Synthesizing Differentially Private Programs

CALVIN SMITH, University of Wisconsin - Madison, USA
AWS ALBARGHOUTHI, University of Wisconsin - Madison, USA

Inspired by the proliferation of data-analysis tasks, recent research in program synthesis has had a strong focus on enabling users to specify data-analysis programs through intuitive specifications, like examples and natural language. However, with the ever-increasing threat to privacy through data analysis, we believe it is imperative to reimagine program synthesis technology in the presence of formal privacy constraints.

In this paper, we study the problem of automatically synthesizing randomized, differentially private programs, where the user can provide the synthesizer with a constraint on the privacy of the desired algorithm. We base our technique on a *linear dependent type system* that can track the *resources* consumed by a program, and hence its privacy cost. We develop a novel *type-directed* synthesis algorithm that constructs randomized differentially private programs. We apply our technique to the problems of synthesizing database-like queries as well as recursive differential privacy mechanisms from the literature.

CCS Concepts: • **Security and privacy** → *Privacy-preserving protocols*; • **Theory of computation** → *Type theory*; Theory of database privacy and security; • **Software and its engineering** → *Programming by example.*

Additional Key Words and Phrases: program synthesis, differential privacy, linear type systems

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

1 INTRODUCTION

The problem of program synthesis from input–output examples has recently received considerable attention. A major focus of the research community has been on enabling programming-by-example for data wrangling, querying, and analytics. The underlying motivation is that accessing data and analyzing it is becoming a common task in our increasingly data-driven world, and program synthesis has the potential to democratize data analysis by enabling end users to specify programs through intuitive specifications.

Multiple techniques have been proposed to tackle a variety of program synthesis problems in the data analysis space—synthesis of spreadsheet data transformations [Gulwani et al. 2012], SQL database queries [Wang et al. 2017a; Yaghmazadeh et al. 2017; Zhang and Sun 2013], data-parallel analytics in frameworks like Apache Spark [Smith and Albarghouthi 2016], table transformations in R [Feng et al. 2017], amongst others. An implicit assumption in existing works is that the user executes the synthesized program on data that is fully accessible to them. However, today data privacy is of paramount importance, and most interesting datasets contain sensitive private information: medical data, personal transactions, web search history—a list that is expanding by the hour. Our goal in this paper is to address synthesis in a setting with formal privacy constraints.

Authors' addresses: Calvin Smith, University of Wisconsin - Madison, 1210 Dayton Street, Madison, Wisconsin, 53706, USA, cjsmith@cs.wisc.edu; Aws Albarghouthi, University of Wisconsin - Madison, 1210 Dayton Street, Madison, Wisconsin, 53706, USA, aws@cs.wisc.edu.

2475-1421/2019/8-ART94
https://doi.org/10.1145/3341698

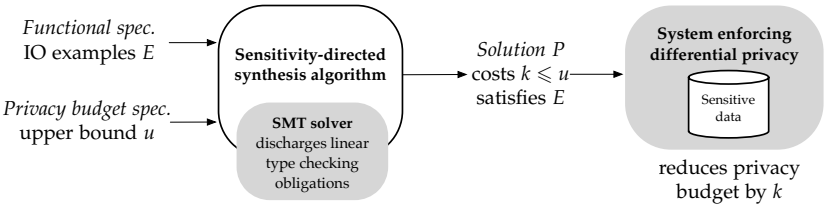

Fig. 1. Overview of synthesis technique and setting. *User-provided input–output examples are not private data—e.g., synthetic examples constructed by the user, or from a public dataset. Further information on the security model is outlined in §5.4.*

Differential privacy (DP) [Dwork and Roth 2014] has emerged as a rigorous probabilistic definition of privacy, where the goal is to guard the personal information of an individual in a dataset. In a nutshell, DP is enforced by adding *random noise* to the output of a program, and assigning a *price* to every program applied to a dataset commensurate with the amount of private information it leaks. A user (or group of users) will have a fixed *privacy budget* to access a dataset; after the budget is exhausted, no further computation is permitted. Therefore, the user needs to judiciously choose the computations they execute. A number of systems have been proposed by the research community for enforcing differential privacy [Johnson et al. 2018; McSherry 2009; Proserpio et al. 2014; Roy et al. 2010], and major corporations, like Google [Erlingsson et al. 2014], Apple [Apple 2017], and Uber [Johnson et al. 2018], have started incorporating differential privacy to protect sensitive customer information in their data-analysis tasks. Furthermore, governmental bodies like the US Census Bureau—which collects personal data—have been actively employing differential privacy [Bureau 2017; Haney et al. 2017].

Synthesizing Differentially Private Programs. We study the problem of automatically synthesizing differentially private programs. We assume a setting where a DP-enforcing system—like PINQ [McSherry 2009], Airavat [Roy et al. 2010], or FLEX [Johnson et al. 2018]—maintains *sensitive data*. Such a system executes randomized (noisy) user-supplied programs against the data, and decreases a user's privacy budget to reflect the total amount of information released.

Our goal is to aid users in constructing programs to execute on DP-enforcing systems. Given a user-provided specification of a program, we seek to synthesize an approximate differentially private program that lies within the user's privacy budget (see Fig. 1). In particular, we are interested in synthesizing a program while ensuring that it has a *low privacy cost*. The user presents a synthesis task as (*i*) input–output examples as a specification of the desired computation, and (*ii*) an upper bound on the privacy cost of the program. The synthesizer ensures that whatever program it constructs satisfies the examples and is within the user's budget. Reasoning about a program's privacy cost is a complex process, akin to reasoning about a program's runtime complexity. Automating such reasoning via synthesis can benefit end users, data scientists, and algorithm designers in performing data analysis in settings where DP is enforced.

Sensitivity-Directed Synthesis Algorithm. We present a synthesis technique for a programming language with a *linear dependent type system*, namely, a variant of DFuzz [Gaboardi et al. 2013]. The DFuzz type system allows us to reason about the *sensitivity* of a randomized program—which is treated as a *resource*—and hence its privacy cost: the more sensitive a program is, the more expensive the program will be deemed from a privacy-budget perspective.

Our technique is inspired by recent breakthroughs in *type-directed synthesis* [Feser et al. 2015; Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016; Smith and Albarghouthi 2016]. These techniques incrementally refine an incomplete program, guided by type information to direct

the search and avoid ill-typed programs. However, they focus on type systems for deterministic computation, and therefore cannot naturally capture a probabilistic hyperproperty like differential privacy, motivating synthesis in a rich type system like DFuzz. Much of the reasoning a synthesis algorithm must do is *dual* to the reasoning a type-checking algorithm does, and so type-directed synthesis algorithms are often constructed as an *inversion* of type-checking algorithms. We would like to construct a type-directed synthesis approach that exploits the full power of the rich linear dependent types provided by DFuzz. Unfortunately, the only known type-checking algorithm for DFuzz [de Amorim et al. 2014] works *bottom-up*, in contrast with type-directed synthesis which must work *top-down*. To enable synthesis in this setting, we address several technical challenges:

Challenge 1 In order to use DFuzz types to guide the search, we must be able to reason about the sensitivity of partial, incomplete programs. To do so, we introduce a language of *symbolic context constraints*, which succinctly characterize the infinitely-many possible typing contexts of an incomplete program. These symbolic constraints are used to direct the search towards low-sensitivity programs. We present a technique for checking the satisfiability of a symbolic context constraint by reduction to the theory of non-linear arithmetic.

Challenge 2 To find subprograms of a type compatible with incremental refinement, we need to reason about linear dependent subtyping. We introduce a subtyping *constraint abduction* procedure that answers the question: *"what sensitivity constraints need to hold to safely insert an expression of type τ in this incomplete program?"*.

Our solutions to these challenges, amongst others, allow us to construct a powerful synthesis algorithm, which we call *sensitivity-directed synthesis*.

Applications of Our Technique. We present two applications of our algorithm: synthesis of (*i*) data-analysis queries and (*ii*) recursive differential privacy algorithms from the literature.

In the first instance, we focus on programs composed of higher-order combinators like *map* and *filter*, the core building blocks of many data-analytics systems, including DP-enforcing ones, like PINQ [McSherry 2009] and Airavat [Roy et al. 2010]. We show how to apply our approach using different *privacy mechanisms*, which establish differential privacy using various randomization schemes. Notably, we show how to synthesize differentially private programs using the *exponential mechanism* [McSherry and Talwar 2007]. In many scenarios, adding noise can make a result useless; or, if the result is not numeric, it is not clear how noise can be added at all. To address those problems, the exponential mechanism assumes that the user provides a *utility function*, where the user indicates what sets of results should be of high utility. We demonstrate how to synthesize the utility function directly from examples.

In the second instance, we use our technique to synthesize iterative differentially private algorithms from the literature, thus utilizing the full spectrum of DFuzz features—recursion, pattern matching, etc. For instance, we can synthesize a version of the *iterative database construction* algorithm [Gupta et al. 2012], which builds an approximate database by sampling queries and updating the approximation to more closely match the expected output. These algorithms demonstrate our technique's ability to synthesize sophisticated mechanisms from the differential privacy literature.

Implementation and Evaluation. We have implemented our sensitivity-directed synthesis approach in a new tool that (*i*) accepts a set of functions typed in DFuzz, and (*ii*) uses the Z3 SMT solver for discharging typing constraints and pruning the search space. We apply our implementation to a wide range of benchmarks from the two applications described above. Our results indicate (*i*) the ability of our technique to synthesize randomized programs in a rich type system and (*ii*) the advantages of our sensitivity-directed technique in comparison with a baseline type-directed approach that does not exploit sensitivities.

Contributions. We summarize our contributions as follows:

- **Synthesis for differential privacy (§3):** We formalize the problem of synthesizing differentially private programs, in which the goal is to satisfy a correctness specification and meet a given privacy budget.
- **Sensitivity-directed synthesis (§4):** We present a novel synthesis algorithm that is guided by a rich linear dependent type system that tracks the cost of computations. The algorithm employs *symbolic context constraints* and *subtyping constraint abduction*.
- **Applications (§5):** We describe two applications of our approach: (1) synthesis of data-analysis queries using higher-order combinators, employing privacy mechanisms like the exponential mechanism and parallel composition; (2) a general application of our approach for synthesizing recursive differentially private algorithms from the literature.
- **Evaluation (§6):** We implement our approach and conduct a through evaluation in which we show our technique's ability to synthesize interesting differentially private programs and the importance of sensitivity-directedness to synthesis performance.

2 ILLUSTRATION AND OVERVIEW

In this section, we discuss two examples of programs that our approach can synthesize. We additionally use them to illustrate differential privacy and the DFuzz type system.

2.1 Example 1: Aggregation Query

Suppose we would like to calculate the number of patients diagnosed with cancer in a hospital without sacrificing the privacy of individual patients. Differential privacy stipulates that a certain amount of random noise needs to be added to the computation. For example, the following randomized function, f, computes the total number of patients diagnosed with cancer (c), assuming the database m is a multiset of pairs of "Patient ID" and "Diagnosis". f returns a noisy version of the true count c by sampling from a Laplace distribution with mean c and scale $1/\epsilon$, where $\epsilon \in \mathbb{R}^{>0}$ is a parameter of the system executing the query that determines the amount of noise to be added.

```
let f = λm. Laplace c
  where c = count m′
  where m′ = filter (λ⟨k, v⟩ . v = "Cancer") m
```

The type of f is ms(row)[∞] $\multimap_\epsilon$ $\bigcirc\mathbb{R}$: it takes a multiset of rows of unbounded size, denoted by the *indexed type* ms(row)[∞], and returns a *sample* from a distribution over real numbers, denoted by the probability monad $\bigcirc\mathbb{R}$. The ϵ denotes the *sensitivity* of the function—how much the output changes if we perturb the input. In our case, adding or deleting one row from the input multiset changes the output probability distribution by a multiplicative factor of ϵ. Formally, the *metric* used over probability distributions by the type system implies *ϵ-differential privacy*: for any two multisets, m_1, m_2, differing by an element and for any subset $S \subseteq \mathbb{R}$, we have

$$\mathbb{P}[f(m_1) \in S] \leqslant e^\epsilon \cdot \mathbb{P}[f(m_2) \in S]$$

As a visual illustration, consider m_1 and m_2 in Fig. 2. The two multisets differ by a single element, patient D. DP ensures that a physician applying f before and after the new patient D is added should not be able to tell whether patient D has cancer, since the two distributions are close to each other, and the physician only observes a single sample. By applying repeated queries, the physician can infer the actual value of the query with high confidence, but the system executing queries on the dataset will enforce a fixed *privacy budget*, protecting from such statistical-inference attacks.

Synthesis Technique. To synthesize the function f above, the user needs to supply (1) a set of input–output examples describing f and (2) an upper bound on the sensitivity of the function, e.g., ϵ. Note that since f is randomized, the input–output examples will refer to the denoised version of the function; i.e., when checking whether f satisfies the examples, our algorithm evaluates f on the examples and replaces privacy mechanisms like `Laplace` c with the function that deterministically returns c (the mean). For instance, the user may provide a toy dataset like m_2 in Fig. 2 as an input example with the corresponding output 2.

Dataset m_1

Patient ID	Disease
A	Heart
B	Heart
C	Cancer

Dataset m_2

Patient ID	Disease
A	Heart
B	Heart
C	Cancer
D	**Cancer**

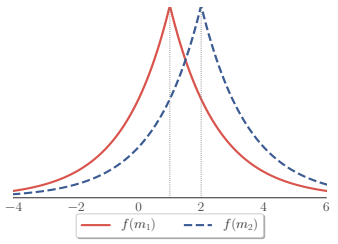

Fig. 2. Applying f to adjacent databases m_1 and m_2.

Our synthesis algorithm operates in a top-down fashion, refining an incomplete program until it is complete, satisfies the input–output examples, and is within the given sensitivity upper bound. For an illustration, consider the following incomplete function, where ● denotes a *wildcard* to be replaced with an expression:

```
let f' = λm. Laplace (sum ●)
```

Here, the function `sum` computes the sum of elements in a multiset of real numbers, so it has type $\mathsf{ms}(\mathbb{R})[\infty] \multimap_\infty \mathbb{R}$. Notice that `sum` is *infinitely* sensitive: adding a number to the input multiset may result in an arbitrary change to the total sum. Our algorithm realizes that it is impossible to complete f' and still result in an ϵ-DP function, as adding any amount of noise to the result of an ∞-sensitive deterministic computation corresponds to an ∞-DP function—i.e., no privacy is guaranteed. Therefore, the whole search space rooted at the incomplete program f' is pruned. The same would hold if we were to replace `sum` with a 2-sensitive function of type $\mathsf{ms}(\mathbb{R})[\infty] \multimap_2 \mathbb{R}$. Our synthesis algorithm would determine that any completion would result in, at best, a 2ϵ-DP function. This is larger than our privacy bound of ϵ, and therefore the search space is pruned.

This *sensitivity-directed pruning* of the search space is guided by a set of type constraints over symbolic sensitivity variables that are maintained along with incomplete programs. We show that these type constraints can be translated to the theory of real closed fields and checked for satisfiability using, e.g., SMT solvers. When those constraints are unsatisfiable, we know that no completion is possible for an incomplete program.

Types of Higher-Order Combinators. To give a better a taste of the type system we are working with, consider the type of the `map` combinator over multisets:

$$\mathtt{map} : \forall \alpha, \beta.\, \forall s.\, (\alpha \to \beta) \multimap_{2 \cdot s} \mathsf{ms}(\alpha)[s] \multimap_1 \mathsf{ms}(\beta)[s]$$

Semantically, `map` takes a function g and a multiset m (with up to s elements) and applies g to every element of m to get a new dataset with the same size upper bound. The type signature of `map` encodes its *privacy* semantics. If the function g is modified in any way, the resulting multiset is different by at most $2 \cdot s$ elements, as denoted by the sensitivity w.r.t. the first argument. Similarly, if the input multiset is modified by adding or deleting one element, this results in a multiset that is different by at most 1 element, as denoted by the sensitivity w.r.t. the second argument.

2.2 Example 2: Iterative Algorithm

The differential privacy literature has a number of privacy-preserving variants of popular algorithms; in this paper, we explore those more complex programs. For instance, we show how to synthesize a

```
let k-means iter centers data = match iter with
 | 0 → return centers
 | n + 1 →
  let-draw centers = k-means n centers data in
   k-step centers data
```

```
let idc iter data queries = match iter with
 | 0 → return init_approx
 | n + 1 →
  let-draw approx = idc n data queries in
  let-draw query = q-select queries approx data in
  let-draw actual = Laplace (eval-q query data) in
   return (dua approx query actual)
```

Fig. 3. Implementation sketches for *k-means* and `idc`, both modified from Gaboardi et al. [2013].

differentially private version of the k-means clustering algorithm, which has the following type:

$$\texttt{k-means} : \forall i, k.\ \underbrace{\mathbb{N}[i]}_{\#\text{ iterations}} \multimap_{\infty} \underbrace{L(\langle \mathbb{R}, \mathbb{R} \rangle)[k]}_{\text{initial clusters}} \multimap_{\infty} \underbrace{\mathsf{ms}(\langle \mathbb{R}, \mathbb{R} \rangle)[\infty]}_{\text{dataset}} \multimap_{3 \cdot i \cdot \epsilon} \underbrace{\bigcirc L(\langle \mathbb{R}, \mathbb{R} \rangle)[k]}_{\text{final clusters}}$$

The goal of k-means is to map a multiset of points to k clusters. It takes the number of iterations of the k-means update procedure, `k-step`, to execute, a list of initial clusters, and a multiset of points (in $\mathbb{R}^2$). Observe that the privacy (sensitivity) of k-means is a function of the number of iterations of a single step: $3 \cdot i \cdot \epsilon$. Given a small example input dataset and expected output, our algorithm is able to synthesize a recursive implementation of differentially private k-means, similar to the one presented by Gaboardi et al. [2013].

An implementation of *k-means* is given in Fig. 3, along with an implementation of the *iterative database construction* algorithm by Gupta et al. [2012], which iteratively learns a synthetic database (using `dua`, a database update mechanism such as multiplicative weights [Hardt et al. 2012]) to accurately answer a set of queries. Our algorithm is also able to synthesize `idc` when provided with a small set of examples.

3 THE SYNTHESIS PROBLEM

In this section, we formally define the synthesis problem. We begin with background on differential privacy and the DFuzz type system.

3.1 Differential Privacy Background

We begin with background on *differential privacy*; see Dwork and Roth [2014] for a thorough exposition.

Let A and B be domains equipped with metrics $d_A : A^2 \to \mathbb{R}^{\geqslant 0}$ and $d_B : B^2 \to \mathbb{R}^{\geqslant 0}$ that define a real-valued distance between two elements of A or B.

Definition 1 (ϵ-DP). *A randomized function $f : A \to B$ is ϵ-differentially private (ϵ-DP), for $\epsilon \in \mathbb{R}^{>0}$, if for all $a_1, a_2 \in A$ such that $d_A(a_1, a_2) \leqslant 1$ and $S \subseteq B$, we have*

$$\mathbb{P}[f(a_1) \in S] \leqslant e^{\epsilon} \cdot \mathbb{P}[f(a_2) \in S]$$

Note that the probabilities of event S in the two applications of f become the same as ϵ approaches 0. Smaller values of ϵ therefore correspond to increased privacy.

Sensitivity and ϵ-DP. To understand how the noise added affects privacy, we need a notion of how *sensitive* a function is.

Definition 2 (Sensitivity). *A deterministic function $f : A \to B$ is c-sensitive if, for all $a_1, a_2 \in A$,*

$$d_B(f(a_1), f(a_2)) \leqslant c \cdot d_A(a_1, a_2)$$

It follows that a c-sensitive function is also c'-sensitive for any $c' \geqslant c$.

The Laplace mechanism makes c-sensitive real-valued functions $c\epsilon$-DP.

Theorem 3.1 (Laplace Mechanism). *Let $f : A \to \mathbb{R}$ be a c-sensitive function, and let $\epsilon \in \mathbb{R}^{>0}$. Let $f'(x) = f(x) + \text{Lap}\,(1/\epsilon)$ be the randomized function that adds a random value drawn from the Laplacian distribution with mean 0 and scale $1/\epsilon$ to the output of f.*[1] *Then f' is $(c \cdot \epsilon)$-DP.*

Theorem 3.1 demonstrates the natural relation between privacy and sensitivity: the more sensitive f is, the weaker the privacy guarantee of f'. Setting the Laplacian distribution's scale parameter to ϵ'/c allows one to achieve any desired level of ϵ'-DP, at the expense of a less accurate f'.

Composition. Consider a 2-sensitive function $f(x) = 2 \cdot x$, and a 3-sensitive function $g(x) = 3 \cdot x$. It is easy to see that the composition $g \circ f$ is 6-sensitive. This principle holds in generality: composition combines sensitivities *multiplicatively*. Naturally, it follows that privacy composes additively. As $e^{\epsilon_1} \cdot e^{\epsilon_2}$ is equivalent to $e^{\epsilon_1 + \epsilon_2}$, the incurred privacy cost of composing computations is the *sum* of the individual privacy costs. This can be formalized as follows:

Theorem 3.2 (Sequential composition). *Let f_1 and f_2 be ϵ_1-DP and ϵ_2-DP, respectively. The function $g(x) = \langle f_1(x), f_2(x) \rangle$ is $(\epsilon_1 + \epsilon_2)$-DP.*

Theorem 3.2 generalizes to an online version: the choice of f_2 can depend on the output of $f_1(x)$.

3.2 The DFuzz Type System

We now present a simplified view of the linear dependent type system DFuzz. Refer to Gaboardi et al. [2013] for a full description.

Differences from DFuzz. The system described below is a *variant* of DFuzz containing minor changes. As the changes are not substantial enough to qualify the variant as a distinct system, we will continue to refer to our type system as DFuzz and simply highlight the changes here.

To support compositions of higher-order operators (such as `map` from §2), we enable type polymorphism by augmenting DFuzz with type quantifiers (similar to de Amorim et al. [2014]).

Databases in DFuzz are elements of a type with no size information. We treat databases as elements of the type $\texttt{ms}(\tau)[S]$, which are multisets containing *at most* S elements of type τ. Including multisets smaller than S in the type allows us to compute the distance between multisets whose dependent indices are not equal by using the resulting subtyping relation (discussed below) to coerce their types into agreeing. Refer to the supplementary material for details of this modification.

Overview. DFuzz uses a *modal* operator $!_k$ from linear logic to keep track of the *sensitivity* of a value. A function of type $!_k\sigma \multimap \tau$ is k-sensitive in its first argument, and a *typing context* containing the assumption $x :!_k \tau$ indicates the typing context can type expressions that are at most k-sensitive in x. We will simplify the syntax of these statements, and instead write them as $\sigma \multimap_k \tau$ and $x :_k \tau$. Furthermore, we use the traditional arrow $\sigma \to \tau$ for ∞-sensitive functions (i.e., $\sigma \multimap_\infty \tau$).

Syntax. DFuzz types, context, and expressions are constructed from the grammar in Fig. 4, which we detail below:

- **Sensitivity and size:** Sensitivity and size expressions (R and S) consist of variables (k and i), constants (0, c, and ∞), and simple arithmetic. These expressions are used in (*i*) modal types and (*ii*) as constraints on our *precise types*.
- **Precise types:** Precise types are types *dependent* on sensitivity and size expressions. $\mathbb{R}[R]$ and $\mathbb{N}[S]$ are the reals and naturals whose value is precisely R or S; $\text{ms}(\tau)[S]$ are multisets with *at most* S elements of type τ, and $L(\tau)[S]$ are lists with precisely S elements of type τ.
- **Probability monad:** Types of probabilistic values of type τ are written using a *monadic type* $\bigcirc\tau$. Values can be lifted to probabilistic values using `return` e, and sampled from probabilistic values using `let-draw` $x = e_1$ `in` e_2 (where x is drawn from distribution e_1 and used in e_2).

[1] Formally, $\text{Lap}\,(y)$ is noise drawn from the distribution with the probability density function $g(x) = \frac{1}{2y} \exp(-|x|/y)$.

Sensitivity and size expressions

k is a *sensitivity variable*, i is a *size variable*, and t is a *type variable*

$$R \coloneqq k \mid c \in \mathbb{R}^{>0} \mid S \quad \text{(sensitivity expression)}$$
$$\mid R + R \mid R \cdot R \mid \infty$$
$$S \coloneqq i \mid 0 \mid S + 1 \mid \infty \quad \text{(size expression)}$$

Linear dependent types

$$\tau \coloneqq a \mid Z \mid \alpha \mid A \multimap \tau \mid \langle \tau, \tau \rangle \mid \bigcirc \tau \quad \text{(types)}$$
$$A \coloneqq !_R \tau \quad \text{(modal types)}$$
$$\alpha \coloneqq \mathbb{R} \mid \text{bool} \mid \ldots \quad \text{(base types)}$$
$$Z \coloneqq \mathbb{R}[R] \mid \mathbb{N}[S] \mid L(\tau)[S] \mid \texttt{ms}(\tau)[S] \quad \text{(precise types)}$$
$$a \coloneqq \forall n.\, \tau \quad \text{(quantifiers)}$$
$$n \coloneqq k \mid i \mid t \quad \text{(kinds of variables)}$$

Constraints and typing context

$$\Phi \coloneqq \top \mid \Phi \wedge \Phi \quad \text{(constraints)}$$
$$\mid S = S \mid R \leqslant R$$
$$\Gamma \coloneqq \emptyset \mid \Gamma, x : A \quad \text{(typing context)}$$

Programs

$$e \coloneqq x \mid f \in \Sigma \quad \text{(expressions)}$$
$$\mid e\ e \quad \text{(application)}$$
$$\mid \lambda x.\, e \mid \texttt{fix}\ x.\, e \quad \text{(abstraction and fixpoints)}$$
$$\mid \texttt{return}\ e \mid \texttt{let-draw}\ x = e\ \texttt{in}\ e \quad \text{(return and bind for } \bigcirc\text{)}$$
$$\mid \texttt{match}_{\mathbb{N}}\ e\ \texttt{with}\ 0 \to e \mid x \to e \quad \text{(pattern-matching)}$$
$$\mid \texttt{match}_{L}\ e\ \texttt{with}\ \texttt{nil} \to e \mid \texttt{cons}\,(x, y) \to e \quad \text{(pattern-matching)}$$
$$\mid e[R] \mid e[\tau] \quad \text{(sens./size and type app.)}$$
$$\mid \Lambda k.\, e \mid \Lambda i.\, e \mid \Lambda t.\, e \quad \text{(sens., size, and type abs.)}$$

Fig. 4. Grammars of DFuzz types and programs

- **Dependent pattern-matching:** Precise types $\mathbb{N}[S]$ and $L(\tau)[S]$ are eliminated using dependent pattern-matching expressions, which use information about the constructors of the precise types to constrain the value of the size term. For example, if we match a precise natural with the constructor 0, we know $S = 0$.
- **Quantifiers:** Quantifiers ($\forall$) bind size, sensitivity, and type variables. We restrict type quantifiers to be *predicative*, but let size and sensitivity quantifiers appear anywhere. We use $\mathit{free}(\tau)$ to denote free variables in τ.
- **Contexts and constraints:** A typing context Γ maps variables to modal types. Our typing judgements will depend on constraints Φ on sensitivity variables.[2] For instance, Φ may specify that $k \leqslant 5$, where k is a sensitivity variable. We use $\textsc{Sat}\,(\Phi)$ to denote that a constraint Φ is satisfiable. A constraint Φ is a conjunction of inequalities over integers and reals, with (*i*) non-linear arithmetic and (*ii*) ∞, following the semantics of de Amorim et al. [2017].[3]
- **Expressions:** We have a simple expression language over variables x and primitives f from a signature Σ, which we call our *synthesis domain*. We also allow for term-, type-, and sensitivity-abstractions and applications as in System F, and recursion using $\texttt{fix}\ x.\, e$.

Typing and Context Arithmetic. DFuzz provides a typing judgement of the form $\Phi, \Gamma \vdash e : \tau$, specifying that e is of type τ under context Γ and assuming constraint Φ holds. We highlight the crucial typing rule—function application—and leave the rest to Gaboardi et al. [2013].

$$\frac{\Phi, \Gamma \vdash f : \sigma \multimap_R \tau \qquad \Phi, \Delta \vdash e : \sigma}{\Phi, \Gamma + R \cdot \Delta \vdash f\ e : \tau}\ (\multimap\text{-}\textsc{Elim.})$$

Observe the typing context arithmetic in the consequent; DFuzz defines *context scaling* and *addition* as a generalization of context union. This arithmetic encodes the fact that sensitivities multiply through function composition in the type system. Formally:

[2]For simplicity, we elide *kinding* contexts and explicitly maintaining kinds of variables, when clear from context.
[3]Multiplying by ∞ is non-commutative: $r \cdot \infty = \infty$, but $\infty \cdot r = 0$ when $r = 0$, and $\infty \cdot r = \infty$ for $r > 0$.

Definition 3 (Context arithmetic). *Let Γ and Δ be typing contexts where, for all variables x, if $(x :_T \sigma) \in \Gamma$ and $(x :_{T'} \tau) \in \Delta$, then $\sigma = \tau$. For sensitivity expressions R, we define:*

$$\begin{aligned}\Gamma + R \cdot \Delta = \{x :_{T+R\cdot T'} \sigma \mid (x :_T \sigma) \in \Gamma,\ (x :_{T'} \sigma) \in \Delta\} \\ \cup \{x :_T \sigma \mid (x :_T \sigma) \in \Gamma,\ x \notin dom(\Delta)\} \\ \cup \{x :_{R\cdot T'} \sigma \mid (x :_{T'} \sigma) \in \Delta,\ x \notin dom(\Gamma)\}\end{aligned}$$

Example 1 (Context arithmetic). Let f be the function $\lambda x : \mathbb{R}.\, 2 \cdot x$ and assume some context Γ gives us the judgement $\top, \Gamma \vdash f : \mathbb{R} \multimap_2 \mathbb{R}$. To apply f to a variable $y : \mathbb{R}$, we can apply ($\multimap$ –Elim) to the natural judgement $\top, \{y :_1 \mathbb{R}\} \vdash y : \mathbb{R}$, producing the consequent $\top, \Gamma + 2 \cdot \{y :_1 \mathbb{R}\} \vdash f\ y : \mathbb{R}$. By Def. 3, $\Gamma + 2 \cdot \{y :_1 \mathbb{R}\} = \Gamma \cup \{y :_2 \mathbb{R}\}$, assuming y is not bound in Γ. ■

Subtyping. DFuzz defines a notion of subtyping using the judgement $\Phi; \Gamma \models \sigma \sqsubseteq \tau$, where the constraint Φ determines whether the relations between sensitivity variables are appropriate to subtype in context Γ. This judgement is defined by a set of inference rules, two of which we present here. The rule

$$\frac{\Phi \models r' \leqslant r \quad \Phi; \Gamma \models \sigma \sqsubseteq \sigma'}{\Phi; \Gamma \models !_r \sigma \sqsubseteq !_{r'} \sigma'}\ (\sqsubseteq .!_r)$$

states that we can replace usage of an r'-sensitive σ' with an r-sensitive σ, as long as we do not decrease the sensitivity (or $r' \leqslant r$). As subtyping functions is standard (contravariant in the domain and covariant in the codomain), with $\sqsubseteq$-reflexivity we can encode the fact that a c-sensitive function is also c'-sensitive when $c \leqslant c'$ in the subtyping system with the valid judgement:

$$R \leqslant T \ \models\ \sigma \multimap_R \tau \sqsubseteq \sigma \multimap_T \tau$$

To subtype databases with size terms, we use the rule

$$\frac{\Phi \models S \leqslant T \quad \Psi; \Gamma \models \sigma \sqsubseteq \tau}{\Phi \wedge \Psi; \Gamma \models \mathtt{ms}(\sigma)[S] \sqsubseteq \mathtt{ms}(\tau)[T]}\ (\sqsubseteq .\mathtt{ms})$$

which allows us to weaken the type of a database by *increasing* the size bound. In the supplementary materials, we show how this rule integrates with DFuzz's metric preservation claims.

3.3 Formalizing the Synthesis Problem

We are now equipped to formalize our synthesis problem.

Definition 4 (Synthesis problem). *A synthesis problem is a tuple $\mathcal{S} = \langle \sigma, \Sigma, \phi_k, \phi_s \rangle$, where*

- *σ is a goal type with free$(\sigma) = \{k\}$,*
- *Σ, the synthesis domain, is a signature containing user-provided components and deterministic alternatives (see Def. 5),*
- *ϕ_k, the budget specification, is a formula constraining a sensitivity variable k, and*
- *ϕ_s, the functional specification, is a formula specifying semantics of the solution.*

A program p is a solution to $\mathcal{S}$ if the sensitivity of p satisfies ϕ_k. For instance, $\phi_k := k \leqslant 1$ imposes an upper bound on sensitivity. Additionally, p should be well-formed, well-typed, and should satisfy the functional specification ϕ_s. If p is a randomized program—that is, p uses components that compute probabilistic values—we cannot check satisfaction of ϕ_s. Instead, we require that a canonical deterministic representative of p satisfies ϕ_s—i.e., p without adding random noise.

To construct a deterministic representative, we assume that we have *deterministic alternatives* c_d for all randomized components $c \in \Sigma$. These alternatives return probability distributions that place support on a single value, and must have the same domain and codomain, *modulo sensitivity annotations*: to ensure the program type-checks when components are replaced by their

Table 1. Deterministic alternatives. Note that sensitivity variables are free and not used elsewhere.

Function Description	Deterministic Alternative
$\texttt{Bern}(p)$	$\texttt{Bern}_d : [0,1] \to \bigcirc\text{bool} = \lambda p.\ \texttt{return}\ (p > 1/2)$ *returns true with probability p, false with probability $1-p$*
$\texttt{Laplace}(x)$	$\texttt{Laplace}_d : \mathbb{R} \multimap_s \bigcirc\mathbb{R} = \lambda x.\ \texttt{return}\ x$ *samples from Laplace distribution with mean x*
$\texttt{ExpMech}^S(u, d)$	$\texttt{ExpMech}_d^S : (\text{ms(row)}[\infty] \multimap_s S \to \mathbb{R}) \to \text{ms(row)}[\infty] \multimap_s \bigcirc S = \lambda u.\ \lambda d.\ \text{argmax}_{s \in S} u(d, s)$ *constructs distribution over and samples from S using utility function u - see §5*

deterministic alternatives, every sensitivity annotation in the type of the alternative is replaced by a *free* and unique sensitivity variable. Semantically, deterministic alternatives return the *most likely output* from the original distribution. For example, the function `Laplace`, which samples from the Laplace distribution, has a deterministic alternative $\texttt{Laplace}_d$ that returns the mean of the distribution. Further examples are given in Table 1.

Using alternatives, we can define a notion of *deterministic satisfaction*:

Definition 5 (Deterministic satisfaction)**.** *A randomized program p using components from synthesis domain Σ* deterministically satisfies *ϕ_s, written $\Sigma \vdash_d p \models \phi_s$, if $p_d \models \phi_s$, where p_d is the deterministic program derived by replacing all uses of components $c \in \Sigma$ with their deterministic alternatives c_d.*

As $\Sigma \vdash_d p \models \phi_s$ is equivalent to $p \models \phi_s$ when p is deterministic, we will also write $p \models \phi_s$ for randomized programs when Σ is clear from context.

Deterministic satisfaction does not ensure *all* executions of a randomized program p satisfy ϕ_c. Rather, as DFuzz models probabilistic states by distributions with finite support, we are simply ensuring that the executions where $p \models \phi_c$ classically have non-zero support. If we further assume that all randomized components operate independently (which may not always be the case, due to control-flow effects), we get a slightly stronger property: *the most likely execution* of p satisfies ϕ_c.

We can now state the requirements on p formally:

Definition 6 (Synthesis solution)**.** *A program p is a* solution *to $\mathcal{S} = \langle \sigma, \Sigma, \phi_k, \phi_s \rangle$ if*

(1) $\Sigma \vdash_d p \models \phi_s$,
(2) p is a valid expression over Σ, and
(3) there is a model M inducing the interpretation $[\![\cdot]\!]_M$ over contexts, expressions, and types that replaces free sensitivity variables with constants such that

$$M \models \phi_k \quad \textit{and} \quad [\![\Sigma]\!]_M \vdash [\![p]\!]_M : [\![\sigma]\!]_M$$

The following example gives a concrete synthesis problem and its solution:

Example 2. Consider the synthesis problem $\langle \mathbb{R} \multimap_k \mathbb{R},\ \Sigma,\ \phi_k,\ \phi_s \rangle$, where

$$\phi_k \coloneqq 0 < k \leqslant 2 \quad \text{and} \quad \phi_s \coloneqq p(0) = 1 \wedge p(2) = 5$$

and the synthesis domain is

$$\Sigma = \{\texttt{square} : \mathbb{R} \multimap_\infty \mathbb{R},\ \texttt{double} : \mathbb{R} \multimap_2 \mathbb{R},\ \texttt{succ} : \mathbb{R} \multimap_1 \mathbb{R}\}$$

A solution is $p = \lambda x.\ \texttt{succ}\ (\texttt{double}\ x) : \mathbb{R} \multimap_k \mathbb{R}$, with witnessing model M setting k to 2. ■

4 SYNTHESIS WITH LINEAR DEPENDENT TYPES

We are now ready to present *sensitivity-directed* synthesis.

Overview. Our synthesis algorithm operates in a top-down fashion, iteratively refining an incomplete program into a complete one that satisfies the specification and budget constraints. The process is enabled and guided by two key ideas:

- **Symbolic context constraints** (scc): A scc succinctly captures all possible typing contexts that can type a program. This allows us to symbolically type-check incomplete programs and prune the search space when the sensitivity budget is insufficient.
- **Constraint abduction**: As discussed in §3.2, DFuzz's subtyping judgement relies on constraints over sensitivity and size variables. During synthesis, to replace a wildcard of type τ with an expression of type $\sigma \sqsubseteq \tau$, we must ask "*what constraints do we need to ensure subtyping works?*" This is a *logical abduction* question. We present a technique to abduce the *most general subtyping constraints*.

Search Structure. Our algorithm is defined by a set of inference rules that let us derive *synthesis states* from a *synthesis state*. A synthesis state $\langle \phi,\ p \rangle$ is a pair consisting of: (*i*) a proof obligation ϕ determining when p is well-typed and satisfies the privacy budget, and (*ii*) an expression p from the program grammar extended with the construction $\bullet_\tau^\Omega$, a *wildcard* maintaining a goal type τ and a *symbolic typing context* Ω (defined shortly in §4.1). A wildcard represents an incomplete, or unrefined, expression. We say an expression is *closed* if it contains no wildcards.

A program can contain multiple wildcards, and all are considered distinct. In our inference rules, we will use the notation $p\,[\bullet]$ to denote the program p containing a single occurrence of the wildcard $\bullet$. We will then use $p\,[e]$ to denote p with the wildcard replaced by expression e.

Invariants. Our algorithm maintains the invariant that if, for some synthesis state $\langle \phi,\ p \rangle$, ϕ is unsatisfiable, then for every synthesis state $\langle \phi',\ p' \rangle$ derived from $\langle \phi,\ p \rangle$, formula ϕ' is also unsatisfiable. This allows us to discontinue inference from unsatisfiable subproblems, as it means they cannot satisfy sensitivity budget constraints.

4.1 Inference Rules and Symbolic Constraints

We now detail our algorithm's inference rules (Fig. 5). Implementation details are left to §6.

4.1.1 Initialization and Termination. The rule Init initiates the synthesis process: starting with the synthesis problem $\langle \sigma, \Sigma, \phi_k, \phi_s \rangle$, we build the synthesis state $\left\langle \phi_k \wedge \Omega = \Sigma,\ \bullet_\sigma^\Omega \right\rangle$ indicating that we want an expression that satisfies the sensitivity requirements of ϕ_k and is of type σ. The functional specification ϕ_s is uniform across all synthesis states derivable from the same synthesis problem, and so we do not explicitly propagate the functional specification. Σ is the typing context $\{c :_\infty \tau\}_{c:\tau \in \Sigma}$ of all components in the synthesis domain Σ.

Rule Finish encodes the definition of a solution to the synthesis problem (Def. 6): a solution p must be closed and must deterministically satisfy the functional specification ϕ_s. Furthermore, the proof obligation ϕ must be satisfiable—our inference rules maintain the invariant that when ϕ is satisfiable, the expression p is well-typed and obeys the sensitivity constraint ϕ_k. In section §4.4 we formalize this notion.

4.1.2 Symbolic Context Constraints. The inference rules can be viewed as inversions of the rules defining the DFuzz typing judgement. Consider the rule App applied to $\left\langle \phi,\ p\left[\bullet_\sigma^\Omega\right] \right\rangle$. App is an inversion of the DFuzz rule for $\multimap$-elimination, from §3.2, which specifies that an expression of type σ can be generated if we have (for some choice of τ)

(1) an expression of type $\tau \multimap_R \sigma$ in some context Ω_1, and
(2) an expression of type τ in some context Ω_2,

where $\Omega = \Omega_1 + R \cdot \Omega_2$. In order to invert this rule to construct App, we *must know how to split* Ω; otherwise, we cannot construct the appropriate wildcards, which track contexts.

Initialization and termination

$$\frac{\langle \sigma, \Sigma, \phi_k, \phi_s \rangle \quad \Omega \text{ fresh}}{\langle \phi_k \wedge \Omega = \Sigma,\ \bullet^{\Omega}_{\sigma} \rangle}\ \textsc{Init} \qquad \frac{\langle \phi,\ p \rangle \quad \textsc{Sat}(\phi) \quad p \models \phi_s \quad p \text{ closed} \quad \textsc{Term}(p)}{\text{return } p}\ \textsc{Finish}$$

Application, abstraction, and variable/function introduction

$$\frac{\langle \phi,\ p\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \Omega_1, \Omega_2, r \text{ fresh}}{\left\langle \phi \wedge \Omega = \Omega_1 + r \cdot \Omega_2,\ p\left[\bullet^{\Omega_1}_{\tau \multimap_r \sigma}\ \bullet^{\Omega_2}_{\tau}\right] \right\rangle}\ \textsc{App} \qquad \frac{\langle \phi,\ p\left[\bullet^{\Omega}_{\tau \multimap_r \sigma}\right] \rangle \quad \Omega_1 \text{ is fresh}}{\left\langle \phi \wedge \Omega_1 = \Omega \oplus \{x :_r \tau\},\ p\left[\lambda x : \tau .\ \bullet^{\Omega_1}_{\sigma}\right] \right\rangle}\ \textsc{Abs}$$

$$\frac{\langle \phi,\ p\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \sigma \prec_{[t/t']} \tau \quad t \text{ is fresh}}{\langle \phi,\ p\left[\bullet^{\Omega}_{\forall t . \tau}[t']\right] \rangle}\ \textsc{TApp} \qquad \frac{\langle \phi,\ p\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \gamma; \psi \vdash \tau \leftsquigarrow \sigma \quad v : \tau \in \textsc{Scope}(\phi, \Omega)}{\langle \phi \wedge \gamma \wedge \psi \wedge \Omega = \{v :_1 \tau\},\ p[v] \rangle}\ \textsc{Id}$$

Recursion and pattern-matching

$$\frac{\langle \phi,\ p\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \Omega_1, \Omega_2 \text{ fresh}}{\left\langle \phi \wedge \Omega = \infty \cdot \Omega_2 \wedge \Omega_1 = \Omega_2 \oplus \{x :_{\infty} \sigma\},\ p\left[\texttt{fix}\, x .\ \bullet^{\Omega_1}_{\sigma}\right] \right\rangle}\ \textsc{Fix}$$

$$\frac{\langle \phi,\ p\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \Omega_1, \Omega_2, \Omega_3, \Omega_4, r \text{ fresh}}{\left\langle \begin{array}{l} \phi \wedge \Omega = \Omega_4 + r \cdot \Omega_1 \wedge \Omega_2 = \Omega_4[s/0] \\ \quad \wedge \Omega_3 = \Omega_4[s/i+1] \oplus \{x :_r \mathbb{N}[i]\},\ p\left[\texttt{match}_{\mathbb{N}}\ \bullet^{\Omega_1}_{\mathbb{N}[s]}\ \texttt{with}\ 0 \to \bullet^{\Omega_2}_{\sigma[s/0]} \mid x[i]+1 \to \bullet^{\Omega_3}_{\sigma[s/i+1]}\right] \end{array} \right\rangle}\ \textsc{Match}_{\mathbb{N}}$$

$$\frac{\langle \phi,\ p\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \Omega_1, \Omega_2, \Omega_3, r \text{ fresh}}{\left\langle \begin{array}{l} \phi \wedge \Omega = \Omega_1 + r \cdot \Delta \wedge \Omega_2 = \Omega_1[s/0] \\ \quad \wedge \Omega_3 = \Omega_1[s/i+1] \oplus \{y :_r \tau, x :_r L(\tau)[i]\},\ p\left[\texttt{match}_L\ \bullet^{\Omega_1}_{L(\tau)[s]}\ \texttt{with}\ \texttt{nil} \to \bullet^{\Omega_2}_{\sigma[s/0]} \mid \texttt{cons}\,(y, x[i]) \to \bullet^{\Omega_3}_{\sigma[s/i+1]}\right] \end{array} \right\rangle}\ \textsc{Match}_{L}$$

Monadic operations

$$\frac{\left\langle \phi,\ p\left[\bullet^{\Omega}_{\bigcirc \sigma}\right] \right\rangle \quad \Omega_1 \text{ fresh}}{\left\langle \phi \wedge \Omega = \infty \cdot \Omega_1,\ p\left[\texttt{return}\ \bullet^{\Omega_1}_{\sigma}\right] \right\rangle}\ \textsc{Return} \qquad \frac{\left\langle \phi,\ p\left[\bullet^{\Omega}_{\bigcirc \sigma}\right] \right\rangle \quad \Omega_1, \Omega_2, \Omega_3 \text{ fresh}}{\left\langle \begin{array}{l} \phi \wedge \Omega = \Omega_1 + \Omega_3 \\ \quad \wedge \Omega_2 = \Omega_3 \oplus \{x :_{\infty} \tau\},\ p\left[\texttt{let-draw}\ x = \bullet^{\Omega_1}_{\bigcirc \tau}\ \texttt{in}\ \bullet^{\Omega_2}_{\bigcirc \sigma}\right] \end{array} \right\rangle}\ \textsc{LetDraw}$$

Fig. 5. Synthesis inference rules. The relation $\prec_{\gamma}$ is *generalizes* (Def. 8), and $\textsc{Term}(\cdot)$ is a termination oracle. $\textsc{Scope}(\phi, \Omega)$ returns the set $\{x_i : \tau_i\}$ such that, for all models $M \models \phi$, for all i we have $M \models x_i : \tau_i \in \Omega$.

Unfortunately, there are infinitely-many choices for contexts Ω_1, Ω_2, and sensitivity expression R. To allow us to express such arbitrary context splits in a finite manner, we introduce *symbolic context constraints* (scc), which succinctly encode infinitely many contexts symbolically.

Definition 7 (Symbolic Context Constraints)**.** *A* symbolic context *is a term C in the grammar*

$$C := \emptyset \mid \{x :_R \tau\} \mid \Omega \mid C + R \cdot C \mid C \oplus C \mid C[s/R],$$

where Ω is a context variable, R is a sensitivity term, τ is a type, and s is a sensitivity variable.

A symbolic context constraint *E is a conjunction of equalities of symbolic contexts, or equivalently, a term in the grammar $E := E \wedge E \mid C = C$.*

The grammar in Def. 7 defines symbolic contexts C, which are expressions containing *symbolic context variables* Ω and a limited set of concrete contexts, namely the empty context $\emptyset$ and the singleton context $\{x :_R \tau\}$. Symbolic contexts can be constructed through linear combinations $C + R \cdot C$, disjoint unions $C \oplus C$ that require the symbolic contexts have no variables in common,

and sensitivity substitutions $C\,[s/R]$ which replace sensitivity variables (s) with arbitrary sensitivity expressions (R). Symbolic context constraints E are conjunctions of equalities over symbolic contexts. An interpretation of a SCC E maps symbolic context variables to concrete contexts. We formalize the interpretation of these constraints in §4.3.

SCCs also appear in the inference rule Abs, which introduces a λ abstraction. The rule specifies that we can construct a function of type $\sigma \multimap_r \tau$ in context Ω, but only if the context in the wildcard $\bullet_\tau^{\Omega_1}$ contains r uses of x. This is expressed in the constraint $\Omega_1 = \Omega \oplus \{x :_r \sigma\}$, which not only ensures that Ω_1 has r uses of x, but that Ω has *no* uses of x (as x is out of scope).

Example 3 (SCCs)**.** Consider the initial synthesis state

$$\left\langle 0 < k \leqslant 2,\ \ \bullet^{\emptyset}_{\mathbb{R} \multimap_k \mathbb{R}} \right\rangle.$$

After applying Abs, we derive the synthesis state

$$\left\langle 0 < k \leqslant 2 \wedge \Omega = \emptyset \oplus \{x :_k \mathbb{R}\},\ \ \lambda x : \mathbb{R}.\, \bullet^{\Omega}_{\mathbb{R}} \right\rangle$$

indicating that the context Ω must contain k copies of x. Then, after applying the rule App, we derive the following synthesis state

$$\left\langle \phi,\ \ \lambda x : \mathbb{R}.\, \bullet^{\Omega_1}_{\mathbb{R} \multimap_l \mathbb{R}}\ \bullet^{\Omega_2}_{\mathbb{R}} \right\rangle$$

where $\phi = 0 < k \leqslant 2 \wedge \Omega = \emptyset \oplus \{x :_k \mathbb{R}\} \wedge \Omega = \Omega_1 + l \cdot \Omega_2$, indicating that if we wish to apply an l-sensitive function (where l is fresh), Ω must contain l uses of variables in context Ω_2. ■

Due to our maintained invariant, an unsatisfiable subproblem can be pruned from the search, as it yields no solutions. The following example illustrates pruning:

Example 4 (Pruning)**.** Let us start from the last synthesis state in Ex. 3. Suppose we replace the $\bullet^{\Omega_1}_{\mathbb{R} \multimap_l \mathbb{R}}$ with the ∞-sensitive function `square` (which squares a real number). Using the rule Id, this results in the subproblem with the following constraints (where we have replaced Ω_1 with $\emptyset$):

$$(0 < k \leqslant 2) \wedge (\Omega = \{x :_k \mathbb{R}\}) \wedge (\Omega = \emptyset + l \cdot \Omega_1) \wedge l \geqslant \infty$$

This simplifies to $0 < k \leqslant 2 \wedge \{x :_k \mathbb{R}\} = \infty \cdot \Omega_2$, which is unsatisfiable: the context on the right has ∞ copies of x, while the left as at most 2 copies. ■

4.1.3 Pattern Matching. Pattern-matching over the precise types $\mathbb{N}[S]$ and $L(\tau)[S]$, introduced using rules $\textsc{Match}_{\mathbb{N}}$ and $\textsc{Match}_L$, gives a mechanism for concretizing our understanding of the sensitivity of an expression by *refining* the value of S. For instance, if we match expression e of type $\mathbb{N}[s]$ with the precise natural constructor 0, in the 0-branch wildcard we may freely assume that $s = 0$. This assumptions is propagated in two ways: (*i*) by substituting all instances of s with 0 in the type annotation of the wildcard, and (*ii*) by restricting the wildcard context via the constraint $\Gamma_1 = \Gamma\,[s/0]$, which requires that Γ_1 is simply Γ with all instances of s replaced by 0.

4.1.4 Recursion. The Fix rule introduces fixpoint expressions $\texttt{fix}\, x.\, e$, allowing for the construction of recursive programs. We must allow for unlimited uses of the recursion variable in e, as constrained by the conjunct $\Omega_1 = \Omega_2 \oplus \{x :_\infty \sigma\}$ in the consequent, and for potentially infinitely-many recursive expansions of e, restricting our context to be infinitely-sensitive with respect to all variables used in e (as expressed by $\Omega = \infty \cdot \Omega_2$).

Recursion can, of course, produce non-terminating programs. We assume the existence of a termination oracle, $\textsc{Term}(p)$, which returns *true* if p terminates for all inputs. In practice, our oracle is a procedure that soundly identifies terminating recursive expansions using the natural well-founded order over the types $\mathbb{N}[S]$ and $L(\tau)[S]$.

4.1.5 Probability Monad. The rules Return and LetDraw are used to enable synthesis of randomized programs. Both rules have monadic types in their antecedent synthesis states, and so the type-directed nature of synthesis ensures that rules constructing probability distributions are only used when a probability distribution is required.

The expression `return` e lifts into the probability monad. As an inversion, rule Return provides a mechanism to construct distributions with support for a single value. The expression generated is ∞-sensitive with respect to all variables in Ω_1 (captured by the constraint $\Omega = \infty \cdot \Omega_1$) because *close values do not result in close distributions.* That is, `return` 1 and `return` 2 are infinitely far apart in the distribution metric, despite 1 and 2 being only distance 1 away. [4]

The rule LetDraw introduces the DFuzz mechanism for *sampling* from distributions, the `let-draw` $x = e_1$ `in` e_2 expression. LetDraw enables synthesis to perform additional computations on sampled values by using x in e_2. Note that e_2 has unrestricted use of x, encoded in the constraints by the conjunct $\Gamma_2 = \Gamma_3 \oplus \{x :_\infty \tau\}$, as once a random value is sampled no post-computation can give any more information about the distribution than that one point.

4.1.6 Polymorphism. The rules TApp, SenApp, and SizeApp are used to enable applying polymorphic functions in our signature, e.g., `map`, by abstracting types, sensitivities, and sizes. We only show TApp: the others are structurally identical.

Each of the three rules operates similarly: given a synthesis state with wildcard $\bullet^{\Omega}_{\sigma}$, they attempt to replace the goal type σ with a *polymorphic* type $\forall n.\, \tau$. To do so, it is necessary to *generalize* the goal type σ to introduce a free variable.

Definition 8 (Generalization)**.** *Let γ be a type-variable substitution. We say a type τ* type-generalizes *a type σ using γ if $\tau = \gamma(\sigma)$ and $dom(\gamma) \subseteq free(\tau)$. We represent this as $\sigma \prec_\gamma \tau$.*
Sensitivity- and size-generalization are defined symmetrically.

In practice, we explore all generalizations, of which there are linearly-many in the size of σ.

Consider the rule TApp: it refines a wildcard $\bullet^{\Omega}_{\sigma}$ by finding a type τ with a free *type* variable t such that $\sigma \prec_{[t/t']} \tau$. This reduces the search to finding a refinement for the wildcard $\bullet^{\Omega}_{\forall t.\tau}$, possibly enabling the introduction of a polymorphic function (such as `map`) using rule Id.

Example 5. Suppose we have a synthesis state $\left\langle \phi,\ \bullet^{\Omega}_{\mathsf{ms(row)}[\infty] \multimap_2 \mathbb{R}} \right\rangle$, and we wish to apply the 1-sensitive polymorphic function `count` : $\forall \beta.\, \forall k.\, \mathsf{ms}(\beta)[k] \multimap_1 \mathbb{R}$. Since

$$\mathsf{ms(row)}[\infty] \multimap_2 \mathbb{R} \ \prec_{[\alpha/\mathsf{row}]}\ \mathsf{ms}(\alpha)[\infty] \multimap_2 \mathbb{R} \ \prec_{[s/\infty]}\ \mathsf{ms}(\alpha)[s] \multimap_2 \mathbb{R},$$

using rules TApp and SizeApp we can *type-generalize* and *size-generalize* $\mathsf{ms(row)}[\infty] \multimap_2 \mathbb{R}$ and generate the synthesis state

$$\left\langle \phi,\ \bullet^{\Omega}_{\forall \alpha.\, \forall s.\, \mathsf{ms}(\alpha)[s] \multimap_2 \mathbb{R}}[\mathsf{row}][\infty], \right\rangle$$

where the inner application is *type-application* and the outer application is *size-application.* An application of the rule Id results in the subtyping abduction rules (§4.2) generating the judgement

$$\top; 1 \leqslant 2 \vdash \forall \beta.\, \forall k.\, \mathsf{ms}(\beta)[k] \multimap_1 \mathbb{R} \leftsquigarrow \forall \alpha.\, \forall s.\, \mathsf{ms}(\alpha)[s] \multimap_2 \mathbb{R},$$

allowing us to replace the wildcard with `count`. ∎

[4] For a fixed ϵ, define $d_{\bigcirc\tau}(\delta_1, \delta_2)$ to be $1/\epsilon \cdot \max_{x \in \tau} |\ln(\delta_1(x)/\delta_2(x))|$. The metric is defined precisely to ensure DFuzz's *metric preservation theorem* guarantees privacy.

$$\frac{v \notin t \quad \textsc{SensFree}(t) = \emptyset}{v = t; \top \vdash_{\{v\}} v \leftsquigarrow t}\ \textsc{LVar} \qquad \frac{v \notin t \quad \textsc{SensFree}(t) = \emptyset}{v = t; \top \vdash_{\{v\}} t \leftsquigarrow v}\ \textsc{RVar} \qquad \frac{}{\top; \top \vdash_\emptyset t \leftsquigarrow t}\ \textsc{Refl}$$

$$\frac{}{\top; S = S' \vdash_\emptyset \mathbb{N}[S] \leftsquigarrow \mathbb{N}[S']}\ \textsc{Nat} \qquad \frac{\gamma; \psi \vdash_A \sigma \leftsquigarrow \tau}{\gamma; \psi \wedge S = S' \vdash_A L(\sigma)[S] \leftsquigarrow L(\tau)[S']}\ \textsc{List}$$

$$\frac{\gamma; \psi \vdash_A \sigma \leftsquigarrow \tau}{\gamma; \psi \wedge S' \leqslant S \vdash_A \mathrm{ms}(\sigma)[S] \leftsquigarrow \mathrm{ms}(\tau)[S']}\ \textsc{MSet} \qquad \frac{\gamma; \psi \vdash_A \sigma \leftsquigarrow \tau}{\gamma; \psi \wedge S' \leqslant S \vdash_A !_S \sigma \leftsquigarrow !_{S'} \tau}\ \textsc{Modal}$$

$$\frac{\gamma; \psi \vdash_A \sigma_2 \leftsquigarrow \sigma_1 \quad \delta; \phi \vdash_B \tau_1 \leftsquigarrow \tau_2}{\gamma \wedge \delta; \psi \wedge \phi \vdash_{A \cup B} \sigma_1 \multimap \tau_1 \leftsquigarrow \sigma_2 \multimap \tau_2}\ \textsc{Arrow} \qquad \frac{\gamma; \psi \vdash_A \sigma \leftsquigarrow \tau}{\gamma; \psi \vdash_A \bigcirc \sigma \leftsquigarrow \bigcirc \tau}\ \textsc{Monad}$$

$$\frac{\gamma; \psi \vdash_A \sigma\,[\alpha/\rho] \leftsquigarrow \tau\,[\beta/\rho] \quad \rho \notin A}{\gamma; \psi \vdash_A \forall \alpha.\, \sigma \leftsquigarrow \forall \beta.\, \tau}\ \textsc{ForAll} \qquad \frac{\gamma; \psi \vdash_A \sigma \leftsquigarrow \tau}{\gamma; \psi \vdash \sigma \leftsquigarrow \tau}\ \textsc{Exit}$$

Fig. 6. Inference rules defining abduction. $\textsc{SensFree}(t)$ is the set of free sensitivity variables in t. The relation $\gamma; \psi \vdash_A \sigma \leftsquigarrow \tau$ is *avoiding abduction*, where the subscript A is a set of type variables to be avoided by quantifiers. Remaining rules are presented in the supplementary material.

4.2 Constraint Abduction

The subtyping judgement for DFuzz (of the form $\Phi, \Gamma \models \sigma \sqsubseteq \tau$) depends on a set of constraints Φ over sensitivity and size variables. During synthesis, if we require an expression of type τ, we will always be satisfied to find an otherwise acceptable expression of type $\sigma \sqsubseteq \tau$.

To appropriately apply rule Id, we must be able to *abduce* the constraint Φ and the type constraints γ under which σ is a subtype of τ. Informally, we need to answer the question: "*what constraints have to be true so that we can use σ in place of τ?*"

We present a set of inference rules in Fig. 6—derived by combining an inversion of DFuzz's subtyping rules with a unification procedure—that define an abduction judgement $\gamma; \psi \vdash \sigma \leftsquigarrow \tau$ stating that, if the type constraint γ (conjunction of equality over type variables and constructors) and the sensitivity constraint ψ hold, then we can use σ in place of τ during synthesis. More formally, we use the following interpretation: if M is an assignment over type and sensitivity variables such that $M \models \gamma \wedge \psi$, then $\top \models [\![\sigma]\!]_M \sqsubseteq [\![\tau]\!]_M$.

The definition of our abduction judgement depends in part on an auxiliary *avoiding abduction* judgement, written $\gamma; \psi \vdash_A \sigma \leftsquigarrow \tau$. A is a set of type variables that have been constrained by abduction, and appears in the antecedent in rule ForAll: to ensure the constraints abduced are the most general, we cannot universally quantify over type variables that have already been constrained.

The following theorem guarantees our abduction procedure abduces the most general constraint, which is necessary to ensure we do not miss solutions by over-constraining the proof obligation:

Theorem 4.1 (Abduction most-generality). *If $\gamma; \psi \vdash \sigma \leftsquigarrow \tau$, then for all constraints δ and ϕ such that $\delta; \phi \vdash \sigma \leftsquigarrow \tau$, the formulas $\delta \wedge \phi \Rightarrow \gamma \wedge \psi$ is valid.*

This abduction judgement appears in only one inference rule. Given a synthesis state $\langle \phi,\ p\,[\bullet_\sigma^\Omega] \rangle$, rule Id lets us replace the wildcard $\bullet_\sigma^\Omega$ with an identifier (either a function or variable in scope) of type τ under the proof obligation $\gamma \wedge \psi$ if $\gamma; \psi \vdash \tau \leftsquigarrow \sigma$. We also accumulate the obligation that

context $\Omega = \{v :_1 \tau\}$, as the expression we replace the wildcard with—the identifier v—is surely 1-sensitive with respect to v.

Example 6 (Abduction)**.** Consider the last synthesis state in Ex. 3. Replacing $\bullet^{\Omega_1}_{\mathbb{R} \multimap_l \mathbb{R}}$ with the 1-sensitive function `succ` using Id invokes abduction. Rules Arrow, Modal, and Refl abduce the constraint $\psi = 1 \leqslant l$ and the empty unification constraint $\top$, indicating that l should be at least the sensitivity of `succ`. ■

4.3 Satisfiability of Proof Obligations

We have discussed the construction of symbolic context constraints, but not their interpretation. In this section, we present a technique for checking satisfiability of SCCs .

Applying rule Finish to a subproblem $\langle \phi,\ p \rangle$ relies on checking satisfiability of ϕ, denoted Sat (ϕ). By construction, the proof obligation ϕ is of the form $\phi_d \wedge \phi_c$, where ϕ_d contains no symbolic context constraints, and ϕ_c is only over symbolic context constraints (following Def. 7).

Formula ϕ_d lies in the combined theory of (*i*) non-linear arithmetic over integers and reals extended with ∞ and (*ii*) equality of uninterpreted functions. However, ϕ_c formulas lie in our context expression language, for which there is no default first-order theory. We now demonstrate how to translate a formula ϕ_c into an *equisatisfiable* formula over arithmetic constraints.

Context Interpretations. A *model* M is a map from symbolic context variables to *concrete* contexts (containing no sensitivity variables) in the DFuzz system; consequently, M also maps sensitivity (resp., size) variables to the reals (resp., naturals). A model M *satisfies* a context constraint ϕ_c (denoted $M \models \phi_c$) if ϕ_c is true when evaluated using variable assignments from M.

Example 7 (Models)**.** Consider $\phi_c \coloneqq (\Omega = \emptyset + 2 \cdot \{x :_k \mathbb{R}\})$. Let M be a model mapping Ω to the context $\{x :_2 \mathbb{R}\}$ and k to 1. Clearly, $M \models \phi$, since the right-hand side of the equality reduces to $\{x :_2 \mathbb{R}\}$ through context arithmetic. ■

Two contexts are equal in our interpretation if (*i*) they contain the same variables and (*ii*) every variable has the same sensitivity and type in both contexts.

Translation. To check satisfiability of ϕ_c, we will transform it into a formula ψ_c over sensitivity and size variables. Let $supp(\phi_c)$ denote the *support* of ϕ_c: the set of expression variables that appear *explicitly* in ϕ_c. For example, if $\phi \coloneqq \Omega = \emptyset + 2 \cdot \{x :_k \tau\}$, then $supp(\phi) = \{x\}$. By definition, ϕ_c can only restrict the sensitivities of variables that appear in $supp(\phi_c)$, and so a translation of ϕ_c need only constrain sensitivities of variables in $supp(\phi_c)$.

We will explicitly state the constraint ϕ_c puts on all variables in the support. For each variable in the support, we compute the *symbolic sensitivity* as follows:

Definition 9 (Symbolic Sensitivity)**.** *Let $x \in supp(\phi_c)$, and let C be a symbolic context term. We define the* symbolic sensitivity of x in C under sensitivity substitution δ*—written $\Delta^{\delta}_x(C)$—recursively by:*

(1) $\Delta^{\delta}_x(\emptyset) \coloneqq \langle 0, \top \rangle$
(2) $\Delta^{\delta}_x(\Omega) \coloneqq \left\langle r^{\Omega}_x, \top \right\rangle$*, where r^{Ω}_x is a fresh sensitivity variable*
(3) $\Delta^{\delta}_x(\{x :_R \tau\}) \coloneqq \langle \delta(R), \top \rangle$
(4) $\Delta^{\delta}_x(C\,[v/s]) \coloneqq \Delta^{\delta'}_x(C)$*, where $\delta' = \delta \circ [v/s]$ and composition is right-associative*
(5) $\Delta^{\delta}_x(C_1 + R \cdot C_2) \coloneqq \langle S_1 + \delta(R) \cdot S_2, \phi_1 \wedge \phi_2 \rangle$*, where $\Delta^{\delta}_x(C_1) \coloneqq \langle S_1, \phi_1 \rangle$ and $\Delta^{\delta}_x(C_2) \coloneqq \langle S_2, \phi_2 \rangle$*
(6) $\Delta^{\delta}_x(C_1 \oplus C_2) \coloneqq \langle S_1 + S_2, \phi_1 \wedge \phi_2 \wedge (S_1 = 0 \vee S_2 = 0) \rangle$*, where $\Delta^{\delta}_x(C_1) \coloneqq \langle S_1, \phi_1 \rangle$ and $\Delta^{\delta}_x(C_2) \coloneqq \langle S_2, \phi_2 \rangle$*

$\Delta_x^\delta(C)$ is a pair $\langle S, \phi \rangle$, where S is a sensitivity expression (i.e., an arithmetic combination of sensitivity variables and constants) and ϕ is a constraint on sensitivity expressions sufficient to ensure equisatisfiability (see Theorem 4.2).

We can now lift a constraint over symbolic contexts to a constraint over symbolic sensitivities:

Definition 10 (Symbolic Sensitivity Constraint)**.** *Let $\phi_c = \bigwedge_{i=1}^n C^{l,i} = C^{r,i}$ be a constraint over symbolic contexts. The associated* symbolic sensitivity constraint*—written $\Delta(\phi_c)$—is computed as follows:*

$$\Delta(\phi_c) \coloneqq \bigwedge_{x \in supp(\phi_c)} \bigwedge_{i=1}^{n} S_x^{l,i} = S_x^{r,i} \wedge \phi_x^{l,i} \wedge \phi_x^{r,i},$$

where $\Delta_x^\iota(C^{l,i}) = \left\langle S_x^{l,i}, \phi_x^{l,i} \right\rangle$, $\Delta_x^\iota(C^{r,i}) = \left\langle S_x^{r,i}, \phi_x^{r,i} \right\rangle$, and ι is the empty substitution.

Example 8. Consider the symbolic context constraint $\phi_c \coloneqq \Omega \oplus 7 \cdot \{x :_k \tau\} = \{x :_{10} \tau\}$. We have $\Delta(\phi_c) \coloneqq r_x^\Omega + 7 \cdot k = 10 \wedge (r_x^\Omega = 0 \vee 7 \cdot k = 0)$. This formula is satisfiable, as witnessed by the model assigning $r_x^\Omega = 10$ and $k = 0$. ■

Importantly, this transformation from symbolic context constraints to symbolic sensitivity constraints *preserves satisfiability*. This gives a mechanism for checking $\textsc{Sat}(\phi_c)$ using modern SMT solvers without extensive modifications. This notion is formalized as follows:

Theorem 4.2. *Consider the constraints $\phi_d \wedge \phi_c$. We have $\textsc{Sat}(\phi_d \wedge \phi_c)$ iff $\textsc{Sat}(\phi_d \wedge \Delta(\phi_c))$.*

4.4 Soundness and Relative Completeness

Our inference rules result in a *sound* synthesis algorithm, meaning that if a program p is returned, it is guaranteed to be a solution to the synthesis problem. Formally:

Theorem 4.3 (Soundness). *Let $\mathcal{S} = \langle \sigma, \Sigma, \phi_k, \phi_s \rangle$ be a synthesis problem, and let p be an expression returned from a sequence of inference rule applications beginning with Init applied to $\mathcal{S}$ and ending with an application of Finish. Then p is a solution to the synthesis problem $\mathcal{S}$.*

Given a signature Σ, our search's inference rules generate every production in our expression grammar *except* type-, size-, and sensitivity-abstractions. Thus, our search is *relatively complete*: assuming we have an oracle for satisfiability and termination checking, our search is able to derive any solution to a synthesis problem that lies in the space of expressions with only term-abstractions and primitives from the provided signature Σ. We have the following formalism:

Theorem 4.4 (Relative completeness). *Let $\mathcal{S} = \langle \sigma, \Sigma, \phi_k, \phi_s \rangle$ be a synthesis problem, and let p be a solution of $\mathcal{S}$ with no sensitivity-, size-, or type-abstractions. Then there is a sequence of inference rule applications beginning with Init applied to $\mathcal{S}$ and ending with $\langle \phi,\ p \rangle$, with $\textsc{Sat}(\phi)$.*

5 APPLICATIONS OF OUR TECHNIQUE

In §4, we described how to solve a synthesis problem $\langle \sigma, \Sigma, \phi_k, \phi_s \rangle$. In this section, we will present two applications of our synthesis algorithm. In §5.1, we focus on synthesizing queries over sensitive databases, and in §5.2 we turn synthesis towards the problem of combining private operators to construct a differential privacy mechanism from scratch. Finally, §5.4 presents an analysis of the security model of using our synthesis algorithm for sensitive applications.

Table 2. Examples of functions in our synthesis domain and privacy mechanisms. The sensitivity ϵ is assumed fixed *a priori*. Note `partition` returns a *list* of key-value pairs, whose distance metric is the *sum* of element-wise distances for lists of the same length, and ∞ otherwise.

Function	Type
`map`	$\forall\alpha, \beta.\ \forall s.\ (\alpha \to \beta) \multimap_{2\cdot s} \text{ms}(\alpha)[s] \multimap_1 \text{ms}(\beta)[s]$
`filter`	$\forall\alpha.\ \forall s.\ (\alpha \to \text{bool}) \multimap_s \text{ms}(\alpha)[s] \multimap_1 \text{ms}(\alpha)[s]$
`partition`	$\forall\alpha, \beta.\ \forall s, n.\ \text{set}(\alpha)[n] \to (\beta \to \alpha) \multimap_s \text{ms}(\beta)[s] \multimap_2 L(\langle\alpha, \text{ms}(\beta)[s]\rangle)[n]$
`count`	$\forall\alpha.\ \forall k.\ \text{ms}(\alpha)[k] \multimap_1 \mathbb{R}$

Mechanism	Type
`lap-mech`	$\forall k.\ (\text{ms(row)}[\infty] \multimap_k \mathbb{R}) \to \text{ms(row)}[\infty] \multimap_{k\cdot\epsilon} \bigcirc\mathbb{R}$
`para-mech`	$\forall\alpha.\ \forall n, k.\ L(\alpha)[n] \to (\text{row} \to \alpha) \to (\text{ms(row)}[\infty] \multimap_k \mathbb{R}) \to \text{ms(row)}[\infty] \multimap_{k\cdot\epsilon} \bigcirc L(\langle\alpha, \mathbb{R}\rangle)[n]$
`exp-mech`	$\forall k.\ D \to (\text{ms(row)}[\infty] \multimap_k D \to \mathbb{R}) \to \text{ms(row)}[\infty] \multimap_{k\cdot\epsilon} \bigcirc D$

5.1 Data Analysis

Data analysis tasks are the main application of differential privacy, and we are interested in using synthesis to aid end-users in interfacing with sensitive data. Rather than reasoning about the complex interactions of sensitivities and information leakage, data analysts should be able to specify a DP query by providing only a *semantic specification* of the query and their *privacy budget*.

We model datasets as multisets of elements of type row, a tuple indexed by *keys*. To facilitate the construction of efficient and expressive queries, we instantiate our synthesis domain Σ with the following sets of components:

- A set of higher-order combinators—`map`, `filter`, etc.—that are prevalent in languages such as Apache Spark [Zaharia et al. 2012] and LINQ [Yu et al. 2008], as well as the DP-system PINQ [McSherry 2009].
- Aggregation operators, such as `sum`, `max`, `count`, and `average`, over multisets of numbers.
- Standard arithmetic operations and Boolean predicates.
- Dataset-dependent constants and projections to extract fields from rows.

Table 2 shows four of the combinators in our data-analysis domain. All are standard, but their type annotations provide a detailed view of their underlying privacy semantics. A strength of synthesis is that it can shield users from having to reason about such complex types.

Privacy Mechanisms. To answer queries in a differentially private manner, dataset maintainers use *privacy mechanisms* that return sensitive information in provably safe ways. Privacy mechanisms take in some specification of a query (and relevant supporting information) and construct a differentially private function from the dataset to the desired output domain. Data analysts can expect to be restricted to interacting with sensitive data via a small set of mechanisms supported by the dataset maintainer. We therefore focus on synthesizing *inputs* to privacy mechanisms.

We will briefly discuss the usage of the privacy mechanisms in Table 2. To simplify the presentation, we assume the synthesis domain Σ and the privacy parameter ϵ are determined *a priori* by the target dataset. As a further simplification, we use *input-output examples* to specify the desired query semantics: if a user presents a set $\mathcal{E}$ of pairs $\langle i_k, o_k\rangle$, we assume they desire a program p that agrees on all $\langle i, o\rangle$ pairs, i.e. $p(i) = o$. Note that, while input-output examples are a straightforward way to encode desired semantics, what follows can be adapted for other forms of specifications.

Laplace Mechanism. Recall that the Laplace mechanism (formalized in §3) is applied to programs with real-valued outputs. When given a set of examples from databases to *real numbers*, we can use the Laplace mechanism to reduce our synthesis problem to a function of type $\text{ms(row)}[\infty] \multimap_k \mathbb{R}$.

More precisely, given a set of input-output examples $\mathcal{E}$ of type $\langle \text{ms(row)}[\infty], \mathbb{R} \rangle$ and a sensitivity budget b, we can construct a synthesis problem

$$\mathcal{S} = \langle \text{ms(row)}[\infty] \multimap_k \mathbb{R},\ \ \Sigma,\ \ k \leqslant b,\ \ \forall \langle i, o \rangle \in \mathcal{E}.\, p(i) = o \rangle\,.$$

Given a program p that is a solution to $\mathcal{S}$, a dataset maintainer can apply the Laplace mechanism (via application of `lap-mech`) to answer the user's query in a privacy-preserving manner.

Parallel Composition. Sequential composition (Theorem 3.2) guarantees that, if a user applies two functions to a dataset, the incurred privacy cost is the *sum* of the costs of the two functions. This is often not the best way to guarantee DP. *Parallel composition* [McSherry 2009] is a property of DP that allows us to evaluate an ϵ-DP function on arbitrarily-many disjoint datasets with a total privacy cost of ϵ. Following PINQ [McSherry 2009], we implement parallel composition by allowing users to *partition* the multiset into disjoint subsets before analyzing each partition independently.

When given a set of examples $\mathcal{E}$ of type $\langle \text{ms(row)}[\infty], \langle \text{key}, \mathbb{R} \rangle \rangle$—maps from databases to real values indexed by *keys*—a sensitivity budget b, *a set of keys* K, and a projection $\pi : \text{row} \multimap K$ selecting the key for every row, we build the synthesis problem

$$\mathcal{S} = \langle \text{ms(row)}[\infty] \multimap_k \mathbb{R},\ \ \Sigma,\ \ k \leqslant b,\ \ \forall \langle i, o \rangle \in \mathcal{E}.\, \texttt{partition}(\pi, K, p, i) = o \rangle\,,$$

where the constraint ensures per-partition correctness. A dataset maintainer given K and a function p that is a solution to $\mathcal{S}$ can construct the query `para-mech` π K p to answer the user's request while using only ϵ of the privacy budget.

Parallel composition is more involved than Laplace mechanism, as now a user must also provide a set of keys along with their input-output examples $\mathcal{E}$. This additional information is required to preserve privacy: `para-mech` only evaluates p on partitions constructed by projecting onto a particular $k \in K$. Restricting evaluation to provided keys ensures the presence or absence of a key in the dataset is revealed in a privacy-preserving manner.

Exponential Mechanism. We are often interested in performing computations whose output is some *categorical type*, e.g., computing the most common medical condition. The mechanisms introduced so far are restricted to numerical outputs. To handle categorical outputs, we use the *exponential mechanism* [McSherry and Talwar 2007]. Instead of adding noise to the output, the exponential mechanism expects the user to provide a *utility function*. This function assigns a real-valued *utility* (or quality) to each output $d \in D$ given an input. The type of such a utility function is $\text{ms(row)}[\infty] \multimap_k D \multimap_\infty \mathbb{R}$. We leave the details of converting such a function to an ϵ-DP query to McSherry and Talwar [2007].

If a user provides a set of input-output examples $\mathcal{E}$ of type $\langle \text{ms(row)}[\infty],\ D \rangle$ and a sensitivity budget b, we can construct the synthesis problem

$$\mathcal{S} = \left\langle \text{ms(row)}[\infty] \multimap_k D \multimap_\infty \mathbb{R},\ \ \Sigma,\ \ k \leqslant b,\ \ \phi^{e}_{\mathcal{E},D} \right\rangle,$$

where the correctness constraint is $\phi^{e}_{\mathcal{E},D} \coloneqq \forall \langle i, o \rangle \in \mathcal{E}.\, \forall d \in D.\, d \neq o \Rightarrow p(i, o) > p(i, d)$, encoding the semantics that o, the *desired output*, has the highest utility of all elements in D.

Unlike the previous mechanisms, the synthesis problem encodes a function whose semantics are *not the same* as the desired query. The requirement that a user builds a utility function that captures their desired semantics while still being appropriately sensitive has prevented full adoption of the exponential mechanism in data analysis. Application of our technique can remove this burden.

5.2 Mechanism Design

While data analysts are the primary *users* of differential privacy, much research is focused on *privacy mechanism design* and implementation. Combining privacy primitives to produce new mechanisms

is non-trivial: proving even simple mechanisms such as `report-noisy-max` require complicated *coupling arguments* [Albarghouthi and Hsu 2018]. Fortunately, the particular combination of features provided by DFuzz—dependent pattern-matching, precise types, the probability monad, and recursion—allow for the typing of full mechanisms such as `k-means` and `idc` (Fig. 3).

The original presentation of DFuzz [Gaboardi et al. 2013] presents three implementations of iterative privacy mechanisms, each of which use an input argument of known size (such as the precise natural `iter` with type $\mathbb{N}[i]$ in *k-means*) to bound the number of iterations. By bounding the number of iterations, the applications of privacy primitives also becomes bounded, which is reflected in the sensitivity of the mechanism, e.g. *k-means* is $3 \cdot i \cdot \epsilon$ sensitive in the argument `data`.

Unlike the applications of synthesis in §5.1, constructing a synthesis problem whose solution is a privacy mechanism is straightforward. The following example demonstrates this using `k-means`.

Example 9. Suppose a mechanism designer wants to synthesize a version of `k-means` by carefully composing applications of the cluster-updating function

$$\texttt{k-step} : \forall k.\, L(\langle \mathbb{R}, \mathbb{R} \rangle)[k] \to \mathsf{ms}(\langle \mathbb{R}, \mathbb{R} \rangle)[\infty] \multimap_{3\cdot\epsilon} \bigcirc L(\langle \mathbb{R}, \mathbb{R} \rangle)[k]$$

that, when given a list of cluster centers and a database, updates the list of cluster centers, accumulating a privacy cost of $3 \cdot \epsilon$. Instead of manually reasoning about the recursion required, the mechanism designer can instead construct the synthesis problem

$$\mathcal{S} = \langle \sigma,\ \{\texttt{k-step}\},\ \top,\ \forall\, \langle i, o \rangle \in \mathcal{E}.\, p(i) = o \rangle\,,$$

where σ is the type of `k-means` given in §2 and $\mathcal{E}$ is a set of input–output examples. The full version of the implementation sketch given for `k-means` in Fig. 3 is a solution to $\mathcal{S}$. ■

5.3 Utility of Synthesis Solutions

In §5.1 and §5.2, we frame instantiations of our technique as a mechanism for finding privacy-aware solutions to a functional specification ϕ_c, usually provided as a set of input-output examples. These instantiations provide a mechanism for easily maintaining privacy while enabling access to sensitive data. However, in most real-world applications, users also desire *accuracy*. For randomized programs, like those we synthesize, accuracy amounts to the intuition that we arrive at close to the right answer most of the time [Dwork and Roth 2014]. As users define "the right answer" by providing ϕ_c, we expect users desire a program p that *maximizes* the probability that $p \models \phi_c$.

Unfortunately, proving accuracy of randomized programs is a challenging task. Most proofs are constructed by hand, and as such are tailored to the algorithm of interest. While there exist program logics for reasoning about accuracy [Barthe et al. 2016], automations of said logics (*i*) are so slow as to dwarf the cost of synthesis (see §6), and (*ii*) themselves reduce to synthesis [Smith et al. 2019]. Considering accuracy in addition to privacy therefore poses a significant challenge to program synthesis, which necessitates our use of the weak notion of *deterministic satisfaction* (§3) to ensure programs have some utility to an end user.

5.4 Usage Scenario and Privacy Guarantee

Here we propose a usage scenario based on the mechanisms in §5.1 and consider the resulting privacy guarantee. The following discussion can be adapted to the application in §5.2, although we expect mechanism designers will not be interacting with real-world sensitive information. We will speak of two distinct entities: (*i*) the dataset maintainer (**DM**), and (*ii*) the user (**U**).

Dataset Maintainer. The dataset maintainer **DM** controls access to a dataset D containing sensitive information. For all authorized users (including **U**), **DM** maintains a privacy budget, the sum of which is the total privacy budget for D. **DM** makes public as much of the *semantic structure*

of D as possible without leaking information. This includes the database schema, descriptions of fields, and *numerical data ranges* when they hold for every possible entry in that field. For instance, a field containing a *percentage* will always have values in the range $[0, 100]$. However, a field recording *age* is theoretically unbounded, and so any range published by **DM** will leak information about the contents of D. Lastly, **DM** makes public Σ, a set of functions whose implementation is trusted and projections with sensitivities derived from the provided data ranges (e.g., a function `grade` of type row $\multimap_{100} \mathbb{R}$ that selects a student's final grade from the relevant row).

As our technique also produces sensitivity proofs, the dataset maintainer can easily verify that executing q on D will be $c\epsilon$-DP. When **DM** receives a c-sensitive query q with a proof of sensitivity from a user, they (*i*) verify the proof of sensitivity, (*ii*) evaluate q on D, and (*iii*) decrement the privacy budget of U appropriately.

User. To query D, the user **U** can synthesize a query and resulting proof of sensitivity. At a minimum, this requires **U** to construct (*i*) input-output examples and (*ii*) a maximal acceptable privacy cost. Input-output examples can be hand-crafted to appropriately disambiguate the desired query, or be randomly-generated inputs and the corresponding outputs. The maximal privacy cost is up to **U**, and might depend on total privacy budget left and the possible value of the query result. Using these inputs, **U** can use a privacy mechanism to instantiate a synthesis problem and send the resulting query and associated proof of privacy to **DM** for consideration.

For numerical fields where **DM** has not provided a data range, **U** can improve Σ by providing best-guess data ranges. These ranges can be converted to projections whose sensitivity is the size of the range using the clipping technique in Airavat [Roy et al. 2010], lowering the privacy cost.

More realistically, we might expect the user **U** to be multiple individuals fulfilling different roles. For example, one could have a *domain expert* that constructs the relevant best-guess data ranges and random examples, and an *analyst* that computes the appropriate outputs to specify some query.

Effective Privacy Guarantee. Consider the usage scenario presented above. Queries are synthesized *independently* of a sensitive dataset, and are built exclusively from trusted code and clipped projections. Numerical data ranges are only provided if they hold for *every possible entry*, and so the only means of egress for sensitive information is through noisy query answers released by the dataset manager. A dataset manager is therefore able to maintain the privacy guarantee on D, as each release of information is $c\epsilon$-DP with a known c.

6 IMPLEMENTATION AND EVALUATION

6.1 Implementation

We implemented our technique in a tool called Zinc, comprising ~4500 lines of OCaml that interface with the Z3 SMT solver [de Moura and Bjørner 2008] for satisfiability checking.

Zinc implements a fair scheduling of the inference rules in §4, so that any rule that is applicable will eventually be applied. The application order of the rules is determined by a heuristic function that assigns costs to subproblems. Zinc takes as input a set of input–output examples and an upper bound on the budget. In the data analysis case, the choice of privacy mechanism to use is determined by the types of the provided examples.

Determinization. Zinc maintains a work-list of candidate partial solutions, and explores candidates in increasing heuristic order. When considering a candidate, all relevant inference rules are applied to generate new candidates. To prevent the search space from exploding, Zinc utilizes two common techniques from the type-directed synthesis literature to minimize the number of term-, type-, and sensitivity-applications. The rules for introducing such applications either introduce free variables or increase the number of wildcards, both of which expand the search space unnecessarily.

Table 3. Datasets Zinc was tested over, including (*i*) the average number of examples per benchmark, (*ii*) the maximum number of rows in an example, and (*iii*) the number of projections added to the signature Σ. For iterative mechanisms, ϵ is a fixed privacy cost.

Data Analysis Benchmark	Mechanism	Sens. Budget
Adult Income (1.5, 5, 8)		
from 1994 census data, contains info on gender, ethnicity, income, profession, and more		
1: number of women who work > 40 hrs a week	Laplace	1
2: cumulative years of education in military	Laplace	20
3: number of people who make > $50k$ in trade	Laplace	1
4: most common gender working in local government	Exponential	1
5: population per ethnicity	Parallel Comp.	1
6: profession with highest total work hours	Exponential	168
7: number of people making > $50k$ per branch of government	Parallel Comp.	1
Student Alcohol Consumption (1, 4, 8)		
Portuguese schools student info on grades, family life, and weekend/weekday alcohol consumption		
8: # of students who drink on the weekend and pay for extra classes	Laplace	1
9: average final grade of students attending for rep. reasons	Laplace	100
10: average weekend alcohol consumption per address type	Parallel Comp.	5
11: family relationship status with highest grade	Exponential	20
12: average final grade of students not drinking on the weekend	Laplace	100
13: total absences per attendance reason	Parallel Comp.	100
14: most common address type for those with poor family relations	Exponential	5
Student Performance (2, 4, 6)		
same as above, with performance, resource usage, participation, and personal info		
15: number of students with satisfied parents and many absences	Laplace	1
16: performance level with the highest average participation	Exponential	100
17: average resource usage per parent satisfaction	Parallel Comp.	100
18: hands raised by students with low discussion activity	Laplace	100
19: average grade in the low performance bracket	Laplace	100
20: are parents satisfied when their child is absent a lot?	Exponential	1
21: total resource usage per performance bracket	Parallel Comp.	100
COMPAS (3, 6, 8)		
ProRebuplica data with info on criminal history and COMPAS *scores, including risk of recidivism*		
22: number of elderly with high risk of violence	Laplace	1
23: average failure to appear for youths with no priors	Laplace	10
24: ethnicity with highest average recidivism	Exponential	10
25: total priors per gender	Parallel Comp.	15
26: # of people with many juvenile felonies and high recidivism risk	Laplace	1
27: average recidivism per ethnicity	Parallel Comp.	10
28: age category with the most priors	Exponential	15

Iterative Privacy Mechanism Benchmark	i Represents...	Sens. Budget
`k-means` - compute k cluster centers from data	number of updates	$3 \cdot i \cdot \epsilon$
`idc` - answers query set by iteratively constructing dataset	number of updates	$2 \cdot i \cdot \epsilon$
`cdf` - given list of buckets, count elements per bucket	number of buckets	$i \cdot \epsilon$

To limit term applications, Zinc only enumerates terms in β-normal form, rather than the equivalent β-expanded terms. That is, Zinc will consider $f\ x$, but not $(\lambda y.\ f\ y)\ x$, despite the two terms being equivalent. Empirically, such terms are smaller and more interpretable. Zinc achieves this by only applying the rule App to a wildcard whose goal type is a *function type*, e.g. $\sigma \multimap_k \tau$.

Zinc limits type and sensitivity applications by only applying rules TApp, SizeApp, and SensApp when necessary to enable an application of rule Id. For example, in Ex. 5, to replace $\bullet^{\Omega}_{\mathsf{ms(row)}[\infty]\multimap_2\mathbb{R}}$ with the function `count` of type $\forall\beta.\forall k.\,\mathsf{ms}(\beta)[k] \multimap_1 \mathbb{R}$, we must first apply TApp and SizeApp to introduce quantifiers in the goal type. Rather than expanding the goal type unnecessarily, Zinc greedily attempts to apply Id, and defaults to applying TApp and the like when abduction fails due to a universal quantifier in the type of the term being substituted.

Pruning Strategy. In §4, we give the invariant that any synthesis state $\langle \phi,\ p \rangle$ that has an unsatisfiable proof obligation ϕ can be pruned without losing relative completeness. In our original

strategy we eagerly called the SMT solver to check satisfiability for every state. While every call typically took less than a second, the overhead of SAT checking far outweighed the benefits of pruning, resulting in an impractical algorithm.

Instead of calling the SMT solver on every subproblem, we implemented a simple constraint solver that only performs *unit propagation*. The solver looks for conjuncts of the form $x = S$, where x is a variable and S is a sensitivity expression, and replaces all occurrences of x by S in the conjunction. If unit propagation fails to prove satisfiability, we use the number of remaining sensitivity variables as a heuristic quantitative measure for *how likely it is for the formula ϕ to be unsatisfiable*. We use this heuristic value to order the states for exploration: intuitively, the more variables we cannot easily eliminate with constant propagation, the harder it will be to satisfy the constraints.

Constraint Solving. Once Zinc has found a *closed* candidate program that satisfies ϕ_s, it attempts to prove that the program satisfies the budget constraints ϕ_k. Theorem 4.2 provides a mechanism for transforming the proof obligation to an equisatisfiable formula in the undecidable theory of mixed real and integer non-linear arithmetic. Fortunately, in our setting, such constraints are reliably checkable [de Amorim et al. 2014]. If necessary, Zinc can *relax* the obligation by treating integers as reals, placing the constraints in the decidable theory of *real closed fields*.

6.2 Evaluation

We curated a set of benchmarks to help us answer the following research questions:

RQ1 Is Zinc able to synthesize interesting differentially private queries in a reasonable time?
RQ2 Is sensitivity-directedness useful in guiding the synthesis process?
RQ3 Can Zinc synthesize full privacy mechanisms?

Benchmarks. One goal with benchmarking was to mimic a setting where a data analyst wants to query a dataset with sensitive information through a DP-enforcing system. We collected 4 real datasets (Table 3) containing personal information that would warrant DP. For example, the Adult Income dataset [Lichman 2013] is census data and contains data on gender, ethnicity, and income.

There is no existing set of queries over the chosen datasets (or any other dataset) that are (*i*) differentially private and (*ii*) designed to stress synthesis tasks. Therefore, for every dataset, we created a number of benchmark queries (data analysis benchmarks in Table 3) that are designed to extract interesting information, be of varying complexity, and exercise the privacy mechanisms from §5.1. We discuss the selection of benchmark datasets and queries further in §6.3.

Our other goal was to explore the synthesis of full privacy mechanisms. We chose three mechanisms whose privacy guarantees can be verified by DFuzz (iterative privacy mechanism benchmarks in Table 3). These mechanisms represent a large portion of the case studies presented in DFuzz [Gaboardi et al. 2013], and are based on real-world privacy mechanisms found throughout the literature. These mechanisms take advantage of every feature of DFuzz our synthesis algorithm supports, including recursion, the probability monad, and dependent pattern-matching.

Experimental Setup. To answer our research questions, we have two instantiations of Zinc:

- **Sensitivity-directed Zinc:** Our primary interest is the *sensitivity-directed* strategy, where Zinc utilizes symbolic context constraints to direct the search, as described in §6.1.
- **Baseline (type-directed):** To contrast with the sensitivity-directed strategy, we built a *baseline* version of Zinc that is type-directed in the style of existing type-directed synthesis tools (Myth [Osera and Zdancewic 2015], λ^2 [Feser et al. 2015], Bigλ [Smith and Albarghouthi 2016]). The baseline searches the space of programs in ascending size order, and does not exploit the generated constraints to guide the search. The choice to construct a baseline, rather than use an existing tool, is discussed in §6.3.

Table 4. Results of evaluating Zinc. Benchmark descriptions specify the privacy mechanism and budget. For all experiments, we report (*i*) the CPU time needed by Zinc and (*ii*) the number of programs explored. For the sens.-directed experiments, we additionally give the speedup in *time* compared to the baseline. Lastly, we report the sizes of the solution in AST nodes and the proof obligation in conjuncts.

	Sensitivity-Directed			*Baseline*			
Benchmark	Time	Speedup	Count	Time	Count	$\|p\|$	$\|\phi\|$
Adult Income							
1: *Lap.*/1ϵ	2.12	1.1x	16,825	2.36	16,721	31	11
2: *Lap.*/20ϵ	3.43	1.9x	26,506	6.65	39,365	33	12
3: *Lap.*/168ϵ	0.04	4.3x	601	0.17	1,722	26	7
4: *Exp.*/1ϵ	1.99	4.2x	11,610	8.44	32,299	26	8
5: *PC*/1ϵ	0.004	2.3x	36	0.009	75	24	9
6: *Exp.*/168ϵ	3.77	8.7x	20,033	32.97	111,466	31	8
7: *PC*/1ϵ	3.53	1.4x	11,856	4.96	13,072	48	21
Student Alcohol Consumption							
8: *Lap.*/1ϵ	0.42	2.3x	4,495	0.95	7,561	25	8
9: *Lap.*/20ϵ	5.03	1.5x	37,539	7.67	45,438	33	12
10: *PC*/5ϵ	3.28	1.4x	12,300	4.72	12,558	44	20
11: *Exp.*/20ϵ	5.27	4.1x	28,503	21.69	81,563	34	12
12: *Lap.*/100ϵ	6.26	3.0x	45,648	18.53	90,664	35	12
13: *PC*/100ϵ	3.35	1.4x	12,549	4.70	12,379	44	20
14: *Exp.*/5ϵ	0.125	9.9x	1,030	1.24	6,161	24	6
School Performance							
15: *Lap.*/1ϵ	0.087	1.7x	1,210	0.15	1,529	26	7
16: *Exp.*/100ϵ	9.67	15.3x	46,069	147.8	432,903	38	12
17: *PC*/100ϵ	5.53	1.1x	19,504	6.19	16,525	44	20
18: *Lap.*/100ϵ	8.11	6.3x	56,131	51.03	213,231	37	12
19: *Lap.*/100ϵ	4.23	1.3x	30,563	5.42	32,167	37	12
20: *Exp.*/1ϵ	1.91	4.1x	11,499	7.77	29,824	26	8
21: *PC*/100ϵ	38.95	2.2x	98,935	85.37	143,448	51	25
COMPAS							
22: *Lap.*/1ϵ	0.015	1.6x	227	0.024	365	17	6
23: *Lap.*/10ϵ	155.21	1.4x	813,878	219.78	858,686	36	15
24: *Exp.*/10ϵ	0.98	2.3x	7,254	2.27	12,603	26	9
25: *PC*/15ϵ	0.042	13.1x	329	0.55	2,110	39	15
26: *Lap.*/1ϵ	7.37	1.3x	55,240	9.38	55,157	33	11
27: *PC*/10ϵ	0.036	9.7x	284	0.35	1339	39	15
28: *Exp.*/15ϵ	0.745	1.9x	6,184	1.41	6897	23	6
Iterative Privacy Mechanisms							
`k-means`	0.278	3.5x	822	0.956	2,100	21	26
`idc`	3.491	2.82x	4440	9.845	10654	40	40
`cdf`	-	N/A	-	-	-	32	-

For each strategy, we ran Zinc on each benchmark with a timeout limit of 5 minutes. Table 4 shows the results. All times reported are in seconds.

RQ1: Synthesis Time. Consider the results in Table 4. Overall, the results demonstrate that our technique can synthesize non-trivial differentially private computations over real datasets in a small amount of time. In all benchmarks, our sensitivity-directed technique was able to discover a solution, and in most benchmarks synthesis terminates in under 10 seconds.

The programs synthesized by Zinc are non-trivial, comprising functions with complex types and involving advanced privacy mechanisms. For example, let us consider the solution to benchmark 23, which takes as input three example datasets of three rows each:

```
let bm23 = λx. avg (map f m) + Laplace(1/ε)
   where f = failure-to-appear
   where m = filter (λy. age-category(y) == "young") m′
   where m′ = filter (λz. priors(z) == none) x
```

```
let cdf buckets data = match buckets with
  | [] → return []
  | x :: xs[i] → let-draw count = ●◯ℝ in let-draw rest = ●◯L(ℝ)[i] in ●◯L(ℝ)[i+1]
```

Fig. 8. Partial synthesis problem for `cdf` [Gaboardi et al. 2013]. Sens.-directed finishes in 190s, while size-directed times out.

The constants `"young"` and `none` are instantiated from the schema of the COMPAS dataset. The solution involves 3 composed higher-order functions, several projections and comparisons, and an aggregation, and is a 10ϵ-DP function (as the projection failure-to-appear maps on a scale from 0 to 10). Note program size $|p|$ reported in Table 4 contains sensitivity annotations, which we elide here for clarity. The proof obligation (also elided) is satisfiable, and consists of 15 conjuncts.

RQ2: Sensitivity-Directed Synthesis. To answer **RQ2**, we compare the synthesis time of the sensitivity-directed configuration and the baseline configuration. Ignoring benchmarks with approximately comparable performance (difference in synthesis time $\leqslant$ 2 seconds), the *average speedup* afforded by incorporating sensitivity into the search is ~3.9x, and the maximum speedup is 15.3x. This significant improvement is a clear indication of the importance of the sensitivity-directed strategy to our algorithm's efficiency.

Consider benchmark 18: here, sensitivity-directed synthesis considers 56,131 programs before discovering the solution, while the baseline technique requires 213,231 programs. Similar patterns are seen across our benchmarks. See Fig. 7 for a clearer picture of the distribution of times across benchmarks.

The sensitivity-directed implementation consistently outperforms the baseline. Most benchmarks lie between the 10x and 1x efficiency lines in Fig. 7, although there are several *above* the 10x line. Note the log-scale on the graph: the benchmark furthest above this line boasts a 30x improvement. This distribution of benchmarks indicates that sensitivity-directed synthesis is an improvement over the baseline.

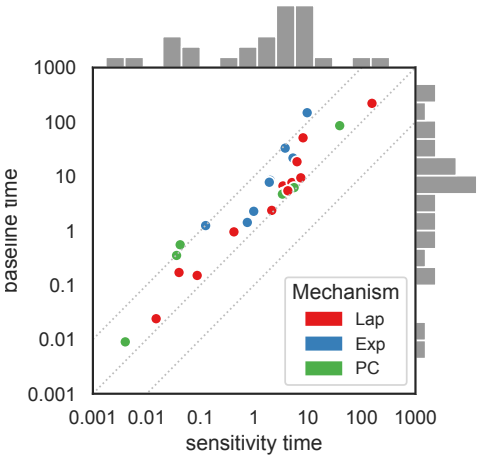

Fig. 7. Each benchmark is a pair in log-space comparing sens.-directed and size-directed performance. Color is the privacy mechanism used. Top, middle, and bottom dotted lines are the 10x, 1x, and 0.1x levels of the efficiency gradient. Marginals projected on the border.

These improvements in the sensitivity-directed synthesis over the baseline come despite the fact that Zinc does not explicitly prune the search space. As the primary difference in implementation is the inclusion of the constant propagation heuristic (§6.1) in the sensitivity-directed approach, we can infer that the reordering of candidate solutions given by the heuristic effectively directs the search towards programs that are *likely* to have satisfiable constraints, and thus be solutions to the synthesis problem. The effectiveness of the heuristic is aided by the fact that, even with relatively simple higher-order combinators, such as `map` and `filter`, the structure of the queries places considerable restrictions on the sensitivity of the input functions.

RQ3: Privacy Mechanism Synthesis. Consider the iterative privacy mechanism results in Table 4. Zinc was able to synthesize `k-means` and `idc` completely in under 4 seconds, but timed out while synthesizing `cdf`. In the two benchmarks that terminated, sensitivity-directedness contributed an average ~3.1x improvement in synthesis speed. The benchmarks that terminate are quite intricate (recall the implementation sketches of the terminating benchmarks in Fig. 3), as they contain recursion, the manipulation of probability distributions, and dependent pattern-matching.

Note that Zinc timing out on a benchmark does not mean Zinc is unhelpful to a mechanism designer. Rarely is an expert in differential privacy synthesizing a mechanism from scratch. To explore the utility of Zinc for *partial program synthesis*, we manually constructed a solution to the `cdf` benchmark (also taken from Gaboardi et al. [2013]) and introduced four *holes* by replacing whole expressions with wildcards. By varying the amount (from 1 to 4) and location of holes, we construct a total of 16 partial program synthesis benchmarks. The sensitivity-directed approach finds completions for the holes in all but one benchmark (in Fig. 8), while the size-directed approach times out on two (Fig. 8 plus one more). When both terminate, however, the sensitivity-directed strategy has a reduction in efficiency of ~0.94x. As 10 benchmarks terminate in under 1 second, and 2 more terminate in under 10 seconds, this tradeoff is likely worthwhile to be able to fill in one more partial program.

6.3 Discussion

Choice of Baseline. Our use of a baseline version of Zinc, as opposed to an existing synthesis tool, is necessitated by the limitations in the implementations of existing type-directed synthesis algorithms (e.g. Myth [Osera and Zdancewic 2015], Myth2 [Frankle et al. 2016], Bigλ [Smith and Albarghouthi 2016], and λ^2 [Feser et al. 2015]) and the complexity of the DFuzz type system. We found no tool that was amenable to modification to handle the combination of recursion, dependent pattern-matching, the probability monad, and sensitivity annotations.

Selection of Benchmarks. Differential privacy is usually benchmarked for (*i*) accuracy of a particular mechanism [Erlingsson et al. 2014; Johnson et al. 2018; Proserpio et al. 2014; Roy et al. 2010], (*ii*) scalability [Narayan and Haeberlen 2012; Proserpio et al. 2014; Roy et al. 2010], or (*iii*) case studies highlighting features of a particular system [McSherry 2009]. None of these cases produce large numbers of queries for testing synthesis. Program synthesis benchmarks on tables and databases [Feng et al. 2017] focus on tasks that are not differentially private.

In light of these limitations, we constructed a set of benchmarks to evaluate Zinc. The benchmark datasets were selected because they have appeared in works on privacy and bias [Datta et al. 2017; Feldman et al. 2015; Jeff Larson and Angwin [n. d.]]. The queries cover a range of query complexities and instantiations of our technique, but are motivated by existing analysis when possible (as in the COMPAS recidivism dataset [Jeff Larson and Angwin [n. d.]]).

Sensitivity. For each of our benchmark queries, we provide a sensitivity budget of the *minimal sensitivity* necessary to compute the desired query, which is derivable from the database schema and the desired semantics of the benchmark query. We do not expect the user to always be able to explicitly compute the appropriate sensitivity upper-bound, but in some cases real-world considerations—e.g., the remaining privacy budget—can inform the choice of sensitivity bound.

Accuracy/Privacy Tradeoff. This work presents mechanisms that add randomness to the output scaled *only* by the privacy parameter ϵ. Often, randomness is dependent on the sensitivity of the query in addition to ϵ [Dwork and Roth 2014]. This allows the mechanism to enforce a stronger privacy guarantee: ϵ-DP instead of $c\epsilon$-DP. Our technique still applies: low sensitivity now represents higher accuracy, instead of a cheaper privacy cost.

7 RELATED WORK

Type-directed Synthesis. Our contributions are inspired by works on type-directed synthesis. Compared to Myth [Frankle et al. 2016; Osera and Zdancewic 2015], λ^2 [Feser et al. 2015], and Bigλ [Smith and Albarghouthi 2016], which use a Hindley–Milner type system, we use a richer type system that adds linear and dependent types to aid in synthesizing programs to meet privacy constraints. Polikarpova et al. [2016] perform synthesis over the powerful refinement type system of liquid types [Rondon et al. 2008]. However, our techniques are incomparable, since liquid types (as defined and used) cannot reason about probabilistic hyperproperties like differential privacy.

Synthesis for Data Manipulation. The main target of DP in general, and this work in particular, is data analysis. Our work follows the tradition of program synthesis for various data-manipulation tasks, e.g., [Gulwani et al. 2012; Le and Gulwani 2014,?; Miltner et al. 2018; Polozov and Gulwani 2015; Smith and Albarghouthi 2016; Wang et al. 2017a,b; Yaghmazadeh et al. 2017; Zhang and Sun 2013]. Perhaps the most closely related work to ours is Bigλ [Smith and Albarghouthi 2016], which targets data-analysis programs composed of higher-order combinators like *map* and *reduce*. Our instantiation of our technique extends these ideas to a DP setting. Additionally, our work can be applied to SQL synthesis [Wang et al. 2017a; Yaghmazadeh et al. 2017; Zhang and Sun 2013] under DP constraints, as we discuss below.

Differential Privacy Systems. Systems enforcing differential privacy on user queries are defined over languages of higher-order combinators [McSherry 2009; Proserpio et al. 2014; Roy et al. 2010] or forms of SQL queries [Johnson et al. 2018; Narayan and Haeberlen 2012]. Our presented approach, based on the DFuzz type system, is general in that it can be applied in a wide range of settings. We instantiated our algorithm with a language of higher-order combinators. SQL queries can also be captured with *join* operators, like the one used in PINQ [McSherry 2009].

Work on DFuzz includes the construction of metric-preserving semantics [de Amorim et al. 2017] and a type-checking algorithm [de Amorim et al. 2014]. Type-checking DFuzz programs is non-trivial: the natural top-down approach requires context splitting, which necessitates a search over sensitivity terms. Our approach avoids this search by using SCCs, while the work of [de Amorim et al. 2014] et al. extends DFuzz sensitivity expressions.

Recently, constraint-based program synthesis techniques have been applied to the problem of proving differential privacy of advanced differentially private algorithms [Albarghouthi and Hsu 2018]. There, the authors use a heavy-weight logical encoding of the space of proofs of ϵ-DP to discover complex proofs using *coupling arguments*. We utilize DFuzz to compute program sensitivities in a more lightweight fashion to ease synthesis.

8 CONCLUSIONS

We introduced program synthesis under differential privacy, where the synthesizer is aware of differential-privacy constraints that impose a budget on the amount of computation that can be performed on a sensitive dataset. We developed a novel type-directed synthesis algorithm that constructs low-cost programs that are within the user's budget. Our technique is based on DFuzz, a linear dependent type system that can track the privacy cost of a program. We demonstrate how our approach can automatically synthesize a wide variety of differentially private programs.

ACKNOWLEDGMENTS

We would like to thank Marco Gaboardi for their extensive comments and advice on earlier drafts, Justin Hsu for the discussion and feedback, and all reviewers for their suggestions. This work is supported by the National Science Foundation CCF under awards 1566015 and 1652140.

REFERENCES

Aws Albarghouthi and Justin Hsu. 2018. Synthesizing coupling proofs of differential privacy. *PACMPL* 2, POPL (2018), 58:1–58:30. https://doi.org/10.1145/3158146

Apple. Accessed 11-11-2017. Differential privacy. https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf. (Accessed 11-11-2017).

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A program logic for union bounds. In *The 43rd International Colloquium on Automata, Languages and Programming* . Rome, Italy. https://doi.org/10.4230/LIPIcs.ICALP.2016.107

US Census Bureau. Accessed 11-11-2017. On The Map. https://onthemap.ces.census.gov/. (Accessed 11-11-2017).

Anupam Datta, Matthew Fredrikson, Gihyuk Ko, Piotr Mardziel, and Shayak Sen. 2017. Use Privacy in Data-Driven Systems: Theory and Experiments with Machine Learnt Programs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 1193–1210. https://doi.org/10.1145/3133956.3134097

Arthur Azevedo de Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type Checking. In *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1-3, 2014*. 5:1–5:12.

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A Semantic Account of Metric Preservation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. 545–556.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

Cynthia Dwork and Aaron Roth. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

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*. ACM, 1054–1067.

Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and Removing Disparate Impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. ACM, New York, NY, USA, 259–268. https://doi.org/10.1145/2783258.2783311

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 422–436.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *POPL*.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 357–370.

Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *CACM* 8 (2012).

Anupam Gupta, Aaron Roth, and Jonathan Ullman. 2012. Iterative constructions and private data release. In *Theory of cryptography conference*. Springer, 339–356.

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*. ACM, 1339–1354.

Moritz Hardt, Katrina Ligett, and Frank Mcsherry. 2012. A Simple and Practical Algorithm for Differentially Private Data Release. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2339–2347. http://papers.nips.cc/paper/4548-a-simple-and-practical-algorithm-for-differentially-private-data-release.pdf

Lauren Kirchner Jeff Larson, Surya Mattu and Julia Angwin. [n. d.]. How We Analyzed the COMPAS Recidivism Algorithm. ([n. d.]). https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm/ Accessed: 2017-11-15.

Noah M. Johnson, Joseph P. Near, and Dawn Xiaodong Song. 2018. Practical Differential Privacy for SQL Queries Using Elastic Sensitivity. *VLDB*. http://arxiv.org/abs/1706.09479

Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *PLDI*.

M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

Frank McSherry and Kunal Talwar. 2007. Mechanism design via differential privacy. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*. IEEE, 94–103.

Frank D McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 19–30.

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing bijective lenses. *PACMPL* 2, POPL (2018), 1:1–1:30. https://doi.org/10.1145/3158089

Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially Private Join Queries over Distributed Databases.. In *OSDI*. 149–162.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *PLDI*.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 522–538.

Oleksander Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *OOPSLA*.

Davide Proserpio, Sharon Goldberg, and Frank McSherry. 2014. Calibrating data to sensitivity in private data analysis: a platform for differentially-private analysis of weighted datasets. *Proceedings of the VLDB Endowment* 7, 8 (2014), 637–648.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*.

Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and privacy for MapReduce.. In *NSDI*, Vol. 10. 297–312.

Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 326–340.

Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace Abstraction Modulo Probability. *Proc. ACM Program. Lang.* 3, POPL, Article 39 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290352

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 452–466.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Synthesis of data completion scripts using finite tree automata. *PACMPL* 1, OOPSLA (2017), 62:1–62:26.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. OOPSLA.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*.

Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *ASE*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 224–234. https://doi.org/10.1109/ASE.2013.6693082