 : \mathcal{B}_{\mathbb{n}_1} \mid C_2 \quad \Gamma \vdash e_3 : \mathcal{B}_{\mathbb{n}_2} \mid C_3}{\Gamma \vdash e_1\ ?\ e_2 : e_3 : \mathcal{B}_{\mathbb{n}_1} \mid C_1 \wedge C_2 \wedge C_3 \wedge (\mathbb{n}_1 = \mathbb{n}_2)}\ (\textsc{T-Select})$$

Transformation rules for commands with form $\vdash \Gamma\ \{c \rightharpoonup c'\}\ \Gamma'$

$$\frac{\Gamma \vdash e : \mathcal{B}_{\mathbb{n}} \mid C \quad \langle \mathbb{d},\ c \rangle = \begin{cases} \langle 0,\ \mathbf{skip} \rangle, & \text{if } \mathbb{n} == 0, \\ \langle *,\ \widehat{x} := \mathbb{n} \rangle, & \text{otherwise} \end{cases}}{\vdash \Gamma\ \{x := e; \rightharpoonup \mathbf{assert}\ (C); x := e; c\}\ \Gamma[x \mapsto \mathcal{B}_{\mathbb{d}}]}\ (\textsc{T-Asgn}) \qquad \frac{\vdash \Gamma\ \{c_1 \rightharpoonup c_1'\}\ \Gamma_1 \quad \vdash \Gamma_1\ \{c_2 \rightharpoonup c_2'\}\ \Gamma_2}{\vdash \Gamma\ \{c_1; c_2 \rightharpoonup c_1'; c_2'\}\ \Gamma_2}\ (\textsc{T-Seq})$$

$$\frac{\Gamma \vdash e : \mathcal{B}_{\mathbb{n}} \mid C}{\vdash \Gamma\ \{\mathbf{return}\ e \rightharpoonup \mathbf{assert}\ (C \wedge \mathbb{n} = 0); \mathbf{return}\ e\}\ \Gamma}\ (\textsc{T-Return}) \qquad \frac{}{\vdash \Gamma\ \{\mathbf{skip} \rightharpoonup \mathbf{skip}\}\ \Gamma}\ (\textsc{T-Skip})$$

$$\frac{\vdash \Gamma \sqcup \Gamma_f\ \{c \rightharpoonup c'\}\ \Gamma_f \quad \Gamma, \Gamma \sqcup \Gamma_f \Rrightarrow c_s \quad \Gamma_f, \Gamma \sqcup \Gamma_f \Rrightarrow c''}{\vdash \Gamma\ \{\mathbf{while}\ e\ \mathbf{do}\ c \rightharpoonup c_s; (\mathbf{while}\ e\ \mathbf{do}\ (\mathbf{assert}\ (⦇e, \Gamma⦈^{\circ}); c'; c''))\}\ \Gamma \sqcup \Gamma_f}\ (\textsc{T-While})$$

$$\frac{\vdash \Gamma \sqcup \Gamma_f\ \{c \rightharpoonup c'\}\ \Gamma_f \quad \Gamma, \Gamma \sqcup \Gamma_f \Rrightarrow c_s \quad \Gamma_f, \Gamma \sqcup \Gamma_f \Rrightarrow c''}{\vdash \Gamma\ \{\mathbf{while\text{-}priv}\ e\ \mathbf{do}\ c \rightharpoonup c_s; (\mathbf{while}\ (e \wedge \mathrm{v}_{\epsilon} \leq \epsilon - \bigcirc)\ \mathbf{do}\ (\mathbf{assert}\ (⦇e, \Gamma⦈^{\circ}); \mathrm{v}_t = \mathrm{v}_{\epsilon}; c'; c''; \mathbf{assert}\ (\mathrm{v}_{\epsilon} - \mathrm{v}_t \leq \bigcirc)))\}\ \Gamma \sqcup \Gamma_f}\ (\textsc{T-While-Priv})$$

$$\frac{\vdash \Gamma\ \{c_i \rightharpoonup c_i'\}\ \Gamma_i \quad \Gamma_i, \Gamma_1 \sqcup \Gamma_2 \Rrightarrow c_i'' \quad i \in \{1, 2\}}{\vdash \Gamma\ \{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightharpoonup \mathbf{if}\ e\ \mathbf{then}\ (\mathbf{assert}\ (⦇e, \Gamma⦈^{\circ}); c_1'; c_1'')\ \mathbf{else}\ (\mathbf{assert}\ (\neg⦇e, \Gamma⦈^{\circ}); c_2'; c_2'')\}\ \Gamma_1 \sqcup \Gamma_2}\ (\textsc{T-If})$$

$$\frac{\mathcal{A} = \mathtt{GenerateTemplate}(\Gamma, \text{All Assertions}) \quad c_a = \mathbf{assert}\ (((\eta + \mathcal{A})\{\eta_1/\eta\} = (\eta + \mathcal{A})\{\eta_2/\eta\} \Rightarrow \eta_1 = \eta_2))}{\vdash \Gamma\ \{\eta := \mathsf{Lap}\ \mathcal{S} \rightharpoonup c_a; \eta := sample[idx]; idx := idx + 1; \mathrm{v}_{\epsilon} := \mathrm{v}_{\epsilon} + |\mathcal{A}|\ /\mathcal{S}; \widehat{\eta} := \mathcal{A};\}\ \Gamma[\eta \mapsto \mathsf{num}_*]}\ (\textsc{T-Laplace})$$

Transformation rules for merging environments

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \quad c = \{\widehat{x} := 0 \mid \Gamma_1(x) = \mathsf{num}_0 \wedge \Gamma_2(x) = \mathsf{num}_*\}}{\Gamma_1, \Gamma_2 \Rrightarrow c}$$

Figure 18: Program transformation rules. $\mathcal{S}$ represents the scale template instrumented in Phase 1. Distinguished variable v_{ϵ} and assertions are added to ensure differential privacy.