CheckDP: An Automated and Integrated Approach for Proving Differential Privacy or Finding Precise Counterexamples

Yuxin Wang, Zeyu Ding, Daniel Kifer, Danfeng Zhang
The Pennsylvania State University
{yxwang,zyding}@psu.edu,{dkifer,zhang}@cse.psu.edu

ABSTRACT

We propose CheckDP, an automated and integrated approach for proving or disproving claims that a mechanism is differentially private. CheckDP can find counterexamples for mechanisms with subtle bugs for which prior counterexample generators have failed. Furthermore, it was able to *automatically* generate proofs for correct mechanisms for which no formal verification was reported before. CheckDP is built on static program analysis, allowing it to be more efficient and precise in catching infrequent events than sampling based counterexample generators (which run mechanisms hundreds of thousands of times to estimate their output distribution). Moreover, its sound approach also allows automatic verification of correct mechanisms. When evaluated on standard benchmarks and newer privacy mechanisms, CheckDP generates proofs (for correct mechanisms) and counterexamples (for incorrect mechanisms) within 70 seconds without any false positives or false negatives.

CCS CONCEPTS

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

KEYWORDS

Differential privacy; formal verification; counterexample detection

ACM Reference Format:
Yuxin Wang, Zeyu Ding, Daniel Kifer, 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 (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3372297.3417282

1 INTRODUCTION

Differential privacy [27] has been adopted in major data sharing initiatives by organizations such as Google [15, 29], Apple [48], Microsoft [22], Uber [36] and the U.S. Census Bureau [1, 17, 35, 41]. It allows these organizations to collect and share data with provable bounds on the information that is leaked about any individual.

Crucial to any differentially private system is the correctness of *privacy mechanisms*, the underlying privacy primitives in larger privacy-preserving algorithms. Manually developing the necessary rigorous proofs that a mechanism correctly protects privacy is a subtle and error-prone process. For example, detailed explanations of significant errors in peer-reviewed papers and systems can be found in [21, 40, 42]. Such mistakes have led to research in the application of formal verification for *proving* that mechanisms satisfy differential privacy [3, 5, 7, 9–11, 51, 52]. However, if a mechanism has a bug making its privacy claim incorrect, these techniques cannot *disprove* the privacy claims – a counterexample detector must be used instead [14, 23, 34]. Finding a counterexample is typically a two-phase process that (1) first searches an infinitely large space for candidate counterexamples and then (2) uses an exact symbolic probabilistic solver like PSI [33] to verify that the counterexample is indeed valid. The search phase currently presents the most problems (i.e., large runtimes or failure to find counterexamples are most often attributed to the search phase). Earlier search techniques were based on sampling (running a mechanism hundreds of thousands of times), which made them slow and inherently imprecise: even with enormous amounts of samples, they can still fail if a privacy-violating section of code is not executed frequently enough or if the actual privacy cost is slightly higher than the privacy claim. Recently, static program analyses were proposed to accomplish both goals [4, 30]. However, they either only analyze a non-trivial but restricted class of programs [4], or rely on heuristic strategies whose effectiveness on many sutble mechanisms is unclear [30].

In this paper, we present CheckDP, an automated and integrated tool for proving or disproving the correctness of a mechanism that claims to be differentially private. Significantly, CheckDP automatically finds counterexamples via static analysis, making it unnecessary to run the mechanism. Like prior work [14], CheckDP still uses PSI [33] at the end. However, replacing sampling-based search with static analysis enables CheckDP to find violations in a few seconds, while previous sampling-based methods [14, 23] may fail even after running for hours. Furthermore, sampling-based methods may still require manual setting of some program inputs (e.g., DP-Finder [14] requires additional arguments to be set manually for Sparse Vector Technique in our evaluation) while CheckDP is fully automated. Furthermore, the integrated approach of CheckDP allows it to efficiently analyze a larger class of differentially privacy mechanisms, compared with concurrent work using static analyses [4, 30].

Meanwhile, CheckDP still offers state-of-the-art verification capability compared with existing language-based verifiers and is further able to automatically generate proofs for 3 mechanisms for which no formal verification was reported before. CheckDP takes the source code of a mechanism along with its claimed level of privacy and either generates a proof of correctness or a verifiable counterexample (a pair of related inputs and a feasible output).

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417282

CheckDP is built upon a proof technique called *randomness alignment* [24, 51, 52], which recasts the task of proving differential privacy into one of finding *alignments* between random variables used by two related runs of the mechanism. CheckDP uses a novel verify-invalidate loop that alternatively improves tentative proofs (in the form of alignments), which are then used to improve tentative counterexamples (and vice versa) until either the tentative proof has no counterexample, or the tentative counterexample has no alignment. We evaluated CheckDP on correct/incorrect versions of existing benchmarks and newly proposed mechanisms. It generated a proof for each correct mechanism within 70 seconds and a counterexample for each incorrect mechanism within 15 seconds.

In summary, this paper makes the following contributions:

(1) CheckDP, one of the first automated tools (with concurrent work [4, 30]) that generates both proofs for correct mechanisms and counterexamples for incorrect mechanisms (Section 2.4).
(2) A syntax-directed translation from the probabilistic mechanism being checked to non-probabilistic target code with explicit proof obligations (Section 3).
(3) An alignment template generation algorithm (Section 3.4).
(4) A novel verify-invalidate loop that incrementally improves tentative proofs and counterexamples (Section 4).
(5) Case studies and experimental comparisons between CheckDP and existing tools using correct/incorrect versions of existing benchmarks and newly proposed mechanisms. For incorrect mechanisms, CheckDP automatically found counterexamples in all cases, even in cases where competing methods [14, 23] failed. For correct mechanisms, CheckDP automatically generated proofs of privacy, including proofs for 3 mechanisms for which no formal verification was reported before (Section 5).

2 PRELIMINARIES AND RUNNING EXAMPLE

2.1 Differential Privacy

Among several popular variants of differential privacy [16, 26, 27, 43], we focus on *pure* differential privacy [27]. The goal of differential privacy is to hide the effect of any person's record on the output of an algorithm. This is achieved by considering all pairs of datasets D and D' that differ on one record. We call such datasets *adjacent* and denote it by $D \sim D'$. To offer privacy, a differentially private algorithm injects carefully calibrated random noise during its computation. Given a pair of datasets (D, D'), we call the execution of an algorithm on D the *original execution* and the execution on (neighboring) D' the *related execution*. Intuitively, we say a randomized algorithm is differentially private if the output distribution of the original execution and its related execution are hard to distinguish for all such dataset pairs:

Definition 1 (Pure Differential Privacy [25]). *Let $\epsilon \geq 0$. A probabilistic computation $M : \mathcal{D} \to \mathcal{O}$ is ϵ-differentially private if for every pair of neighboring datasets $D \sim D' \in \mathcal{D}$ and every output $o \in \mathcal{O}$, $\mathbb{P}[M(D) = o] \leq e^{\epsilon}\, \mathbb{P}[M(D') = o]$.*

Often, a differentially private algorithm M interacts with a dataset D through a list of queries $f_1, f_2, \ldots$: it iteratively runs a query f_i on D to get an exact answer q_i, then performs some randomized computation on the set of query answers $\{q_j \mid j \leq i\}$. We call the vector $(q_1, q_2, \ldots)$ along with other data-independent parameters to M (e.g., privacy parameter ϵ) an *input* to M. The notion of adjacent datasets translates into the notion of *sensitivity* on those queries:

Definition 2 (Global Sensitivity [28]). *The global sensitivity of a query f is $\Delta_f = \sup_{D \sim D'} \left|f(D) - f(D')\right|$.*

We say two inputs $inp = \{(q_1, q_2, \ldots), \text{params}\}$ and $inp' = \{(q_1', q_2', \ldots), \text{params}\}$ are adjacent with respect to the queries $f_1, f_2, \ldots$, and write $inp \sim inp'$, if the params are the same and there exist two adjacent datasets D and D' such that $(f_1(D), f_2(D), \ldots) = (q_1, q_2, \ldots)$ and $(f_1(D'), f_2(D'), \ldots) = (q_1', q_2', \ldots)$. Note that this implies that $\left|q_i - q_i'\right| \leq \Delta_{f_i}, \forall i$. It follows that differential privacy can be proved by showing that for all pair of inputs $inp \sim inp'$ and all outputs $o \in \mathcal{O}$, $\mathbb{P}[M(inp) = o] \leq e^{\epsilon}\, \mathbb{P}[M(inp') = o]$. As standard, we assume that the sensitivity of inputs are either manually specified or computed by sensitivity analysis tools (e.g., [32, 44]).

Many mechanisms are built on top of the Laplace Mechanism [27] which adds Laplace noise to query answers:

Theorem 1 (Laplace Mechanism [27]). *Let $\epsilon > 0$, let D be a dataset, let f be a query with sensitivity Δ_f and let $q = f(D)$. The Laplace Mechanism which, on input q, outputs $q + \eta$ (where η is sampled from the Laplace distribution with mean 0 and scale parameter Δ_f/ϵ) satisfies ϵ-differential privacy.*

We sometimes abuse notation and refer to the sensitivity Δ_q of a numerical value q – we always take this to mean as the sensitivity of the function that produced q.

2.2 Randomness Alignment

Randomness alignment is a simple yet powerful proof technique that underpins the verification tools LightDP [52] and its successor ShadowDP [51]. Precise reasoning using this proof technique was used to improve a variety of algorithms, allowing them to release strictly more information at the same privacy cost [24]. Given two executions of a randomized algorithm M on D and D' respectively, a randomness alignment is a mapping between the random variables in the first execution to random variables in the second execution that will cause the second execution to always produce the same output as the first. Upper bounds on privacy parameters depend on how much the random variables change under this mapping [52].

We use the Laplace Mechanism [28] to illustrate the key ideas behind randomness alignment. Let $D \sim D'$ be a pair of neighboring datasets and let f be a query with sensitivity Δ_f. Let $q = f(D)$ and $q' = f(D')$ be the respective query answers. If we use the Laplace Mechanism to answer these queries with privacy, on input q (resp. q') it will output $q + \eta$ (resp. $q' + \eta'$) where η (resp. η') is a Laplace random variable with scale Δ_f/ϵ. In order for the Laplace Mechanism to produce the same output in both executions, we need $q + \eta = q' + \eta'$ and therefore $\eta' = \eta + q - q'$. This creates a "mapping" between the values of random noises: if we change the input from q to q', we need to adjust the random noise by an amount of $q - q'$ (i.e., this is the *distance* we need to move η' to get to η). Clearly $\left|q - q'\right| \leq \Delta_f$ by definition of sensitivity. The privacy proof follows from the fact that if two random samples η and η' (from the Laplace distribution with scale Δ_f/ϵ) are at most distance Δ_f apart, the ratio of their probabilities is at most e^{ϵ}. Hence, the *privacy cost*, the natural log of this ratio, is bounded by ϵ.

Thus randomness alignment can be viewed in terms of *distances* that we need to move random variables. Let $q \sim q'$ be query answers from neighboring datasets and M be a randomized algorithm which uses a set of random noises $H = \{\eta\}$. We associate to every random variable η a numeric value $\widehat{\eta}$ which tracks precisely the amount in value we need to change η in order to obtain the same output when the input to M is changed from q to q'. In other words, the output of M with input q and random values $\{\eta\}$ is the same as that of M with input q' and random values $\{\eta + \widehat{\eta}\}$. Taking M to be the Laplace Mechanism, then the alignment in the previous paragraph is $\{\widehat{\eta} = q - q'\}$. Note that the alignment is a function that depends on M as well as q and q'.

If all of the random variables are Laplace, the cost of an alignment is the summation of $\frac{\text{distance}}{\text{noise scale}}$ for each random variable. To find the overall privacy cost (e.g., the ϵ in differential privacy), we then find an upper bound on the alignment cost for all related q and q'.

2.3 Privacy Proof and Counterexample

Not all randomness alignments serve as proofs of differential privacy. To form a proof, one must show that (1) the alignment forces the two related executions to produce the same output, (2) the privacy cost of an alignment must be bounded by the promised level of privacy, and (3) the alignment is injective. Hence, in this paper, an (alignment-based) privacy proof refers to a randomness alignment that satisfies these requirements.

On the other hand, to show that an algorithm violates differential privacy, it suffices to demonstrate the existence of a counterexample. Formally, if an algorithm M claims to satisfy ϵ-differential privacy, a *counterexample* to this claim is a triple (inp, inp', o) such that $inp \sim inp'$ and $\mathbb{P}[M(inp) = o] > e^{\epsilon}\,\mathbb{P}[M(inp') = o]$.

Challenges. LightDP [52] and ShadowDP [51] can check if a manually generated alignment is an alignment-based privacy proof. On the other hand, an exact symbolic probabilistic solver, such as PSI [33], can check if a counterexample, either generated manually or via a sampling-based generator, witnesses violation of differential privacy. To the best of our knowledge, CheckDP is the first tool that *automatically generates* alignment-based proofs/counterexamples via static program analysis.[1] To do so, *a key challenge* is to tackle the infinite search space of proofs (i.e., alignments) and counterexamples. CheckDP uses a novel proof template generation algorithm to reduce the search space of candidate alignments (Section 3) and uses a novel verify-invalidate loop (Section 4) to find tentative proofs, counterexamples showing their privacy cost is too high, improved proofs, improved counterexamples, etc.

2.4 Running Examples

To illustrate our approach, we now discuss two variants of the Sparse Vector Technique [28], one correct and one incorrect. Using the two variants, we sketch how CheckDP automatically proves/disproves (as appropriate) their claimed privacy properties.

Sparse Vector Technique (SVT) [28]. A powerful mechanism proven to satisfy differential privacy. It can be used as a building block for many advanced differentially private algorithms. This mechanism is designed to solve the following problem: given a series of queries and a preset public threshold, we want to identify the first N queries whose answers are above the threshold, but in a privacy-preserving manner. To achieve this, it adds independent Laplace noise both to the threshold and each query answer, then it returns the identities of the first N queries whose *noisy* answers are above the noisy threshold. The standard implementation of SVT outputs `true` for the above-threshold queries and `false` for the others (and terminates when there are a total of N outputs equal to `true`). We use two variants of SVT for an overview of CheckDP.

GapSVT. This is an improved (and correct) variant of SVT which provides numerical information about some queries. When a noisy query exceeds the noisy threshold, it outputs the difference between these noisy values; otherwise it returns `false`. This provides an estimate for how much higher a query is compared to the threshold. The algorithm was first proposed and verified in [51]; its pseudo code is shown in Figure 1. Here, **Lap** $(2/\epsilon)$ draws one sample from Laplace distribution with mean 0 and scale factor of $2/\epsilon$. This random value is then added to the public threshold T (stored as noisy threshold T_η). For each query answer, another independent Laplace noise $\eta_2 =$ **Lap** $(4N/\epsilon)$ is added. If the noisy query answer `q[i]` + η_2 is above the noisy threshold T_η, the gap between them (`q[i]` + $\eta_2 - T_\eta$) is added to the output list `out`, otherwise `0` is added.

One key observation from the manual proofs of SVT and its variants [21, 24, 28, 40] is that the privacy cost is only paid for the queries whose noisy answers are above the noisy threshold. In other words, outputting `false` does not incur any new privacy cost. Correspondingly, the correct alignment for GapSVT [24, 51] (that is, the distance that η_1 and η_2 need to be moved to ensure the output is the same when the input changes from $q[i]$ to $q'[i] \equiv q[i] + \widehat{q}[i]$, for all i) is: $\eta_1 : 1$ and $\eta_2 :$ `q[i] + `$\eta_2 \geq T_\eta$ `? (1 - `$\widehat{\text{q}}$`[i]) : 0`.

Note that η_2 is aligned with non-zero distance only under the `true` branch; hence, no privacy cost is paid in the other branch. It is easy to verify that if every query has sensitivity 1, the cost of this alignment is bounded by ϵ.

BadGapSVT. We also consider a variant of SVT (and GapSVT) that incorrectly tries to release numerical information. When a noisy query answer is larger than the noisy threshold, the variant releases that noisy query answer (that is, it *does not* subtract from it the noisy threshold); otherwise it outputs `false`. This is an incorrect variant of SVT [45] that was reported in [40] and was called iSVT4 in [23]. More precisely, BadGapSVT replaces line 7 of GapSVT with `out := (q[i] + `η_2`)::out;`. This small change makes it not ϵ-differentially private [40]. The reason why is subtle, but the intuition is the following. Suppose BadGapSVT returns a noisy query answer `q[i]` + $\eta_2 = 3$, the attacker is able to deduce that $T_\eta \leq 3$. Once this information is leaked, outputting `false` in the else branch is no longer "free"; every output incurs a privacy cost.

2.5 Approach Overview

We use GapSVT and BadGapSVT to illustrate how CheckDP generates proofs and counterexamples.

Code Transformation (Section 3). CheckDP first takes the probabilistic algorithm being checked, written in the CheckDP language

[1] Prior work [3] automatically generates coupling proofs, an alternative language-based proof technique for differential privacy. But all existing verifiers using alignment-based proofs[51, 52] require manually provided alignments.

function GapSVT (T, N, size : num$_0$, q : list num$_*$)
returns (out : list num$_0$), **check**(ϵ)
precondition $\forall$ i. $-1 \le \widehat{\mathrm{q}}$[i] ≤ 1

```
1  η1 := Lap (2/ε)
2  Tη := T + η1;
3  count := 0; i := 0;
4  while (count < N ∧ i < size)
5    η2 := Lap (4N/ε)
6    if (q[i] + η2 ≥ Tη) then
7      out := (q[i] + η2 - Tη)::out;
8      count := count + 1;
9    else
10     out := false::out;
11   i := i + 1;
```

function Transformed GapSVT (T, N, size, q, $\widehat{\mathrm{q}}$, *sample*, θ)
returns (out)

```
12 vε := 0; idx = 0;
13 η1 := sample[idx]; idx := idx + 1;
14 vε := vε + |A1| × ε/2; η̂1 := A1;
15 Tη := T + η1;
16 T̂η := η̂1;
17 count := 0; i := 0;
18 while (count < N ∧ i < size)
19   η2 := sample[idx]; idx := idx + 1;
20   vε := vε + |A2| × ε/4N; η̂2 := A2;
21   if (q[i] + η2 ≥ Tη) then
22     assert(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η);
23     assert(q̂[i] + η̂2 - T̂η == 0);
24     out := (q[i] + η2 - Tη)::out;
25     count := count + 1;
26   else
27     assert(¬(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η));
28     out := false::out;
29   i := i + 1;
30 assert(vε ≤ ε);
```

Figure 1: GapSVT and its transformed code, where underlined parts are added by CheckDP. The transformed code contains two alignment templates for η_1 and η_2: $\mathcal{A}_1 = \theta[0]$ and $\mathcal{A}_2 = (q[i] + \eta_2 \ge T_\eta)\ ?\ (\theta[1] + \theta[2] \times \widehat{T_\eta} + \theta[3] \times \widehat{q}[i]) : (\theta[4] + \theta[5] \times T_\eta + \theta[6] \times \widehat{q}[i])$. The random variables and θ are inserted as part of the function input.

(Section 3.1), and generates the non-probabilistic target code with *assertions* and *alignment templates* (i.e. templates for possible alignments). The bottom of Figure 1 shows the transformed code of GapSVT with alignment templates. The transformed code is distinguished from the source code in a few important ways: (1) The probabilistic sampling commands (at lines 1 and 5) are replaced by non-probabilistic counterparts that read samples from the instrumented function input *sample*. (2) An alignment template (e.g., $\mathcal{A}_1, \mathcal{A}_2$) is generated for each sampling command; each template contains a few holes, i.e., θ, which is also instrumented as function input. (3) A distinguished variable $\mathbf{v}_\epsilon$ is added to track the overall privacy cost and lines 14 and 20 update the cost variable in a sound way. (4) Assertions are inserted in the transformed code (lines 22,23,27,30) to ensure the following soundness property:

if $M(inp)$ is transformed to $M'(inp, \widehat{inp}, sample, \theta)$, then

$$\exists\theta.\ \forall inp, \widehat{inp}, sample.\ \textit{all assertions in } M' \textit{ pass} \implies M \text{ is differentially private}$$

We note that the transformed code forms the basis for both proof and counterexample generation in CheckDP.

Proof/Counterexample Generation (Section 4). Inspired by the Counterexample Guided Inductive Synthesis (CEGIS) [46] technique, originally proposed for program synthesis, CheckDP uses a verify-invalidate loop to simultaneously generate proofs and counterexamples. Unlike CEGIS, however, the verify-invalidate loop is *bidirectional*, in the sense that it internally records all previous counterexamples (resp. proofs) to generate one proof (resp. counterexample) as the algorithm output. On the other hand, the CEGIS loop is *unidirectional*: it only collects and uses a set of inputs to guide synthesis internally. At a high level, the verify-invalidate loop of CheckDP includes two integrated sub-loops, one for proof generation and the other for counterexample generation.

Verify Sub-loop. Its goal is to generate a proof (i.e., an instantiation of θ) such that $\forall inp, \widehat{inp}, sample.\ \textit{all assertions in } M' \textit{ pass}$

This is done by two iterative phases:

(1) Generating invalidating inputs: Given a proof candidate (i.e., an instantiation of θ), it is *incorrect* if
$$\exists inp, \widehat{inp}, sample.\ \textit{some assertion in } M' \textit{ fails}$$
We use I to denote a triple of $inp, \widehat{inp}, sample$. Hence, given any instantiation of θ, we use an off-the-shelf symbolic execution tool such as KLEE [18] to find invalidating inputs when possible.
(2) Generating proof candidates: with a set of invalidating inputs found so far $I_1, \cdots, I_i$, we can try to generate a new proof candidate to satisfy $\exists\theta.\ M'(I_1, \theta) \wedge \cdots \wedge M'(I_i, \theta)$

Starting from a default instantiation (e.g., one that sets $\forall i.\ \theta[i] = 0$), CheckDP iteratively repeats Phases 1 and 2. Since CheckDP uses all invalidating inputs found so far in Phase 2, the proof candidate after each iteration is improving. When Phase 1 gets stuck, CheckDP obtains a proof candidate θ which is a privacy proof if
$$\forall inp, \widehat{inp}, sample.\ M'(inp, \widehat{inp}, sample, \theta)$$
due to the soundness property above. Hence, a proof (alignment) can be validated by program verification tools such as CPAChecker [13]. For GapSVT, CheckDP generates and verifies (via CPAChecker) that $\theta = \{1, 1, 0, -1, 0, 0, 0\}$ results in a proof that GapSVT satisfies ϵ-differential privacy.

Invalidate Sub-loop. While the verify sub-loop is conceptually similar to a CEGIS loop [46], CheckDP also employs an invalidate sub-loop (integrated with the verify sub-loop); its goal is to generate *one invalidating input* I such that $\forall\theta.\ \textit{some assertion in } M' \textit{ fail}$. This is done by two iterative phases:[2]

(1) Generating proof candidates: Given an invalidating input I, it is *incorrect* if $\exists\theta.\ M'(I, \theta)$. Hence, given any I, we can use KLEE [18] to find an alignment when possible.

[2]Note that a set of invalidating inputs $I_1, \cdots, I_i$, generated from Phase 2 of the verify sub-loop is not a counterexample candidate, since by definition, a differential privacy counterexample consists of only one invalidating input.

Reals	r	$\in$	$\mathbb{R}$
Booleans	b	$\in$	$\{\texttt{true}, \texttt{false}\}$
Vars	x	$\in$	V
Rand Vars	η	$\in$	H
Linear Ops	$\oplus$	::=	$+ \mid -$
Other Ops	$\otimes$	::=	$\times \mid /$
Comparators	$\odot$	::=	$<\mid>\mid=\mid\leq\mid\geq$
Bool Exprs	$\mathbb{b}$	::=	$\texttt{true} \mid \texttt{false} \mid x \mid \neg\mathbb{b} \mid \mathbb{n}_1 \odot \mathbb{n}_2$
Num Exprs	$\mathbb{n}$	::=	$r \mid x \mid \eta \mid \mathbb{n}_1 \oplus \mathbb{n}_2 \mid \mathbb{n}_1 \otimes \mathbb{n}_2 \mid \mathbb{b}\ ?\ \mathbb{n}_1 : \mathbb{n}_2$
Expressions	e	::=	$\mathbb{n} \mid \mathbb{b} \mid e_1 :: e_2 \mid e_1[e_2]$
Commands	c	::=	$\textbf{skip} \mid x := e \mid \eta := g \mid c_1; c_2 \mid$ $\textbf{if}\ e\ \textbf{then}\ (c_1)\ \textbf{else}\ (c_2) \mid$ $\textbf{while}\ e\ \textbf{do}\ (c) \mid \textbf{return}\ e$
Rand Exps	g	::=	$\textbf{Lap}\ r$
Types	τ	::=	$\texttt{num}_{\mathbb{d}} \mid \texttt{bool} \mid \texttt{list}\ \tau$
Distances	$\mathbb{d}$	::=	$0 \mid *$

Figure 2: CheckDP: language syntax.

(2) Generating counterexamples: with a set of previously found alignments $\theta_1, \cdots, \theta_i$, we try to find a new invalidating input to satisfy $\exists I. \neg M'(I, \theta_1) \wedge \cdots \wedge \neg M'(I, \theta_i)$

To integrate with the verify sub-loop, Phase 1 of the invalidate sub-loop starts when Phase 2 of the verify sub-loop gets stuck with a set of invalidating inputs $I_1, \cdots, I_i$; it uses I_i to proceed since it is the most promising one. When Phase 1 of invalidate sub-loop gets stuck, CheckDP obtains a counterexample candidate, which can be validated by PSI [33] (this is necessary since a mechanism might be differentially private even if no alignment-based proof exists).

For example, the counterexample found for BadGapSVT sets the threshold $T = 0$, $N = 1$ (max number of outputs equal to `true` before termination), neighboring inputs $q = [0, 0, 0, 0, 0]$ and $q' = [1, 1, 1, 1, -1]$, and the following output to examine $[0, 0, 0, 0, 1]$. PSI confirms that the probability of this output when q is an input is $\geq e^{\epsilon}$ times the probability of this output when q' is the input.

When Phase 1 of the invalidate sub-loop generates a new alignment θ, which happens in our empirical study (Section 5), Phase 2 follows to generate an "improved" invalidating input, which is then used to start Phase 2 of the validate sub-loop.

3 PROGRAM TRANSFORMATION

CheckDP takes a probablistic program along with an adjacency specification (i.e., how much two adjacent inputs can differ) and the claimed level of differential privacy as inputs. It translates the source code into a non-probabilistic program with assertions to ensure differential privacy. The transformed code forms the basis of finding a proof or a counterexample (Section 4).

3.1 Syntax

The syntax of CheckDP source code is listed in Figure 2. Most of the syntax is standard with the following features:

- Real numbers, booleans and their standard operations;
- Ternary expressions $\mathbb{b}\ ?\ \mathbb{n}_1 : \mathbb{n}_2$, it returns $\mathbb{n}_1$ when $\mathbb{b}$ evaluates to `true` or $\mathbb{n}_2$ otherwise;
- List operations: $e_1 :: e_2$ appends element e_1 to list e_2, and $e_1[e_2]$ gets the e_2^{th} element of list e_1;
- Loop with keyword **`while`** and branch with keyword **`if`**;
- A final return command **`return`** e.

We now introduce other interesting parts that are needed for developing differentially private algorithms.

Random Expressions. Differential privacy relies heavily on probabilistic computations: many mechanisms achieve differential privacy by adding appropriate random noise to variables. To model this behavior, we embed a sampling command $\eta := \textbf{Lap}\ r$ in CheckDP, which draws a sample from the Laplace distribution with mean 0 and scale of r. In this paper, we only focus on the most interesting sampling command **`Lap`** r (which is used in Laplace Mechanism and GapSVT in Section 2). However, we note that it is fairly easy to add new sampling distributions to CheckDP.

For clarity, we distinguish variables holding random values, denoted by $\eta \in H$, from other ones, denoted by $x \in V$.

Types with Distances. To enable alignment-based proof, one important aspect of the type system in CheckDP is the ability to compute and track the distances for each program variable. Motivated by verification tools using alignments (e.g., LightDP [52] and ShadowDP [51]), types in the source language of CheckDP have the form of $\mathcal{B}_0$ or $\mathcal{B}_*$, where $\mathcal{B}$ is the base type such as numerics (`num`), booleans (`bool`) and lists ($\texttt{list}\ \tau$). The subscript of each type is the key to alignment-based proofs: it explicitly tracks the *exact* difference between the value of a variable in two related runs.

In the source language of CheckDP, the distances can either be 0 or $*$: the former indicates the variables stay the same in the related runs; the latter means that the variable might hold different values in two related runs and the value difference is stored in a distinguished variable $\widehat{x}$ added by the program transformation (i.e., a syntactic sugar for dependent sum type $\sum_{(\widehat{x}:\ \texttt{num}_0)}\ \mathcal{B}_{\widehat{x}}$). For example, inputs `T,N,size` are annotated with distance 0 in Figure 1, meaning that they are public parameters to the algorithm; query answers q are annotated with distance $*$, meaning that each $q[i]$ differ by exactly $\widehat{q}[i]$ in two related runs. The type system distinguishes zero-distance variables as an optimization: as we show shortly, it helps to reduce the code size for later stages (Section 3.3) as well as aids proof template generation (Section 3.4).

Note that boolean types (`bool`) and list types ($\texttt{list}\ \tau$) cannot be associated with numeric distances, hence omitted in the syntax. However, nested cases such as $\texttt{list}\ \texttt{num}_*$ still accurately track the distances of the elements inside the list.

The semantics of CheckDP follows the standard definitions of probabilistic programs [37]; the formal semantics can be found in the full version of this paper [50]. Finally, CheckDP also supports shadow execution, a technique that underpins ShadowDP [51] and is crucial to the verification of challenging mechanisms such as Report Noisy Max [25]. However, in order to focus on the most interesting parts of CheckDP, we first present the transformation without shadow execution, and later discuss how to support it.

3.2 Program Transformation

CheckDP is equipped with a flow-sensitive type system whose typing rules are shown in Figure 3. At command level, each rule has the following format: $\vdash \Gamma\ \{c \rightharpoonup c'\}\ \Gamma'$ where a typing environment Γ tracks for each program variable its type with distance, c and c' are the source and target programs respectively, and the flow-sensitive type system also updates typing environment to Γ' after command

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

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

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

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

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

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

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

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

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

$$\frac{\vdash \Gamma \sqcup \Gamma_f\ \{c \rightharpoonup c'\}\ \Gamma_f \quad \Gamma, \Gamma \sqcup \Gamma_f \Rrightarrow c_s \quad \Gamma_f, \Gamma \sqcup \Gamma_f \Rrightarrow c''}{\vdash \Gamma\ \{\mathtt{while}\ e\ \mathtt{do}\ c \rightharpoonup c_s; (\mathtt{while}\ e\ \mathtt{do}\ (\mathtt{assert}((\!|e, \Gamma|\!)^\circ); c'; c''))\}\ \Gamma \sqcup \Gamma_f}\ \text{(T-While)}$$

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

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

Transformation rules for merging environments

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

Figure 3: Program transformation rules. Distinguished variable $\mathbf{v}_\epsilon$ and assertions are added to ensure differential privacy.

c. At a high-level, the type system transforms the probabilistic source code c into the non-probabilistic target code c' in a way that if all assertions in c' holds, then c is differentially private.

CheckDP's program transformation is motivated by those of LightDP and ShadowDP [51, 52], all built on randomness alignment proof. However, there are a few important differences:

- CheckDP generates an alignment template for each sampling instruction, rather than requiring manually provided alignments.
- CheckDP defers all privacy-related checks to assertions. This is crucial since information needed for proof and counterexample generation is unavailable in a lightweight static type system.
- CheckDP only tracks if a variable has the same value in two related runs (with distance 0) or not (with distance $*$). This design aids alignment template generation and reduces the size of transformed code.

Checking Expressions. Each typing rule for expression e computes the correct distance for its resulting value: $\Gamma \vdash e : \mathcal{B}_{\mathbb{n}} \mid C$, which reads as: expression e has type $\mathcal{B}$ and distance $\mathbb{n}$ under the typing environment Γ if the constraints C are satisfied. The reason to collect constraints C instead of statically checking them, is to defer all privacy-related checks to later stages.

Most of the expression rules are straightforward: they check the base types (just like a traditional type system) and compute the distance of e's value in two related runs. For example, all constants must be identical (Rules (T-Num,T-Boolean)) and the distance of a variable is retrieved from the environment (T-VarZero,T-VarStar) (note that rule (T-VarStar) just desugers the $*$ notation). For linear operation ($\oplus$), the distance of the result is computed in a precise way (Rule (T-OPlus)), while the other operations are treated in a more conservative way: constraints are generated to ensure that the result is identical in Rules (T-OTimes, T-ODot). For example, (T-ODot) ensures boolean value of $e_1 \odot e_2$ will be the same in two related runs by adding a constraint

$$(e_1 \odot e_2) \Leftrightarrow (e_1 + \mathbb{n}_1) \odot (e_2 + \mathbb{n}_2)$$

(T-Cons) restricts constructed list elements to have 0-distance (note that the restriction does not apply to input lists), while (T-Index) requires the index to have zero-distance. Rule (T-Select) restricts e_1 and e_2 to have the same distance. The constraints gathered in the

expression rules will later be explicitly instrumented as assertions in the translated programs, which we will explain shortly.

3.3 Checking Commands

For each program statement, the type system updates the typing environment and if necessary, instruments code to update $\widehat{x}$ variables to the correct distances. Moreover, it ensures that the two related runs take the same branch in if-statement and while-statement.

Flow-Sensitivity. Each typing rule updates the typing environment to track if a variable has zero-distance. When a variable has non-zero distance, it instruments the source code to properly maintain the corresponding $\widehat{x}$ variables. The most interesting rules are: rule (T-Asgn) properly promotes the type of x to be $\mathcal{B}_*$ (tracked by distance variables) in Γ' if the distance of e is not 0. Meanwhile it optimizes away updates to $\widehat{x}$ and properly downgrades type to $\mathcal{B}_0$ if e has a zero-distance. For example, line 16 in GapSVT (Figure 1) is instrumented to update distance of T_η, according to the distance of $T + \eta_1$. Moreover, variable `count` in GapSVT always has the type $\texttt{num}_0$; therefore its distance variable never appears in the translated program due to the optimization in (T-Asgn).

Rule (T-If) and (T-While) are more complicated since they both need to merge environments. In rule (T-If), as c_1 and c_2 might update Γ to Γ_1 and Γ_2 respectively, we need to merge them in a natural way: the distance of a type form a two-level lattice with $0 \sqsubset *$. Thus we define a union operator $\sqcup$ for distances $\mathbb{d}$ as:

$$\mathbb{d}_1 \sqcup \mathbb{d}_2 \triangleq \begin{cases} \mathbb{d}_1 & \text{if } \mathbb{d}_1 = \mathbb{d}_2 \\ * & \text{otherwise} \end{cases}$$

therefore the union operator for two environments are defined as follows: $\Gamma_1 \sqcup \Gamma_2 = \lambda x.\ \Gamma_1[x] \sqcup \Gamma_2[x]$.

Moreover, we use an auxiliary function $\Gamma_1, \Gamma_2 \Rrightarrow c$ to "promote" a variable to star type. For example, with $\Gamma(x) = *, \Gamma(y) = *$ and $\Gamma(b) = 0$, rule (T-If) translates the source code
if b **then** $x := y$ **else** $x := 1$ to the following:
if b **then** $(x := y; \widehat{x} := \widehat{y};)$ **else** $(x := 1; \widehat{x} := 0)$
where $\widehat{x} := \widehat{y}$ is instrumented by (T-Asgn) and $\widehat{x} := 0$ is instrumented due to the promotion.

Similarly, the typing environments are merged in rule (T-While), except that it requires a fixed point Γ_f such that $\vdash \Gamma \sqcup \Gamma_f \ \{c\}\ \Gamma_f$. We follow the construction in [51] to compute a fixed point, noting that the computation always terminates since all of the translation rules are monotonic and the lattice only has two levels.

Assertion Generation. To ensure differential privacy, the type system inserts assertion in various rules:

- To ensure that two related runs take the same control flow, (T-If) and (T-While) asserts that the value of the branch condition stays the same across two related executions. A helper function $(\!|e, \Gamma|\!)^\circ$ is used to compute the value of e in the aligned execution; its full definition can be found in the Appendix.
- To ensure that the final output value is differentially private, rule (T-Return) asserts that its distance is zero (i.e., identical in two related runs).
- To ensure all constraints collected in the expression rules are satisfied, assignment rules (T-Asgn) and (T-AsgnStar) also insert corresponding assertions.

3.4 Checking Sampling Commands

Rule (T-Laplace) performs a few important tasks:

Replacing Sampling Command. Rule (T-Laplace) removes the sampling instruction and assign to η the next (unknown) sample value `sample[idx]`, where `sample` is a parameter of type `list num` added to the transformed code. The typing rule also increments `idx` so that the next sampling command will read out the next value.

Checking Injectivity. T-Laplace adds an assertion c_a to check the injectivity of the generated alignment (a fundamental requirement of alignment-based proofs): the same aligned value of η implies the same value of η in the original execution.

Tracking Privacy Cost. A distinguished privacy cost variable $\mathbf{v}_\epsilon$ is also instrumented to track the cost for aligning the random variables in the program. Due to the properties of Laplace distribution, for a sampling command $\eta :=$ **Lap** r with alignment template $\mathcal{A}$, we have $\mathbb{P}(\eta)/\mathbb{P}(\eta + \mathcal{A}) \leq e^{|\mathcal{A}|/r}$. Hence, the privacy cost for aligning η by $\mathcal{A}$ is $|\mathcal{A}|/r$. Note that the symbols in gray, including $\mathcal{A}$, are placeholders when the rule is applied, since function `GenerateTemplate` takes all assertions in the transformed code as inputs. Once translation is complete, the placeholders are filled in by the algorithm that we discuss in Section 4.

Alignment Template Generation. For each sampling command $\eta :=$ **Lap** r, an alignment of η is needed in a randomness alignment proof. In its most flexible form, the alignment can be written as any numerical expression $\mathbb{n}$, which is prohibitive for our goal of automatic proof generation. On the other hand, using simple heuristics such as only considering constant alignment does not work: for example, the correct alignment for η_2 in GapSVT is written as "$(\texttt{q[i]} + \eta_2 \geq T_\eta)\ ?\ (1 - \widehat{\texttt{q[i]}}) : 0$", where the alignment actually depends on which branch is taken during the execution.

To tackle the challenges, CheckDP generates an *alignment template* for each sampling instruction; a template is a numerical expression with "holes" whose values are to be searched for in later stages. For example, the template generated for η_2 in GapSVT is

$$(\texttt{q[i]} + \eta_2 \geq T_\eta)?(\theta[0] + \theta[1] \times \widehat{T_\eta} + \theta[2] \times \widehat{\texttt{q[i]}}) : (\theta[3] + \theta[4] \times \widehat{T_\eta} + \theta[5] \times \widehat{\texttt{q[i]}})$$

where $\theta[0] - \theta[5]$ are symbolic coefficients to be found later.

In general, for each sampling command $\eta =$ **Lap** r, CheckDP first uses static program analysis to find a set of relevant program expressions, denoted by $\mathbb{E}$, and a set of relevant program variables, denoted by $\mathbb{V}$ (as described shortly). Second, it generates an alignment template as follows:

$$\mathcal{A}_{\mathbb{E}} ::= \begin{cases} e_0\ ?\ \mathcal{A}_{\mathbb{E}\setminus\{e_0\}} : \mathcal{A}_{\mathbb{E}\setminus\{e_0\}}, \text{when } \mathbb{E} = \{e_0, \cdots\} \\ \theta_0 + \sum_{v_i \in \mathbb{V}} \theta_i \times v_i \text{with fresh } \theta_0, \cdots, \theta_{|\mathbb{V}|}, \text{otherwise} \end{cases}$$

where θ denotes coefficients ("holes") to be filled out out by later stages and each of them is generated fresh.

To find proper $\mathbb{E}$ and $\mathbb{V}$, our insight is that the alignments serve to "cancel out" the differences between two related runs (i.e., to make all assertions pass). Algorithm 1 follows the insight to compute $\mathbb{E}$ and $\mathbb{V}$ for each sampling instruction: it takes Γ_s, the typing environment right before the sampling instruction and A, all assertions in the transformed code, as inputs. It also assumes an oracle

`Depends(e, x)` which returns `true` whenever the expression e depends on the variable x. We note that the oracle can be implemented as standard program dependency analysis [2, 31] or information flow analysis [12]; hence, we omit the details in this paper.

Algorithm 1: Template generation for $\eta :=$ **Lap** r

```
input: Γ_s: typing environment at sampling command
       A: set of the generated assertions in the program
1  function GenerateTemplate(Γ_s, A):
2      𝔼 ← ∅, 𝕍 ← ∅
3      foreach assert(e) ∈ A do
4          if Depends(e, η) then
5              if assert(e) is generated by (T-If) then
6                  e' ← the branch condition of if
7                  𝔼 ← 𝔼 ∪ {e'}
8              foreach v ∈ Vars ∪ {e1[e2] | e1[e2] ∈ e} do
9                  if Γ_s ⊬ v : ℬ0 ∧ Depends(e, v) then
10                     𝕍 ← 𝕍 ∪ {v}
11     foreach e ∈ 𝔼 ∪ 𝕍 do
12         remove e from 𝔼 and 𝕍 if not in scope
13     return 𝔼,𝕍;
```

The algorithm first checks (at line 4) if aligning η has a chance to make an assertion pass. If so, it will increment $\mathbb{E}$ and $\mathbb{V}$ as follows. For $\mathbb{E}$, we notice that only for the assertions generated by rule (T-If), depending on the branch condition allows the alignment to have different values under different branches. Hence, we add the branch condition to $\mathbb{E}$ in this case. For $\mathbb{V}$, our goal is to use the alignment to "cancel" the differences caused by other variables and array elements such as $q[i]$ used in e. Hence, we only need to consider $\widehat{v}$ if (1) v is different between two related runs (i.e., $\Gamma_s \not\vdash v : \mathcal{B}_0$) and (2) v contributes the assertion (i.e., e depends on v).

Finally, the algorithm performs a "scope check": if any element in $\mathbb{E}$ or $\mathbb{V}$ contains out-of-scope variables, then the element is excluded; for example, η_1 should not depend on $q[i]$ in GapSVT since $q[i]$, essentially an iterator of q, is not in scope at that point.

Consider η_1 and η_2 in GapSVT. The assertions in the translated programs are (we only list the assertion in the `true` branch since the constraint in `false` branch is symmetric) :

(1) **assert**(`q[i]` + η_2 + $\widehat{\texttt{q[i]}}$ + $\widehat{\eta_2}$ $\geq$ T_η + $\widehat{T_\eta}$)

(2) **assert**($\widehat{\texttt{q[i]}}$ + $\widehat{\eta_2}$ - $\widehat{T_\eta}$ = 0)

For η_1, we have $\Gamma_s = \{\texttt{q} : *\}$ (we omit the base types and the variables that have 0 distance for brevity) and both assertions depend on η_1. Since both assertions depend on η_1 and `q[i]`, Algorithm 1 adds $\widehat{\texttt{q[i]}}$ into $\mathbb{V}$. Moreover, assertion (1) is generated by rule (T-If). Thus,the algorithm adds `q[i]` + η_2 $\geq$ T_η into $\mathbb{E}$. Finally, since `q[i]` is out of scope at the sampling instruction, expression using `q[i]` and variable `q[i]` are excluded, resulting $\mathbb{V} = \{\}$ and $\mathbb{E} = \{\}$.

For η_2, we have $\Gamma_s = \{\texttt{q}: *, T_\eta: *\}$. Since both assertions depend on η_2 and `q[i]` and T, Algorithm 1 adds $\widehat{\texttt{q[i]}}$ and $\widehat{T_\eta}$ into $\mathbb{V}$. Similar to η_1, the algorithm also adds `q[i]` + η_2 $\geq$ T_η into $\mathbb{E}$. Finally, all expressions and variable are in scope, resulting $\mathbb{V} = \{\widehat{\texttt{q[i]}}, \widehat{T_\eta}\}$ and $\mathbb{E} = \{\texttt{q[i]} + \eta_2 \geq T_\eta\}$.

3.5 Function Signature Rewrite

Finally, CheckDP rewrites the function signature to reflect the extra parameters and holes introduced in the transformed code. In general, $M(inp)$ is transformed to a new function signature $M'(inp, \widehat{inp}, sample, \theta)$ where $\widehat{inp}$ are the distance variables associated with inputs whose distance is not zero (e.g., $\widehat{q}$ is associated with q in GapSVT), *sample* is a list of random values used in M, and θ are the missing holes in alignment templates.

3.6 Shadow Execution

To tackle challenging mechanisms such as Report Noisy Max [25], CheckDP uses *shadow execution* [51]. Intuitively, the shadow execution tracks another program execution where the injected noises are always the same as those in the original execution. Therefore, values computed in the shadow execution incur no privacy cost. The aligned execution can then switch to shadow execution when certain conditions are met, allowing extra permissiveness [51].

Supporting shadow execution only requires a few modifications:

(1) Expressions will have a pair of distances ($\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle$), where the extra distance $\mathbb{d}^\dagger$ tracks the distance in the shadow execution;
(2) Since the branches and loop conditions in shadow execution are not aligned, they might diverge from the original execution. Hence, a separate shadow branch/loop is generated to correctly update the shadow distances for the variables.

Since the extended transformation rules largely follow the corresponding typing rules of ShadowDP, we present the complete set of rules with detailed explanations in the Appendix.

3.7 Soundness

CheckDP enforces a fundamental property: suppose $M(inp)$ is transformed to $M'(inp, \widehat{inp}, sample, \theta)$, then $M(inp)$ is differentially private if there is a list of values of θ, such that all assertions in M' hold for all $inp, \widehat{inp}, sample$. Recall that an alignment template $\mathcal{A}$ is a function of θ. Hence, we have a concrete alignment $\mathcal{A}(\theta)$ (i.e., a proof) when such values of θ exist.

We build the soundness of CheckDP based on that of ShadowDP [51]. The main difference is that ShadowDP requires every sampling command $\eta :=$ **Lap** r to be manually annotated. Thus, we can easily rewrite a program M in CheckDP to a program $\tilde{M}$ in ShadowDP by adding the following annotations:

$\eta :=$ **Lap** $r \rightarrow \eta :=$ **Lap** r; $\circ$; $\mathcal{A}_\eta(\theta)$ (CheckDP to ShadowDP)

where $\mathcal{A}_\eta$ is the alignment template for η. We formalize the main soundness results next; the full proof can be found in the full version of this paper [50].

Theorem 2 (Soundness). *Let M be a mechanism written in CheckDP. With a list of concrete values of θ, let $\tilde{M}$ be the corresponding mechanism in ShadowDP by rule* (CheckDP to ShadowDP). *If (1) M type checks, i.e., $\vdash \Gamma \{M \rightharpoonup M'\} \Gamma'$ and (2) the assertions in M' hold for all inputs. Then $\tilde{M}$ type checks in ShadowDP, and the assertions in $\tilde{M}'$ (transformed from $\tilde{M}$ by ShadowDP) pass.*

Theorem 3 (Privacy). *With exactly the same notation and assumption as Theorem 2, M satisfies ϵ-differential privacy.*

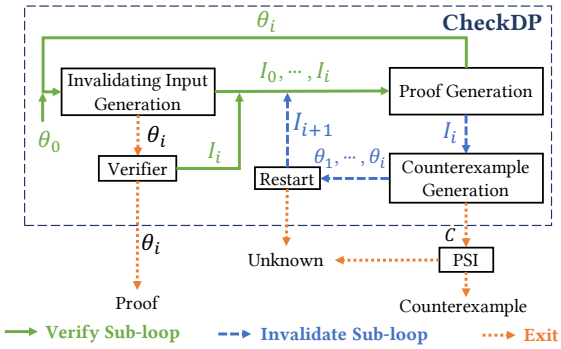


Figure 4: Overview of the verify-invalidate loop of CheckDP.

4 PROOF AND COUNTEREXAMPLE GENERATION

Recall that the transformed source code has the form of the following: $M'(inp, \widehat{inp}, sample, \theta)$. For brevity, Let I denote a triple of $(inp, \widehat{inp}, sample)$, and C denote a counterexample in the form of $C = (inp, inp', o)$ as defined in Section 2.3. Proof/counterexample generation is divided into two tasks:

- Proof Generation: find an instantiation of θ such that assertions in M' never fail for any input I, or
- Counterexample Generation: find an instantiation of I, such that no θ exists to make all assertions in M' pass, and then construct a counterexample C based on I.

The key challenge here is the infinite search space of both θ and I. Our insight is to use a verify-invalidate loop, as depicted in Figure 4, to improve θ and I after each iteration. At a high-level, the iterative process involves two sub-loops: the (green) verify sub-loop generates proofs, and the (blue) invalidate sub-loop generates counterexamples. Moreover, the two sub-loops are integrated: starting from a default θ_0 where $\theta_0[i] = 0$. $\forall i$, the procedure generates sequences of proofs and invalidating inputs in the form of $\theta_0, I_0, \theta_1, I_1, \cdots$. The final θ_k or I_k is used to construct proof or counterexamples correspondingly.

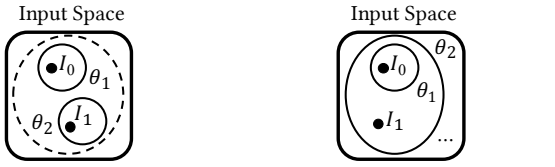
4.1 Verify Sub-Loop

The verify sub-loop that involves Invalidating Input Generation and Proof Generation components is responsible of generating a sequence of improving alignments $\theta_0, \theta_1, \cdots, \theta_i$ such that, if the mechanism is correct, θ_i is a privacy proof (i.e, $\forall I.\ M'(I, \theta_i)$).

Invalidating Input Generation. This component takes a proof candidate θ_i and then tries to find an input I_i such that $\neg M'(I_i, \theta_i)$ (meaning that at least one assertion in M' fails).

Intuitively, θ_i is the currently "best" proof candidate (initially, a default null proof $\theta_0 = [0, \cdots]$ is used to bootstrap the process) that is able to validate all previously found inputs $(I_0, \cdots, I_{i-1})$. An input I_i, if any, shows that θ_i is in fact not a valid proof (recall that a proof needs to ensure $M'(I, \theta_i)\ \forall I$). Hence, we call such I_i an *invaliding input* of θ_i and feed it with all previously identified invalidating inputs to the Proof Generation component following the "Verify Sub-loop" edge.

Take GapSVT (Figure 1) for example. Since the initial null proof $\theta_0 = [0, \cdots]$ does not align any random variable, any input, say I_0, that diverges on the branch $q[i] + \eta_2 \geq T$ will trigger an assertion violation at Line 22. Hence, the identified invalidating input I_0 is fed to the Proof Generation component.



(a) A case where θ_i cannot be improved. **(b) Iteratively improving the alignment θ_i.**

Figure 5: Tentative alignments and invalidating inputs.

Proof Generation. This component takes in a series of invalidating inputs $I_0, \cdots, I_i$ seen so far, and tries to find an proof candidate θ_i such that: $M'(I_0, \theta_i) \wedge \cdots \wedge M'(I_i, \theta_i)$.

Intuitively, the goal is to find a proof candidate θ_i that successfully "covers" all invalidating inputs seen so far. Most likely, an improved proof candidate θ_i that is able to align randomness for more inputs is generated by the component. Then θ_i is fed back to the Invalidating Input Generation component, closing the loop.

Consider the GapSVT example again. In order to align randomness for the invalidating input I_0, one possible θ_1 is to align the random variable η_2 by $-\widehat{q}[i]$ to cancel out the difference introduced by $q[i]$. Note that this tentative proof θ_1 does not work for *all possible* inputs: it only serves as the "best" proof given I_0. With the Verify Sub-loop, such imperfect proof candidates enable the generation of more invalidating inputs, such as an invalidating input I_1 where the query answers are mostly below the threshold T (I_1 invalidates θ_1 since a privacy cost incurs whenever any branch is taken, which eventually exhausts the given privacy budget). Therefore, a more general proof that leverages the conditional expression $q[i] + \eta_2 \geq T\ ?\ \bullet : \bullet$ in the alignment template can be discovered by Proof Generation. For GapSVT, the Verify sub-loop eventually terminates with a correct proof (Section 5).

Exit Edges. The verify loop has two exit edges. First, when no invalidating input is generated, θ_i is likely a valid proof. Hence, θ_i is passed to a verifier with the following condition: $\forall I.\ M'(I, \theta_i)$. Due to the soundness result (Theorem 3), we have a proof of differential privacy when the verifier passes (the "Exit" edge from Verifier component). Otherwise, CheckDP uses the counterexample returned by the verifier to construct I_i (the "Verify Sub-loop" edge). We note that the verification step is required since KLEE, the symbolic executor that we use to find invalidating inputs, is unsound (i.e., it might miss an invalidating input) in theory; however, we did not experience any such unsound case of KLEE in our experience.

Second, the Proof Generation component might fail to find an alignment for $I_0, \cdots, I_i$, a case that will eventually occur for incorrect mechanisms. This exit edge leads to the invalidate sub-loop that we discuss next.

4.2 Invalidate Sub-Loop

The invalidate sub-loop involves Counterexample Generation and Restart; it is responsible of generating *one single* invalidating input I such that, if the mechanism is incorrect, I cannot be aligned (i.e, $\nexists \theta.\ M'(I, \theta)$). At first glance, it could be attempting to directly use I_i from the Verify Sub-Loop. However, this is problematic both in

theory and in practice: no alignment for $I_0, \cdots, I_i$ does not imply no alignment of I_i alone. In practice, we found such a naive approach fails for BadSmartSum and BadGapSVT in Section 5.

Counterexample Generation. This component takes an invalidating input I_i and then tries to find an alignment θ_i such that $M'(I_i, \theta_i)$ (meaning that I_i is not a counterexample since it can be aligned by θ_i). For example, consider a corner case in Figure 5a, where Proof Generation fails to find a common proof of both I_0 and I_1, but each of I_0 and I_1 has a proof (illustrated by the two solid circles around them). Mostly likely, this occurs when the program being analyzed is incorrect (hence, no common proof) but neither I_1 nor I_2 is a good candidate for counterexample of differential privacy, since each of them can be aligned in isolation.

Restart. This component is symmetric to the Invalidating Input Generation component in the verify sub-loop: it takes all previously found proof candidates $\theta_1, \cdots, \theta_i$ and tries to find an invalidating input I_{i+1} such that: $\neg M'(I_{i+1}, \theta_1) \wedge \cdots \neg M'(I_{i+1}, \theta_i)$.

If found, I_{i+1} will intuitively be out of scope of all found proofs and serve as a "better" invalidating input. In theory, we can close the invalidate sub-loop by feeding I_{i+1} back to Counterexample Generation. However, doing so will make proof and counterexample generation isolated tasks. Instead, we take an integrated approach, which we discuss shortly, where the verify and invalidate sub-loops communicate to generate proofs and counterexamples in a more efficient and simultaneous way.

Exit Edges. If no θ is found to prove $I_i = (inp, \widehat{\text{inp}}, sample)$, a counterexample $C = (inp, inp + \widehat{inp}, M'(inp, \widehat{inp}, sample, \theta_0))$ can be formed and sent to an external exact probabilistic solver PSI [33] for validation. In theory, the Restart component might fail to find a new invalidating input given $\theta_1, \cdots, \theta_i$. However, this "unknown" state never showed up in our experience.

4.3 Integrating Verify and Invalidate Sub-Loops

We integrate the verify and invalidate sub-loops as follows: following the "Invalidate Sub-loop" edge of the Proof Generation component, the latest invalidating input I_i (i.e., the "best" invalidating input so far) is passed to the Counterexample Generation component to start the invalidate sub-loop. Moreover, the newly generated invalidating input I_i from the Restart component is fed back to the Proof Generation component to start the verify sub-loop.

We note that by the design of the verify-invalidate loop, it alternatively runs Invalidating Input Generation and Proof Generation components. By doing so, the proof keeps improving while the invalidating inputs are getting closer to a true counterexample (since the most recent one violates a "better" proof). More intuitively, consider an invalidating input I_0 as a point in the entire input space, illustrated in Figure 5b. A proof candidate θ_1 is able to prove the algorithm for a subset of inputs including I_0 (indicated by the circle around I_0). The Invalidating Input Generation component then tries to find another invalidating I_1 that violates θ_1 (falls outside of the θ_1 circle). Next, the Proof Generation component finds better proof candidate θ_2 which proves ("covers") both I_0 and I_1.

We also note that it is crucial to consider all invalidating inputs so far rather than the last input I_i in the Proof Generation component: the efficiency of our approach crucially relies on "improving" the proofs quantified by validating more invalidating inputs. Without the improving proofs, the iterative procedure might fail to terminate in case shown in Figure 5a: the procedure might repeat $I_0, \theta_1, I_1, \theta_2, I_0, \theta_1, \cdots$. This is confirmed in our empirical study.

Unknown State. Due to the soundness result (Theorem 3), the program being analyzed is verified whenever CheckDP returns with a proof. Moreover, a validated counterexample by PSI disproves an incorrect mechanism. However, two reasons might lead to the "unknown" state in the Figure 4: the generated counterexample is invalid or the Restart component fails to find a new invalidating input. However, for all the correct and incorrect examples we explored, the unknown state never showed up.

5 IMPLEMENTATION AND EVALUATION

We implemented CheckDP in Python[3]. The *Program Transformation* phase is implemented as a trans-compiler from CheckDP code (Figure 2) to C code. Following the transformation rules in Figure 3, the trans-compiler tracks the typing environment, gathers the needed constraints for the expressions, and more importantly, instruments corresponding statements when appropriate. Moreover, it adds a final assertion **assert**($\mathbf{v}_\epsilon \leq \epsilon_b$) before each **return** command, where ϵ_b is the annotated privacy bound to be checked. Once all assertions are generated, the trans-compiler generates one alignment template for each sampling instruction as described in Algorithm 1. For the *Proof and Counterexample Generation* phase (i.e., verify-invalidate loop in Section 4), we used an efficient symbolic executor KLEE [18] for most tasks. Due to limited support of unbounded lists in KLEE, we fix the length of lists to be 5 in our evaluation. Also, to speed up the search, KLEE is configured to exit once an assertion is hit. We note that the use of KLEE is to *discover* alignments and counterexamples, where alignments are eventually verified by our sound Verifier component with arbitrary array length; counterexamples are confirmed by PSI. Moreover, CheckDP automatically extends the array length until either a verified proof or verified counterexample is produced.

Finally, we deploy a verification tool CPAChecker [13] for the Verifier component in CheckDP, which is capable of automatically verifying C programs with given configuration (*predicateAnalysis* is used). Note that CPAChecker is able to generate counterexamples for a failed verification. If the verification fails (which did not happen in our evaluation), CheckDP can feed the counterexample back to the Proof and Counterexample Generation component.

5.1 Case Studies

Aside from GapSVT, we also evaluate CheckDP on the standard benchmark used in previous mechanism verifiers [3, 51, 52] and counterexample generators [14, 23],[4] including correct ones such as NumSVT, PartialSum, and SmartSum, as well as the incorrect variants of SVT reported in [40] and BadPartialSum. To show the power of CheckDP and expressiveness of our template generation algorithm, we also evaluate on a couple of correct/incorrect mechanisms that, to the best of our knowledge, have not been proved/disproved

[3]Publically available at https://github.com/cmla-psu/checkdp.

[4]We note that like all tools designed for privacy mechanisms (e.g., [3, 14, 23, 51, 52]), the benchmark do not include iterative programs that are built on those privacy mechanisms, such as k-means clustering, k-medians, since they are out of scope.

Table 1: Detected counterexamples for the incorrect algorithms and comparisons with other sampling-based counterexample detectors. $\#t$ stands for `true` and $\#f$ stands for `false`.

Mechanism	q	q′	Extra Args	Output	Iterations	Time(s)	StatDP [23]	DP-Finder [14]	DiPC [4]
BadNoisyMax	[0, 0, 0, 0, 0]	[−1, 1, 1, 1, 1]	N/A	0	3	5.7	11.2	2561.5	N/A
BadSVT1	[0, 0, 0, 0, 1]	[1, 1, 1, 1, 0]	T: 0, N: 1	$[\#f, \#f, \#f, \#f, \#t]$	4	3.2	4.9	3847.5 (Semi-Manual)	N/A
BadSVT2	[0, 0, 0, 0, 1]	[1, 1, 1, 1, −1]	T: 0, N: 1	$[\#f, \#f, \#f, \#f, \#t]$	4	2.0	15.6	4126.1 (Semi-Manual)	N/A
BadSVT3	[0, 0, 0, 0, 1]	[1, 1, 1, 1, −1]	T: 0, N: 1	$[\#f, \#f, \#f, \#f, \#t]$	4	2.1	9.1	3476.2 (Semi-Manual)	269
BadGapSVT	[0, 0, 0, 0, 0]	[1, 1, 1, 1, −1]	T: 0, N: 1	[0, 0, 0, 0, 1]	4	5.7	10.6	11611.6 (Semi-Manual)	N/A
BadAdaptiveSVT	[0, 0, 0, 0, 2]	[1, 1, 1, 1, −1]	T: 0, N: 1	[0, 0, 0, 0, 17]	8	14.2	Search Failed	Search Failed	N/A
Imprecise SVT	[0, 0, 0, 0, 1]	[1, 1, 1, 1, −1]	T: 0, N: 1	$[\#f, \#f, \#f, \#f, \#t]$	4	8.6	Search Failed	Search Failed	N/A
BadSmartSum	[0, 0, 0, 0, 0]	[0, 0, 0, 1, 0]	T: 3, M: 4	[0, 0, 0, 0, 0]	4	6.3	22.4 (Semi-Manual)	Search Failed	N/A
BadPartialSum	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 1]	N/A	0	3	3.7	3.8	1128.5	N/A

Table 2: Alignments found for the correct algorithms. Ω_* stands for the branch condition in each mechanism, where $\Omega_{NM} = q[i] + \eta > bq \vee i = 0$, $\Omega_{SVT} = q[i] + \eta_2 \geq T_\eta$, $\Omega_{Top} = q[i] + \eta_2 - T_\eta \geq \sigma$, $\Omega_{Middle} = q[i] + \eta_3 - T_\eta \geq 0$

Mechanism	Alignment			Iterations	Time (s)	ShadowDP [51]	Coupling [3]	DiPC [4]
	η_1	η_2	η_3					
ReportNoisyMax	Ω_{NM} ? $1 - \widehat{q}[i] : 0$	N/A	N/A	10	69.3	Manual	22	193
PartialSum	$-\widehat{sum}$	N/A	N/A	2	5.6	Manual	14	N/A
SmartSum	$-\widehat{sum} - \widehat{q}[i]$	$-\widehat{q}[i]$	N/A	6	6.8	Manual	255	N/A
SVT	1	Ω_{SVT} ? $1 - \widehat{q}[i] : 0$	N/A	4	6.2	Manual	580	825
Monotone SVT (Increase)	0	Ω_{SVT} ? $1 - \widehat{q}[i] : 0$	N/A	8	18.4	N/A	N/A	N/A
Monotone SVT (Decrease)	0	Ω_{SVT} ? $-\widehat{q}[i] : 0$	N/A	8	20.5	N/A	N/A	N/A
GapSVT	1	Ω_{SVT} ? $1 - \widehat{q}[i] : 0$	N/A	6	13.5	Manual	N/A	N/A
NumSVT	1	Ω_{SVT} ? 2 : 0	$-\widehat{q}[i]$	4	8.8	Manual	5	N/A
AdaptiveSVT	1	Ω_{Top} ? $1 - \widehat{q}[i] : 0$	Ω_{Middle} ? $1 - \widehat{q}[i] : 0$	10	25.6	N/A	N/A	N/A

by existing verifiers and counterexample generators. This set of mechanisms include: Sparse Vector with monotonic queries [40], AdaptiveSVT (called Adaptive Sparse Vector with Gap in [24]) as well as new incorrect variants of SVT, AdaptiveSVT and SmartSum. For all mechanisms we explore, CheckDP is able to: (1) provide a proof if it satisfies differential privacy, or (2) provide a counterexample if it violates the claimed level of privacy. Neither false positives nor false negatives were observed. In this section, we discuss the new cases; detailed explanations can be found in the Appendix.

Sparse Vector with Monotonic Queries. The queries in some usages of SVT are monotonic. In such cases, a **Lap** $2N/\epsilon$ noise (instead of **Lap** $4N/\epsilon$ in SVT) is sufficient for ϵ-privacy [40].

AdaptiveSVT, BadAdaptiveSVT and BadSmartSum. Ding et al. [24] recently proposed a new variant of SVT which adaptively allocates privacy budget, saving privacy cost when *noisy* query answers are much larger than the noisy threshold. The difference from standard (correct) SVT is that it first draws a $\eta_2 :=$ **Lap** $8N/\epsilon$ noise (instead of **Lap** $4N/\epsilon$ in SVT) and checks if the gap between noisy query and noisy threshold T_η is larger than a preset hyper-parameter σ (**if** $q[i]$ + η_2 - $T_\eta \geq \sigma$). If the test succeeds, the gap is directly returned, hence costing only $\epsilon/(8N)$ (instead of $\epsilon/(4N)$) privacy budget. Otherwise, it draws $\eta_3 :=$ **Lap** $4N/\epsilon$ and follows the same procedure as SVT. We also create an incorrect variant called BadAdaptiveSVT. It directly releases the noisy query answer instead of the gap after the first test. Sampling-based methods can have difficulty detecting the privacy leakage because the privacy-violating branch of the BadAdaptiveSVT code is not executed frequently. We also create an incorrect variant of SmartSum by releasing a noise-less sum of queries in an infrequent branch. Details of SmartSum and this variant can be found in the Appendix.

SVT with Wrong Privacy Claims (Imprecise SVT). We also study another interesting yet quite challenging violation of differential privacy: suppose a mechanism satisfies 1.1-differential privacy but claims to be 1-differentially private. This slight violation requires precise reasoning about the privacy cost and poses challenges for prior sampling-based approaches. We thus evaluate a variant of SVT, referred to as Imprecise SVT, which is $\epsilon = 1.1$-differentially private but with an incorrect claim of $\epsilon = 1$ (**check**(1) in the signature).

5.2 Experiments

We evaluate CheckDP on a Intel® Xeon® E5-2620 v4 CPU machine with 64 GB memory. To compare CheckDP with the state-of-the-art tools, we either directly run tools on the benchmark when they are publicly available (including ShadowDP [51], StatDP [23] and DP-Finder [14]), or cite the reported results from the corresponding papers (including Coupling [3] and DiPC [4]).[5] For the latter case, we note that the numbers are for reference only, due to different settings, including hardware, used in the experiments.

Counterexample Generation. Table 1 lists the counterexamples (i.e., a pair of related inputs and a feasible output that witness the violation of claimed level of privacy) automatically generated by CheckDP for the incorrect algorithms. For all incorrect algorithms, CheckDP is able to provide a counterexample (validated by PSI [33]) in 15 seconds and 8 iterations.[6]

Notably, both StatDP and DP-Finder fail to find the privacy violations in BadSmartSum and BadAdaptiveSVT, as well as the violation of $\epsilon = 1$-privacy in Imprecise SVT after hours of searching.[7] This is due to the limitations of sampling-based approaches. In certain cases, we can help these sampling-based algorithms by *manually*

[5] Default settings are used in our evaluation: 100K/500K samples for event selection/hypothesis testing components of StatDP; 50 iterations for sampling and optimization components of DP-Finder where each iteration collects 409,600 samples on average.
[6] We note that the counterexample of BadSmartSum is validated on a slightly modified algorithm since PSI does not support modulo operation.
[7] For StatDP, we use 1000X of the default number of samples to confirm the failure.

providing proper values for the extra arguments that some of the mechanisms require (4^{th} column of Table 1). This extra advantage (labeled Semi-Manual in the table) sometimes allows the sampling-based methods to find counterexamples. We note that CheckDP, in contrast, generates all inputs automatically.

Verification. Table 2 lists the automatically generated proofs (i.e., alignments) for each random variable in the correct algorithms. Due to the soundness of CheckDP, all returned proofs are valid. We note that correct algorithms on average take more iterations (and hence, time) to verify; still all of them are verified within 70 seconds. . Report Noisy Max is the only example that uses shadow execution; the selector generated is $\mathcal{S} = q[i] + \eta_2 \geq bq \vee i = 0 \,?\, \dagger : \circ$, the same as the manually generated one in [51].

Performance. We note that all examples finish within 10 iterations. We contribute the efficiency to the reduced search space of Algorithm 1 (e.g., the alignment template for GapSVT only contains 7 "holes") as well as our novel verify-invalidate loop that allows verification and counterexample generation components to communicate in meaningful ways. Compared with StatDP and DP-Finder, CheckDP is more efficient on the cases where they do find counterexamples. Compared with static tools [3, 4], we note that CheckDP is much faster on BadSVT3, SmartSum and SVT. In summary, CheckDP is mostly more efficient compared to counterexample detectors and automated provers.

6 RELATED WORK

Proving and Disproving Differential Privacy. Concurrent works [4, 30] also target both proving and disproving differential privacy. Barthe et al. [4] identify a non-trivial class of programs where checking differential privacy is decidable. Their work also supports approximate differential privacy. However, the decidable programs only allow finite inputs and outputs, while CheckDP is applicable to a larger class of programs. Moreover, CheckDP is more scalable, as observed in our evaluation. Farina [30] builds a relational symbolic execution framework, which when combined with probabilistic couplings, is able to prove differential privacy or generate failing traces for SVT and its two incorrect variants. However, it is unclear if the employed heuristic strategies work on other mechanisms, such as Report Noisy Max. Moreover, CheckDP is likely to be more scalable since their approach treats both program inputs and proofs in a symbolic way, whereas in the novel verify-invalidate loop of CheckDP, either program inputs or proofs are concrete.

Formal Verification of Differential Privacy. From the verification perspective, CheckDP is mostly related to LightDP [52] and ShadowDP [51] – all use randomness alignment. The type system of CheckDP is directly inspired by that of [51, 52]. However, the most important difference is that CheckDP is *the first* that automatically generates alignment-based proofs; both LightDP and ShadowDP assume manually-provided proofs. As discussed in Section 3, CheckDP also simplifies the previous type systems and defers all privacy-related checks to later stages. Both changes are important for automatically generating proofs and counterexamples.

Besides alignment-based proofs, probabilistic couplings and liftings [3, 7, 9] have also been used in language-based verification of differential privacy. Most notably, Albarghouthi and Hsu [3] proposed the first automated tool capable of generating *coupling proofs* for complex mechanisms. Coupling proofs are known to be more general than alignment-based proofs, while alignment-based proofs are more light-weight. Since CheckDP and [3] are built on different proof techniques, the proof generation algorithm in [3] is not directly applicable in our context. Moreover, [3] does not generate counterexamples and we do not see an obvious way to extend the Synthesize-Verify loop of [3] to do so .

With verified privacy mechanisms, such as SVT and Report Noisy Max, we still need to verify that the larger program built on top of them is differentially private. An early line of work [8, 10, 11, 32, 44] uses (variations of) relational Hoare logic and linear indexed types to derive differential privacy guarantees. For example, Fuzz [44] and its successor DFuzz[32] combine linear indexed types and light-weight dependent types to allow rich sensitivity analysis and then use the composition theorem to prove overall system privacy. We note that CheckDP and those systems are largely orthogonal: those systems rely on trusted mechanisms (e.g., SVT and Report Noisy Max) without verifying them, while CheckDP is likely less scalable; they can be combined for sophisticated verification tasks.

Counterexample Generation. Ding et al. [23] and Bichsel et al. [14] proposed counterexample generators that rely on *sampling* – running an algorithm hundreds of thousands of times to estimate the output distribution of mechanisms (this information is then used to find counterexamples). The strength of these methods is that they do not rely on external solvers, and more importantly, they are not tied to (the limitation of) any particular proof technique (e.g., randomness alignment and coupling). However, sampling also make the counterexample detectors imprecise and more likely to fail in some cases, as confirmed in the evaluation.

7 CONCLUSIONS AND FUTURE WORK

We proposed CheckDP, an integrated tool based on static analysis for automatically proving or disproving that a mechanism satisfies differential privacy. Evaluation shows that CheckDP is able to provide proofs for a number of algorithms, as well as counterexamples for their incorrect variants within 2 to 70 seconds. Moreover, all generated proofs and counterexamples are validated.

For future work, CheckDP relies on the underlying randomness alignment technique; hence it is subject to its limitations, including lack of support for (ϵ, δ)-differential privacy and renyi differential privacy [43]. We plan to extend the underlying proof technique for other variants of differential privacy.

Moreover, subtle mechanisms such as PrivTree [53] and private selection [39], where the costs of intermediate results are dependent on the data but the cost of sum is data-independent, is still out of reach for formal verification (including CheckDP).

Finally, CheckDP is designed for DP mechanisms, rather than larger programs built on top of them. An interesting area of future work is integrating CheckDP with tools like DFuzz [32], which are more efficient on programs built on top of DP mechanisms (but don't verify the mechanisms themselves).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedbacks. This work was supported by NSF Awards CNS-1702760.

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, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations (CSFW '04)*. IEEE Computer Society, Washington, DC, USA, 100–.

[7] 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.

[8] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving Differential Privacy in Hoare Logic. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium (CSF '14)*. IEEE Computer Society, Washington, DC, USA, 411–424.

[9] 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.

[10] 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.

[11] 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.

[12] 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

[13] 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.

[14] 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.

[15] 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

[16] Mark Bun and Thomas Steinke. 2016. Concentrated Differential Privacy: Simplifications, Extensions, and Lower Bounds. In *Proceedings, Part I, of the 14th International Conference on Theory of Cryptography - Volume 9985*. Springer-Verlag New York, Inc., New York, NY, USA, 635–658.

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

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[19] 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.

[20] Rui Chen, Qian Xiao, Yu Zhang, and Jianliang Xu. 2015. Differentially Private High-Dimensional Data Publication via Sampling-Based Inference. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. ACM, New York, NY, USA, 129–138. https://doi.org/10.1145/2783258.2783379

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

[22] 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

[23] 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 '18)*. ACM, New York, NY, USA, 475–489.

[24] 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

[25] 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

[26] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy via Distributed Noise Generation. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques* (St. Petersburg, Russia) *(EUROCRYPT'06)*. Springer-Verlag, Berlin, Heidelberg, 486–503.

[27] 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.

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

[29] 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.

[30] Gian Pietro Farina. 2020. *Coupled Relational Symbolic Execution*. Ph.D. Dissertation. State University of New York at Buffalo.

[31] 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.

[32] 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

[33] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 62–83.

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

[35] 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

[36] 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.

[37] Dexter Kozen. 1981. Semantics of probabilistic programs. *J. Comput. System Sci.* 22, 3 (1981), 328 – 350.

[38] Jaewoo Lee and Christopher W. Clifton. 2014. Top-k Frequent Itemsets via Differentially Private FP-trees. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD '14)*. ACM, New York, NY, USA, 931–940. https://doi.org/10.1145/2623330.2623723

[39] Jingcheng Liu and Kunal Talwar. 2019. Private Selection from Private Candidates. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing* (Phoenix, AZ, USA) *(STOC 2019)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3313276.3316377

[40] 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.

[41] 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

[42] Frank McSherry. 2018. Uber's differential privacy .. probably isn't. https://github.com/frankmcsherry/blog/blob/master/posts/2018-02-25.md (retrieved 11/15/2019).

[43] I. Mironov. 2017. Rényi Differential Privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, Piscataway, NJ, USA, 263–275. https://doi.org/10.1109/CSF.2017.11

[44] 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

[45] Aaron Roth. 2011. The Sparse Vector Technique. http://www.cis.upenn.edu/~aaroth/courses/slides/Lecture11.pdf.

[46] 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

[47] Ben Stoddard, Yan Chen, and Ashwin Machanavajjhala. 2014. Differentially Private Algorithms for Empirical Machine Learning. arXiv:1411.5428 [cs.LG]

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

[49] Tachio Terauchi and Alex Aiken. 2005. Secure information flow as a safety problem. In *International Static Analysis Symposium*. Springer, 352–367.

[50] Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng Zhang. 2020. CheckDP: An Automated and Integrated Approach for Proving Differential Privacy or Finding Precise Counterexamples. arXiv:2008.07485 [cs.PL]

[51] 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

[52] 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.

[53] Jun Zhang, Xiaokui Xiao, and Xing Xie. 2016. PrivTree: A Differentially Private Algorithm for Hierarchical Decompositions. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 155–170. https://doi.org/10.1145/2882903.2882928

A SHADOW EXECUTION

We show how to extend the program transformation in Figure 3 to support shadow execution. At a high level, the extension encodes the selectors (which requires manual annotations in ShadowDP [51]) and integrates them with the generated templates. With the extra "holes" in the templates, the verify-invalidate loop will automatically find alignments (including selectors)/counterexamples. The complete set of transformation rules with shadow execution is shown in Figure 7, where the extensions are highlighted in gray.

Syntax and Expressions. Since a new shadow execution is tracked, types for each variable would be expanded to include a pair of distances $\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle$. More specifically, the types should now be defined as: $\tau ::= \texttt{num}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle} \mid \texttt{bool} \mid \texttt{list}\ \tau$.

With the modified types, corresponding modifications to the transformation rules for expressions are straightforward and minimal: the handling of shadow distances are essentially the same as that of aligned distances.

Normal Commands. Following the type system of ShadowDP, a program counter $pc \in \{\top, \bot\}$ is introduced to each transformation rule for commands to capture potential divergence of shadow execution. Specifically, $pc \vdash c \rightharpoonup c'$. $pc = \top$ (resp. $\bot$) means that the branch / loop command might diverge in the shadow execution (resp. must stay the same). The value of pc is used to guide how each rule should handle the shadow distances (e.g., (T-Asgn)), which we will explain shortly. Therefore, another auxiliary function `updatePC` is added to track the value of pc.

$$(\!|r, \Gamma|\!)^\star = r \quad (\!|\texttt{true}, \Gamma|\!)^\star = \texttt{true} \quad (\!|\texttt{false}, \Gamma|\!)^\star = \texttt{false}$$

$$(\!|x, \Gamma|\!)^\star = \begin{cases} x + \mathbb{n}^\dagger & \text{, if } \Gamma \vdash x : \texttt{num}_{\langle \mathbb{n}^\circ, \mathbb{n}^\dagger \rangle} \\ x & \text{, else} \end{cases}$$

$$(\!|e_1\ \texttt{op}\ e_2, \Gamma|\!)^\star = (\!|e_1, \Gamma|\!)^\star\ \texttt{op}\ (\!|e_2, \Gamma|\!)^\star \text{ where } \texttt{op} = \oplus \cup \otimes \cup \odot$$

$$(\!|e_1[e_2], \Gamma|\!)^\star = \begin{cases} e_1[e_2] + \widehat{e_1}^\dagger[e_2] & \text{, if } \Gamma^\dagger \vdash e_1 : \texttt{list num}_* \\ e_1[e_2] & \text{, else} \end{cases}$$

$$(\!|e_1 :: e_2, \Gamma|\!)^\star = (\!|e_1, \Gamma|\!)^\star :: (\!|e_2, \Gamma|\!)^\star \qquad (\!|\neg e, \Gamma|\!)^\star = \neg (\!|e, \Gamma|\!)^\star$$

$$(\!|e_1\ ?\ e_2 : e_3, \Gamma|\!)^\star = (\!|e_1|\!)^\star\ ?\ (\!|e_2, \Gamma|\!)^\star : (\!|e_3, \Gamma|\!)^\star$$

$$(\!|\texttt{skip}, \Gamma|\!)^\star = \texttt{skip} \qquad \frac{(\!|c_1; \Gamma|\!)^\star = c_1' \quad (\!|c_2; \Gamma|\!)^\star = c_2'}{(\!|c_1; c_2, \Gamma|\!)^\star = c_1'; c_2'}$$

$$(\!|x := e, \Gamma|\!)^\star = (\widehat{x}^\dagger := (\!|e, \Gamma|\!)^\star - x)$$

$$\frac{(\!|c_i, \Gamma|\!)^\star = c_i' \quad i \in \{1, 2\}}{(\!|\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \Gamma|\!)^\star = \texttt{if}\ (\!|e, \Gamma|\!)^\star\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'}$$

$$\frac{(\!|c, \Gamma|\!)^\star = c'}{(\!|\texttt{while}\ e\ \texttt{do}\ c, \Gamma|\!)^\star = \texttt{while}\ (\!|e, \Gamma|\!)^\star\ \texttt{do}\ c'}$$

Figure 6: Transformation of expressions and commands for aligned and shadow execution, where $\star \in \{\circ, \dagger\}$.

Compared with the type system of ShadowDP, the first major difference is in (T-Asgn). If $pc = \bot$, shadow distances are handled as the aligned distances. However, when $pc = \top$ (shadow execution diverges), it updates the shadow distance of the variable to make sure the value in shadow execution (i.e., $x + \widehat{x}^\dagger$) remains the same after the assignment. For example, Line 19 in Figure 8 is instrumented to maintain the value of bq in the shadow execution ($\texttt{bq} + \widehat{\texttt{bq}}^\dagger$), so that the branch at Line 25 is not affected by the new assignment of bq.

As previously explained, a separate shadow branch / loop has to be generated to correctly track the shadow distances of the variables. More specifically, Rules (T-If) and (T-While) is extended to include an extra shadow execution command $c^\dagger$ *when pc transits from* $\bot$ *to* $\top$. The shadow execution is constructed by an auxiliary function $(\!|c, \Gamma|\!)^\dagger$, as defined in Figure 6, which is the same as the ones in ShadowDP [51]. It essentially replaces each variable with its correspondence (e.g., variable x to $x + \widehat{x}^\dagger$), as is standard in self-composition [6, 49]. Note that the value of an expression e in an aligned execution (i.e., $(\!|e, \Gamma|\!)^\circ$ used in Rules (T-If) and (T-While)) are defined in a similar way.

Sampling Commands. The most interesting rule is (T-Laplace). In order to enable the automatic discovery of the selectors, our `GenerateTemplate` algorithm needs to be extended to return a selector template $\mathcal{S}$.

Transformation rules for expressions with form $\Gamma \vdash e : \mathcal{B}_{\langle \mathbb{m}^\circ, \mathbb{m}^\dagger \rangle}$

$$\frac{}{\Gamma \vdash r : \mathsf{num}_{\langle 0,0 \rangle} \mid \mathtt{true}} \text{ (T-Num)} \qquad \frac{}{\Gamma \vdash b : \mathtt{bool} \mid \mathtt{true}} \text{ (T-Boolean)} \qquad \frac{\Gamma \vdash e : \mathtt{bool} \mid C}{\Gamma \vdash \neg e : \mathtt{bool} \mid C} \text{ (T-Neg)}$$

$$\frac{\Gamma(x) = \mathcal{B}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle} \quad \mathbb{m}^\star = \begin{cases} \widehat{x}^\star & \text{if } \mathbb{d}^\star = * \\ 0 & \text{otherwise} \end{cases} \quad \star \in \{\circ, \dagger\}}{\Gamma \vdash x : \mathcal{B}_{\langle \mathbb{m}^\circ, \mathbb{m}^\dagger \rangle} \mid \mathtt{true}} \text{ (T-Var)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{num}_{\langle \mathbb{m}_1, \mathbb{m}_2 \rangle} \mid C_1 \quad \Gamma \vdash e_2 : \mathsf{num}_{\langle \mathbb{m}_3, \mathbb{m}_4 \rangle} \mid C_2}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{num}_{\langle \mathbb{m}_1 \oplus \mathbb{m}_3, \mathbb{m}_2 \oplus \mathbb{m}_4 \rangle} \mid C_1 \wedge C_2} \text{ (T-OPlus)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{num}_{\langle \mathbb{m}_1, \mathbb{m}_2 \rangle} \mid C_1 \quad \Gamma \vdash e_2 : \mathsf{num}_{\langle \mathbb{m}_3, \mathbb{m}_4 \rangle} \mid C_2}{\Gamma \vdash e_1 \otimes e_2 : \mathsf{num}_{\langle 0,0 \rangle} \mid C_1 \wedge C_2 \wedge \begin{pmatrix} (\mathbb{m}_1 = \mathbb{m}_2 = \\ \mathbb{m}_3 = \mathbb{m}_4 = 0) \end{pmatrix}} \text{ (T-OTimes)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{num}_{\langle \mathbb{m}_1, \mathbb{m}_2 \rangle} \mid C_1 \quad \Gamma \vdash e_2 : \mathsf{num}_{\langle \mathbb{m}_3, \mathbb{m}_4 \rangle} \mid C_2}{\Gamma \vdash e_1 \odot e_2 : \mathtt{bool} \mid C_1 \wedge C_2 \wedge (e_1 \odot e_2) \Leftrightarrow (e_1 + \mathbb{m}_1) \odot (e_2 + \mathbb{m}_3) \wedge (e_1 \odot e_2) \Leftrightarrow (e_1 + \mathbb{m}_2) \odot (e_2 + \mathbb{m}_4)} \text{ (T-ODot)}$$

$$\frac{\Gamma \vdash e_1 : \mathcal{B}_{\langle \mathbb{m}_1, \mathbb{m}_2 \rangle} \mid C_1 \quad \Gamma \vdash e_2 : \mathtt{list}\ \mathcal{B}_{\langle \mathbb{m}_3, \mathbb{m}_4 \rangle} \mid C_2}{\Gamma \vdash e_1 :: e_2 : \mathtt{list}\ \mathcal{B}_{\langle \mathbb{m}_3, \mathbb{m}_4 \rangle} \mid C_1 \wedge C_2 \wedge (\mathbb{m}_1 = \mathbb{m}_2 = \mathbb{m}_3 = \mathbb{m}_4 = 0)} \text{ (T-Cons)} \qquad \frac{\Gamma \vdash e_1 : \mathtt{list}\ \tau \mid C_1 \quad \Gamma \vdash e_2 : \mathsf{num}_{\langle \mathbb{m}_1, \mathbb{m}_2 \rangle} \mid C_2}{\Gamma \vdash e_1[e_2] : \tau \mid C_1 \wedge C_2 \wedge (\mathbb{m}_1 = \mathbb{m}_2 = 0)} \text{ (T-Index)}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \mid C_1 \quad \Gamma \vdash e_2 : \mathcal{B}_{\langle \mathbb{m}_1, \mathbb{m}_2 \rangle} \mid C_2 \quad \Gamma \vdash e_3 : \mathcal{B}_{\langle \mathbb{m}_3, \mathbb{m}_4 \rangle} \mid C_3}{\Gamma \vdash e_1 \,?\, e_2 : e_3 : \mathcal{B}_{\langle \mathbb{m}_1, \mathbb{m}_2 \rangle} \mid C_1 \wedge C_2 \wedge C_3 \wedge (\mathbb{m}_1 = \mathbb{m}_2 = \mathbb{m}_3 = \mathbb{m}_4)} \text{ (T-Select)}$$

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

$$\frac{\Gamma \vdash e : \mathcal{B}_{\langle \mathbb{m}^\circ, \mathbb{m}^\dagger \rangle} \mid C \quad \langle \mathbb{d}^\circ,\ c^\circ \rangle = \begin{cases} \langle 0,\ \mathbf{skip} \rangle, & \text{if } \mathbb{m}^\circ == 0, \\ \langle *,\ \widehat{x}^\circ := \mathbb{m}^\circ \rangle, & \text{otherwise} \end{cases} \quad \langle \mathbb{d}^\dagger, c^\dagger, c' \rangle = \begin{cases} \langle 0, \mathbf{skip}, \mathbf{skip} \rangle, & \text{if } pc = \bot \wedge \mathbb{m}^\dagger = 0 \\ \langle *, \widehat{x}^\dagger := \mathbb{m}^\dagger, \mathbf{skip} \rangle, & \text{if } pc = \bot \wedge \mathbb{m}^\dagger \neq 0 \\ \langle *, \mathbf{skip}, \widehat{x}^\dagger := x + \mathbb{m}^\dagger - e \rangle, & \text{otherwise} \end{cases}}{pc \vdash \Gamma\ \{x := e \rightharpoonup \mathtt{assert}(C); c'; x := e; c^\circ;\ c^\dagger\}\ \Gamma[x \mapsto \mathcal{B}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle}]} \text{ (T-Asgn)}$$

$$\frac{pc \vdash \Gamma\ \{c_1 \rightharpoonup c_1'\}\ \Gamma_1 \quad pc \vdash \Gamma_1\ \{c_2 \rightharpoonup c_2'\}\ \Gamma_2}{pc \vdash \Gamma\ \{c_1; c_2 \rightharpoonup c_1'; c_2'\}\ \Gamma_2} \text{ (T-Seq)} \qquad \frac{}{pc \vdash \Gamma\ \{\mathbf{skip} \rightharpoonup \mathbf{skip}\}\ \Gamma} \text{ (T-Skip)}$$

$$\frac{\Gamma \vdash e : \mathcal{B}_{\langle \mathbb{m}^\circ, \mathbb{m}^\dagger \rangle} \mid C}{pc \vdash \Gamma\ \{\mathbf{return}\ e \rightharpoonup \mathtt{assert}(C \wedge \mathbb{m}^\circ = 0); \mathbf{return}\ e\}\ \Gamma} \text{ (T-Return)}$$

$$\frac{\begin{array}{l} pc \vdash \Gamma \sqcup \Gamma_f\ \{c \rightharpoonup c'\}\ \Gamma_f \quad \Gamma, \Gamma \sqcup \Gamma_f, pc' \Rightarrow c_s \\ pc' = \mathsf{updatePC}(pc, \Gamma, e) \quad \Gamma_f, \Gamma \sqcup \Gamma_f, pc \Rightarrow c'' \end{array} \quad c^\dagger = \begin{cases} \mathbf{skip}, & \text{if } (pc = \top \vee pc' = \bot) \\ (\!|\mathbf{while}\ e\ \mathbf{do}\ c, \Gamma \sqcup \Gamma_f|\!)^\dagger, & \text{else} \end{cases}}{pc \vdash \Gamma\ \{\mathbf{while}\ e\ \mathbf{do}\ c \rightharpoonup c_s; (\mathbf{while}\ e\ \mathbf{do}\ (\mathbf{assert}((\!|e, \Gamma|\!)^\circ); c'; c''));\ c^\dagger\}\ \Gamma \sqcup \Gamma_f} \text{ (T-While)}$$

$$\frac{pc \vdash \Gamma\ \{c_i \rightharpoonup c_i'\}\ \Gamma_i \quad \begin{array}{l} pc' = \mathsf{updatePC}(pc, \Gamma, e) \\ \Gamma_i, \Gamma_1 \sqcup \Gamma_2, pc' \Rightarrow c_i'' \quad i \in \{1,2\} \end{array} \quad c^\dagger = \begin{cases} \mathbf{skip}, & \text{if } (pc = \top \vee pc' = \bot) \\ (\!|\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \Gamma_1 \sqcup \Gamma_2|\!)^\dagger, & \text{else} \end{cases}}{pc \vdash \Gamma\ \{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightharpoonup (\mathbf{if}\ e\ \mathbf{then}\ (\mathbf{assert}((\!|e, \Gamma|\!)^\circ); c_1'; c_1'')\ \mathbf{else}\ (\mathbf{assert}(\neg(\!|e, \Gamma|\!)^\circ); c_2'; c_2''));\ c^\dagger\}\ \Gamma_1 \sqcup \Gamma_2} \text{ (T-If)}$$

$$\frac{pc = \bot \quad \begin{array}{c} \mathcal{A}, \mathcal{S} = \mathtt{GenerateTemplate}(\Gamma, \text{All Assertions}) \\ c_a = \mathtt{assert}(((\eta + \mathcal{A})\{\eta_1/\eta\} = (\eta + \mathcal{A})\{\eta_2/\eta\} \Rightarrow \eta_1 = \eta_2)) \\ \Gamma' = \lambda x.\ \langle \mathbb{d}^\circ \sqcup \mathbb{d}^\dagger, \mathbb{d}^\dagger \rangle \text{ where } \Gamma(x) = \mathsf{num}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle} \end{array} \quad \begin{array}{c} c' = \{\widehat{x}^\circ := 0 \mid \Gamma'(x) = \mathsf{num}_{\langle *, \mathbb{d}^\dagger \rangle} \wedge \Gamma(x) = \mathsf{num}_{\langle 0, \mathbb{d}^\dagger \rangle}\} \\ c'' = \{\widehat{x}^\circ := \mathbb{m}^\dagger \mid \Gamma' \vdash x : \mathsf{num}_{\langle -, \mathbb{m}^\dagger \rangle}\} \\ c_{\mathbb{d}} = (\mathbf{if}\ \mathcal{S}\ \mathbf{then}\ c'\ \mathbf{else}\ c'') \end{array}}{pc \vdash \Gamma\ \{\eta := \mathbf{Lap}\ r \rightharpoonup c_a;\ \eta := sample[idx]; idx := idx + 1; \mathrm{v}_\epsilon := (\mathcal{S}\ ?\ \mathrm{v}_\epsilon : 0) + |\mathcal{A}|/r; \widehat{\eta} := \mathcal{A}; c_{\mathbb{d}}\}\ \Gamma'[\eta \mapsto \mathsf{num}_{\langle *, 0 \rangle}]} \text{ (T-Laplace)}$$

Transformation rules for merging environments

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \quad \begin{array}{l} c^\circ = \{\widehat{x}^\circ := 0 \mid \Gamma_1(x) = \mathsf{num}_{\langle 0, \mathbb{d}_1 \rangle} \wedge \Gamma_2(x) = \mathsf{num}_{\langle *, \mathbb{d}_2 \rangle}\} \\ c^\dagger = \{\widehat{x}^\dagger := 0 \mid \Gamma_1(x) = \mathsf{num}_{\langle \mathbb{d}_1, 0 \rangle} \wedge \Gamma_2(x) = \mathsf{num}_{\langle \mathbb{d}_2, * \rangle}\} \end{array} \quad c' = \begin{cases} c^\circ; c^\dagger & \text{if } pc = \bot \\ c^\circ & \text{if } pc = \top \end{cases}}{\Gamma_1, \Gamma_2, pc \Rightarrow c'}$$

PC update function

$$\mathsf{updatePC}(pc, \Gamma, e) = \begin{cases} \bot, & \text{if } pc = \bot \wedge \Gamma \vdash e : \mathsf{num}_{\langle -, 0 \rangle} \\ \top, & \text{else} \end{cases}$$

Figure 7: Rules for transforming probabilistic programs into deterministic ones with shadow execution extension. Differences that shadow execution introduce are marked in gray boxes.

Intuitively, a selector expression $\mathcal{S}$ with the following syntax decides if the aligned or shadow execution is picked:

$$\begin{array}{lllll} \text{Var Versions} & k & \in & \{\circ, \dagger\} \\ \text{Selectors} & \mathcal{S} & ::= & e\ ?\ \mathcal{S}_1 : \mathcal{S}_2 \mid k \end{array}$$

The definition of the selector template is then similar to the alignment template, where the value can depend on the branch conditions:

$$\mathcal{S}_{\mathbb{E}} ::= \begin{cases} e_0\ ?\ \mathcal{S}_{\mathbb{E}\setminus\{e_0\}} : \mathcal{S}_{\mathbb{E}\setminus\{e_0\}}, \text{when } \mathbb{E} = \{e_0, \cdots\} \\ \theta \text{ with fresh } \theta, \text{otherwise} \end{cases}$$

Compared with other holes (θ) in the alignment template ($\mathcal{A}_{\mathbb{E}}$), the only difference is that θ in $\mathcal{S}_{\mathbb{E}}$ has Boolean values representing whether to stay on aligned execution ($\circ$), or switch to shadow execution ($\dagger$).

To embed shadow execution into CheckDP, the type system dynamically instruments an auxiliary command (c_{d}) according to the selector template $\mathcal{S}$. Once a switch is made ($\mathcal{S} = \dagger$), the distances of all variables are replaced with their shadow versions by this command. Moreover, the privacy cost $\mathbf{v}_\epsilon$ will also be properly reset according to the selector.

B EXTRA CASE STUDIES

In this section we list the pseudo-code of the algorithms we evaluated in the paper for completeness. The incorrect part for the incorrect algorithms is marked with a box.

B.1 Report Noisy Max

Report Noisy Max [25]. This is an important building block for developing differentially private algorithms. It generates differentially private synthetic data by finding the identity with the maximum (noisy) score in the database. Here we present this mechanism in a simplified manner: for a series of query answers q, where each of them can differ at most one in the adjacent underlying database, its goal is to return the index of the maximum query answer in a privacy-preserving way. To achieve differential privacy, the mechanism first adds $\eta = \texttt{Lap}\ 2/\epsilon$ noise to each of the query answer, then returns the index of the maximum noisy query answers $\texttt{q[i]} + \eta$, instead of the true query answers q[i]. The pseudo code of this mechanism is shown in Figure 8.

To prove its correctness using randomness alignment technique, we need to align the only random variable η in the mechanism (Line 3). Therefore, a corresponding privacy cost of aligning η would be incurred for each iteration of the loop. However, manual proof [25] suggests that we only need to align the random variable added to the actual maximum query answer. In other words, we need an ability to "reset" the privacy cost upon seeing a new current maximum noisy query answer.

Bad Noisy Max. We also created an incorrect variant of Report Noisy Max. This variant directly returns the maximum noisy query answer, instead of the *index*.

More specifically, it can be obtained by changing Line 5 in Figure 8 from `max := i` to `max := q[i] + η`. CheckDP is then able to find a counterexample for this incorrect variant.

function NoisyMax (size : $\texttt{num}_{\langle 0,0\rangle}$, q : $\texttt{list num}_{\langle *,*\rangle}$)
returns max : $\texttt{num}_{\langle 0,-\rangle}$
precondition $\forall\, \texttt{i}.\ -1 \le \widehat{\texttt{q}}^{\circ}[\texttt{i}] \le 1 \wedge \widehat{\texttt{q}}^{\dagger}[\texttt{i}] = \widehat{\texttt{q}}^{\circ}[\texttt{i}]$

```
1  i := 0; bq := 0; max := 0;
2  while (i < size)
3    η := Lap (2/ε);
4    if (q[i] + η > bq ∨ i = 0)
5     max := i;
6     bq := q[i] + η;
7    i := i + 1;
```

function Transformed NoisyMax (size, q, $\widehat{\texttt{q}}$, *sample*, θ)
returns (max)

```
8   v_ε := 0; idx := 0;
9   i := 0; bq := 0; max := 0;
10  bq̂° := 0; bq̂† := 0; max̂° := 0; max̂† := 0;
11  while (i < size)
12    η := sample[idx]; v_ε := (S ? v_ε : 0) + |A| × ε/2;
13    η̂ := A;
14    if (S) bq̂° := bq̂†; max̂° := max̂†;
15    if (q[i] + η > bq ∨ i = 0)
16      assert(q[i] + q̂°[i] + η + η̂° > bq + bq̂° ∨ i = 0);
17      max := i;
18      max° := 0;
19      bq̂† := bq + bq̂† - (q[i] + η);
20      bq := q[i] + η;
21      bq̂° := q̂°[i] + η̂°;
22    else
23      assert(¬(q[i] + q̂°[i] + η + η̂° > bq + bq̂° ∨ i = 0));
24    // shadow execution
25    if (q[i] + q̂†[i] + η > bq + bq̂† ∨ i = 0)
26      bq̂† := q[i] + q̂†[i] + η − bq;
27      max̂† := i - max;
28    i := i + 1;
29  assert(max̂° = 0);
30  assert(v_ε ≤ ε);
```

Figure 8: Report Noisy Max and its transformed code, where $\mathcal{S} = \texttt{q[i]} + \eta > \texttt{bq} \vee \texttt{i = 0}\ ?\ \theta[0] : \theta[1]$ and $\mathcal{A} = \texttt{q[i]} + \eta > \texttt{bq} \vee \texttt{i = 0} ? \theta[2] + \theta[3] \times \widehat{\texttt{q}}^{\circ}\texttt{[i]} + \theta[4] \times \widehat{\texttt{bq}}^{\circ} : \theta[5] + \theta[6] \times \widehat{\texttt{q}}^{\circ}\texttt{[i]} + \theta[7] \times \widehat{\texttt{bq}}^{\circ}$

B.2 Variants of Sparse Vector Technique

SVT. We first show a correctly-implemented standard version of SVT [40]. This standard implementation is less powerful than running example GapSVT, as it outputs `true` instead of the gap between noisy query answer and noisy threshold. This can be obtained by changing Line 7 in Figure 1 from `out := (q[i] + η2)::out;` to `out := true::out;`.

SVT with Monotonic Queries. There exist use cases with SVT where the queries are monotonic. More formally, queries are monotonic if for related queries $q \sim q'$, $\forall i.\ q_i \le q'_i$ or $\forall i.\ q_i \ge q'_i$. As shown in [40]. When the queries are monotonic, it suffices to add η_2 `:= Lap` $2N/\epsilon$ to each queries (Line 5 in Figure 1) and the algorithm still satisfies ϵ-DP.

Thanks to the flexibility of CheckDP, it only requires one change in the function specification in order to verify this variant: modify the constraint on $\widehat{\texttt{q}}$[i] in the precondition.

function SVT (T, N, size : num$_0$, q : list num$_*$)
returns (out : list bool), **check**(ϵ)
precondition $\forall$ i. $-1 \le \widehat{\mathtt{q}}[\mathtt{i}] \le 1$

```
1  η1 := Lap (2/ε)
2  Tη := T + η1;
3  count := 0; i := 0;
4  while (count < N ∧ i < size)
5    η2 := Lap (4N/ε)
6    if (q[i] + η2 ≥ Tη) then
7      out := true::out;
8      count := count + 1;
9    else
10     out := false::out;
11   i := i + 1;
```

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

```
12 vε := 0; idx = 0;
13 η1 := sample[idx]; idx := idx + 1;
14 vε := vε + |A1| × ε/2; η̂1 := A1;
15 Tη := T + η1;
16 T̂η := η̂1;
17 count := 0; i := 0;
18 while (count < N ∧ i < size)
19   η2 := sample[idx]; idx := idx + 1;
20   vε := vε + |A2| × ε/4N; η̂2 := A2;
21   if (q[i] + η2 ≥ Tη) then
22     assert(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η);
23     out := true::out;
24     count := count + 1;
25   else
26     assert(¬(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η));
27     out := false::out;
28   i := i + 1;
29 assert(vε ≤ ε);
```

Figure 9: Standard Sparse Vector Technique and its transformed code, where underlined parts are added by CheckDP. The transformed code contains two alignment templates for η_1 and η_2: $\mathcal{A}_1 = \theta[0]$ and $\mathcal{A}_2 = (\mathbf{q[i]} + \eta_2\mathbf{[i]} \ge T_\eta)\ ?\ (\theta[1] + \theta[2] \times \widehat{T_\eta} + \theta[3] \times \widehat{\mathbf{q[i]}}) : (\theta[4] + \theta[5] \times T_\eta + \theta[6] \times \widehat{\mathbf{q[i]}})$.

Specifically, the new precondition for SVT with monotonic queries becomes $\forall\, \mathtt{i}.\, 0 \le \widehat{\mathtt{q}}[\mathtt{i}] \le 1$ for the $\forall i.\, q_i \le q_i'$ and $\forall\, \mathtt{i}. -1 \le \widehat{\mathtt{q}}[\mathtt{i}] \le 0$ for the other case. The final found alignment by CheckDP is the same as the ones reported in the manual randomness alignment based proofs [24]:

$$\eta_1 : 0 \quad \eta_2 : \begin{cases} \mathtt{q[i]} + \eta_2 \ge T_\eta\ ?\ 1 - \widehat{\mathtt{q}}[\mathtt{i}] : 0, & \text{if } \forall i.\ q_i \le q_i' \\ \mathtt{q[i]} + \eta_2 \ge T_\eta\ ?\ -\widehat{\mathtt{q}}[\mathtt{i}] : 0, & \text{otherwise} \end{cases}$$

To the best of our knowledge, no prior verification works have automatically verified this variant.

Adaptive SVT. As mentioned in Section 5, we list the pseudo code of Adaptive SVT in Figure 10.

function ADAPTIVESVT (T, N, size : num$_0$, q : list num$_*$)
returns (out : list num$_0$), **check**(ϵ)
precondition $\forall$ i. $-1 \le \widehat{\mathtt{q}}[\mathtt{i}] \le 1$

```
1  cost := 0;
2  η1 := Lap (2/ε);
3  cost := cost + ε/2;
4  Tη := T + η1;
5  i := 0;
6  while (cost ≤ ε - 2×ε/4N ∧ i < size)
7    η2 := Lap (8N/ε);
8    if (q[i] + η2 − Tη ≥ σ) then
9      out := (q[i] + η2 - Tη)::out;
10     cost := cost + 2×ε/(8N);
11   else
12     η3 := Lap (4N/ε);
13     if (q[i] + η3 − Tη ≥ 0) then
14       out := (q[i] + η3 - Tη)::out;
15       cost := cost + 2×ε/(4N);
16     else
17       out := 0::out;
18   i := i + 1;
```

function TRANSFORMED ADAPTIVESVT (T, N, size, q, $\widehat{\mathtt{q}}$, *sample*, θ)
returns (out)

```
12 vε := 0; idx = 0;
13 η1 := sample[idx]; idx := idx + 1;
14 vε := vε + |A1| × ε/2; η̂1 := A1;
15 Tη := T + η1;
16 T̂η := η̂1;
17 count := 0; i := 0;
18 while (cost ≤ ε - 2×ε/4N ∧ i < size)
19   η2 := sample[idx]; idx := idx + 1;
20   vε := vε + |A2| × ε/8N; η̂2 := A2;
21   if (q[i] + η2 - Tη ≥ σ) then
22     assert(q[i] + η2 + q̂[i] + η̂2 - (Tη + T̂η) ≥ σ);
23     assert(q̂[i] + η̂2 - T̂η == 0);
24     out := (q[i] + η2 - Tη)::out;
25     cost := cost + 2×ε/(8N);
26   else
27     assert(¬(q[i] + η2 + q̂[i] + η̂2 - (Tη + T̂η) ≥ σ));
28     η3 := sample[idx]; idx := idx + 1;
29     vε := vε + |A3| × ε/4N; η̂2 := A3;
30     if (q[i] + η3 - Tη ≥ 0)
31       assert(q[i] + η3 + q̂[i] + η̂3 - (Tη + T̂η ≥ 0) ;
32       assert(q̂[i] + η̂3 - T̂η == 0);
33       out := (q[i] + η3 - Tη)::out;
34       cost := cost + 2×ε/(4N);
35     else
36       assert(¬(q[i] + η3 + q̂[i] + η̂3 - (Tη + T̂η) ≥ 0));
37       out := false::out;
38   i := i + 1;
39 assert(vε ≤ ε);
```

Figure 10: Adaptive SVT and its transformed code, where underlined parts are added by CheckDP. The transformed code contains three alignment templates for η_1 and η_2: $\mathcal{A}_1 = \theta[0]$, $\mathcal{A}_2 = \Omega_{Top}\ ?\ (\theta[1] + \theta[2] \times \widehat{T_\eta} + \theta[3] \times \widehat{\mathbf{q[i]}}) : (\theta[4] + \theta[5] \times T_\eta + \theta[6] \times \widehat{\mathbf{q[i]}})$ and $\mathcal{A}_3 = \Omega_{Middle}\ ?\ (\theta[1] + \theta[2] \times \widehat{T_\eta} + \theta[3] \times \widehat{\mathbf{q[i]}}) : (\theta[4] + \theta[5] \times T_\eta + \theta[6] \times \widehat{\mathbf{q[i]}})$, where Ω_* denotes the corresponding branch condition at Line 8 and 13.

function NumSVT (T, N, size : num$_0$, q : list num$_*$)
returns (out : list bool), **check**(ϵ)
precondition $\forall$ i. $-1 \le \widehat{\text{q}}$[i] ≤ 1

```
1  η1 := Lap (3/ε)
2  Tη := T + η1;
3  count := 0; i := 0;
4  while (count < N ∧ i < size)
5    η2 := Lap (6N/ε)
6    if (q[i] + η2 ≥ Tη) then
7      η3 := Lap (3N/ε);
8      out := (q[i] + η3)::out;
9      count := count + 1;
10   else
11     out := false::out;
12   i := i + 1;
```

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

```
12 vε := 0; idx = 0;
13 η1 := sample[idx]; idx := idx + 1;
14 vε := vε + |A1| × ε/3; η̂1 := A1;
15 Tη := T + η1;
16 T̂η := η̂1;
17 count := 0; i := 0;
18 while (count < N ∧ i < size)
19   η2 := sample[idx]; idx := idx + 1;
20   vε := vε + |A2| × ε/6N; η̂2 := A2;
21   if (q[i] + η2 ≥ Tη) then
22     assert(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η);
23     η3 := sample[idx]; idx := idx + 1;
24     vε := vε + |A3| × ε/3N; η̂3 := A3;
25     assert(q̂[i] + η̂3 = 0);
26     out := (q[i] + η3)::out;
27     count := count + 1;
28   else
29     assert(¬(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η));
30     out := false::out;
31   i := i + 1;
32 assert(vε ≤ ε);
```

Figure 11: Numerical Sparse Vector Technique and its transformed code, where underlined parts are added by CheckDP. The transformed code contains three alignment templates for η_1, η_2 and η_3 respectively: $\mathcal{A}_1 = \theta[0]$, $\mathcal{A}_2 = (\mathbf{q[i]} + \eta_2\mathbf{[i]} \ge T_\eta)\ ?\ (\theta[1] + \theta[2] \times \widehat{T_\eta} + \theta[3] \times \widehat{\mathbf{q[i]}}) : (\theta[4] + \theta[5] \times T_\eta + \theta[6] \times \widehat{\mathbf{q[i]}})$, $\mathcal{A}_3 = \theta[7] + \theta[8] \times \widehat{T_\eta} + \theta[9] \times \widehat{\mathbf{q[i]}}$

NumSVT. Numerical Sparse Vector (NumSVT) [28] is another interesting correct variant of SVT which outputs a numerical answer when the input query is larger than the noisy threshold. It follows the same procedure as Sparse Vector Technique, the difference is that it draws a fresh noise η_3 in the `true` branch, and outputs $q[i] + \eta_3$ instead of `true`. Note that this is very similar to our running example GapSVT and BadGapSVT, the key difference is that the freshly-drawn random noise hides the information about T_η, unlike the BadGapSVT. This variant can be obtained by making the following changes in Figure 1: (1) Line 1 is changed from **Lap** $2/\epsilon$ to **Lap** $3/\epsilon$; (2) Line 5 is changed from **Lap** $4N/\epsilon$ to **Lap** $6N/\epsilon$; (3) Line 7 is change from `out := (q[i] + η)::out;` to "η_3 := **Lap** $(3N/\epsilon)$; `out := (q[i] + η3)::out;`". CheckDP finds the same alignment as shown in [52] with which CPAChecker is able to verify the algorithm with this generated alignment.

function BadSVT1 (T, N, size : num$_0$, q : list num$_*$)
returns (out : list bool), **check**(ϵ)
precondition $\forall$ i. $-1 \le \widehat{\text{q}}$[i] ≤ 1

```
1  η1 := Lap (2/ε);
2  Tη := T + η1;
3  count := 0; i := 0;
4  while ( i < size )
5    η2 := 0;
6    if (q[i] + η2 ≥ Tη) then
7      out := true::out;
8      count := count + 1;
9    else
10     out := false::out;
11   i := i + 1;
```

function Transformed BadSVT1 (T, N, size, q, $\widehat{\text{q}}$, *sample*, θ)
returns (out)

```
12 vε := 0; idx = 0;
13 η1 := sample[idx]; idx := idx + 1;
14 vε := vε + |A1| × ε/2; η̂1 := A1;
15 Tη := T + η1;
16 T̂η := η̂1;
17 count := 0; i := 0;
18 while (i < size)
19   η2 := 0;
20   if (q[i] + η2 ≥ Tη) then
21     assert(q[i] + η2 + q̂[i] ≥ Tη + T̂η);
22     out := true::out;
23     count := count + 1;
24   else
25     assert(¬(q[i] + η2 + q̂[i] ≥ Tη + T̂η));
26     out := false::out;
27   i := i + 1;
28 assert(vε ≤ ε);
```

Figure 12: BadSVT1 and its transformed code, where underlined parts are added by CheckDP. The transformed code contains a alignment template for η_1: $\mathcal{A}_1 = \theta[0]$.

BadSVT1 - 3. We now study other three incorrect variants of SVT collected from [40]. All three variants are based on the classic SVT algorithm we have seen (i.e., Line 7 in Figure 1 is `out := true::out;`).

BadSVT1 [47] adds no noise to the query answers and has no bounds on the number of `true`'s it can output. This variant is obtained by changing Line 4 from **while** (`count<N∧i<size`) to **while** (`i<size`) and Line 5 from **Lap** $4N/\epsilon$ to 0. Another variant BadSVT2 [20] has no bounds on the number of `true`'s it can output as well. It keeps outputting `true` even if the given privacy budget has been exhausted. Moreover, the noise added to the queries does not scale with parameter N. Specifically, based on BadSVT1, Line 5 is changed to **Lap** $2/\epsilon$. BadSVT3 [38] is an interesting case since it tries to spend its privacy budget in a different allocation strategy between the threshold T and the query answers `q[i]` ($1:3$ instead of $1:1$). However, the noise added to η_2 does not scale with parameter N. The 3/4 privacy budget is allocated to each of the queries where it should be shared among them. To get this variant, based on SVT

algorithm, the noise generation commands (Line 1 and Line 5) are changed to η_1 := **Lap** $4/\epsilon$ and η_2 := **Lap** $4/(3\times\epsilon)$, respectively.

function BadSVT2 (T, N, size : num$_0$, q : list num$_*$)
returns (out : list bool), **check**(ϵ)
precondition $\forall$ i. $-1 \le \widehat{\text{q}}$[i] ≤ 1

```
1  η1 := Lap (2/ε);
2  Tη := T + η1;
3  count := 0; i := 0;
4  while ( i < size )
5    η2 := Lap (2/ε);
6    if (q[i] + η2 ≥ Tη) then
7      out := true::out;
8      count := count + 1;
9    else
10     out := false::out;
11   i := i + 1;
```

function Transformed BadSVT2 (T, N, size, q, $\widehat{\text{q}}$, *sample*, θ)
returns (out)

```
12 vε := 0; idx := 0;
13 η1 := sample[idx]; idx := idx + 1;
14 vε := vε + |A1| × ε/2; η̂1 := A1;
15 Tη := T + η1;
16 T̂η := η̂1;
17 count := 0; i := 0;
18 while (i < size)
19   η2 := sample[idx]; idx := idx + 1;
20   vε := vε + |A2| × ε/2; η̂2 := A2;
21   if (q[i] + η2 ≥ Tη) then
22     assert(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η);
23     out := true::out;
24     count := count + 1;
25   else
26     assert(¬(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η));
27     out := false::out;
28   i := i + 1;
29 assert(vε ≤ ε);
```

Figure 13: BadSVT2 and its transformed code, where underlined parts are added by CheckDP. The transformed code contains two alignment templates for η_1 and η_2: $\mathcal{A}_1 = \theta[0]$ and $\mathcal{A}_2 = (\mathbf{q[i]} + \eta_2 \ge T_\eta)\ ?\ (\theta[1] + \theta[2]\times\widehat{T_\eta} + \theta[3]\times\widehat{\mathbf{q[i]}}) : (\theta[4] + \theta[5]\times T_\eta + \theta[6]\times\widehat{\mathbf{q[i]}})$.

Note that apart from BadSVT1, which does not sample η_2, the generated templates are identical to the GapSVT since they all have similar typing environments.

Interestingly, since the errors are very similar among them (no bounds on number of outputs / wrong scale of added noise), CheckDP finds a common counterexample $[0,0,0,0,0]$, $[1,1,1,1,-1]$ where $T = 0$ and $N = 1$ within 6 seconds, and this counterexample is further validated by PSI.

function BadSVT3 (T, N, size : num$_0$, q : list num$_*$)
returns (out : list bool), **check**(ϵ)
precondition $\forall$ i. $-1 \le \widehat{\text{q}}$[i] ≤ 1

```
1  η1 := Lap (4/ε);
2  Tη := T + η1;
3  count := 0; i := 0;
4  while (count < N ∧ i < size)
5    η2 := Lap (4/3ε);
6    if (q[i] + η2 ≥ Tη) then
7      out := true::out;
8      count := count + 1;
9    else
10     out := false::out;
11   i := i + 1;
```

function Transformed BadSVT3 (T, N, size, q, $\widehat{\text{q}}$, *sample*, θ)
returns (out)

```
12 vε := 0; idx := 0;
13 η1 := sample[idx]; idx := idx + 1;
14 vε := vε + |A1| × ε/4; η̂1 := A1;
15 Tη := T + η1;
16 T̂η := η̂1;
17 count := 0; i := 0;
18 while (count < N ∧ i < size)
19   η2 := sample[idx]; idx := idx + 1;
20   vε := vε + |A2| × 3ε/4; η̂2 := A2;
21   if (q[i] + η2 ≥ Tη) then
22     assert(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η);
23     out := true::out;
24     count := count + 1;
25   else
26     assert(¬(q[i] + η2 + q̂[i] + η̂2 ≥ Tη + T̂η));
27     out := false::out;
28   i := i + 1;
29 assert(vε ≤ ε);
```

Figure 14: BadSVT3 and its transformed code, where underlined parts are added by CheckDP. The transformed code contains two alignment templates for η_1 and η_2: $\mathcal{A}_1 = \theta[0]$ and $\mathcal{A}_2 = (\mathbf{q[i]} + \eta_2 \ge T_\eta)\ ?\ (\theta[1] + \theta[2]\times\widehat{T_\eta} + \theta[3]\times\widehat{\mathbf{q[i]}}) : (\theta[4] + \theta[5]\times T_\eta + \theta[6]\times\widehat{\mathbf{q[i]}})$.

B.3 Partial Sum

Next, we study a simple algorithm PartialSum (Figure 15) which outputs the sum of queries in a privacy-preserving manner: it directly computes sum of all queries and adds a **Lap** $1/\epsilon$ to the final output sum. Note that similar to SmartSum, it has the same adjacency requirement (only one query can differ by at most one). The alignment is easily found for η by CheckDP which is to "cancel out" the distance of sum variable (i.e., $-\widehat{\text{sum}}$). With the alignment CPAChecker verifies this algorithm.

An incorrect variant for PartialSum called BadPartialSum is created where Line 5 is changed from $1/\epsilon$ to $1/(2\times\epsilon)$, therefore making it fail to satisfy ϵ-differential privacy (though it actually satisfies 2ϵ-differential privacy). A counterexample $[0,0,0,0,0]$, $[0,0,0,0,1]$ is found by CheckDP and further validated by PSI.

function PartialSum (size : $\texttt{num}_0$, q : list $\texttt{num}_*$)
returns (out : $\texttt{num}_0$), **check**(ϵ)
precondition $\forall \texttt{i}.\ -1 \le \hat{\texttt{q}}[\texttt{i}] \le 1 \wedge (\forall \texttt{i}.\ (\hat{\texttt{q}}[\texttt{i}] \neq 0) \Rightarrow (\forall \texttt{j}.\ \hat{\texttt{q}}[\texttt{j}] = 0))$

```
1  sum := 0; i := 0;
2  while (i < size)
3    sum := sum + q[i];
4    i := i + 1;
5  η = Lap (1/ε);
6  out := sum + η;
```

function Transformed PartialSum (size, q, $\hat{\texttt{q}}$, *sample*, θ)
returns (out)

```
7  v_ε := 0; sum^ := 0;
8  sum := 0; i := 0;
9  while (i < size)
10   sum := sum + q[i];
11   sum^ := sum^ + q^[i];
12   i := i + 1;
13 v_ε := v_ε + |A| × ε; η^ := A;
14 assert(sum^ + η^ = 0);
15 out := sum + η;
16 assert(v_ε ≤ ε);
```

Figure 15: PartialSum and its transformation using CheckDP, where $\mathcal{A} = \theta[\mathbf{0}] + \theta[\mathbf{1}] \times \widehat{\mathbf{sum}} + \theta[\mathbf{2}] \times \hat{\mathbf{q}}[\mathbf{i}]$.

B.4 SmartSum and BadSmartSum

SmartSum [19] continually releases aggregated statistics with privacy protections. For a finite sequence of queries $q[0], q[1], \cdots, q[T]$, where T is the length of q, the goal of SmartSum is to release the prefix sum: $q[0], q[0] + q[1], \cdots, \sum_{i=0}^{T} q[i]$ in a private way. To achieve differential privacy, SmartSum first divides the sequence into non-overlapping blocks $B_0, \cdots, B_l$ with size M, then maintains the noisy version of each query and noisy version of the block sum, both by directly adding **Lap** $1/\epsilon$ noise. Then to compute the k^{th} component of the prefix sum sequence $\sum_{i=0}^{k} q[i]$, it only has to add up the noisy block sum that covers before k, plus the remaining $(k+1)$ mod M noisy queries. The pseudo code is shown in Figure 16. The **if** branch is responsible for dividing the queries and summing up the block sums (stored in sum variable), where **else** branch adds the remaining noisy queries.

Notably, SmartSum satisfies 2ϵ-differential privacy instead of ϵ-differential privacy. Moreover, the adjacency requirement of the inputs is that only one of the queries can differ by at most one. These two requirements are specified in the function signature (**check**(2ϵ) and **precondition**).

An incorrect variant of SmartSum, called BadSmartSum, is obtained by changing Line 4 to $\eta_1 := 0$ in Figure 16. It directly releases sum + q[i] without adding any noise (since $\eta_1 = 0$), where sum stores the accurate, non-noisy sum of queries (at Line 11), hence breaking differential privacy. Interestingly, the violation only happens in a rare branch **if** ((i + 1) mod M = 0), where the accurate sum is added to the output list out. In other words, out contains mostly private data with only a few exceptions. This rare event makes it challenging for sampling-based tools to find the violation.

function SmartSum (M, T, size : $\texttt{num}_0$, q : list $\texttt{num}_*$)
returns (out : list $\texttt{num}_0$), **check**(2ϵ)
precondition $\forall \texttt{i}.\ -1 \le \hat{\texttt{q}}[\texttt{i}] \le 1 \wedge (\forall \texttt{i}.\ (\hat{\texttt{q}}[\texttt{i}] \neq 0) \Rightarrow (\forall \texttt{j}.\ \hat{\texttt{q}}[\texttt{j}] = 0))$

```
1  next := 0; i := 0; sum := 0;
2  while (i < size ∧ i ≤ T)
3    if ((i + 1) mod M = 0) then
4      η1 := Lap (1/ε);
5      next := sum + q[i] + η1;
6      sum := 0;
7      out := next::out;
8    else
9      η2 := Lap (1/ε);
10     next:= next + q[i] + η2;
11     sum := sum + q[i];
12     out := next::out;
13   i := i + 1;
```

function Transformed SmartSum (M, T, size, q, $\hat{\texttt{q}}$, *sample*, θ)
returns (out)

```
14 v_ε := 0; idx := 0;
15 next := 0; i := 0; sum := 0;
16 sum^ := 0; next^ := 0;
17 while (i < size ∧ i ≤ T)
18   if ((i + 1) mod M = 0) then
19     η1 := sample[idx]; idx := idx + 1;
20     v_ε := v_ε + |A1| × ε; η1^ := A1;
21     next := sum + q[i] + η1;
22     next^ := sum^ + q^[i] + η1^;
23     sum := 0;
24     sum^ := 0;
25     assert(next^ = 0);
26     out := next::out;
27   else
28     η2 := sample[idx]; idx := idx + 1;
29     v_ε := v_ε + |A2| × ε; η2^ := A2;
30     next := next + q[i] + η2;
31     next^ := next^ + q^[i] + η2^;
32     sum := sum + q[i];
33     sum^ := sum^ + q^[i];
34     assert(next^ = 0);
35     out := next::out;
36 i := i + 1;
37 assert(v_ε ≤ 2ε);
```

Figure 16: SmartSum and its transformed code. Underlined parts are added by CheckDP. $\mathcal{A}_1 = \theta[0] + \theta[1] \times \widehat{\mathbf{sum}} + \theta[2] \times \hat{\mathbf{q}}[\mathbf{i}] + \theta[3] \times \widehat{\mathbf{next}}$ and $\mathcal{A}_2 = \theta[4] + \theta[5] \times \widehat{\mathbf{sum}} + \theta[6] \times \hat{\mathbf{q}}[\mathbf{i}] + \theta[7] \times \widehat{\mathbf{next}}$.