Learning Differentially Private Mechanisms

Subhajit Roy[1]
Computer Science and Engineering
Indian Institute of Technology Kanpur
subhajit@iitk.ac.in

Justin Hsu
Department of Computer Sciences
University of Wisconsin–Madison
justhsu@cs.wisc.edu

Aws Albarghouthi[2]
Department of Computer Sciences
University of Wisconsin–Madison
aws@cs.wisc.edu

***Abstract*—Differential privacy is a formal, mathematical definition of data privacy that has gained traction in academia, industry, and government. The task of correctly constructing differentially private algorithms is non-trivial, and mistakes have been made in foundational algorithms. Currently, there is no automated support for converting an existing, non-private program into a differentially private version. In this paper, we propose a technique for automatically learning an accurate and differentially private version of a given non-private program. We show how to solve this difficult program synthesis problem via a combination of techniques: carefully picking representative example inputs, reducing the problem to continuous optimization, and mapping the results back to symbolic expressions. We demonstrate that our approach is able to learn foundational algorithms from the differential privacy literature and significantly outperforms natural program synthesis baselines.**

I. Introduction

Today, private data from individuals is commonly aggregated in large databases maintained by companies or governmental organizations. There is clear value in using this data to learn information about a general population, but there are also serious privacy concerns—seemingly innocuous data analyses can unintentionally leak highly sensitive information about specific individuals [1], [2].

To address these concerns, differential privacy [3] is a rigorous, mathematical definition of data privacy that has attracted a flurry of interest across academia, industry, and government. Intuitively, differential privacy defines privacy as a kind of *sensitivity* property: given a program that takes a private dataset as input, adding or removing an individual's data should only have a small effect on the program's output distribution. While differentially private programs were originally designed by theoreticians to solve specific statistical tasks, programmers in many areas are now looking to use differentially-private programs for their application domain [4], [5].

However, applying differential privacy is far from easy. First, to satisfy the mathematical guarantee, differentially private programs must be carefully crafted to add probabilistic noise at certain key points. The task of correctly constructing differentially private algorithms is non-trivial, and mistakes have been discovered in commonly used algorithms [6]. Second, although there are numerous software packages that provide private building-blocks and methods for safely combining components while respecting the privacy guarantee [7], [8], existing frameworks for differential privacy are intended for writing new private programs from scratch: practitioners who have domain-specific code developed without considering privacy cannot automatically convert their code to differentially private programs. Instead, programmers must *reimplement* their code, manually figure out where to add random noise—while keeping in mind that suboptimal choices may trivially achieve privacy by adding so much noise as to ruin accuracy—and then *prove* that the resulting algorithm is indeed differentially private. For many applications, the programmer burden is simply too high for differential privacy to be a realistic approach.

Mechanism synthesis problem. In this paper, we are interested in *automatically synthesizing a differentially private program* from an existing, non-private program. More specifically, we pose the question as follows:

> Given a non-private algorithm M, can we generate an algorithm $M^\star$ that is differentially private and "accurate" with respect to M?

This problem is difficult for several reasons: (1) The space of algorithms is infinite and discrete, and it is not clear how to search through it to find $M^\star$. Changing the amount of noise added at some program locations may sharply change the privacy level of the whole algorithm, while adjusting the noise level at other program locations may have no effect on privacy. (2) Given a candidate $M^\star$ that is not differentially private, it may be difficult to find a counterexample; given a counterexample, it is not obvious how to make the candidate more private. (3) While a testing/verification tool is only concerned with proving or disproving differential privacy, a synthesis method has the additional goal of finding a mechanism that adds as little noise as possible to achieve the target privacy level.

To address these problems, we present a set of novel contributions to synthesize *pure* differentially private (i.e., ϵ-differentially private) mechanisms.

Our approach. First, we restrict the search space for $M^\star$ to variants of M with noise added to some selected program expressions. These expressions can be selected by a domain expert and provided to our algorithm as a *mechanism sketch*. The question now becomes: *how much noise should we add?* In general, the amount of noise may need to be a function of the algorithm's inputs and ϵ, the privacy parameter.

To search the space of noise functions, we employ ideas from *inductive program synthesis* [9], [10], where candidate pro-

[1] The work was primarily done during his visit to UW–Madison.

[2] Author's name in native alphabet: أوس البرغوثي

grams are proposed and then refined through counterexamples generated by a testing tool; we rely on a state-of-the-art tool for detecting violations of differential privacy, called StatDP [11]. However, a naïve search strategy runs very slowly, as the testing tool can take a long time to discover counterexamples to privacy for every candidate we provide. To make the search process more efficient, we demonstrate how to set up a continuous optimization problem that allows us to hone in on the most-likely candidates, eliminating unlikely candidates from the search space without calling the testing tool. Our optimization problem also guides our search towards noise parameters such that the privacy guarantee is *tight*, i.e., our synthesis procedure aims to find programs that are ϵ-differentially private, but not ϵ'-differentially private for any smaller ϵ'. This strategy ensures that there are no obvious places where too much noise is being added, and enables our tool to find private algorithms that have been proposed in the privacy literature, as well as *new variants*.

Evaluation. We have implemented and applied our approach to synthesize a range of ϵ-differentially private algorithms. Our results show that (1) our approach is able to discover sophisticated algorithms from the literature, including the sparse vector technique (SVT), and (2) our continuous optimization-guided search improves performance significantly over simpler approaches.

Contributions. We offer the following technical contributions.

- We present a sketch-based methodology to construct an ϵ-differentially private mechanism M^* from a given non-private program M.
- Our technique combines a number of novel ideas: (a) bootstrapping the learning process with a number of carefully selected examples, (b) solving an approximate continuous optimization variant of the problem to guide an enumerative program-synthesis approach, and (c) using the privacy loss to help rank the proposed programs, treating it as a proxy for the tightness of the privacy guarantee.
- We implement our approach in a tool called Kolahal,[1] and evaluate it on synthesizing several foundational algorithms from the differential privacy literature, e.g., the *sparse vector technique* (SVT). We compare our approach to a series of successively stronger baseline procedures, demonstrating the importance of our algorithmic choices.

Taken together, our work is the first to automatically synthesize complex differentially private mechanisms.

Limitations. While there are now many known approaches for verifying differential privacy, existing methods are too slow to be used in our counterexample-guided synthesis loop. Hence, our synthesis procedure leverages an efficient, counterexample generation tool to check if candidates are not private. Note that this tool is *unsound for verification*: failure to find a counterexample does not prove differential privacy. Instead, after our tool produces a ranked list of candidates, each candidate must be analyzed by a sound verifier as a final check.

[1] "Kolahal" (कोलाहल) is a Hindi word meaning *loud noise*.

Secondly, our proposal only attempts to synthesize suitable noise expressions in mechanism *sketches*—our algorithm does not look to transform the sketch. Finally, we have only investigated our method for pure, $(\epsilon, 0)$-privacy, not variants such as (ϵ, δ)-privacy, or Rényi differential privacy. Please refer to Section VII for details.

II. Background

We begin by introducing key definitions and existing algorithmic tools that we will use in our synthesis procedure. Readers interested in a more thorough presentation of differential privacy should consult the textbook by Dwork and Roth [12].

A. Differential privacy

Differential privacy [3] is a quantitative definition of privacy for programs that take private information as input, and produce a randomized answer as output.

Mechanism. A *mechanism* (or program) M takes as input a database d of private information, a *privacy parameter* $\epsilon \in \mathbb{R}_{>0}$, and potentially other inputs (e.g., queries to be answered on the private database), then returns a noisy output of type T.

Neighboring (or adjacent) databases. We shall assume a relation Δ over pairs of databases. If $(d_1, d_2) \in \Delta$, then we say that d_1 and d_2 are *neighboring* (or *adjacent*) databases. Intuitively, Δ relates pairs of databases that differ in a single individual's private data. For instance, Δ might relate one database to a neighboring database where one individual's record has been added, removed, or modified. We assume that this relation is provided as part of the input specification.

Privacy loss. For any pair of databases $(d_1, d_2) \in \Delta$, privacy parameter $\epsilon > 0$, and event $E \subseteq T$, we define the *privacy loss* $L(M, d_1, d_2, \epsilon, E)$ to be:

$$\max\left(\frac{\mathbb{P}[M(d_1,\epsilon) \in E]}{\mathbb{P}[M(d_2,\epsilon) \in E]}, \frac{\mathbb{P}[M(d_2,\epsilon) \in E]}{\mathbb{P}[M(d_1,\epsilon) \in E]}\right)$$

Pure differential privacy. Using our definitions, *ϵ-differential privacy* (DP)—also known as pure differential privacy [3]—is defined as follows: M is ϵ-DP iff for all $(d_1, d_2) \in \Delta$, $\epsilon > 0$, and event $E \subseteq T$, the privacy loss is upper bounded by e^ϵ:

$$L(M, d_1, d_2, \epsilon, E) \leqslant e^\epsilon$$

Accordingly, a *counterexample* to ϵ-DP is a tuple $\langle d_1, d_2, \epsilon, E\rangle$ such that $(d_1, d_2) \in \Delta$, $\epsilon > 0$, and $L(M, d_1, d_2, \epsilon, E) > e^\epsilon$.

B. Programming language: syntax and semantics

We will work with an imperative language with a random sampling command, described in Fig. 1. The syntax and semantics are largely standard; readers who are more interested in the synthesis procedure can safely skip ahead.

The syntax is summarized in Fig. 1a. Here, Var is a countable set of program variables, and Exp is a set of program expressions. Expressions may be boolean- or integer-valued; we implicitly assume that all expressions are well-typed. Besides the usual commands in an imperative language, there are two

$$
\begin{aligned}
&\text{Var } x ::= a \mid b \mid \cdots \\
&\text{Exp } e ::= x \mid \mathsf{true} \mid \mathsf{false} \mid e = e' \mid e < e' \mid e > e' \mid \neg e \\
&\qquad \mid \mathbb{Z} \mid e + e' \mid e \cdot e' \mid e / e' \\
&\text{Com } c ::= \mathsf{skip} \mid x \leftarrow e \mid x \leftarrow \mathsf{Lap}(e)(e') \\
&\qquad \mid c; c' \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c' \mid \mathsf{while}\ e\ \mathsf{do}\ c
\end{aligned}
$$

(a) Language Syntax

$$
\begin{aligned}
[\![\mathsf{skip}]\!]m &\triangleq \mathsf{unit}(m) \\
[\![x \leftarrow e]\!]m &\triangleq \mathsf{unit}(m[x \mapsto [\![e]\!]m]) \\
[\![x \leftarrow e + \mathsf{Lap}(e')]\!]m &\triangleq \mathsf{bind}(\mathcal{L}_{[\![e']\!]m}([\![e]\!]m), v \mapsto m[x \mapsto v]) \\
[\![c; c']\!]m &\triangleq \mathsf{bind}([\![c]\!]m, m' \mapsto [\![c']\!]m') \\
[\![\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']\!]m &\triangleq \begin{cases} [\![c]\!]m & : \text{if } [\![e]\!]m = \mathsf{true} \\ [\![c']\!]m & : \text{if } [\![e]\!]m = \mathsf{false} \end{cases} \\
[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]m &\triangleq \lim_{n \to \infty} [\![(\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{skip})^n]\!]m
\end{aligned}
$$

(b) Language Semantics

Fig. 1: Programming Language

constructs that might be less familiar: skip is the do-nothing command, and $x \leftarrow e + \mathsf{Lap}(e')$ draws a random sample from the Laplace distribution with mean e and scale e', and stores the result in variable x; this commands is one of the building blocks for differentially private algorithms. Throughout, we will use the abbreviation:

$$\mathsf{if}\ e\ \mathsf{then}\ c \triangleq \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{skip}$$

Our semantics for commands, summarized in Fig. 1b, is also largely standard. The program state is modeled by a memory $m \in \mathsf{Mem}$, where Mem is the set of maps from variables in Var to values. We model commands c as functions $[\![c]\!] : \mathsf{Mem} \to \mathsf{D}(\mathsf{Mem})$, taking an input memory to a distribution over output memories, where a distribution $\mu \in \mathsf{D}(\mathsf{Mem})$ is a map $\mu : \mathsf{Mem} \to [0,1]$ such that μ has countable support (i.e., there are countably many $m \in \mathsf{Mem}$ such that $\mu(m) \neq 0$) and the weights in μ sum up to 1.

The formal semantics in Fig. 1b uses two operations on distributions. First, given $a \in A$, the *Dirac distribution* $\mathsf{unit}(a) \in \mathsf{D}(A)$ is defined via

$$\begin{cases} \mathsf{unit}(a)(a') \triangleq 1 & : a = a' \\ \mathsf{unit}(a)(a') \triangleq 0 & : a \neq a'. \end{cases}$$

That is, unit is simply the point mass distribution. Second, given $\mu \in \mathsf{D}(A)$ and $f : A \to \mathsf{D}(B)$, the *distribution bind* $\mathsf{bind}(\mu, f) \in \mathsf{D}(B)$ is defined via

$$\mathsf{bind}(\mu, f)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b).$$

Intuitively, bind sequences a distribution together with a continuation, leading to a single distribution on outputs.

We make two brief remarks about the semantics. First, the semantics of sampling involves the (discrete) Laplace distribution $\mathcal{L}_b(z)$, where $z \in \mathbb{Z}$ is the *mean* of the distribution and $b \in \mathbb{R}$ is a positive *scale* parameter. The distribution $\mathcal{L}_b(z) \in \mathsf{D}(\mathbb{Z})$ is defined as follows:[2]

$$\mathcal{L}_b(z)(z') \triangleq \frac{\exp(-|z - z'|/b)}{\sum_{y \in \mathbb{Z}} \exp(-|z - y|/b)}$$

[2]The standard Laplace distribution has support over the real numbers; we take the discretized version to simplify the technical development.

Intuitively, the scale parameter b controls how broadly spread the distribution is—larger values of b lead to a more spread out distribution. Second, the semantics of loops is well-defined provided that the loop terminates with probability 1 on any input. Since this property holds for all private (and non-private) programs of interest, we will assume this throughout the paper.

C. Testing for differential privacy

At a high level, our synthesis procedure iteratively tries different settings of the unknown noise parameters. Initial candidates will almost certainly fail to satisfy differential privacy. To make progress, our procedure leverages StatDP [11], a counterexample generation tool for differential privacy. Given a mechanism M and a target privacy level ϵ_0, StatDP constructs a set of candidate counterexamples $\{(d_1, d_2, E)\}$ that may witness a differential privacy violation; here d_1, d_2 are adjacent databases and E is a subset of outputs that is much more likely on one database than on the other.

Since our synthesis procedure leverages more specialized information provided by StatDP, we briefly describe how StatDP operates. StatDP uses a set of patterns to generate test databases (d_1, d_2), and a set of heuristics to construct test events E. For each test, StatDP estimates the probabilities $\tilde{\rho_1}, \tilde{\rho_2}$ of the output being in E starting from inputs d_1 and d_2 respectively, by repeatedly running the given mechanism. Then, it runs a *hypothesis test* (Fisher's exact test) to decide how likely the true probabilities ρ_1 and ρ_2 are to satisfy the guarantee that the program is differential private at some *test* ϵ values in the neighborhood of target level of privacy ϵ_0. For instance, if $\epsilon_0 = 0.5$, StatDP will test whether the mechanism is ϵ-DP for $\epsilon \in \{0.4, 0.5, 0.6\}$. If the hypothesis test indicates that ρ_1 and ρ_2 are highly unlikely to satisfy the privacy guarantee, then StatDP returns (d_1, d_2, E) as a candidate counterexample. Along with this tuple, StatDP also reports the *p-value* of the statistical test, a number in $[0, 1]$ measuring the confidence: a small p-value indicates that the candidate is likely to be a true counterexample to ϵ-differential privacy, while a large p-value indicates that the candidate is unlikely to be a true counterexample.

Figure 2 shows the p-values produced by a run of StatDP on a particular 0.5-differentially private mechanism M, checking

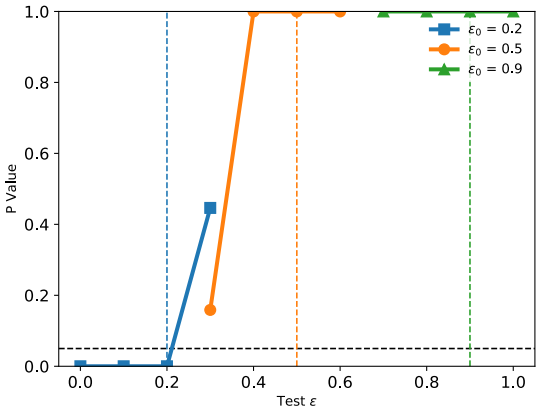


Fig. 2: StatDP at 0.2, 0.5, and 0.9

against target privacy levels ϵ_0 of 0.2, 0.5 and 0.9. Since M is 0.5-differentially private, it is automatically also 0.9-differentially private; however, M is not necessarily 0.2-differentially private. The candidate counterexamples returned by StatDP reflect this description. For small test ϵ, StatDP produces a candidate counterexample with a p-value close to zero, meaning that the candidate is highly likely to be a counterexample to ϵ-DP. For large test ϵ, StatDP still produces a candidate counterexample but with a p-value close to one—this indicates that the candidate is unlikely to be a true counterexample to ϵ-DP. Near the true privacy level ($\epsilon \in [0.2, 0.4]$), the reported p-value is in the middle of the range $[0, 1]$—StatDP is uncertain whether it has found a true counterexample or not.

Our synthesis approach crucially relies on the p-value reported by StatDP to judge the difficulty of the candidate counterexamples. Difficult cases—ones where p is in the middle of the range $[0, 1]$—are used as examples to improve the privacy of the synthesized mechanism. We will discuss this heuristic further in Section IV, but intuitively, challenging counterexamples to a mechanism represent pairs of inputs and sets of outputs where the privacy guarantee is nearly *tight*: lowering the target privacy level ϵ_0 would cause these challenging counterexamples to turn into true counterexamples to ϵ_0-privacy. As a result, challenging counterexamples witness the fact that a particular mechanism is *almost exactly* ϵ_0-private, and not more private than desired. Mechanisms without challenging counterexamples are likely more private than necessary, adding more noise than is needed.

We survey other approaches for testing and verification of differential privacy in related work (Section VI).

III. Overview

A. An illustrative example

To better illustrate the goal of our algorithm, consider the three programs in Fig. 3. The first program, Fig. 3a, takes in a database d of private information, a list of numeric queries $\boldsymbol{q}$, and a numeric threshold T, and checks if any query has answer (*ans*) above the threshold (T) when evaluated on the database. We suppose that this program is written by someone without taking privacy into account—the program adds no noise, and it does not satisfy differential privacy.

Our aim is to *automatically convert* this algorithm into a differentially-private version. To start with, we create a *sketch* of the program, Fig. 3b, where the desired privacy level ϵ is given as an input, and where certain assignment instructions are marked to add noise from the *Laplace distribution*, a standard numerical distribution used in differential privacy. These program locations can be identified by a domain expert, or they can simply be taken to be every assignment instruction. However, selecting *where* to add noise is not enough—the Laplace distribution requires a *scale parameter* describing the amount of noise that is added. Setting parameters incorrectly will lead to an program that is not ϵ-differentially private, or a program that adds too much noise. These unknown parameters are indicated by boxes in Fig. 3b; note that the second location adds a *vector* of independent Laplace draws with the same, unknown scale parameter to the vector $\boldsymbol{q}(d)$.

Then, our algorithm searches for a *symbolic expression* for each of the boxed locations so that the resulting program is ϵ-differentially private. One of the possible solutions from our algorithm is shown in Fig. 3c. Note that our procedure sets the scale parameter of the last location η_3 to $\perp$, indicating that this location does not need any noise. The completed program—generated by our approach—is known as the *Above Threshold* algorithm in the differential privacy literature [12]. Along with this version, our algorithm also finds other differentially private variants; e.g., a version where the first two noise locations add noise with scale $3/\epsilon$.

This example shows some of the challenges in converting non-private programs to private versions. First, the target noise parameters are not just constants: they are often *symbolic expressions*. For instance, the parameters for aboveT depend on the symbolic privacy parameter ϵ, and in general, parameters can depend on other arguments (e.g., the size of an input list). Second, not all locations need noise. For example, a different way of completing aboveT would also add noise at the last location, when assigning ans. While this version would also satisfy ϵ-differential privacy, it is inferior to the version in Fig. 3c since it adds noise unnecessarily.

B. The Mechanism-Synthesis problem

We formalize the overall problem as a synthesis problem.

Mechanism sketch. A *mechanism sketch* $M^\bullet$ is an incomplete program with *holes* at certain program locations; these holes represent unknown noise parameters that must be synthesized in order to produce a differentially private program. The sketch also specifies (i) the name of the input variable holding the private input, (ii) an adjacency relation on private inputs, and (iii) the names of non-private, auxiliary inputs, which we call *arguments*. For aboveT, for example, the private input is d, while the list of queries ($\boldsymbol{q}$) and the threshold (T) appear as the arguments. Pairs of databases d, d' where $\boldsymbol{q}(d)$ and $\boldsymbol{q}(d')$ differ by at most 1 in each coordinate are adjacent.

```
aboveT(d, q, T) :
  i ← 1;
  done ← false;
  t ← T;
  a ← q(d);
  while i ⩽ |q| ∧ ¬done do :
    if a_i > t then
      done ← true;
    i ← i + 1;
  if done
    ans ← i − 1;
  else
    ans ← 0;
  out ← ans;
  return out;
```

(a) Original program

```
aboveT•(d, q, T, ε) :
  i ← 1;
  done ← false;
  t ← T + Lap(□η1);
  a ← q(d) + ⟶Lap(□η2);
  while i ⩽ |q| ∧ ¬done do :
    if a_i > t then
      done ← true;
    i ← i + 1;
  if done
    ans ← i − 1;
  else
    ans ← 0;
  out ← ans + Lap(□η3);
  return out;
```

(b) Input: Sketch program

```
aboveT⋆(d, q, T, ε) :
  i ← 1;
  done ← false;
  t ← T + Lap(2/ε);
  a ← q(d) + ⟶Lap(4/ε);
  while i ⩽ |q| ∧ ¬done do :
    if a_i > t then
      done ← true;
    i ← i + 1;
  if done
    ans ← i − 1;
  else
    ans ← 0;
  out ← ans + Lap(⊥);
  return out;
```

(c) Output: Private program

Fig. 3: From a non-private sketch to a private program.

To complete the mechanism, we need to replace each hole with a well-typed expression constructed from the program inputs. We will use $\boldsymbol{\eta}$ to denote a vector of expressions, and $M^{\boldsymbol{\eta}}$ to denote the completion of sketch $M^\bullet$ with the expressions in $\boldsymbol{\eta}$. Recall Fig. 3b for an example of a sketch, where $\boldsymbol{\eta}$ is a vector of length 3.

Given a sketch $M^\bullet$, there are infinitely many ways of completing the sketch. To make this problem more tractable, we restrict the space of expressions using a finite grammar G whose elements we can enumerate.

A proxy for accuracy. Even after restricting the possible expressions, there are often many possible solutions giving ϵ-DP mechanisms; for instance, it is usually quite easy to construct a ϵ-DP mechanism by selecting an enormous noise parameter at every location. However, a solution that adds too much noise is less accurate and less useful. Since directly estimating a concrete mechanism's accuracy is challenging—for instance, it is often not clear how accuracy should be defined, and inputs that lead to inaccurate results may be hard to find—we will use *privacy loss* as a proxy for accuracy. Intuitively, we want to find a mechanism with a privacy loss that is *exactly* equal to e^ϵ—such a mechanism satisfies ϵ-DP tightly, in the sense that it adds just enough noise to satisfy differential privacy for the target level of ϵ. Mechanisms with privacy loss below e^ϵ satisfy ϵ'-DP for $\epsilon' < \epsilon$, a stronger guarantee that requires adding more noise.

We will say that an ϵ-DP mechanism M *dominates* M', denoted $M' \sqsubseteq M$, iff for all pairs of databases $(d_1, d_2) \in \Delta$, privacy parameters ϵ, and events E, we have

$$e^\epsilon \geqslant L(M, d_1, d_2, \epsilon, E) \geqslant L(M', d_1, d_2, \epsilon, E).$$

Note that $\sqsubseteq$ is a partial order, as some mechanisms are incomparable. We are now ready to define our mechanism synthesis problem.

Definition 3.1 (Problem statement): Given a sketch $M^\bullet$, an optimal solution to the *synthesis problem* is a vector of expressions $\boldsymbol{\eta}$ such that given any privacy parameter $\epsilon > 0$, the completed mechanism $M^{\boldsymbol{\eta}}$ is

1) ϵ-differentially private, and
2) a maximal mechanism per the ordering $\sqsubseteq$.

The first point ensures that the mechanism is differentially private, while the second point ensures that we cannot lower the amount of noise while still meeting the target differential privacy guarantee. While it is not feasible to certify that the second condition holds, the ordering helps guide our search towards more accurate private mechanisms.

IV. Mechanism Synthesis Algorithm

In this section, we present our technique for synthesizing optimal mechanisms. A simplistic strategy to synthesizing the mechanism would be to leverage an enumerative synthesis strategy that proposes expressions from a grammar G, and uses a verifier to accept or reject solutions. However, there are several interrelated challenges:

Challenge 1: Infinitely many inputs and output events. Even with a finite grammar G, finding an optimal solution $M^{\boldsymbol{\eta}}$ requires showing that it is differentially private and more accurate than (or incomparable to) all other mechanisms $M^{\boldsymbol{\eta}'}$ that are differentially private. This is challenging due to the universal quantifier over neighboring databases, privacy parameter, and events. *We solve this challenge by approximating the universal quantifier with a finite number of carefully chosen neighboring databases, program arguments, and events.*

Challenge 2: Expensive search over noise expression. Even if we have fixed an input, checking every possible expression to see if it satisfies ϵ-DP on those inputs is an expensive hypothesis testing process. *To reduce the cost, we adopt a*

two-phase approach: we first approximate the search problem using a fast continuous optimization procedure where we solve for constant instantiations of the noise values, then search for symbolic expressions that are close to these constants.

Challenge 3: Achieving privacy while limiting noise. For a given program sketch and a given level of ϵ, there are many possible ways of adding noise so that the program is ϵ-DP; for instance, privacy can usually be ensured by adding a large amount of noise at every location. However, adding too much noise reduces the accuracy of the algorithm. *To guide our search towards better private algorithms, our optimization objective takes the tightness of the privacy guarantee and the sparsity of the noise parameters into account.*

A. High-level mechanism synthesis algorithm

Our high-level algorithm proceeds in three steps, as shown in Figure 4. First, we fix all of the mechanism's inputs (γ) besides the private database to concrete values using fixParams; these initial arguments can be drawn from a fixed set of default values, or supplied by the user. The resulting sketch $\gamma(M^\bullet)$ has only one input—the private database—but it still has holes for unknown noise expressions. Since we are searching for target expressions that are formed from the mechanism's non-private inputs, concretizing these inputs means that we can search for concrete instantiations—real numbers—for each hole. This transforms the more challenging expression search problem into a simpler (but still challenging) numerical search problem.

Next, we generate a set of *challenging examples* for $\gamma(M^\bullet)$ using selectExamples, such that ensuring ϵ-differential privacy for these examples is likely to ensure ϵ-differential privacy *for all inputs*. An example is a pair of neighboring databases and an output event. These challenging examples are generated by repeatedly concretizing the program holes with real numbers, and searching for counterexamples Ex for $\gamma(M^\bullet)$ to ϵ-DP on StatDP. Then, for the challenging examples, our algorithm searches for a setting of hole completions so that ϵ-DP holds with the smallest loss to accuracy. To do so, we construct an optimization problem over possible concrete values of the noise expressions that aims to make the least-private example in Ex as tight as possible. By approximately solving this optimization problem (in the procedure getNoiseRegion), we can extract a region (R) of possible noise values ensuring ϵ-differential privacy on the given examples.

Finally, we employ an *enumerative synthesis loop* (findExpr) to *generalize* the concrete instantiations of the noise parameters into a ranked list of candidate completions $\boldsymbol{\eta}_1, \ldots, \boldsymbol{\eta}_n$—vectors of symbolic expressions—that give ϵ-differentially private mechanisms for the arguments γ as well as other arguments γ'. Expressions whose concrete values on γ do not belong to the region R can be pruned immediately, without testing differential privacy. This allows us to *focus* the expensive task of testing differential privacy to a few selected symbolic expressions whose concretizations lie in this noise region R.

We now describe each step in detail. We will use the program aboveT from Section III as our running example; Fig. 3b shows a possible sketch, and Fig. 3c shows a completion of the program satisfying differential privacy.

$\mathsf{synth}(M^\bullet, G)$:
  $\gamma \leftarrow \mathsf{fixParams}(M^\bullet)$
  $Ex \leftarrow \mathsf{selectExamples}(\gamma(M^\bullet))$
  $R \leftarrow \mathsf{getNoiseRegion}(\gamma(M^\bullet), Ex)$
  $\boldsymbol{\eta}_1, \ldots, \boldsymbol{\eta}_n \leftarrow \mathsf{findExpr}(M^\bullet, G, R, Ex)$

Fig. 4: High-level algorithm synth

B. Fixing arguments and selecting examples

Differentially-private mechanisms often take inputs besides the private database; we call such auxiliary inputs *arguments*. We first fix all arguments to be some initial values γ, so that the only input of the sketch $\gamma(M^\bullet)$ is the private database. We assume that sketches take the target level of privacy ϵ as a parameter, so γ also fixes this variable to some concrete number $\gamma(\epsilon)$. (Our tool, described in Section V, actually uses multiple settings of γ that helps when synthesizing symbolic expressions; for simplicity, we will present our core algorithm using a single setting of γ.)

There are several reasonable ways to choose γ. If representative inputs are available—perhaps from the original, non-private program—these inputs are natural choices for γ. Otherwise, we can leverage tools capable of producing counterexamples to differential privacy; these tools produce settings for all inputs to the program, including the non-private arguments. For instance, the StatDP tool [11] uses a combination of symbolic execution and statistical testing to find counterexample inputs.

Next, we find a set of examples to bootstrap our synthesis algorithm. More precisely, an example is a tuple $\langle d_1, d_2, E\rangle$ consisting of a pair of neighboring databases and a set of outputs (an *event*). Intuitively, we use the examples to quickly screen out choices for the noise scales that *don't* lead to differentially-private programs—if M is differentially private, it must have similar probabilities of producing an output in E from inputs d_1 and d_2; this local property can be quickly checked without running the whole testing tool. However, not all examples are equally useful. Mechanisms may need only a low level of noise to satisfy the privacy condition at easier examples, but may need higher levels of noise to satisfy the privacy condition at more difficult examples. Since the differential privacy property quantifies over all pairs of adjacent databases, we need to find *challenging* examples that maximizes the privacy loss.

To discover such examples, we leverage StatDP to generate pairs of databases and output events. Since StatDP requires a complete mechanism as input, rather than just a sketch, we first complete $\gamma(M^\bullet)$ by filling holes with concrete noise values (i.e., real numbers). To search the space of noise, a naïve grid search over the space of noise values will generate many values of $\boldsymbol{c}$, but calling StatDP for each $\boldsymbol{c}$ is expensive. Thus instead of performing a full grid search for noise values in each dimension, we choose a set *Dir* of predefined directions

and perform a line search along each direction. The directions are essentially chosen to contain a basis of the noise space, and hence, together, are likely to create a good representation of the space. For example, with $n = 2$ noise locations, we choose the vectors as $\{\langle 1,1\rangle, \langle 1,0\rangle, \langle 0,1\rangle\}$.

Finally, we apply StatDP on the completed sketch to check if the mechanism is $\gamma(\epsilon)$-differentially private. If the tester judges the mechanism to not be differentially private, it returns a counterexample and a p-value, indicating the degree of confidence that the counterexample is a true counterexample. If the p-value of the discovered counterexample is in the *zone of confusion*, $\mathit{Conf} \subset [0,1]$—indicating that the tool had a hard time proving or disproving privacy—then we consider the counterexample to be challenging and we keep it. Summing up, the set $\mathsf{selectExamples}(M^\bullet)$ is defined as:

$$\{\langle d_1, d_2, E\rangle \mid (\langle d_1, d_2, E\rangle, p) = \mathsf{test}(\gamma(M^{\boldsymbol{c}}), \gamma(\epsilon)), \\ p \in \mathit{Conf}, \boldsymbol{c} \in \mathit{Dir}\}$$

Running example: Above Threshold. The sketch $\mathsf{aboveT}^\bullet$ from Fig. 3b has three arguments (non-private inputs): the queries $\boldsymbol{q}$, the threshold T, and the target privacy level ϵ. One possible setting of the arguments is:

$$\gamma = \{\boldsymbol{q} \mapsto (q_1, \ldots, q_5), T \mapsto 2, \epsilon \mapsto 0.5\}$$

where we abbreviate the queries $q_1, \ldots, q_5$. For this setting, our tool identifies several challenging counterexamples $\langle d_1, d_2, E\rangle$, including $\langle r, t, \{3\}\rangle$ and $\langle s, t, \{3\}\rangle$, where:

$$q_1(r), \ldots, q_5(r) = 0,0,0,0,0$$
$$q_1(s), \ldots, q_5(s) = 2,2,0,0,0$$
$$q_1(t), \ldots, q_5(t) = 1,1,1,1,1.$$

C. Reducing to a continuous optimization problem

After identifying a set of challenging examples Ex, we try to find a concrete instantiation $\boldsymbol{c}$ of $\gamma(M^{\boldsymbol{c}})$ that (1) is differentially private at every example in Ex, and (2) maximizes the privacy loss while remaining ϵ-DP; this second criteria biases the search towards noise parameters that achieve a tight privacy guarantee. Note that this task is in line with our problem statement (Definition 3.1), except that, (a) we work with a *finite set* of examples rather than all pairs of neighboring databases and output events, and (b) we complete $M^\bullet$ with *concrete numbers*, not symbolic expressions.

To find a concrete noise value $\boldsymbol{c}$, we set up the following optimization problem:

$$\operatorname*{argmin}_{\boldsymbol{c}} \left| \left(\max_{\langle d_1, d_2, E\rangle \in Ex} L(\gamma(M^{\boldsymbol{c}}), d_1, d_2, \gamma(\epsilon), E) \right) - e^{\gamma(\epsilon)} \right|$$

This optimization objective looks for a concrete completion $\boldsymbol{c}$ that makes the privacy loss of the worst example in Ex as close as possible to the target privacy loss $e^{\gamma(\epsilon)}$. Intuitively, taking the objective function to be the absolute difference between the target privacy level and the privacy loss at the challenging examples induced by a particular noise vector $\boldsymbol{c}$ penalizes concrete noise values $\boldsymbol{c}$ that add too much noise or too little noise. If the privacy loss at $\boldsymbol{c}$ is above $e^{\gamma(\epsilon)}$ then the concretized mechanism should be rejected since it fails to satisfy $\gamma(\epsilon)$-privacy. If the privacy losss at $\boldsymbol{c}$ is below $e^{\gamma(\epsilon)}$, then the concretized mechanism satisfies a differential privacy guarantee that is *stronger* than the target guarantee of $\gamma(\epsilon)$-privacy; this is not preferred as it adds *more* noise than necessary, typically leading to a less accurate mechanism.

To bias the search towards solutions that add noise at fewer locations, we also regularize the objective with the L_0 (sparsity) norm of $\boldsymbol{c}$; recall that this norm counts the number of non-zero entries. The final optimization function is as follows:

$$\operatorname*{argmin}_{\boldsymbol{c}} \left| \left(\max_{\langle d_1, d_2, E\rangle \in Ex} L(\gamma(M^{\boldsymbol{c}}), d_1, d_2, \gamma(\epsilon), E) \right) - e^{\gamma(\epsilon)} \right| + \lambda \|\boldsymbol{c}\|_0$$

where $\lambda \in \mathbb{R}$ is a regularization parameter.

Unfortunately, the optimization problem is not easy to solve—the objective is not convex or differentiable, and the loss function is expensive to evaluate even approximately, as it involves computing the probability of invocations of mechanisms returning certain events.

To approximately solve this optimization problem, we employ an evolutionary algorithm called *differential evolution* [13]. The algorithm maintains a set of candidate solutions, referred to as a *population*. In each iteration, every candidate in the current population is moved around using a simple randomized heuristic; the candidate's new position is retained if it reduces the objective function (referred to as the *fitness function*). As a result, the population tends to stabilize in the low loss regions. As the algorithm does not require any other information about the fitness function except its value at a given candidate, it can be used for noisy problems that are not differentiable, and not even continuous. Though differential evolution does not guarantee convergence to the optimal, it is useful when we are interested in estimating a "near optimal" *region* of solutions—captured by the final population.

After multiple rounds of evolutionary refinement, the candidate noise vectors tends to stabilize, providing us with a set of instantiations that minimize the objective; we call these candidates the *noise region* R.

Computing the objective efficiently. Each round in the evolutionary search requires computing the objective for each candidate noise vector $\boldsymbol{c}_i$ in the population—the objective depends on the probabilities of output events $\gamma(M^{\boldsymbol{c}_i}(d)) \in E$, where $\gamma(M^{\boldsymbol{c}_i}(d))$ is the sketch $M^\bullet$ with concrete noise scales $\boldsymbol{c}_i$, applied to the input arguments from γ and database d. The main new difficulty in our setting, compared to work on prior testing/verification of differential privacy [14], [11], [15], [16], is that the noise scales are not fixed in advance: the search considers a population of mechanisms with concrete different noise scales, and the scale parameters are repeatedly adjusted as the optimization unfolds.

Since computing the fitness function is the primary bottleneck for an evolutionary algorithm, it is crucial to efficiently compute the probabilities of output events for different mechanisms. While there are existing symbolic methods for computing probabilities exactly, these methods are too slow for our purpose.

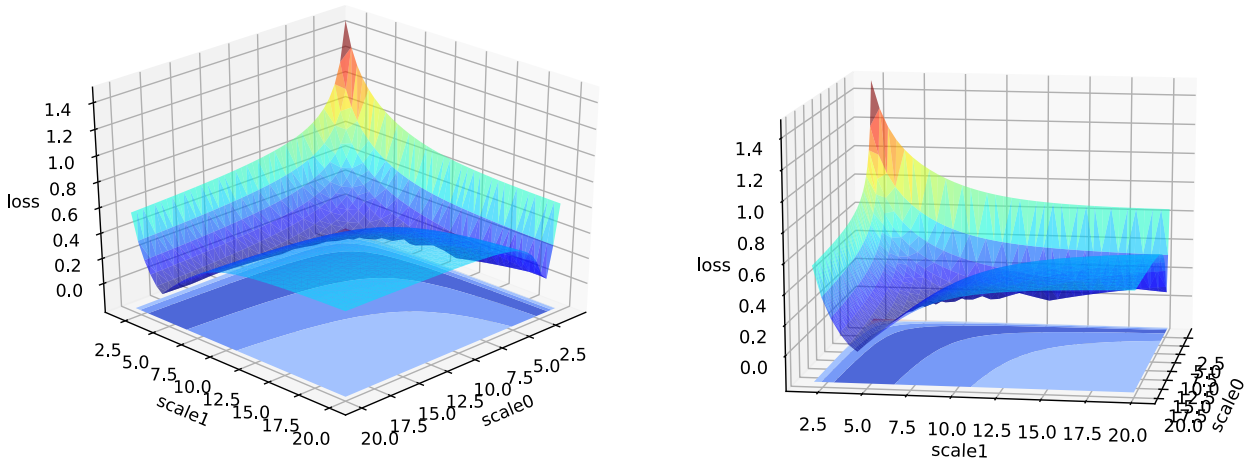


Fig. 5: Optimization objective for $\gamma(\mathsf{aboveT}^\bullet)$

Instead, a natural idea is to estimate output probabilities by repeated sampling. This is more efficient than exact computation, but there are still difficulties: estimating probabilities separately for each candidate $\boldsymbol{c}_i$ introduces a large overhead when this procedure is employed on large populations across many optimization steps, and stochastic variations across sampling steps due to random sampling introduce more instability into the optimization procedure, leading to slower convergence.

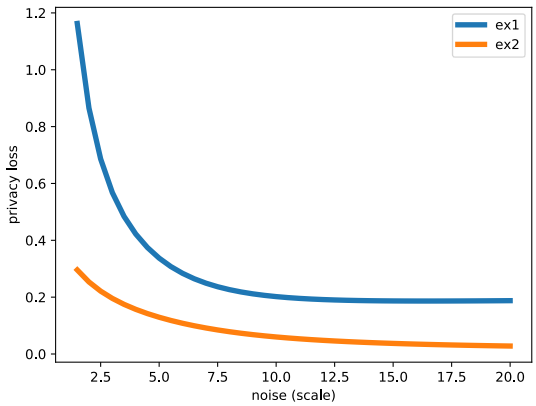
To speed up this process, we take advantage of the structure of the objective. For every example $\langle d_1, d_2, E\rangle \in Ex$, computing the privacy loss L for a candidate mechanism M amounts to computing the probability of E on d_1, computing the probability of E on d_2, and then taking the ratio of the probabilities. Both of these steps can be optimized:

- **Computing the probability of an output event.** Instead of drawing random samples separately for every mechanism in the population, we can preselect a single set of random samples for each sampling statement in the program, and reuse these samples—across all candidates and over all optimization steps—to estimate the probability of the output event E. This leads to a significant reduction in the sampling time. Of course, since a specific noise value may have different probabilities under the different candidates, we must weight each trial differently for each candidate when estimating each probability of E.
  In more detail, suppose the sketch $M^\bullet$ has n holes. We draw m uniform vectors $\mathbf{v}_j \in \mathbb{R}_+^n$ of non-negative real numbers representing the *results* of the sampling statements in $M^\bullet$. Then, each probability of $[\gamma(M^{\boldsymbol{c}_i}(d)) \in E]$ can be estimated by counting how many runs of $\gamma(M^\bullet)(d)$ with noise $\mathbf{v}_j$ produce an output in E, weighted by the probability of drawing $\mathbf{v}_j$ from the Laplace distribution with noise scale $\boldsymbol{c}_i$; this probability can be computed analytically and efficiently, without sampling. The latter step is essentially an application of Monte Carlo integration; to reduce variance further, our implementation performs importance sampling using a fixed Laplace (or Exponential) distribution as the proposal distribution.
- **Computing the privacy loss at an output event.** After computing the probability of E on inputs d_1 and d_2, we must take the ratio of these probabilities to compute the privacy loss, and then take a maximum over all examples. To further reduce the number of samples required, we can estimate the probability on d_1 and d_2 using the same set of samples; this correlated sampling method was previously used by [14].

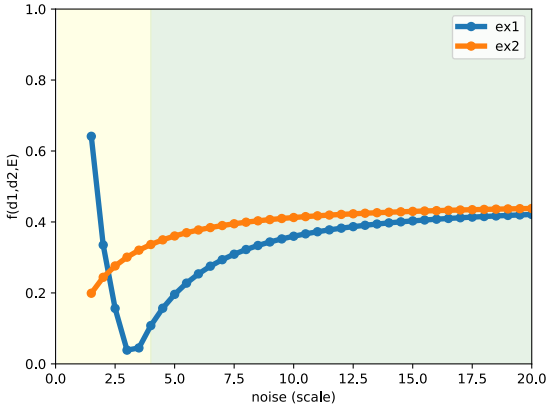
Running example: Above Threshold. To give a better idea of the optimization problem for our running example, Fig. 5 shows the objective (sans regularization) as we plug in different concrete noise values for the holes in $\mathsf{aboveT}^\bullet$, with arguments fixed to γ. To make the space easier to visualize, the plot only varies noise for the first two locations while η_3 is set to $\perp$. The plot shows that there isn't a single noise setting that minimizes the objective; rather, there is a broad *region* where the objective is approximately minimized (visualized in the contour map in dark blue).

D. Enumerative synthesis

To complete synthesis, we want to produce symbolic expressions that lead to noise close to the noise region R; these completions should work for various settings of the mechanism arguments, not just γ. Our procedure to search for an expression, denoted findExpr in Fig. 4, inspired by the classical synthesize-check loop from the formal-methods literature, enumerates all expressions $\boldsymbol{\eta}$ in the grammar G. However, it only considers expressions where the concrete values $\gamma(\boldsymbol{\eta})$ are in a *neighborhood* of the noise region R, denoted $\mathit{Nbhd}(R)$ (defined on the L_1 distance of instantiations



(a) Privacy loss for different examples



(b) $f(d_1, d_2, E)$ at $\gamma(\epsilon) = 0.5$; the green shade captures the region where this algorithm is 0.5-differentially private

Fig. 6: Comparing the implication of the choice of examples

$$
\begin{array}{l}
\mathsf{NoisyMax}^{\bullet}(d, \boldsymbol{q}, \epsilon) : \\
\quad m, v \leftarrow \perp, \perp; \\
\quad \boldsymbol{a} \leftarrow \boldsymbol{q}(d); \\
\quad \mathsf{for}\ i \in \{1 \ldots |q|\} \\
\quad\quad b \leftarrow \boldsymbol{a}_i + \mathsf{Lap}(\boxed{\eta}); \\
\quad\quad \mathsf{if}\ b \geqslant v \\
\quad\quad\quad m, v \leftarrow i, b; \\
\quad \mathsf{return}(m);
\end{array}
$$

Fig. 7: NoisyMax1: Sketch program

in R). This pruning step is the key to efficient synthesis (see Section V for a more thorough evaluation). Formally, we generate the following set of candidate expressions:

$$cand = \{\boldsymbol{\eta} \mid \boldsymbol{\eta} \in G,\ \gamma(\boldsymbol{\eta}) \in Nbhd(R)\}$$

In general, this set contains multiple candidate expressions. To narrow down this list, we augment Ex with additional test examples Ex_{test}—varying the arguments γ—and then rank the candidates according to (1) how many examples in Ex_{test} they violate (fewer is better), (2) their privacy loss on examples in Ex_{test} (higher is better), and (3) the magnitude of noise injected (smaller is better). Note that we may accept expressions that violate some small number of examples, since there is probabilistic noise involved when checking whether an example is violated. Finally, we call the testing tool on the mechanisms from the top few completions in $cand$, and output a ranked list of passing mechanisms (following the same ranking as above) as candidate solutions to the mechanism synthesis problem.

Importance of selecting challenging examples. A key aspect of our approach is selecting "challenging" examples. To better understand why certain examples may be easier or more difficult than others, let us consider the *Noisy Max* mechanism (Fig. 7): it returns the index of an approximately maximum query while preserving privacy by perturbing every query by Laplace distributed noise. It is known that this mechanism achieves ϵ-differential privacy when the noise scale η is taken to be $2/\epsilon$ [12].

Figure 6a plots the privacy loss (vertical axis) of this mechanism for different concrete values of η (horizontal axis) for two different examples $\langle d_1, d_2, E\rangle$, $ex_1 : \langle 111, 022, 1\rangle$ and $ex_2 : \langle 111, 022, 2\rangle$. The plot shows that ex_1 is a more *challenging example*: it incurs a higher privacy loss than ex_2 for all values of η; so any expression synthesized to satisfy ex_1 will also satisfy ex_2. Conversely, symbolic expressions synthesized to ensure differential privacy at ex_2 may not be capable of ensuring differential privacy at ex_1. When deciding which examples to use, our synthesis procedure should seek to keep examples like ex_1, and discard examples like ex_2—the former example is *more useful* than the latter example.

However, given two examples, it is not easy to tell which (if either) example is more challenging. The objective of our optimization function can be viewed as a rough heuristic, preferring more useful examples. To see this, consider the following function that captures the behavior of our objective function on a given example $\langle d_1, d_2, E\rangle$:

$$f(d_1, d_2, E)(\boldsymbol{c}) = |\log(L(\gamma(M^{\boldsymbol{c}}), d_1, d_2, \gamma(\epsilon), E)) - \gamma(\epsilon)|$$

That is, for a provided example $\langle d_1, d_2, E\rangle$, $f(d_1, d_2, E)$ is a function that maps $\boldsymbol{c}$ to the value of the objective function when the arguments to the noise distribution is $\boldsymbol{c}$. Roughly speaking, this function measures how closely the mechanism satisfies $\gamma(\epsilon)$-differential privacy at the given example: noise scales $\boldsymbol{c}$ where the mechanism $M^{\boldsymbol{c}}$ satisfies a much weaker or a much stronger guarantee have larger objective, and the optimization process avoids these noise vectors $\boldsymbol{c}$. While we do not have a rigorous proof that examples with lower objective are more useful, we found this to be the case empirically.

For an example of this heuristic in action, the plot in Fig. 6b at $\gamma(\epsilon) = 0.5$ shows that the objective function is lower at

$\mathsf{Histogram}^\bullet(d, \boldsymbol{q}, \epsilon)$:
  $\boldsymbol{hist} \leftarrow \boldsymbol{q}(d)$;
  $\boldsymbol{ans} \leftarrow \boldsymbol{hist} + \overrightarrow{\mathsf{Lap}}(\boxed{\eta})$;
  return $\boldsymbol{ans}$;

Fig. 8: Histogram: Sketch program

ex_1 than at ex_2 for all noise scales where the mechanism does satisfy differential privacy. However, this plot also shows that the example ex_1 is not the most challenging example possible. To see why this is so, recall that *Noisy Max* achieves ϵ-differential privacy when the scale parameter on the noisy distribution is $2/\epsilon$, e.g., for $\epsilon = 0.5$, the noise scale should be 4. Thus, an example where the privacy guarantee is tight should achieve ϵ-differential privacy only when the noise scale is 4. The (blue) curve for ex_1 in Figure 6b achieves a minimum around noise scale $\eta \approx 3$, so achieving ϵ-differential privacy at example ex_1 requires less noise than is required for achieving ϵ-differential privacy for the whole mechanism (i.e., at all examples). While ex_1 is not optimal in this sense, this example was sufficiently good to guide our implementation to a successful synthesis of the optimal symbolic expression of the noise parameter; we will describe our implementation in more detail in Section V.

Running example: Above Threshold. On this example, we use a grammar of expressions of the following form:

$$[-] \times |\boldsymbol{q}|^{[-]} \times T^{[-]} \times (1/\epsilon)^{[-]} \quad \text{or} \quad \perp$$

where each unknown $[-]$ ranges over non-negative integers, and $\perp$ represents no noise added. In general, our expression grammar combines powers of basic numeric expressions involving inputs of the target sketch.

For the sketch $\mathsf{aboveT}^\bullet$, our tool produces expression vectors $(2/\epsilon, 4/\epsilon, \perp)$ and $(3/\epsilon, 3/\epsilon, \perp)$ as candidate solutions. The first setting recovers the textbook version of the Above Threshold algorithm [12], while the latter gives a new variant that is also differentially private. The privacy proof for the second variant follows by adjusting parameters in existing privacy proofs of Above Threshold (e.g., [17]).

V. Experiments

We developed Kolahal, an implementation of our synthesis procedure. Our implementation is written in the Julia programming language [18], and uses the *BlackBoxOptim* package [19] for optimization.

Benchmark examples.

We used our tool to synthesize nine mechanisms from the differential privacy literature. We briefly describe our benchmarks here. We chose this set of benchmarks because they represent foundational algorithms and have been previously considered in the formal verification and testing literature [11], [14], [15], [20], [17].

Sum One of the simplest benchmarks, *Sum* (Fig. 9) computes the sum of a private list of numbers. To ensure privacy, each list element is perturbed by noise drawn from a Laplace distribution.

Histogram Given a input database, *Histogram* (Fig. 8) computes a histogram of frequency counts of number of elements in predefined buckets. It ensures privacy by adding noise from a Laplace distribution to each of the frequency counts.

Above Threshold The benchmark *AboveT1* (Fig. 12) is a sketch of the AboveThreshold algorithm [12]. This program takes a numeric threshold and a list of numeric queries, and returns the index of the first query whose answer is (approximately) above the threshold. We also consider a variant *AboveT2* (Fig. 3b), where the sketch has more noise locations than required.

Sparse Vector Technique The benchmark *SVT* (Fig. 13) is a sketch of the Sparse Vector Technique, an algorithm that has been rediscovered numerous times in the differential privacy literature [12]. The variant of SVT that we use [21] returns a vector to indicate which of the queries are above/below a noisy threshold; the mechanism halts after it outputs N above noisy threshold ($\top$) responses.

Noisy Max The benchmark *NoisyMax1* (Fig. 7) is a sketch of the Report-Noisy-Argmax algorithm from the differential privacy literature [12], which takes a list of numeric queries and releases the index of the query with (approximately) the highest answer. The sketch for *NoisyMax2* (Fig. 10) has more locations than required, and we also consider a variant *ExpNoisyMax* (Fig. 11) where the sketch specifies noise drawn from the Exponential distribution instead of the Laplace distribution.

SmartSum The benchmark *SmartSum* (Fig. 14) implements the two-level counter mechanism for computing all running sums of a sequence [22], [23]; roughly speaking, it chunks the sequence into blocks and adds noise to each block. The algorithm requires addition of noise at two program locations.

Comparison with simpler procedures. Our synthesis method involves quite a few moving parts. To demonstrate the importance of each phase in our algorithm, we evaluate Kolahal against four simpler baselines. In order of increasing sophistication:

naïve The naïve baseline applies a brute-force strategy: it enumerates all expressions at all program locations, queries the tester StatDP [11] as an oracle to accept or reject each choice of expressions, and then finally ranks all combinations of expressions. This baseline is perhaps the simplest synthesis method.

unlim This baseline improves upon brute-force enumeration by including an unbounded *counterexample cache* to memoize "good" counterexamples that have been able to cause violations for past instances. The idea behind this baseline is the hypothesis that a few "good" counterexamples can cause violations for most of the candidates being enumerated, so there is no need to find fresh counterexamples for every candidate. The cache is sorted by (1) *utility* (the

```
Sum•(d, q, ε) :
  a ← q(d);
  s ← 0;
  for i ∈ {1 . . . |q|} :
    s ← s + a_i + Lap(η);
  return s;
```

Fig. 9: Sum: Sketch program

```
NoisyMax2•(d, q, ε) :
  m, v ← ⊥, ⊥;
  a ← q(d);
  for i ∈ {1 . . . |q|} :
    b ← a_i + Lap(η_1);
    if b > v
      m, v ← i, b;
  ans ← m + Lap(η_2);
  return ans;
```

Fig. 10: NoisyMax2: Sketch program

```
ExpNoisyMax•(d, q, ε) :
  m, v ← ⊥, ⊥;
  a ← q(d);
  for i ∈ {1 . . . |q|} :
    b ← a_i + Exp(η_1);
    if b > v
      m, v ← i, b;
  ans ← m + Exp(η_2);
  return ans;
```

Fig. 11: ExpNoisyMax: Sketch program

```
AboveT1•(d, q, T, ε) :
  i ← 1;
  done ← false;
  t ← T + Lap(η_1);
  a ← q(d) + Lap(η_2);
  while i ⩽ |q| ∧ ¬done do
    if a_i > t then
      done ← true;
    i ← i + 1;
    if done then
      ans ← i − 1;
    else
      ans ← 0;
  return ans;
```

Fig. 12: AboveT1: Sketch program

```
SVT•(d, q, N, T, ε) :
  out ← [ ]; i ← 1;
  count ← 0;
  t ← T + Lap(η_1);
  a ← q(d);
  while i ⩽ |q| :
    qans = a_i + Lap(η_2);
    if qans > t;
      append(out, ⊤);
      count ← count + 1;
      if count ⩾ N then
        break;
    else
      append(out, ⊥);
    i ← i + 1;
  return out;
```

Fig. 13: SVT: Sketch program

```
SmartSum•(d, q, M, ε) :
  n ← 0; i ← 1;
  next ← 0; sum ← 0;
  a ← q(d);
  r ← [ ];
  while i ⩽ |q| :
    sum ← sum + a_i;
    if i mod M = 0 then
      n ← n + sum + Lap(η_1);
      sum ← 0;
      next ← n;
    else
      next ← next + a_i + Lap(η_2);
    prepend(r, next);
    i ← i + 1;
  return r;
```

Fig. 14: SmartSum: Sketch program

number of expression vectors that created a violation for this counterexample), and (2) *recency* (how new the counterexample is). Candidate mechanisms are evaluated, in order, on the examples in the counterexample cache before spawning the expensive tester StatDP to discover a new counterexample; if StatDP can produce a new counterexample, it is added to the cache.

lim When the counterexample cache grows too large, the *unlim* baseline wastes time searching through the cache on "bad" counterexamples at the tail end of the cache. In this optimized version, we limit the counterexample cache to the top-5 counterexamples (we empirically found this size to be a good setting).

noopt This version operates similarly to Kolahal but it does not use optimization to find a region R of noise values: all expressions that satisfy the grammar of expressions (G) are ranked according to the heuristics described in Section IV-D, and the top-k ranked expressions (k=5× #locations) are sent for verification. Compared to the other baselines, this version includes the initialization phase of Kolahal where challenging examples are selected, and it uses a top-5 limited counterexample cache.

We ran our experiments on a cluster with 4 worker threads, provisioning for 4 cores and 8 GB memory for each task. Table I shows our comparisons with the baseline: the second column (#locs) shows the number of locations in the sketch where noise can be added. The blank cells in the experimental results (Table I) correspond to jobs that did not complete on the cluster over two days.

We briefly comment on the performance of the baseline solutions. The *naïve* baseline performs reasonably well for mechanisms that require noise at a few locations with simple noise expressions (like *Histogram* and *Sum*); however, it struggles when the number of locations and the complexity of the expressions increase.

The baselines with the counterexample cache (*lim* and *unlim*) have improved performance in some cases (e.g., *NoisyMax1*). For the simpler cases (*Sum* and *Histogram*), it seems that the baseline loses too much time in warming up the cache. However, for a slightly more involved benchmark (*NoisyMax1*), the cache pays off. Also, in general, limiting the cache size (*lim*) seems to be a better configuration than an unlimited cache (*unlim*). The behavior of *AboveT* is a bit surprising: it fails to complete for *unlim*, which it understandable, but draws similar runtimes for both *naïve* and *lim*; it seems that the overhead of using the cache cancels out its gains.

The performance of the *noopt* baseline is a significant improvement over the previous baselines: it synthesizes most of the mechanisms in reasonable time and solves many more benchmarks, especially, the more involved ones. This shows the importance of identifying "good" counterexamples.

TABLE I: KOLAHAL versus baselines (time rounded to the nearest minute)

Mechanism		KOLAHAL (time in minutes and rank)						Baselines (time in minutes)			
benchmark	#locs	init	opti	enum	verify	**total**	rank	naïve	unlim	lim	noopt
Histogram	1	4	1	< 1	15	**20**	1	55	98	74	60
Sum	1	3	< 1	< 1	10	**15**	1	55	69	68	52
NoisyMax1	1	9	1	1	20	**32**	1	84	63	61	54
NoisyMax2	2	14	2	6	48	**72**	1	-	-	-	-
ExpNoisyMax	2	32	1	7	50	**90**	1	-	-	-	217
SmartSum	2	31	6	12	42	**91**	4	-	-	-	289
SVT	2	18	2	4	18	**44**	1	-	-	-	128
AboveT1	2	13	2	6	38	**60**	1	1582	-	1647	115
AboveT2	3	24	7	41	54	**126**	2	-	-	-	-

Finally, KOLAHAL, by including the optimization phase, is $2\times$ to $20\times$ faster than *noopt*. Furthermore, it solves a couple of benchmarks that *noopt* could not complete. This shows the value of the optimization phase, especially for involved mechanisms with more noise locations.

Overall, the above experiment shows that selecting challenging counterexamples and identifying a noise region for the search for expressions are crucial to the success of KOLAHAL.

Time spent in different phases. KOLAHAL spends most of its time in the tester STATDP, either in search of representative counterexamples (*init*) or in the final verification (*verify*). The optimization phase (*opti*) and enumerative synthesis (*enum*) are reasonably fast. The optimization phase owes its speed to our technique of approximating the probability estimates using importance sampling and the heuristic of reusing noise samples across candidates. The total time (*total*) is the end-to-end time taken by KOLAHAL, including time for logging.

Ranking of mechanisms. For almost all benchmarks, the noise setting corresponding to the textbook versions of these mechanisms is discovered by our tool and ranked high. Recall that we run a continuous optimization to prepare an preliminary ranking of the candidates, and the top ranked examples are passed to STATDP for a more rigorous test for the final ranking.

For each of *Sum*, *Histogram* and *NoisyMax1*, the textbook mechanism is ranked among the top two candidates even after the preliminary ranking, which, then, emerges as the topmost candidate after the final ranking. This behavior does not change for *NoisyMax2* where we add an additional noise location in the sketch, showing that our method can ignore irrelevant noise locations—when sketches are obtained by annotating existing, non-private code, many of the possible noise locations may be unnecessary for privacy. Our tool performed similarly for *ExpNoisyMax*, which not only contains an additional noise location but also uses an Exponential noise distribution instead of the Laplace distribution, showing that our method can be applied to sketches with noise distributions besides Laplace.

For *AboveT1*, the top two solutions that emerged are $(2/\epsilon, 4/\epsilon)$ and $(3/\epsilon, 3/\epsilon)$. The former is the classic version of the algorithm [12], while the latter is a *new* variant identified by KOLAHAL. The proof of privacy of the new variant follows the proof for the standard variant (see, e.g., [17]). In fact, the existing privacy proof applies to *exactly* these two variants, and no other variants. This benchmark shows that our tool is able to automatically discover new versions of well-known private algorithms.

For *SVT*, the standard version of the mechanism was not the topmost in the preliminary ranking, though it was high enough to be selected for the final phase, where it was identified it as the topmost candidate.

The only benchmark in which our tool had difficulty was *SmartSum*, where the ideal solution ended up being ranked fourth in the final ranking. While the expected noise expressions are $(2/\epsilon, 2/\epsilon)$, our procedure proposed the mechanism with noise scale $(1/\epsilon, 2/\epsilon)$ as the top ranking candidate because STATDP could not find counterexamples against the mechanism with $(1/\epsilon, 2/\epsilon)$, even though this mechanism is not ϵ-differentially private. Thus, STATDP did not generate high-quality examples (in selectExamples) to direct the search away from the incorrect expression and towards the correct expressions. In most of our benchmarks, however, we found that STATDP performed quite well.

VI. RELATED WORK

Program synthesis. Program synthesis is an active area of research; we summarize the most related directions here. Closest to our work is the recent paper [24] that develops a technique relying on user-defined examples to synthesize private programs in a strongly-typed functional language. However, this approach can only synthesize simple mechanisms where the privacy analysis follows from standard composition theorems; even if provided with an infinite number of examples, their system is not be able to synthesize mechanisms like NoisyMax, SVT, AboveT, and SmartSum. Our synthesis technique is also radically different: rather than using a type-directed approach, we perform a reduction to continuous optimization.

In terms of synthesizing randomized algorithms generally, most work has focused on programs where all inputs are

known; in that setting, the target specification for the synthesis problem is simpler—there is no need to quantify over all inputs, unlike the universal quantification over pairs of databases in our setting [25], [26]. Our general approach of using a small number of examples to guide the search appears in various forms of *counterexample-guided inductive synthesis* techniques [26]

Finding counterexamples to DP. There have been a few proposals for finding violations to differential privacy. As mentioned, our approach builds heavily on StatDP [11], a counterexample generation tool for differential privacy using statistical tests.

A different approach, DP-Finder [14], reduces the search for counterexamples to an optimization problem by approximating the mechanism by a differential surrogate function, thereby allowing the use of numerical optimization methods. The solution of the optimization on the surrogate function is a candidate counterexample. An exact solver (eg. PSI [27]), or an approximate, sampling-based estimator is then used to check if the candidate is a true counterexample on the actual mechanism. In spirit, our use of presampling is similar to derandomization of candidates in DP-Finder. Our approach also relies on an optimization problem but instead of transforming the optimization space via a surrogate function, we first concretize the mechanism to transfer the search over symbolic expressions to a search over real vectors, and then use a black-box optimizer that does not require gradients.

CheckDP [28] combines verification and falsification of differential privacy. Unlike StatDP, CheckDP relies on symbolic, rather than statistical methods to prove privacy and generate counterexamples. As a result, counterexamples do not come with a p-value or measure of tightness, a measure that is crucial to our example-selection process. It would be very interesting to see if the counterexamples and more powerful analysis afforded by CheckDP could be used to drive a synthesis approach, like ours.

Verifying DP. Differential privacy has been a prime target for formal verification ever since it was introduced by Dwork et al. [3], due to its compelling motivation, rigorous foundations, and clean composition properties. There are too many verification approaches to survey here, applying techniques like runtime verification, various kinds of type systems, and program logics. Unlike our approach, all of these approaches assume that the mechanism to be verified is fully specified. The most advanced examples considered in our benchmarks (e.g., *NoisyMax*, *AboveT*, *SVT*) have only recently been verified [17], [15]; they have also been tricky for human experts to design correctly [6]. LightDP [16] proposes a language for verifying privacy-preserving mechanisms and dependent type system for annotations to synthesize proofs of differential privacy. It constructs proofs by *randomness alignment* via an *alignment function* that "aligns" the noise on the executions corresponding to the adjacent databases. ShadowDP [20] attempts to construct a randomness alignment by instrumenting *shadow executions* to transform a probabilistic program to a program where privacy costs appear explicitly. This allows the transformed program to be verified by off-the-shelf verification tools.

VII. Discussion

We propose the first technique for automatically synthesizing complex differential privacy mechanisms (like NoisyMax, SVT, AboveT, and SmartSum) from sketches. Our approach does have certain limitations which opens up opportunities for interesting future work.

Perhaps the primary limitation of Kolahal is its dependence on StatDP for challenging counterexamples; developing techniques to generate high-quality, "worst-case" counterexamples would likely improve the synthesis procedure.

In the absence of a fast and robust verifier for differential privacy, we used a testing tool as a stand-in for a verification oracle. We assume that the failure to reject the null hypothesis (at a $\alpha = 0.05$) is an indication that the algorithm is DP at the provided privacy budget. This is a clear limitation of our choice of using a tester as a verifier, given that testers and verifiers answer complimentary questions: while a verifier ensures soundness (that a verification instance that is claimed to be verified is indeed so), a tester, on the other hand, guarantees completeness (any counterexample generated does indicate a violation). Nevertheless, we found it to be a good choice in practice. Once the candidates are ranked by Kolahal, an existing differential privacy verifier can then be used as the final step to prove that the synthesized program is private; besides the new variant of Above Threshold, the target mechanisms in our examples have all been certified by existing automatic verifiers [15], [16].

Our algorithm requires, as inputs, a sketch of a mechanism with noise expressions as holes and a finite grammar G for noise expressions. The algorithm is not capable of performing any syntactic transformations over the input sketch. It will fail to find a solution if no such noise expressions exist for the provided sketch within the provided grammar.

All our benchmarks were run with the same heuristics and the same setting of the hyperparameters (see Appendix); however, synthesizing more mechanisms would give a better assessment of the generality of these heuristics. Finally, it would be interesting to consider other forms of privacy, e.g., (ϵ, δ)-DP and Rényi differential privacy [29], [30].

Acknowledgements. The first author is grateful to the United States-India Educational Foundation for their support. This work is partially supported by the NSF (CNS-2023222, CCF-1943130, CCF-1652140), and grants from Facebook.

References

[1] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *IEEE Symposium on Security and Privacy (S&P), Oakland, California*, 2008, pp. 111–125. [Online]. Available: https://doi.org/10.1109/SP.2008.33

[2] J. A. Calandrino, A. Kilzer, A. Narayanan, E. W. Felten, and V. Shmatikov, ""You might also like:" privacy risks of collaborative filtering," in *IEEE Symposium on Security and Privacy (S&P), Berkeley, California*, 2011, pp. 231–246. [Online]. Available: https://doi.org/10.1109/SP.2011.40

[3] C. Dwork, F. McSherry, K. Nissim, and A. D. Smith, "Calibrating noise to sensitivity in private data analysis," in *IACR Theory of Cryptography Conference (TCC), New York, New York*, ser. Lecture Notes in Computer Science, vol. 3876. Springer-Verlag, 2006, pp. 265–284.

[4] Ú. Erlingsson, V. Pihur, and A. Korolova, "Rappor: Randomized aggregatable privacy-preserving ordinal response," in *ACM SIGSAC Conference on Computer and Communications Security (CCS), Scottsdale, Arizona*, 2014, pp. 1054–1067.

[5] N. Johnson, J. P. Near, and D. Song, "Towards practical differential privacy for SQL queries," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 526–539, 2018.

[6] M. Lyu, D. Su, and N. Li, "Understanding the Sparse Vector Technique for differential privacy," *Proceedings of the VLDB Endowment*, vol. 10, no. 6, pp. 637–648, 2017, appeared at the International Conference on Very Large Data Bases (VLDB), Munich, Germany. [Online]. Available: https://arxiv.org/abs/1603.01699

[7] F. D. McSherry, "Privacy integrated queries: an extensible platform for privacy-preserving data analysis," in *ACM SIGMOD International Conference on Management of Data (SIGMOD), Providence, Rhode Island*, 2009, pp. 19–30.

[8] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for MapReduce," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Jose, California*, vol. 10, 2010, pp. 297–312.

[9] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *International Conference on Architectural Support for Programming Langauages and Operating Systems (ASPLOS), San Jose, California*, 2006, pp. 404–415.

[10] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-guided synthesis*. IEEE, 2013.

[11] Z. Ding, Y. Wang, G. Wang, D. Zhang, and D. Kifer, "Detecting violations of differential privacy," in *ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, Ontario*, 2018, pp. 475–489.

[12] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014. [Online]. Available: https://dx.doi.org/10.1561/0400000042

[13] K. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series)*. Berlin, Heidelberg: Springer-Verlag, 2005.

[14] B. Bichsel, T. Gehr, D. Drachsler-Cohen, P. Tsankov, and M. Vechev, "DP-finder: Finding differential privacy violations by sampling and optimization," in *ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, Ontario*, 2018, pp. 508–524.

[15] A. Albarghouthi and J. Hsu, "Synthesizing coupling proofs of differential privacy," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, 2018.

[16] D. Zhang and D. Kifer, "LightDP: Towards automating differential privacy proofs," in *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France*, 2017, pp. 888–901. [Online]. Available: https://arxiv.org/abs/1607.08228

[17] G. Barthe, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub, "Proving differential privacy via probabilistic couplings," in *IEEE Symposium on Logic in Computer Science (LICS), New York, New York*. IEEE, 2016, pp. 1–10.

[18] "The Julia Language," https://julialang.org.

[19] "Blackboxoptim.jl," https://github.com/robertfeldt/BlackBoxOptim.jl.

[20] Y. Wang, Z. Ding, G. Wang, D. Kifer, and D. Zhang, "Proving differential privacy with shadow execution," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Phoenix, Arizona*, 2019, pp. 655–669.

[21] M. Lyu, D. Su, and N. Li, "Understanding the sparse vector technique for differential privacy," *Proc. VLDB Endow.*, vol. 10, no. 6, pp. 637–648, Feb. 2017. [Online]. Available: https://doi.org/10.14778/3055330.3055331

[22] T.-H. H. Chan, E. Shi, and D. Song, "Private and continual release of statistics," *ACM Transactions on Information and System Security*, vol. 14, no. 3, p. 26, 2011. [Online]. Available: https://eprint.iacr.org/2010/076.pdf

[23] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum, "Differential privacy under continual observation," in *ACM SIGACT Symposium on Theory of Computing (STOC), Cambridge, Massachusetts*, 2010, pp. 715–724. [Online]. Available: https://www.mit.edu/~rothblum/papers/continalobs.pdf

[24] C. Smith and A. Albarghouthi, "Synthesizing differentially private programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, 2019.

[25] A. Albarghouthi, L. D'Antoni, and S. Drews, "Repairing decision-making programs under uncertainty," in *International Conference on Computer Aided Verification (CAV), Heidelberg, Germany*. Springer, 2017, pp. 181–200.

[26] S. Chaudhuri, M. Clochard, and A. Solar-Lezama, "Bridging Boolean and quantitative synthesis using smoothed proof search," in *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California*, 2014, pp. 207–220.

[27] T. Gehr, S. Misailovic, and M. Vechev, "PSI: Exact symbolic inference for probabilistic programs," in *International Conference on Computer Aided Verification (CAV), Toronto, Ontario*. Cham: Springer International Publishing, 2016, pp. 62–83.

[28] Y. Wang, Z. Ding, D. Kifer, and D. Zhang, "CheckDP: An automated and integrated approach for proving differential privacy or finding precise counterexamples," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[29] M. Bun and T. Steinke, "Concentrated differential privacy: Simplifications, extensions, and lower bounds," in *IACR Theory of Cryptography Conference (TCC), Beijing, China*, ser. Lecture Notes in Computer Science, vol. 9985. Springer-Verlag, 2016, pp. 635–658.

[30] I. Mironov, "Rényi differential privacy," in *IEEE Computer Security Foundations Symposium (CSF), Santa Barbara, California*, 2017, pp. 263–275. [Online]. Available: https://arxiv.org/abs/1702.07476

APPENDIX

We use the following setting of hyperparameters for *all* the mechanisms we evaluated (i.e. we did not tune them separately for each case). The hyperparameters were selected via a set of preliminary experiments on a few mechanisms. These values continued to hold well as our benchmark set was expanded with more mechanisms. Nevertheless, a more exhaustive study can be done to evaluate the generality of this setting.

Selecting Examples

- We pick the zone of confusion on *p-values* $\in [0.05, 0.9]$.

Region Selection

- For importance sampling, we use a distribution of the same family (Laplace or Exponential) as provided in the sketch and a scale of 4.0;
- We use $\lambda = 1$ for the regularization parameter in the objective function in the optimization phase i.e. we weigh each of the objective function and the simplicity of the expression equally.

Differential Evolution

- We ran our optimizer for (500×#locations) steps, where #locations refers to the number of noise locations specified in the sketch;
- The size of the population was set to 50.

Enumerative Synthesis

- Our enumeration of expressions is over:

$$[1-4] \times |\boldsymbol{q}|^{[0-2]} \times (1/\epsilon)^{[1-2]} \quad \text{or} \quad \perp$$

  (noise expressions must be directly proportional to $\boldsymbol{q}$ and inversely proportional to ϵ);
- We define the neighborhood (Nbhd) of the region R as all instantiations lying within an L1 distance of 3 from instantiations in R;
- From the set of ranked expression vectors emitted by the enumerative synthesis phase, we select the top-(5×#locations) for rigorous verification.