2021

Language-Based Analysis Of Differential Privacy

Chukwunweike Abuah
University of Vermont

LANGUAGE-BASED ANALYSIS OF DIFFERENTIAL PRIVACY

A Dissertation Presented

by

Chukwunweike (Chiké) Abuah

to

The Faculty of the Graduate College of

The University of Vermont

In Partial Fulfilment of the Requirements
For the Degree of Doctor of Philosophy
Specializing in Computer Science

October, 2021

Defense Date: September 7, 2021
Dissertation Examination Committee:

Joseph P. Near, PhD., Advisor
Christelle Vincent, PhD., Chairperson
David Darais, PhD.
Christian Skalka, PhD.
Cynthia J. Forehand, Ph.D., Dean of the Graduate College

Abstract

Differential privacy (Dwork, 2006; Dwork et al., 2006a) has achieved prominence over the past decade as a rigorous formal foundation upon which diverse tools and mechanisms for performing private data analysis can be built.

The guarantee of differential privacy is that it protects privacy at the individual level: if the result of a differentially private query or operation on a dataset is publicly released, any individual present in that dataset can claim *plausible deniability*. This means that any participating individual can deny the presence of their information in the dataset based on the query result, because differentially private queries introduce enough random noise/bias to make the result indistinguishable from that of the same query run on a dataset which actually *does not* contain the individual's information. Additionally, differential privacy guarantees are resilient against any form of *linking attack* in the presence of *auxiliary information* about individuals.

Both static and dynamic tools have been developed to help non-experts write differentially private programs: static analysis tools construct a proof without needing to run the program; dynamic analysis tools construct a proof while running the program, using a dynamic monitor executed by the unmodified runtime system. The resulting proof may apply only to that execution of the program.

Many of the static tools take the form of statically-typed programming languages, where correct privacy analysis is built into the soundness of the type system. Meanwhile dynamic systems typically take either a *prescriptive* or *descriptive* approach to analysis when running the program.

This dissertation proposes new techniques for language-based analysis of differential privacy of programs in a variety of contexts spanning static and dynamic analysis. Our approach towards differential privacy analysis makes use of ideas from linear type systems and static/dynamic taint analysis. While several prior approaches towards differential privacy analysis exist, this dissertation proposes techniques which are designed to, in several regards, be more flexible and usable than prior work.

Dedication

to my family and friends

to the essential workers on the COVID-19 front lines

Acknowledgements

I would like to say a very special thanks (roughly in the order I met them) to Jon Edwards, Karen Edwards, Brenda Strong, Gary & Valorie Larsson, Mani Tiwaree, Michelle Kim, Leo Bremer, Jerod Weinman, Sam Rebelsky, John Stone, Kim Spasaro, Yanjun Chen, Tanya Santiago, Andrew Larson, Ben Nydegger, Christine Tran, John Brady, Mai Pham, Rikin Shah, Jay McCarthy, Lindsey Reams, Joel Savitz, Matthias Felleisen, Christian Skalka, Sami Connolly, Ryan Estes, Emma Tosch, John Foley, and Christelle Vincent.

A very special thanks to David Darais and Joseph P. Near.

Contents

Part I

Introduction

Chapter 1

Significance & Contributions

Differential privacy has achieved prominence over the past decade as a rigorous formal foundation upon which diverse tools and mechanisms for performing private data analysis can be built. The guarantee of differential privacy is that it protects privacy at the individual level: if the result of a differentially private query or operation on a dataset is publicly released, any individual present in that dataset can claim *plausible deniability*. This means that any participating individual can deny the presence of their information in the dataset based on the query result, because differentially private queries introduce enough random noise/bias to make the result indistinguishable from that of the same query run on a dataset which actually *does not* contain the individual's information. Additionally, differential privacy guarantees are resilient against any form of *linking attack* in the presence of *auxiliary information* about individuals.

High profile tech companies such as Google have shown a commitment to differential privacy by developing projects such as RAPPOR (Erlingsson et al., 2014) as well as several open-source privacy-preserving technologies (Guevara, 2019; Guevara et al., 2020; Wilson et al., 2019). Facebook recently released

an unprecedented social dataset, protected by differential privacy guarantees, which contains information regarding people who publicly shared and engaged with about 38 million unique URLs, as an effort to help researchers study social media's impact on democracy and the 2020 United States presidential election (Nayak, 2020; Kifer et al., 2020; King and Persily, 2020; Evans and King, 2021; Evans et al., 2021). The US Census Bureau has also adopted differential privacy to safeguard the 2020 census results (Abowd, 2018).

Both static and dynamic tools have been developed to help non-experts write differentially private programs: static analysis tools construct a proof without needing to run the program; dynamic analysis tools construct a proof while running the program, using a dynamic monitor executed by the unmodified runtime system. The resulting proof may apply only to that execution of the program.

Many of the static tools take the form of statically-typed programming languages, where correct privacy analysis is built into the soundness of the type system.

Meanwhile dynamic systems typically take either a *prescriptive* or *descriptive* approach to analysis when running the program. Intuitively, the *prescriptive* approach represents the scenario in which the analyst wishes to enforce an upper bound on the privacy leakage and that bound is known *a priori*. The *descriptive* approach represents the scenario in which the analyst only wishes to record the privacy leakage incurred, and no previously determined bound on privacy leakage is enforced during program execution.

The contributions of this dissertation cover both static and dynamic analysis.

1.1 Significance

Differential privacy has become the standard for protecting the privacy of individuals with formal guarantees of plausible deniability. In this dissertation we propose new techniques for language-based analysis of differential privacy of programs in a variety of contexts spanning static and dynamic analysis. Our approach towards differential privacy analysis makes use of ideas from linear type systems and static/dynamic taint analysis.

1.1.1 Challenges of Implementing Differential Privacy

- Differential privacy is a definition of privacy which is proven to be resilient against linking attacks. While this is an attractive prospect theoretically, in practice differential privacy can be difficult to analyze and implement. This dissertation proposes several techniques for automatic analysis and implementation of differential privacy.

- Violations of differential privacy are *silent*, and in most cases are impossible to catch by human observation, even by experts. This makes differential privacy an especially important field in which to apply verification techniques. The nature of privacy bugs necessitates an approach to privacy analysis which can be pervasive in all of the tools and software that can access and manipulate sensitive data (i.e. language-based techniques). This dissertation discusses approaches to privacy analysis which make this possible and convenient in practice.

- The techniques needed for validating DP depend on the implementation language.

1.1.2 Practical Impact of this Dissertation

This dissertation proposes several techniques which make it possible for regular programmers (who are not experts in differential privacy) to safely create and modify statically and dynamically typed programs which satisfy differential privacy, with little or no domain knowledge required.

1.1.3 Technical Contributions of this Dissertation

We will show a series of works that, in short, demonstrate the following key ideas:

- A pure linear typing discipline is sufficient to perform accurate analysis of differential privacy, even for its advanced variants. This is relevant for performing static privacy analysis of differential privacy alongside other important security properties in modern type systems. Linear type systems have gained popularity in mainstream languages such as Rust and Haskell for tracking/analysis of various security properties (II).

- Prescriptive and descriptive dynamic analysis of differential privacy is possible with low overhead for general-purpose programming languages. This is important because it demonstrates that privacy analysis can be embedded as a library in some of the most popular languages for data analysis and scripting in modern times, such as Python and JavaScript (III).

- Mainstream statically typed languages can be made to perform differential privacy analysis as part of their standard typechecking process, without any runtime execution or information. This is important because it shows that static differential privacy analysis can be convenient and accurate with full data-independence even in the absence of linear types (IV).

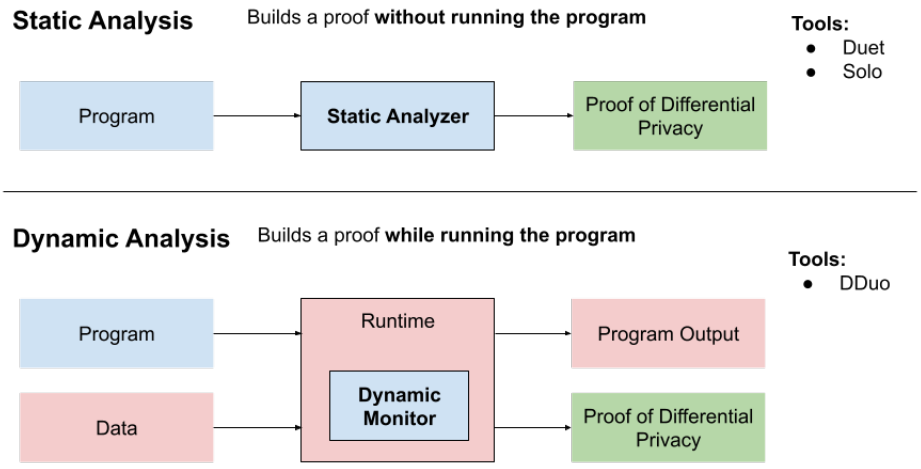

Figure 1.1: Static vs. Dynamic Privacy Analysis

- Language-based analysis of differential privacy can be practical and convenient in any given scenario.

It is our hope that the usability and strong guarantees of the works contained in this dissertation will inspire data analysts, technology corporations, researchers, and students in computer science to continue to build a community and culture of verifiable data privacy.

The following 3 sections briefly outline the high-level motivation and contributions for the major works which make up this dissertation.

1.2 DUET: An Expressive Higher-Order Language and Type System for Differential Privacy

Goal: Linear types based analysis for advanced differential privacy variants.

1.2.1 Motivation

Linear Type Systems Type-system-based solutions to proving that a program adheres to differential privacy began with Reed and Pierce's FUZZ language (Reed and Pierce, 2010), which is based on linear typing. FUZZ, as well as subsequent work based on linear types aided by SMT solvers (Gaboardi et al., 2013a), supports type inference of privacy bounds with type-level dependency and higher-order composition of programs. However, these systems only support the original and most basic variant of differential privacy called ϵ-*differential privacy*. More recent variants, like (ϵ, δ)-differential privacy (Dwork et al., 2014a) and others (Mironov, 2017a; Bun and Steinke, 2016; Bun et al., 2018), improve on ϵ-differential privacy by providing vastly more accurate answers for the same amount of privacy "cost" (at the expense of introducing a negligible chance of failure).

1.2.2 Contribution

The strengths of Duet w.r.t. prior work are summarized as follows: (1) DUET supports sensitivity analysis in combination with higher order programming, program composition, and compound datatypes, building on ideas from FUZZ (SA+HO); (2) DUET supports type-level dependency on values, which enables differentially private algorithms to be verified w.r.t. symbolic privacy parameters, building on ideas from DFuzz (DT); (3) DUET supports calculation of independent privacy costs for multiple program arguments via a novel approach (MA); and (4)

DUET supports (ϵ, δ)-differential privacy—in addition to other recent powerful variants, such as Rényi, zero-concentrated and truncated concentrated differential privacy—via a novel approach ((ϵ, δ)-DP, Rényi/ZC/TC)). DUET is able to achieve all of these features while maintaining a *pure linear typing* discipline.

In striking this balance, DUET comes with known limitations: (1) DUET is not easy to extend with new relational properties (Rel-ext); and (2) DUET is not suitable for verifying implementations of low-level mechanisms, such as the implementation of advanced composition, gradient operations, and the sparse-vector technique (SVT-imp).

1.3 DDUO: General-Purpose Dynamic Analysis for Differential Privacy

Goal: dynamic analysis of differential privacy as a library.

1.3.1 Motivation

Both static and dynamic tools have been developed to help non-experts write differentially private programs. Many of the static tools take the form of statically-typed programming languages, where correct privacy analysis is built into the soundness of the type system. However, existing language-oriented tools for compositional verification of differential privacy impose significant burden on the programmer (in the form of additional type annotations) (Reed and Pierce, 2010; Gaboardi et al., 2013b; Near et al., 2019b; de Amorim et al., 2019; Zhang et al., 2019a; Winograd-Cort et al., 2017; Barthe et al., 2019, 2012, 2013, 2016b; Sato et al., 2019; Albarghouthi and Hsu, 2018; Zhang and Kifer, 2017; Wang et al., 2019; Bichsel et al., 2018; Ding et al., 2018; Wang et al., 2020) (see Chapter 25 for a longer discussion).

The best-known dynamic tool is PINQ (McSherry, 2009), a dynamic analysis for sensitivity and privacy. It features an extensible system which allows non-experts in differential privacy to execute SQL-like queries against relational databases. However, PINQ comes with several restrictions that limit its applicability. For example, PINQ's expressiveness is limited to a subset of the SQL language for relational databases. Methods in PINQ are assumed to be side-effect free, which is necessary to preserve their privacy guarantee.

1.3.2 Contribution

We introduce DDuo, a dynamic analysis for enforcing differential privacy. DDuo is usable by non-experts: its analysis is automatic and it requires no additional type annotations. DDuo can be implemented *as a library* for existing programming languages; we present a reference implementation in Python. Our goal in this work is to answer the following four questions, based on the limitations of PINQ:

- Can a PINQ-style dynamic analysis extend to base types in the programming language, to allow its use pervasively?
- Is the analysis sound in the presence of side effects?
- Can we use this style of analysis for complex algorithms like differentially private gradient descent?
- Can we extend the privacy analysis beyond pure ϵ-differential privacy?

We answer all four questions in the affirmative, building on PINQ in the following ways:

- DDuo provides a dynamic analysis for base types in a general purpose language (Python). DDuo supports general language operations, such as

mapping arbitrary functions over lists, and tracks the sensitivity (stability) and privacy throughout.

- Methods in DDuo are not required to be side-effect free and allow programmers to mutate references inside functions which manipulate sensitive values.
- DDuo supports various notions of sensitivity and arbitrary distance metrics (including L_1 and L_2 distance).
- DDuo is capable of leveraging advanced privacy variants such as (ϵ, δ) and Rényi differential privacy.

1.4 Solo: A Lightweight Static Analysis for Differential Privacy

Goal: static analysis of differential privacy as a library.

1.4.1 Motivation

All current approaches for statically enforcing differential privacy in higher order languages make use of either linear or relational refinement types. A barrier to adoption for these approaches is the lack of support for expressing these "fancy types" in mainstream programming languages. For example, no mainstream language supports relational refinement types, and although Rust and modern versions of Haskell both employ some linear typing techniques, they are inadequate for embedding enforcement of differential privacy, which requires "full" linear types a la Girard. Recent work has made significant progress towards techniques for static verification of differentially private programs (Reed and Pierce, 2010; Near et al., 2019b; Barthe et al., 2015; Gaboardi et al., 2013b).

However, the specialized features they rely on do not exist in mainstream programming languages.

1.4.2 Contribution

We propose a new type system that enforces differential privacy, avoids the use of linear and relational refinement types, and can be easily embedded in mainstream richly typed programming languages such as Scala, OCaml and Haskell.

We introduce Solo, a novel type system for static verification of differential privacy, with a reference implementation as a Haskell library. Solo is similar to Fuzz (Reed and Pierce, 2010) and its descendants in expressive power, but Solo does not rely on linear types and can be implemented entirely in Haskell with no additional language extensions. In particular, Solo's sensitivity and privacy tracking mechanisms are compatible with higher-order functions, and leverage Haskell's type inference system to minimize the need for additional type annotations.

1.5 Personal Contributions

I personally contributed to the works in this dissertation in the following ways:

- I worked on the development of the interpreter and typechecker for Duet, helped iterate on the design and metatheory behind the language, and made major contributions to the research paper describing Duet.
- I lead the design and implementation of the DDuo system, and wrote the research paper describing DDuo.
- I lead the conceptualization, design and implementation of the Solo system, and wrote the research paper describing Solo.

1.6 Future Work

Below we briefly outline potential future work regarding this dissertation:

- A limitation of DDuo and similar software is the inability to automatically discover the sensitivity of arbitrary functions, particularly functions imported from third party software libraries. Property-based testing could potentially be used to fully automate sensitivity analysis for large software libraries.
- Gradual Differential Privacy could be used to bridge the gap between statically and dynamically typed approaches towards privacy analysis.

Chapter 2

Background

2.1 Differential Privacy.

Differential privacy is a formal notion of privacy; certain algorithms (called *mechanisms*) can be said to *satisfy* differential privacy. Intuitively, the idea behind a differential privacy mechanism is that: given inputs which differ in the data of a single individual, the mechanism should return statistically indistinguishable answers. This means that the data of any one individual should not have any significant effect on the outcome of the mechanism, effectively protecting privacy on the individual level. Formally, differential privacy is parameterized by the privacy parameters ϵ, δ which control the strength of the guarantee.

We say that two inputs x and x' are *neighbors* when $d_A(x, x') = 1$. To provide meaningful privacy protection, two neighboring inputs are normally considered to differ in the data of a single individual. Thus, the definition of differential privacy ensures that the probably distribution over $\mathcal{K}$'s outputs will be roughly the same, whether or not the data of a single individual is included in the input.

The strength of the guarantee is parameterized by the *privacy parameters* ϵ and δ. The case when $\delta = 0$ is often called *pure* ϵ-differential privacy; the case when $\delta > 0$ is often called *approximate* or (ϵ, δ)-differential privacy. When $\delta > 0$, the δ parameter can be thought of as a *failure probability*: with probability $1 - \delta$, the mechanism achieves pure ϵ-differential privacy, but with probability δ, the mechanism makes no guarantee at all (and may violate privacy arbitrarily). The δ parameter is therefore set very small—values on the order of 10^{-5} are often used. Typical values for ϵ are in the range of 0.1 to 1.

2.2 Sensitivity.

The core mechanisms for differential privacy (described below) rely on the notion of *sensitivity* (Dwork et al., 2006b) to determine how much noise is needed to achieve differential privacy. Intuitively, function sensitivity describes the rate of change of a function's output relative to its inputs, and is a scalar value that bounds this rate, in terms of some notion of distance. Formally:

Definition 2.2.1 (Global Sensitivity)**.** *Given distance metrics d_A and d_B, a function $f \in A \to B$ is said to be* s-sensitive *if* $\forall s' \in \mathbb{R},\ (x, y) \in A.\ d_A(x, y) \leq s' \implies d_B(f(x), f(y)) \leq s' \cdot s$.

For example, the function $\lambda x.\ x + x$ is 2-sensitive, because its output is twice its input. Determining tight bounds on sensitivity is often the key challenge in ensuring differential privacy for complex algorithms.

2.3 Core Mechanisms.

The core mechanisms that are often utilized to achieve differential privacy are the *Laplace mechanism* (Dwork et al., 2014b) and the *Gaussian mechanism* (Dwork et al., 2014b). Both mechanisms are defined for scalar values as well as vectors;

the Laplace mechanism requires the use of the L_1 distance metric and satisfies ϵ-differential privacy, while the Gaussian mechanism requires the use of the L_2 distance metric (which is often much smaller than L_1 distance) and satisfies (ϵ, δ)-differential privacy (with $\delta > 0$).

Definition 2.3.1 (Laplace Mechanism)**.** *Given a function $f : A \to \mathbb{R}^d$ which is s-sensitive under the L_1 distance metric $d_{\mathbb{R}}(x, x') = \| x - x' \|_1$ on the function's output, the Laplace mechanism releases $f(x) + Y_1, \ldots, Y_d$, where each of the values $Y_1, \ldots, Y_d$ is drawn iid from the Laplace distribution centered at 0 with scale $\frac{s}{\epsilon}$; it satisfies ϵ-differential privacy.*

Definition 2.3.2 (Gaussian Mechanism)**.** *Given a function $f : A \to \mathbb{R}^d$ which is s-sensitive under the L_2 distance metric $d_{\mathbb{R}}(x, x') = \| x - x' \|_2$ on the function's output, the Gaussian mechanism releases $f(x) + Y_1, \ldots, Y_d$, where each of the values $Y_1, \ldots, Y_d$ is drawn iid from the Gaussian distribution centered at 0 with variance $\sigma^2 = \frac{2s^2 \ln(1.25/\delta)}{\epsilon^2}$; it satisfies (ϵ, δ)-differential privacy for $\delta > 0$.*

Definition 2.3.3 (Differential privacy)**.** *Given a distance metric $d_A \in A \times A \to \mathbb{R}$, a randomized algorithm (or* mechanism*) $\mathcal{M} \in A \to B$ satisfies (ϵ, δ)-differential privacy if for all $x, x' \in A$ such that $d_A(x, x') \leq 1$ and all possible sets $S \subseteq B$ of outcomes, $\Pr[\mathcal{M}(x) \in S] \leq e^{\epsilon} \Pr[\mathcal{M}(x') \in S] + \delta$.*

Differential privacy is *compositional*: running two mechanisms $\mathcal{M}_1$ and $\mathcal{M}_2$ with privacy costs of (ϵ_1, δ_1) and (ϵ_2, δ_2) respectively has a total privacy cost of $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$. *Advanced composition* (Dwork et al., 2014b) improves on this composition bound for iterative algorithms; several variants of differential privacy (e.g. Rényi differential privacy (Mironov, 2017b) and zero-concentrated differential privacy (Bun and Steinke, 2016)) have been developed that improve the bound even further. Importantly, sequential composition theorems for differential privacy do not necessarily allow the privacy parameters to be chosen *adaptively*, which presents a special challenge.

Variant	SequentialComposition	k-Loop	Basic Mechanism
ϵ-DP (*Dworket al., 2014a*)	$\epsilon_1 + \epsilon_2 \triangleq \epsilon_1 + \epsilon_2$	$k\epsilon$	*Laplace*
(ϵ, δ)-DP (*Dworket al., 2014a*)	$(\epsilon_1, \delta_1) + (\epsilon_2, \delta_2) \triangleq (\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$	$(k\epsilon, k\delta)$	*Gaussian*
RDP (*Mironov, 2017a*)	$(\alpha, \epsilon_1) + (\alpha, \epsilon_2) \triangleq (\alpha, \epsilon_1 + \epsilon_2)$	$(\alpha, k\epsilon)$	*Gaussian*
zCDP (*BunandSteinke, 2016*)	$\rho_1 + \rho_2 \triangleq \rho_1 + \rho_2$	$k\rho$	*Gaussian*
tCDP (*Bunet al., 2018*)	$(\rho_1, \omega_1) + (\rho_2, \omega_2) \triangleq (\rho_1 + \rho_2, \min(\omega_1, \omega_2))$	$(k\rho, \omega)$	*Sinh-normal*

Figure 2.1: Variants of Differential Privacy

2.4 Composition.

Multiple invocations of a privacy mechanism on the same data degrade in an additive or compositional manner. For example, the law of *sequential composition* states that:

Theorem 2.4.1 (Sequential Composition)**.**
If two mechanisms $\mathcal{K}_1$ and $\mathcal{K}_2$ with privacy costs of (ϵ_1, δ_1) and (ϵ_2, δ_2) respectively are executed on the same data, the total privacy cost of running both mechanisms is $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$.

For iterative algorithms, *advanced composition* (Dwork et al., 2014b) can yield tighter bounds on total privacy cost. Advanced variants of differential privacy, like Rényi differential privacy (Mironov, 2017a) and zero-concentrated differential privacy (Bun and Steinke, 2016), provide even tighter bounds on composition.

The *moments accountant* was introduced by Talwar et al. (Abadi et al., 2016) specifically for stochastic gradient descent in deep learning applications. It provides tight bounds on privacy loss in iterative applications of the Gaussian mechanism, as in SGD. The Rényi differential privacy and zero-concentrated differential privacy generalize the ideas behind the moments accountant.

2.5 Variants of differential privacy.

In addition to ϵ and (ϵ, δ)-differential privacy, other variants of differential privacy with significant benefits have recently been developed. Three examples are Rényi differential privacy (RDP) (Mironov, 2017a), zero-concentrated differential privacy (zCDP) (Bun and Steinke, 2016), and truncated concentrated differential privacy (tCDP) (Bun et al., 2018). Each one has different privacy parameters and a different form of sequential composition, summarized in Figure 2.1. The basic mechanism for RDP and zCDP is the Gaussian mechanism; tCDP uses a novel *sinh-normal* mechanism (Bun et al., 2018) which decays more quickly in its tails. All three can be converted to (ϵ, δ)-differential privacy, allowing them to be compared and composed with each other. These three variants provide asymptotically tight bounds on privacy cost under composition, while at the same time eliminating the "catastrophic" privacy failure that can occur with probability δ under (ϵ, δ)-differential privacy.

2.6 Group privacy.

Differential privacy is normally used to protect the privacy of individuals, but it turns out that protection for an individual also translates to (weaker) protection for *groups* of individuals. A mechanism which provides pure ϵ-differential privacy for individuals also provides $k\epsilon$-differential privacy for groups of size k (Dwork et al., 2014a). Group privacy also exists for (ϵ, δ)-differential privacy, RDP, zCDP, and tCDP, but the scaling of the privacy parameters is nonlinear.

Part II

DUET: An Expressive Higher-Order Language and Type System for Differential Privacy

Chapter 3

Introduction

This section discusses DUET, an expressive higher-order language, linear type system and tool for automatically verifying differential privacy of general-purpose higher-order programs. In addition to general purpose programming, DUET supports encoding machine learning algorithms such as stochastic gradient descent, as well as common auxiliary data analysis tasks such as clipping, normalization and hyperparameter tuning–each of which are particularly challenging to encode in a statically verified differential privacy framework.

We present a core design of the DUET language and linear type system, and complete key proofs about privacy for well-typed programs. We then show how to extend DUET to support realistic machine learning applications and recent variants of differential privacy which result in improved accuracy for many practical differentially private algorithms. Finally, we implement several differentially private machine learning algorithms in DUET which have never before been automatically verified by a language-based tool, and we present experimental results which demonstrate the benefits of DUET's language design in terms of accuracy of trained machine learning models.

DUET supports (1) general purpose programming features like compound datatypes and higher-order functions, (2) library functions for matrix-based computations, and (3) multiple state-of-the-art variants of differential privacy—(ϵ, δ)-differential privacy (Dwork et al., 2014a), Rényi differential privacy (Mironov, 2017a), zero-concentrated differential privacy (zCDP) (Bun and Steinke, 2016), and truncated-concentrated differential privacy (tCDP) (Bun et al., 2018)—and can be easily extended to new ones. DUET strikes a strategic balance between generality, practicality, extensibility, and precision of computed privacy bounds.

The design of DUET consists of *two* separate, mutually embedded languages, each with its own type system. The *sensitivity language* uses linear types *with* metric scaling (as in FUZZ (Reed and Pierce, 2010)) to bound function sensitivity. The *privacy language* uses linear types *without* metric scaling (novel in DUET) to compose differentially private computations. Disallowing the use of scaling in the privacy language is essential to encode more advanced variants of differential privacy (like (ϵ, δ)) in a linear type system, because these definitions do not scale linearly, a requirement imposed by the usual scaled-metric interpretation of linear typing. E.g., FUZZ requires that the underlying definition of privacy supports linear scaling—which is true of the simplest variant of differential privacy (pure ϵ)—and it is well known that FUZZ cannot be used with more advanced variants of differential privacy for this reason. Restricting FUZZ to disallow scaling would severely limit the language's ability to reason about sensitivity.

Linear typing is a general-purpose type discipline, which can be applied to programming paradigms that track some notion of resource.

Linear typing (Barber, 1996; Girard, 1987) is a good fit for both privacy and sensitivity analysis because resources are tracked per-variable and combined additively. In particular, our linear typing approach to *privacy* allows for independent privacy costs for multiple function arguments, a feature shared

by Fuzz and DFuzz (which only support pure ϵ-differential privacy), but not supported by prior type systems for (ϵ, δ)-differential privacy. This limitation of prior work is due to treating privacy as a computational "effect"—a property of the output via an indexed monad—as opposed to our treatment of privacy as a "co-effect"—a property of the context via linear typing.

Our main idea is to co-design *two* separate languages for privacy and sensitivity, and our main insight is that a linear type system can (1) model more powerful variants of differential privacy (like (ϵ, δ)) when strengthened to disallow scaling, and (2) interact seamlessly with a sensitivity-type system which does allow scaling. Each language embeds inside the other, and the privacy mechanisms of the underlying privacy definition (e.g. the Gaussian mechanism (Dwork et al., 2014a)) form the interface between the two languages. Both languages use similar syntax and identical types. The two languages aid type checking, the proof of type soundness, and our implementation of type inference; programmers need not be intimately aware of the multi-language design.

In addition to differential-privacy primitives like the Gaussian mechanism, we provide a core language design for matrix-based data analysis tasks, such as aggregation, clipping and gradients. Key challenges we overcome in our design are how these features compose in terms of function sensitivity, and how to statically track bounds on vector norms (due to clipping, for the purposes of privacy)—and each in a way that is general enough to support a wide range of useful applications.

We demonstrate the usefulness of Duet by implementing and verifying several differentially private machine learning algorithms from the literature, including private stochastic gradient descent (Bassily et al., 2014a) and private Frank-Wolfe (Talwar et al., 2015), among many others. We also implement a variant of stochastic gradient descent suitable for deep learning. For each of

these algorithms, no prior work has demonstrated an automatic verification of differential privacy, and DUET is able to automatically infer privacy bounds that equal *and in some cases improve upon* previously published manual privacy proofs.

We have implemented a typechecker and interpreter for DUET, and we use these to perform an empirical evaluation comparing the accuracy of models trained using our implementations. Although the "punchline" of the empirical results are unsurprising due to known advantages of the differential privacy definitions used (e.g., that using recent variants like zero-concentrated differential privacy results in improved accuracy), our results show the extent of the accuracy improvements for specific algorithms and further reinforce the idea that choosing the best definition consistently results in substantially better accuracy of the trained model.

Chapter 4

DUET's Contributions.

In summary, DUET makes the following contributions:

- We present DUET, a language, linear type system and tool for expressing and automatically verifying differentially private programs. DUET supports a combination of (1) general purpose, higher order programming, (2) advanced definitions of differential privacy, (3) independent tracking of privacy costs for multiple function arguments, and (4) auxiliary differentially-private data analysis tasks such as clipping, normalization, and hyperparameter tuning.
- We formalize DUET's type system and semantics, and complete key proofs about privacy of well-typed programs.
- We demonstrate a battery of case studies consisting of medium-sized, real-world, differentially private machine learning algorithms which are successfully verified with optimal (or near-optimal) privacy bounds. In some cases, DUET infers privacy bounds which improve on the best previously published manually-verified result.
- We conduct an experimental evaluation to demonstrate DUET's feasibility

	SA	HO	DT	MA	Rel-ext	ϵ-DP	(ϵ, δ)-DP	Rényi/zCDP/tCDP	SVT-imp
Fuzz	✓	✓	✗	✓	✗	✓	✗	✗	✗
DFuzz	✓	✓	✓	✓	✗	✓	✗	✗	✗
PathC	✓	✓	✗[1]	✗	✓	✓	✓	✓[2]	✗
HOARe[2]	✓	✓	✓	✗	✓	✓	✓	✓[2]	✗
LightDP	✗	✗	✓	✓	✗	✓	✓	✓[2]	✓
Fuzzi	✓	✗	✗[1]	✓	✓	✓	✓	✓	✓
Duet	✓	✓	✓	✓	✗	✓	✓	✓	✗

Figure 4.1: Legend: SA = capable of sensitivity analysis; HO = support for higher order programming, program composition, and compound datatypes; DT = support for dependently typed privacy bounds; MA = support for distinct privacy bounds of multiple input arguments; Rel-ext = supports extensions to support non-differential-privacy relations; ϵ-DP = supports ϵ-differential-privacy; (ϵ, δ)-DP = supports (ϵ, δ)-differential-privacy; Rényi/zCDP/tCDP: supports Rényi, zero-concentrated and truncated concentrated differential privacy; SVT-imp: supports verified *implementation* of the sparse vector technique. 1: This limitation is not fundamental and could be supported by simple extension to underlying type theory. 2: Not described in prior work, but could be achieved through a trivial extension to existing support for (ϵ, δ)-differential privacy.

in practice by training two machine learning algorithms on several non-toy real-world datasets using Duet's interpreter. These results demonstrate the effect of improved privacy bounds on the accuracy of the trained models.

4.1 Our Approach

We show the strengths and limitations of Duet in relation to approaches from prior work in Figure 4.1. In particular, strengths of Duet w.r.t. prior work are: (1) Duet supports sensitivity analysis in combination with higher order programming, program composition, and compound datatypes, building on ideas from Fuzz (SA+HO); (2) Duet supports type-level dependency on values, which enables differentially private algorithms to be verified w.r.t. symbolic privacy parameters, building on ideas from DFuzz (DT); (3) Duet supports calculation of independent privacy costs for multiple program arguments via a novel approach (MA); and (4) Duet supports (ϵ, δ)-differential privacy—in addition to other recent powerful variants, such as Rényi, zero-concentrated and truncated concentrated differential privacy—via a novel approach ((ϵ, δ)-DP, Rényi/ZC/TC)).

In striking this balance, Duet comes with known limitations: (1) Duet is not easy to extend with new relational properties (Rel-ext); and (2) Duet

is not suitable for verifying implementations of low-level mechanisms, such as the implementation of advanced composition, gradient operations, and the sparse-vector technique (SVT-imp).

Chapter 5

The DUET Language

5.1 DUET: A Language for Privacy

This section describes the syntax, type system and formal properties of DUET. Our design of DUET is the result of two key insights.

1. *Linear typing, when restricted to disallow scaling, can be a powerful foundation for enforcing* (ϵ, δ)*-differential privacy.* Privacy bounds in (ϵ, δ)-differential privacy do not scale linearly, and cannot be accurately modeled by linear type systems which permit unrestricted scaling.

2. *Sensitivity and privacy cost are distinct properties, and warrant distinct type systems to enforce them.* Our design for DUET is a co-design of two distinct, mutually embedded languages: one for sensitivity which leverages linear typing *with* scaling a la FUZZ, and one for privacy which leverages linear typing *without* scaling and is novel in this work.

Before describing the syntax, semantics and types for each of DUET's two languages, we first provide some context which motivates each design decision made.

We do this through several small examples and type signatures drawn from state-of-the-art type systems such as Fuzz Reed and Pierce (2010), HOARe^2 Barthe et al. (2015) and Azevedo de Amorim et al's *path construction* de Amorim et al. (2018).

5.1.1 Design Challenges

Higher-Order Programming

An important design goal of Duet is to support sensitivity analysis of higher-order, general purpose programs. Prior work (Fuzz and HOARe^2) has demonstrated exactly this, and we build on their techniques. In Fuzz, the types for the higher-order map function and a list of reals named xs looks like this:

$$\begin{aligned} \text{map} &: (\tau_1 \multimap_s \tau_2) \multimap_\infty \text{list } \tau_1 \multimap_s \text{list } \tau_2 \\ xs &: \text{list } \mathbb{R} \end{aligned}$$

The type of map reads: "Take as a first argument an s-sensitive function from τ_1 to τ_2 which map is allowed to use as many times as it wants. Take as second argument a list of τ_1, and return a result list of τ_2 which is s-sensitive in the list of τ_1." Two programs that use map might look like this:

$$\begin{aligned} &\text{map } (\lambda\ x \to x + 1)\ \text{xs} && (1) \\ &\text{map } (\lambda\ x \to x + x)\ \text{xs} && (2) \end{aligned}$$

The Fuzz type system reports that *(1)* is 1-sensitive in xs, and that *(2)* is 2-sensitive in xs. To arrive at this conclusion, the Fuzz type checker is essentially counting how many times x is used in the body of the lambda, and type soundness for Fuzz means that these counts correspond to the semantic property of function sensitivity.

In HOARE^2 the type for map is instead:

$$\begin{aligned}\text{map} \;:\; & (\forall s'.\ \{x :: \tau_1 \mid \mathfrak{D}_{\tau_1}(x_\triangleleft, x_\triangleright) \leq s'\} \to \{y :: \tau_2 \mid \mathfrak{D}_{\tau_2}(y_\triangleleft, y_\triangleright) \leq s{\cdot}s'\}) \\ & \to \forall s'.\ \{xs :: list\ \tau_1 \mid \mathfrak{D}_{(list\ \tau_1)}(xs_\triangleleft, xs_\triangleright) \leq s'\} \\ & \to \{ys :: list\ \tau_2 \mid \mathfrak{D}_{(list\ \tau_2)}(ys_\triangleleft, ys_\triangleright) \leq s{\cdot}s'\}\end{aligned}$$

This type for map means the same thing as the FUZZ type shown above, and HOARE^2 likewise reports that *(1)* is 1-sensitive and *(2)* is 2-sensitive, each in xs, and where $\mathfrak{D}_\tau$ is some family of distance metrics indexed by types τ. To arrive at this conclusion, HOARE^2 generates relational verification conditions (where, e.g., $x_\triangleleft$ is drawn from a hypothetical "first/left run" of the program, and $x_\triangleright$ is drawn from a hypothetical "second/right run" of the program) which are discharged by an external solver (e.g., SMT). In this approach, sensitivity is not concluded via an interpretation of a purely *syntactic* type system (e.g., linear typing in FUZZ), rather the relational *semantic* property of sensitivity (and its scaling) is embedded directly in the relational refinements of higher-order function types.

In designing DUET, we follow the design of FUZZ in that programs adhere to a linear type discipline, i.e., the mechanics of our type system is based on counting variables and (in some cases) scaling, and we prove a soundness theorem that says well-typed programs are guaranteed to be sensitive/private programs. Our type for map is identical to the one shown above for FUZZ.

Non-Linear Scaling

FUZZ encodes an ϵ-differentially private function as an ϵ-sensitive function which returns a monadic type $\bigcirc\ \tau$. The Laplace differential privacy mechanism is then encoded in FUZZ as an ϵ-sensitive function from $\mathbb{R}$ to $\bigcirc\ \mathbb{R}$:

$$\text{laplace} : \mathbb{R} \multimap_\epsilon \bigcirc\ \mathbb{R}$$

Because the metric on distributions for pure ϵ-differential privacy scales linearly, laplace can be applied to a 2-sensitive argument to achieve 2ϵ-differential privacy, e.g.:

$$\text{laplace}\ (x + x)$$

gives 2ϵ-differential privacy for x. Adding more advanced variants of differential privacy like (ϵ, δ) to Fuzz has proved challenging because these variants do not scale linearly. Azevedo de Amorim et al's *path construction* successfully adds (ϵ, δ)-differential privacy to Fuzz by tracking privacy "cost" as an index on the monadic type operator $\bigcirc_{\epsilon,\delta}$. However, in order to interpret a function application like the one shown, the group privacy property for (ϵ, δ)-differential privacy must be used, which results in undesirable non-linear scaling of the privacy cost. The derived bound for this program using group privacy (for $k = 2$) is not $(2\epsilon, 2\delta)$ but $(2\epsilon, 2e^{\epsilon}\delta)$ Dwork et al. (2014a). As a result, achieving a desired ϵ and δ by treating an s-sensitive function as 1-sensitive and leveraging group privacy requires adding much more noise than simply applying the Gaussian mechanism with a sensitivity of s.

In HOARE^2, the use of scaling which might warrant the use of group privacy is explicitly disallowed in the stated relational refinement type. This is in contrast to sensitivity, which likewise must explicitly allow arbitrary scaling. The type for gauss in HOARE^2 (the analogous mechanism to laplace in the (ϵ, δ)-differential privacy setting) is written:

$$\text{gauss} : \{x :: \mathbb{R} \mid \mathfrak{D}_{\mathbb{R}}(x_{\triangleleft}, x_{\triangleright}) \leq 1\} \to \mathbf{M}_{\epsilon,\delta}\ \mathbb{R}$$

Notice the assumed sensitivity of x to be bounded by 1, not some arbitrary s' to be scaled in the output refinement (as was seen in the type for map in

HOARE^2 above). In this way, HOARE^2 is able to restrict *uses* of gauss to strictly 1-sensitive arguments, a restriction that is not possible in a pure linear type system where arbitrary program composition is allowed and interpreted via scaling.

In DUET, we co-design two languages which are mutually embedded inside one another. The sensitivity language is nearly identical to FUZZ, supports arbitrary scaling, and is never interpreted to mean privacy. The privacy language is also linearly typed, but restricts function call parameters to be strictly 1-sensitive—a property established in the sensitivity fragment. The gauss mechanism in DUET is (essentially) given the type:

$$\text{gauss} : \mathbb{R}@\langle\epsilon, \delta\rangle \multimap^* \mathbb{R}$$

where $\multimap^*$ is the function space in DUET's privacy language, and the annotation $@\langle\epsilon, \delta\rangle$ tracks the privacy cost of that argument following a linear typing discipline.

Multiple Private Parameters Both HOARE^2 and the *path construction* track (ϵ, δ)-differential privacy via an indexed monadic type, notated $\mathbf{M}_{\epsilon,\delta}$ and $\bigcirc_{\epsilon,\delta}$ respectively. E.g., a program that returns an (ϵ, δ)-differentially private real number has the type $\mathbf{M}_{\epsilon,\delta}(\mathbb{R})$ in HOARE^2. These monadic approaches to privacy inherently follow an "effect" type discipline, and as a result the monad index must track the *sum total of all privacy costs to any parameter*. For example, a small program that takes two parameters, applies a mechanism to enforce differential privacy for each parameter, and adds them together, will report a double-counting of privacy cost. E.g., in this HOARE^2 program (translated to

Haskell-ish "do"-notation):

$$\texttt{let}\ f = \lambda\ x\ y \to \texttt{do}\ \{\ r_1 \leftarrow gauss_{\epsilon,\delta}\ x\ ;\ r_2 \leftarrow gauss_{\epsilon,\delta}\ y\ ;\ return\ (r_1 + r_2)\ \}$$

The type of f in HOARE^2 reports that it costs $(2\epsilon, 2\delta)$ privacy:

$$f : \{x :: \mathbb{R} \mid \mathfrak{D}_{\mathbb{R}}(x_{\triangleleft}, x_{\triangleright}) \leq 1\} \to \{y :: \mathbb{R} \mid \mathfrak{D}_{\mathbb{R}}(y_{\triangleleft}, y_{\triangleright}) \leq 1\} \to \mathbf{M}_{2\epsilon,2\delta}\ \mathbb{R}$$

This bound is too conservative in many cases: it is the best bound in the case that f is applied to the same variable for both arguments (e.g., in $f\ a\ a$), however, if f is applied to *different* variables (e.g., in $f\ a\ b$) then a privacy cost of $(2\epsilon, 2\delta)$ is still claimed, interpreted as *for either or both variables* $2\epsilon, 2\delta$ privacy is consumed. A better accounting of privacy in this second case should report (ϵ, δ)-differential privacy *independently* for both variables a and b, and such accounting is not possible in either HOARE^2 or the *path construction*.

In DUET, we track privacy following a *co-effect* discipline (linear typing without scaling), as opposed to an effect discipline, in order to distinguish privacy costs independently for each variable. The type of the above program in DUET is:

$$f : (\mathbb{R}@\langle\epsilon, \delta\rangle, \mathbb{R}@\langle\epsilon, \delta\rangle) \multimap^* \mathbb{R}$$

indicating that f "costs" (ϵ, δ) for each parameter independently, and only when f is called with two identical variables as arguments are they combined as $(2\epsilon, 2\delta)$.

Due to limitations of linear logic in the absence of scaling, privacy lambdas must be multi-argument in the core design of DUET—they cannot be recovered by single-argument lambdas. As a consequence, our privacy language is not Cartesian closed.

5.1.2 DUET by Example

Sensitivity. DUET consists of two languages: one for tracking sensitivities (typeset in green), and one for tracking privacy cost (typeset in red). The sensitivity language is similar to that of DFuzz Gaboardi et al. (2013a); its typing rules track the sensitivity of each variable by annotating the context. For example, the expression $x + x$ is 2-sensitive in x; the typing rules in Figure 5.3 allow us to conclude:

$$\{x :_2 \mathbb{R}\} \vdash x + x : \mathbb{R}$$

In this case, the context $\{x :_2 \mathbb{R}\}$ tells us that the expression is 2-sensitive in x. The same idea works for functions; for example:

$$\emptyset \vdash \lambda x : \mathbb{R} \Rightarrow x + x : \mathbb{R} \multimap_2 \mathbb{R}$$

Here, the context is empty; instead, the function's sensitivity to its argument is encoded in an annotation on its type (the 2 in $\mathbb{R} \multimap_2 \mathbb{R}$). Applying such a function to an argument *scales* the sensitivity of the argument by the sensitivity of the function. This kind of scaling is appropriate for sensitivities, and even has the correct effect for higher-order functions. For example:

$$\{y :_2 \mathbb{R}\} \vdash (\lambda x : \mathbb{R} \Rightarrow x + x)\ y : \mathbb{R}$$
$$\{y :_4 \mathbb{R}\} \vdash (\lambda x : \mathbb{R} \Rightarrow x + x)\ (y + y) : \mathbb{R}$$
$$\{y :_4 \mathbb{R}, z :_2 \mathbb{R}\} \vdash (\lambda x : \mathbb{R} \Rightarrow x + x)\ (y + y + z) : \mathbb{R}$$
$$\{y :_1 \mathbb{R}\} \vdash \lambda x : \mathbb{R} \Rightarrow y : \mathbb{R} \multimap_0 \mathbb{R}$$
$$\{y :_1 \mathbb{R}, z :_0 \mathbb{R}\} \vdash (\lambda x : \mathbb{R} \Rightarrow y)\ z : \mathbb{R}$$
$$\{y :_2 \mathbb{R}, z :_0 \mathbb{R}\} \vdash (\lambda f : \mathbb{R} \multimap_0 \mathbb{R} \Rightarrow (f\ z) + (f\ z))\ (\lambda x : \mathbb{R} \Rightarrow y) : \mathbb{R}$$

Privacy. Differentially private mechanisms like the Gaussian mechanism Dwork et al. (2014a) specify how to add noise to a function with a particular sensitivity in order to ensure differential privacy. In Duet, such mechanisms form the interface between the sensitivity language and the privacy language. For example:

$$\{x :_{\epsilon,\delta} \mathbb{R}\} \vdash \texttt{gauss}[\mathbb{R}^{+}[2.0], \epsilon, \delta] \; <x> \; \{x + x\} : \mathbb{R}$$

In a `gauss` expression, the first three elements (inside the square brackets) represent the maximum allowed sensitivity of variables in the expression's body, and the desired privacy parameters ϵ and δ. The fourth element (here, $<x>$) is a list of variables whose privacy we are interested in tracking. Variables not in this list will be assigned infinite privacy cost.

The value of the `gauss` expression is the value of its fifth element (the "body"), plus enough noise to ensure the desired level of privacy. The body of a `gauss` expression is a sensitivity expression, and the `gauss` expression is well-typed only if its body typechecks in a context assigning a sensitivity to each variable of interest which does not exceed the maximum allowed sensitivity. For example, the expression $\texttt{gauss}[\mathbb{R}^{+}[1.0], \epsilon, \delta] \; <x> \; \{x + x\}$ is not well-typed, because $x + x$ is 2-sensitive in x, but the maximum allowed sensitivity is 1.

Privacy expressions like the example above are typed under a *privacy context* which records privacy cost for individual variables. The context for this example ($\{x :_{\epsilon,\delta} \mathbb{R}\}$) says that the expression provides (ϵ, δ)-differential privacy for the variable x. Tracking privacy costs using a co-effect discipline allows precise tracking of the privacy cost for programs with multiple inputs:

$$\{x :_{\epsilon,\delta} \mathbb{R}, y :_{\epsilon,\delta} \mathbb{R}\} \vdash \texttt{gauss}[\mathbb{R}^{+}[1.0], \epsilon, \delta] \; <x, y> \; \{x + y\} : \mathbb{R}$$

The Bind rule encodes the sequential composition property of differential privacy. For example:

$\{x :_{2\epsilon,2\delta} \mathbb{R}\} \vdash$
$\quad v_1 \leftarrow \texttt{gauss}[\mathbb{R}^+[1.0], \epsilon, \delta] <x> \{x\} ;$
$\quad v_2 \leftarrow \texttt{gauss}[\mathbb{R}^+[1.0], \epsilon, \delta] <x> \{x\} ;$
$\quad \texttt{return } v_1 + v_2$
$: \mathbb{R}$

$\{x :_{\epsilon,\delta} \mathbb{R}, y :_{\epsilon,\delta} \mathbb{R}\} \vdash$
$\quad v_1 \leftarrow \texttt{gauss}[\mathbb{R}^+[1.0], \epsilon, \delta] <x> \{x\} ;$
$\quad v_2 \leftarrow \texttt{gauss}[\mathbb{R}^+[1.0], \epsilon, \delta] <y> \{y\} ;$
$\quad \texttt{return } v_1 + v_2$
$: \mathbb{R}$

In the example on the left, the Gaussian mechanism is applied to x twice, so the total privacy cost for x is $(2\epsilon, 2\delta)$. In the example on the right, x and y are each used once, and their privacy costs are tracked separately. The Return rule provides a second interface between the sensitivity and privacy languages: a *return* expression is part of the privacy language, but its argument is a sensitivity expression. The value of a *return* expression is exactly the value of its argument, so the variables used in its argument are assigned *infinite* privacy cost. *return* expressions are therefore typically used to compute on values which are *already* differentially private (like v_1 and v_2 above), since infinite privacy cost is not a problem in that case.

Gradient descent. Machine learning problems are typically defined in terms of a *loss function* $\mathcal{L}(\theta; X, y)$ on a *model* θ, *training samples* $X = (x_1, x_2, \ldots, x_n)$ (in which each sample is typically represented as a *feature vector*) and corresponding *labels* $y = (y_1, y_2, \ldots, y_n)$ (i.e. the prediction target). The training task is to find a model $\hat{\theta}$ which minimizes the loss on the training samples (i.e. $\hat{\theta} = \text{argmin}_\theta \mathcal{L}(\theta; X, y)$.

One solution to the training task is *gradient descent*, which starts with an initial guess for θ and iteratively moves in the direction of an improved θ until the current setting is close to $\hat{\theta}$. To determine which direction to move, the

algorithm evaluates the *gradient* of the loss, which yields a vector representing the direction of greatest *increase* in $\mathcal{L}(\theta; X, y)$. Then, the algorithm moves in the *opposite* direction.

To ensure differential privacy for gradient-based algorithms, we need to bound the sensitivity of the gradient computation. The gradients for many kinds of convex loss functions are 1-*Lipschitz* Wu et al. (2017): if each sample in $X = (x_1, \ldots, x_n)$ has bounded $L2$ norm (i.e. $\| x_i \|_2 \leq 1$), then for all models θ and labelings y, the gradient $\nabla\ (\theta; X, y)$ has $L2$ sensitivity bounded by 1. For now, we will assume the existence of a function called gradient with this property (more details in Section 6.1).

$$\text{gradient} : \mathbb{M}^{\mathtt{U}}_{\mathtt{L2}}[\mathtt{1}, \mathtt{n}]\ \mathbb{R} \multimap_{\infty} \mathbb{M}^{\mathtt{U}}_{\mathtt{L}\infty}[\mathtt{m}, \mathtt{n}]\ \mathbb{D} \multimap_{\frac{1}{\mathtt{m}}} \mathbb{M}^{\mathtt{U}}_{\mathtt{L}\infty}[\mathtt{m}, \mathtt{1}]\ \mathbb{D} \multimap_{\frac{1}{\mathtt{m}}} \mathbb{M}^{\mathtt{U}}_{\mathtt{L2}}[\mathtt{1}, \mathtt{n}]\ \mathbb{R}$$

The function's arguments are the current θ, a $m \times n$ matrix X containing n training samples, and a $1 \times n$ matrix y containing the corresponding labels. In Duet, the type $\mathbb{M}^{\mathtt{U}}_{L\infty}[m, n]\ \mathbb{D}$ represents a $m \times n$ matrix of *discrete* real numbers; neighboring matrices of this type differ arbitrarily in a single row. The function's output is a new θ of type $\mathbb{M}^{\mathtt{U}}_{L2}[1, n]\ \mathbb{R}$, representing a matrix of real numbers with bounded $L2$ sensitivity (see Section 6.1 for details on matrix types). We can use the gradient function to implement a differentially private gradient descent algorithm:

$$\begin{array}{l}
\text{noisy-gradient-descent}(X, y, k, \epsilon, \delta) \triangleq \\
\quad \mathtt{let}\ \theta_0 = \text{zeros}\ (\text{cols}\ \mathtt{X_1})\ \mathtt{in} \\
\quad \mathtt{loop}[\delta']\ \mathtt{k}\ \mathtt{on}\ \theta_0\ \langle \mathtt{X_1}, \mathtt{y} \rangle\ \{\mathtt{t}, \theta \Rightarrow \\
\qquad \mathtt{g_p} \leftarrow \mathtt{mgauss}[\tfrac{1}{\mathtt{m}}, \epsilon, \delta]\ \langle \mathtt{X}, \mathtt{y} \rangle\ \{\text{gradient}\ \theta\ \mathtt{X}\ \mathtt{y}\}\ ; \\
\qquad \mathtt{return}\ \theta - g_p \quad \}
\end{array}$$

The arguments to our algorithm are the training data (X and y), the desired

number of iterations k, and the privacy parameters ϵ and δ. The first line constructs an initial model θ_0 consisting of zeros for all parameters. Lines 2-4 represent the iterative part of the algorithm: k times, compute the gradient of the loss on X and y with respect to the current model, add noise to the gradient using the Gaussian mechanism, and subtract the gradient from the current model (thus moving in the opposite direction of the gradient) to improve the model.

The typing rules presented in Figure 5.3 allow us to derive a privacy bound for this algorithm which is equivalent to manual proof of Bassily et al. Bassily et al. (2014b). Based on the type of the gradient function, the $\multimap$-E rule allows us to conclude that the gradient operation is $\frac{1}{m}$-sensitive in the training data, which is reflected by the sensitivity annotations in the context:

$$\{\theta :_{\infty} \tau_1, X :_{\frac{1}{m}} \tau_2, y :_{\frac{1}{m}} \tau_3\} \vdash \text{gradient } \theta\ X\ y : \mathbb{M}^{\mathtt{U}}_{L2}[1, n]\ \mathbb{R}$$

$$\textit{where } \tau_1 = \mathbb{M}^{\mathtt{U}}_{L2}[1, n]\ \mathbb{R}$$

$$\tau_2 = \mathbb{M}^{\mathtt{U}}_{L\infty}[m, n]\ \mathbb{D}$$

$$\tau_3 = \mathbb{M}^{\mathtt{U}}_{L\infty}[m, 1]\ \mathbb{D}$$

Next, the MGauss rule represents the use of the Gaussian mechanism, and transitions from the sensitivity language (implementing the gradient) to the privacy language (in which we use the noisy gradient). The rule allows us to conclude that since the sensitivity of the gradient computation is $\frac{1}{m}$, our use of the Gaussian mechanism satisfies (ϵ, δ)-differential privacy. This context is a *privacy* context, and its annotations represent privacy costs rather than sensitivities.

$$\{\theta :_{\infty} \tau_1, X :_{\langle\epsilon,\delta\rangle} \tau_2, y :_{\langle\epsilon,\delta\rangle} \tau_3\} \vdash \mathtt{mgauss}[\tfrac{1}{m}, \epsilon, \delta]\ <X, y>\ \{\text{gradient } \theta\ X\ y\}$$
$$: \mathbb{M}^{\mathtt{U}}_{L2}[1, n]\ \mathbb{R}$$

Finally, the Loop rule for advanced composition allows us to derive a bound on

the total privacy cost of the iterative algorithm, based on the number of times the loop runs:

$$\{\theta :_{\infty} \tau_1, X :_{\langle \epsilon', k\delta+\delta' \rangle} \tau_2, y :_{\langle \epsilon', k\delta+\delta' \rangle} \tau_3\} \vdash \texttt{loop}[\delta'] \texttt{ k on } \theta_0 \texttt{ <X}_1\texttt{, y>} \{\texttt{t}, \theta \Rightarrow \ldots\}$$
$$: \mathbb{M}^{\mathtt{U}}_{L2}[1, n]\ \mathbb{R}$$
$$\textit{where } \epsilon' = 2\epsilon\sqrt{2k\log(1/\delta')}$$

Variants of Differential Privacy. The typing rules presented in Figure 5.3 are specific to (ϵ, δ)-differential privacy, but the same framework can be easily extended to support the other variants described in Figure 2.1. New variants can be supported by making three simple changes: (1) Modify the *privacy cost* syntax p to describe the privacy parameters of the new variant; (2) Modify the sum operator _+_ to reflect sequential composition in the new variant; and (3) Modify the typing for basic mechanisms (e.g. `gauss`) to reflect corresponding mechanisms in the new variant. The extended version of this paper Near et al. (2019a) includes typing rules for the variants in Figure 2.1.

As an example, considering the following variant of the noisy gradient descent algorithm presented earlier, but with ρ-zCDP instead of (ϵ, δ)-differential privacy. There are only two differences: the `loop` construct under zCDP has no δ' parameter, since standard composition yields tight bounds, and the `mgauss` construct has a single privacy parameter (ρ) instead of ϵ and δ.

$$\begin{aligned}
&\text{noisy-gradient-descent}(X, y, k, \rho) \triangleq \\
&\quad \texttt{let } \theta_0 = \text{zeros (cols } \texttt{X}_1\text{) } \texttt{in} \\
&\quad \texttt{loop k on } \theta_0 \texttt{ <X}_1\texttt{, y>} \{\texttt{t}, \theta \Rightarrow \\
&\qquad \texttt{g}_\texttt{p} \leftarrow \texttt{mgauss}[\tfrac{1}{\texttt{m}}, \rho] \texttt{ <X, y>} \{\text{gradient } \theta \texttt{ X y}\} \texttt{ ;} \\
&\qquad \texttt{return } \theta - g_p \quad \}
\end{aligned}$$

Typechecking for this version proceeds in the same way as before, with the

modified typing rules; the resulting privacy context gives both X and y a privacy cost of $k\rho$.

Mixing Variants. DUET allows mixing variants of differential privacy in a single program. For example, the total privacy cost of an algorithm is often given in (ϵ, δ) form, to enable comparing the costs of different algorithms; we can use this feature of DUET to automatically derive the cost of our zCDP-based gradient descent in terms of ϵ and δ.

$$
\begin{array}{l}
\text{noisy-gradient-descent}(X, y, k, \rho, \delta) \triangleq \\
\quad \texttt{let}\ \theta_0 = \text{zeros}\ (\text{cols}\ \texttt{X}_1)\ \texttt{in} \\
\quad \texttt{ZCDP}\ [\delta]\ \{\ \texttt{loop k on}\ \theta_0\ \texttt{<X}_1, \texttt{y>}\ \{\texttt{t}, \theta \Rightarrow \\
\qquad\qquad \texttt{g}_\texttt{p} \leftarrow \texttt{mgauss}[\tfrac{1}{\texttt{m}}, \rho]\ \texttt{<X, y>}\ \{\text{gradient}\ \theta\ \texttt{X y}\}\ ; \\
\qquad\qquad \texttt{return}\ \theta - g_p \quad \}\ \ \}
\end{array}
$$

The `ZCDP` $\{\ldots\}$ construct represents embedding a mechanism which satisfies ρ-zCDP in another mechanism which provides (ϵ, δ)-differential privacy. The rule for typechecking this construct encodes the property that if a mechanism satisfies ρ-zCDP, it also satisfies $(\rho + 2\sqrt{\rho \log(1/\delta)}, \delta)$-differential privacy Bun and Steinke (2016). Using this rule, we can derive a total privacy cost for the gradient descent algorithm in terms of ϵ and δ, but using the tight bound on composition that zCDP provides.

$$
\begin{array}{l}
\{X :_{\langle \epsilon', \delta \rangle} \tau_2, y :_{\langle \epsilon', \delta \rangle} \tau_3, k :_\infty \tau_4, \rho :_\infty \tau_5\} \vdash \text{noisy-gradient-descent}(X, y, k, \rho, \delta) \\
: \mathbb{M}^{\mathtt{U}}_{L2}[1, n]\ \mathbb{R} \\
\textit{where}\ \epsilon' = k\rho + 2\sqrt{k\rho \log(1/\delta)},\ \ \tau_3 = \mathbb{R}^+[k],\ \ \textit{and}\ \ \tau_5 = \mathbb{R}^+[\rho]
\end{array}
$$

We might also want to nest these conversions. For example, when the dimensionality of the training data is very small, the Laplace mechanism might yield more accurate results than the Gaussian mechanism (due to the shape of the

distribution). To use the Laplace mechanism in an iterative algorithm which satisfies zCDP, we can use the fact that any ϵ-differentially private mechanism also satisfies $\frac{1}{2}\epsilon^2$-zCDP; by nesting conversions, we can determine the total cost of the algorithm in terms of ϵ and δ.

$$
\begin{array}{l}
\text{noisy-gradient-descent}(X, y, k, \epsilon, \delta) \triangleq \\
\quad \texttt{let}\ \theta_0 = \text{zeros}\ (\text{cols}\ \texttt{X}_1)\ \texttt{in} \\
\quad \texttt{ZCDP}\ [\delta]\ \{\ \texttt{loop k on}\ \theta_0\ \texttt{<X}_1, \texttt{y>}\ \{\texttt{t}, \theta \Rightarrow \\
\qquad\qquad \texttt{g}_\texttt{p} \leftarrow \texttt{EPS\_DP}\ \{\ \texttt{mlaplace}[\tfrac{1}{\texttt{m}}, \epsilon]\ \texttt{<X, y>}\ \{\text{gradient}\ \theta\ \texttt{X y}\}\ \}\ ; \\
\qquad\qquad \texttt{return}\ \theta - g_p \quad \}\ \ \}
\end{array}
$$

$$
\{X :_{\langle \epsilon', \delta \rangle} \tau_2, y :_{\langle \epsilon', \delta \rangle} \tau_3, k :_\infty \mathbb{R}^+, \rho :_\infty \mathbb{R}^+\} \vdash \text{noisy-gradient-descent}(X, y, k, \epsilon, \delta)
$$

$$
: \mathbb{M}^{\mathtt{U}}_{L2}[1, n]\ \mathbb{R}
$$

where $\epsilon' = \frac{1}{2}k\epsilon^2 + 2\sqrt{\frac{1}{2}k\epsilon^2 \log(1/\delta)}$

Such nestings are sometimes useful in practice: in Section 7.1, we will define a variant of the Private Frank-Wolfe algorithm which uses the exponential mechanism (which satisfies ϵ-differential privacy) in a loop for which composition is performed with zCDP, and report the total privacy cost in terms of ϵ and δ.

Contextual Modal Types

A new problem arises in the design of Duet governing the interaction of sensitivity and privacy languages: in general—and for very good reasons which are detailed in the next section—let-binding intermediate results in the privacy language doesn't always preserve typeability. Not only is let-binding intermediate results desirable for code readability, it can often be *essential* in order to achieve desirable performance. Consider a loop body which performs an expensive

operation that does not depend on the inner-loop parameter:

$$\lambda\ xs\ \theta_0 \to \texttt{loop}\ k\ \texttt{times on}\ \theta_0\ \{\ \theta \to \\ \text{gauss}_{\epsilon,\delta}\ (f\ (\text{expensive}\ xs)\ \theta))\ \}$$

A simple refactoring achieves much better performance:

$$\lambda\ xs\ \theta_0 \to \texttt{let}\ \text{temp} = \text{expensive}\ xs\ \texttt{in} \\ \texttt{loop}\ k\ \texttt{times on}\ \theta_0\ \{\ \theta \to \\ \text{gauss}_{\epsilon,\delta}\ (f\ \text{temp}\ \theta)\ \}$$

However instead of providing (ϵ, δ)-differential privacy for xs, as was the case before the refactor, the new program provides (ϵ, δ)-differential privacy for temp—an intermediate variable we don't care about—and makes no guarantees of privacy for xs.

To accommodate this pattern we borrow ideas from *contextual modal type theory* Nanevski et al. (2008) to allow "boxing" a sensitivity context, and "unboxing" that context at a later time. In terms of differential privacy, the argument that the above loop is differentially private relies on the fact that $\text{temp} \equiv \text{expensive}(xs)$ is 1-sensitive in xs (assuming expensive is 1-sensitive), a property which is lost by the typing rule for `let` in the privacy language. We therefore "box" this sensitivity information outside the loop, and "unbox" it inside the loop, like so:

$$\lambda\ xs\ \theta_0 \to \texttt{let}\ \text{temp} = \texttt{box}\ (\text{expensive}\ xs)\ \texttt{in} \\ \texttt{loop}\ k\ \texttt{times on}\ \theta_0\ \{\ \theta \to \\ \text{gauss}_{\epsilon,\delta}\ (f\ (\texttt{unbox}\ \text{temp})\ \theta)\ \}$$

In this example, the type of temp is a $\square[xs@1]$ `data` (a "box of data 1-sensitive in xs") indicating that when unboxed, temp will report 1-sensitivity w.r.t xs, not temp. f is then able to make good on its promise to gauss that the result of

$$
\begin{array}{rcll}
 & m, n \in \mathbb{N} \qquad r \in \mathbb{R} \qquad \dot{r}, \epsilon, \delta \in \mathbb{R}^{+} \qquad x, y \in \mathrm{var} & & \\
 & s \in \mathrm{sens} ::= \dot{r} \mid \infty \qquad\qquad p \in \mathrm{priv} ::= \epsilon, \delta \mid \infty & & \\
\tau \in \mathrm{type} & ::= & \mathbb{N} \mid \mathbb{R} \mid \mathbb{N}[\mathtt{n}] \mid \mathbb{R}^{+}[\mathtt{r}] \mid \mathtt{box}[\Gamma_s]\ \tau & \textit{numeric and box} \\
 & \mid & \tau \multimap_{\mathtt{s}} \tau \mid (\tau@\mathtt{p}, \ldots, \tau@\mathtt{p}) \multimap^{*} \tau & \textit{functions} \\
\Gamma_s \in \mathrm{tcxt}_s & \triangleq & \mathrm{var} \rightharpoonup \mathrm{sens} \times \mathrm{type} ::= \{x :_s \tau, \ldots, x :_s \tau\} & \textit{sens. contexts} \\
\Gamma_p \in \mathrm{tcxt}_p & \triangleq & \mathrm{var} \rightharpoonup \mathrm{priv} \times \mathrm{type} ::= \{x :_p \tau, \ldots, x :_p \tau\} & \textit{priv. contexts} \\
e_s \in \mathrm{exp}_s & ::= & \mathbb{N}[\mathtt{n}] \mid \mathbb{N}[\mathtt{r}] \mid n \mid r \mid \mathtt{real}\ e & \textit{numeric literals} \\
 & \mid & \mathtt{e} + \mathtt{e} \mid \mathtt{e}\text{-}\mathtt{e} \mid \mathtt{e} \cdot \mathtt{e} \mid 1/\mathtt{e} \mid e\ \mathtt{mod}\ e & \textit{arithmetic} \\
 & \mid & x \mid \mathtt{let}\ \mathtt{x} = \mathtt{e}\ \mathtt{in}\ \mathtt{e} \mid e\ e & \textit{let/sens. app.} \\
 & \mid & \mathtt{s}\lambda\ \mathtt{x} : \tau \Rightarrow \mathtt{e} \mid \mathtt{p}\lambda\ (\mathtt{x} : \tau, \ldots, \mathtt{x} : \tau) \Rightarrow \mathtt{e} & \textit{sens./priv. fun.} \\
 & \mid & \mathtt{box}\ e \mid \mathtt{unbox}\ e & \textit{sensitivity capture} \\
e_p \in \mathrm{exp}_p & ::= & \mathtt{return}\ e \mid \mathtt{x} \leftarrow \mathtt{e}\ ;\ \mathtt{e} \mid \mathtt{e}(\mathtt{e}, \ldots, \mathtt{e}) & \textit{ret/bind/priv. app.} \\
 & \mid & \mathtt{loop}[\mathtt{e}]\ \mathtt{e}\ \mathtt{on}\ \mathtt{e}\ \texttt{<}\mathtt{x}, \ldots, \mathtt{x}\texttt{>}\ \{\mathtt{x}, \mathtt{x} \Rightarrow \mathtt{e}\} & \textit{finite iteration} \\
 & \mid & \mathtt{gauss}[\mathtt{e}, \mathtt{e}, \mathtt{e}]\ \texttt{<}\mathtt{x}, \ldots, \mathtt{x}\texttt{>}\ \{\mathtt{e}\} & \textit{gaussian noise}
\end{array}
$$

Figure 5.1: Core Types and Terms

f is 1-sensitive in xs (assuming f is 1-sensitive in its first argument), and gauss properly reports its privacy "cost" in terms of xs, not temp.

We use exactly this pattern in many of our case studies, where *expensive* is a pre-processing operation on the input data (e.g., clipping or normalizing), and f is a machine-learning training operation, such as computing an improved model based on the current model θ and the pre-processed input data *temp*.

5.1.3 Duet Syntax & Typing Rules

Figure 22.1 shows a core subset of syntax for both languages. We only present the privacy fragment for (ϵ, δ)-differential privacy in the core formalism, although support for other variants (and combined variants) is straightforward as sketched in the previous section. See the extended version of this paper Near et al. (2019a) for the complete presentation of the full language including all advanced variants of differential privacy. We use color coding to distinguish between the sensitivity language, privacy language, and shared syntax between languages. The sensitivity and privacy languages share syntax for variables and types, which

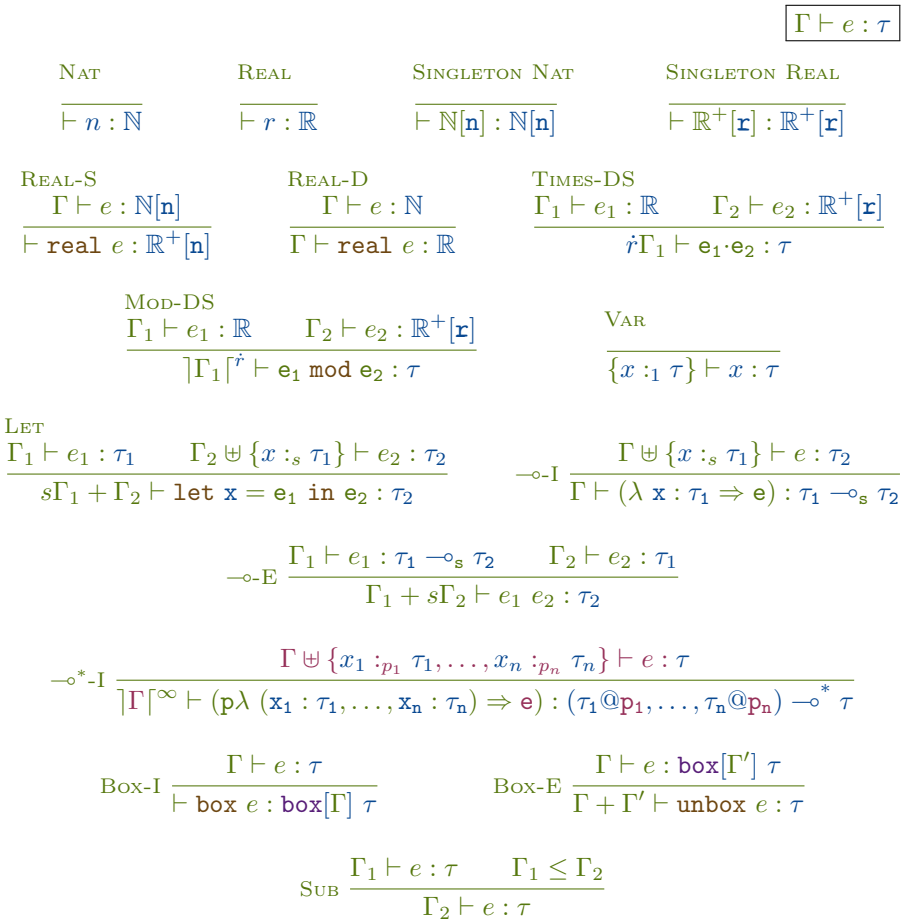

Figure 5.2: Core Typing Rules: Sensitivity

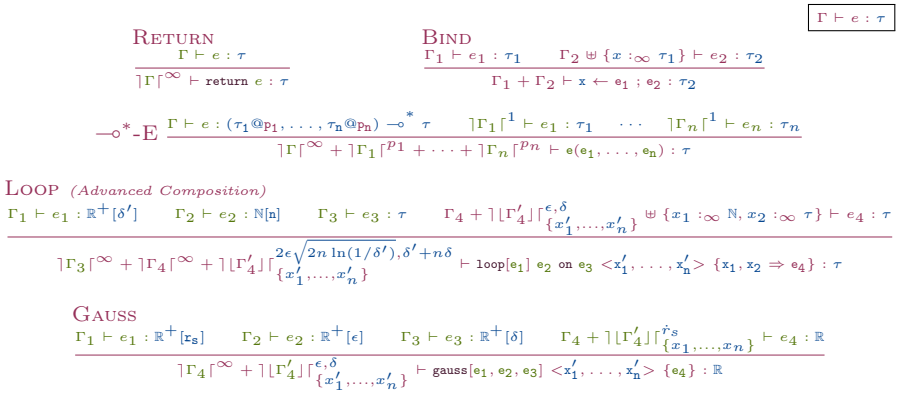

Figure 5.3: Core Typing Rules: Privacy

are typeset in blue. Expressions in the sensitivity language are typeset in green, while expressions in the privacy language are typeset in red.[1]

Types τ include base numeric types $\mathbb{N}$ and $\mathbb{R}$ and their treatment is standard. We include singleton numeric types $\mathbb{N}[n]$ and $\mathbb{R}^+[\dot{r}]$; these types classify runtime numeric values which are identical to the static index n or $\dot{r}$, e.g., $\mathbb{N}[n]$ is a type which exactly describes its runtime value as the number n. Static reals only range over non-negative values, and we write $\dot{r}$ for elements of the non-negative reals $\mathbb{R}^+$. Singleton natural numbers are used primarily to construct matrices with some statically known dimension, and to execute loops for some statically known number of iterations. Singleton real numbers and are used primarily for tracking sensitivity and privacy quantities. Novel in DUET is a "boxed" type $\text{box}[\Gamma_s]\ \tau$ which delays the "payment" of a value's sensitivity, to be unboxed and "paid for" in a separate context. Boxing is discussed in more detail later in this section. The sensitivity function space (a la Fuzz) is written $\tau_1 \multimap_s \tau_2$ and encodes an s-sensitive function from τ_1 to τ_2. The privacy function space (novel in DUET) is written $(\tau_1 @ p_1, \ldots, \tau_n @ p_n) \multimap^* \tau$ and encodes a multi-arity function that preserves p_i-privacy for its ith argument. Privacy functions are

[1] Colors were chosen to minimize ambiguity for colorblind persons following a colorblind-friendly palette: http://mkweb.bcgsc.ca/colorblind/img/colorblindness.palettes.png

multi-arity because functions of multiple arguments cannot be recovered from iterating functions over single arguments in the privacy language, as can be done in the sensitivity language.

In our implementation and extended presentation of DUET in the extended version of this paper, we generalize the static representations of natural numbers and reals to symbolic expression η, which may be arbitrary symbolic polynomial formulas including variables. E.g., suppose ϵ is a type-level variable ranging over real numbers and $x{:}\mathbb{N}[\epsilon]$, then $2x{:}\mathbb{N}[2\epsilon]$. Our type checker knows this is the same type as $\mathbb{N}[\epsilon{+}\epsilon]$ using a custom solver we implemented but do not describe in this paper. Because the typelevel representation of a natural number can be a variable, its value is therefore not statically *determined*, rather it is statically *tracked* via typelevel symbolic formulas.

Type contexts in the sensitivity language Γ_s track the *sensitivity* s of each free variable whereas in the privacy language Γ_p they track *privacy cost* p. Sensitivities are non-negative reals $\dot{r}$ extended with a distinguished infinity element ∞, and privacy costs are specific to the current privacy *mode*. In the case of (ϵ, δ)-differential privacy, p has the form ϵ, δ or ∞ where ϵ and δ range over $\mathbb{R}^+$.

We reuse notation conventions from Fuzz for manipulating contexts, e.g., $\Gamma_1{+}\Gamma_2$ is partial and defined only when both contexts agree on the type of each variable; adding contexts adds sensitivities pointwise, i.e., $\{x{:}_{s_1+s_2}\tau\} \in \Gamma_1{+}\Gamma_2$ when $\{x{:}_{s_1}\tau\} \in \Gamma_1$ and $\{x{:}_{s_2}\tau\} \in \Gamma_2$; and scaling contexts scales sensitivities pointwise, i.e., $\{x{:}_{ss'}\tau\} \in s\Gamma$ when $\{x{:}_{s'}\tau\} \in \Gamma$.

We introduce a new operation not shown in prior work called *truncation* and written $\rceil s_1 \lceil^{s_2}$ for truncating a sensitivity and $\rceil \Gamma \lceil^{s}$ for truncating a sensitivity context, which is pointwise truncation of sensitivities. Sensitivity truncation

$\rceil\_\lceil^s$ maps 0 to 0 and any other value to s:

$$\rceil\_\lceil - \in \text{sens} \times \text{sens} \to \text{sens} \qquad \rceil s_1 \lceil^{s_2} \triangleq \begin{cases} 0 & \textit{if} \quad s_1 = 0 \\ s_2 & \textit{if} \quad s_1 \neq 0 \end{cases}$$

Truncation is defined analogously for privacies $\rceil p_1 \lceil^{p_2}$, for converting between sensitivities and privacies $\rceil s \lceil^p$ and $\rceil p \lceil^s$, and also for liftings of these operations pointwise over contexts $\rceil \Gamma \lceil^p$, $\rceil \Gamma \lceil^p$ and $\rceil \Gamma \lceil^s$. Sensitivity truncation is used for typing the modulus operator, and truncating between sensitivities and privacies is always to ∞/∞ and appears frequently in typing rules that embed sensitivity terms in privacy terms and vice versa.

The syntax and language features for both sensitivity and privacy languages are discussed next alongside their typing rules. Figure 5.3 shows a core subset of typing rules for both languages. In the typing rules, the languages embed within each other—sensitivity typing contexts are transformed into privacy contexts and vice versa. Type rules are written in logical style with an explicit subsumption rule, although a purely algorithmic presentation is possible (not shown) following ideas from Azevedo de Amorim et al De Amorim et al. (2014) which serves as the basis for our implementation.

5.1.4 Sensitivity Language

DUET's sensitivity language is similar to that of DFuzz Gaboardi et al. (2013a), except that we extend it with significant new tools for machine learning in Section 6.1. We do not present standard linear logic connectives such as sums, additive products and multiplicative products (a la Fuzz), or symbolic type-level expressions (a la DFuzz), although each are implemented in our tool and described formally in the extended version of this paper Near et al. (2019a). We do not formalize or implement general recursive types in order to ensure

that all DUET programs terminate. Including general recursive types would be straightforward in DUET (following the design of Fuzz), however such a decision comes with known limitations. As described in Fuzz Reed and Pierce (2010), requiring that all functions terminate is necessary in order to give both sound and useful types to primitives like set-filter. The design space for the combination of sensitivity types and nontermination is subtle, and discussed extensively in prior work Reed and Pierce (2010); Azevedo de Amorim et al. (2017).

Typing for literal values is immediate (NAT, REAL). Singleton values are constructed using the same syntax as their types, and where the type level representation is identical to the literal (SINGLETON NAT, SINGLETON REAL). Naturals can be converted to real numbers through the explicit conversion operation `real` (REAL-S, REAL-D). For the purposes of sensitivity analysis, statically known numbers are considered constant, and as a consequence any term that uses one is considered 0-sensitive in the statically known term. The result of this is that the sensitivity environment Γ associated with the subterm at singleton type is dropped from the output environment, e.g., in REAL-S. This dropping is justified by our metric space interpretation $[\![\mathbb{N}[\mathtt{n}]]\!]$ for statically known numbers as singleton sets $\{n\}$, and because for all $x, y \in [\![\mathbb{N}[\mathtt{n}]]\!]$, $x = y$ and therefore $|x - y| = 0$.

Type rules for arithmetic operations are given in multiple variations, depending on whether or not each argument is tracked statically or dynamically. We show only the rule for multiplication when the left argument is dynamic and the right argument is static (TIMES-DS). The resulting sensitivity environment reports the sensitivities of e_1 scaled by $\dot{r}$—the statically known value of e_2—and the sensitivities for e_2 are not reported because its value is fixed and cannot vary, as discussed above. When both arguments are dynamic, the resulting sensitivity environment is $\infty(\Gamma_1 + \Gamma_2)$, i.e., all potentially sensitive variables

for each expression are bumped to infinity. The modulus operation is similar to multiplication in that we have cases for each variation of static or dynamic arguments, however the context is truncated rather than scaled in the case of one singleton-typed parameter; we show only this static-dynamic variant in the figure (Mod-DS).

Typing for variables (Var) and functions ($\multimap$-I, $\multimap$-E) is the same as in Fuzz: variables are reported in the sensitivity environment with sensitivity 1; and closures are created by annotating the arrow with the sensitivity s of the argument in the body, and by reporting the rest of the sensitivities Γ from the function body as the sensitivity of whole closure as a whole; and function application scales the argument by the function's sensitivity s.

The first new (w.r.t. DFuzz) term in our sensitivity language is the privacy lambda. Privacy lambdas are multi-arity (as opposed to single-arity sensitivity lambdas) because the privacy language does not support currying to recover multi-argument functions. Privacy lambdas are created in the *sensitivity* language with $\mathtt{p}\lambda\ (\mathtt{x} : \tau, \ldots, \mathtt{x} : \tau) \Rightarrow \mathtt{e}$ and applied in the *privacy* language with $\mathtt{e}(\mathtt{e}, \ldots, \mathtt{e})$. The typing rule for privacy lambdas ($\multimap^*$-I) types the body of the lambda in a privacy type context extended with its formal parameters, and the privacy cost of each parameter is annotated on its function argument type. Unlike sensitivity lambdas, the privacy cost of variables in the closure environment are not preserved in the resulting typing judgment. The reason for this is twofold: (1) the final "cost" for variables in the closure environment depends on how many times the closure is called, and in the absence of this knowledge, we must conservatively assume that it could be called an infinite number of times, and (2) the interpretation of an ∞-sensitive function coincides with that of an ∞-private function, so we can soundly convert between ∞-privacy-cost and ∞-sensitivity contexts freely using truncation.

The final two new terms in our sensitivity language are introduction and elimination forms for "boxes" (Box-I and Box-E). Boxes have no operational behavior and are purely a type-level mechanism for tracking sensitivity. The rules for box introduction capture the sensitivity context of the expression, and the rule for box elimination pays for that cost at a later time. Boxes are reminiscent of *contextual modal type theory* Nanevski et al. (2008)—they allow temporary capture of a linear context via boxing—thereby deferring its payment—and re-introduction of the context at later time via unboxing. In a linear type system that supports scaling, this boxing would not be necessary, but it becomes necessary in our system to achieve the desired operational behavior when interacting with the privacy language, which does not support scaling. E.g., in many of our examples we perform some pre-processing on the database parameter (such as clipping) and then use this parameter in the body of a loop. Without boxing, the only way to achieve the desired semantics is to re-clip the input (a deterministic operation) every time around the loop—boxing allows you to clip on the outside of the loop and remember that privacy costs should be "billed" to the initial input.

5.1.5 Privacy Language

Duet's privacy language is designed specifically to enable the composition of individual differentially private computations. It has a linear type system, but unlike the sensitivity language, annotations instead track privacy cost, and the privacy language *does not allow scaling* of these annotations, that is, the notation $p\Gamma$ is not used and cannot be defined. Syntax `return` e and `x←e;e` (pronounced "bind") are standard from Fuzz, as are their typing rules (Return, Bind), except for our explicit conversion from a sensitivity context Γ to a privacy context Γ by truncation to infinity in the conclusion of Return. Bind encodes exactly the

post-processing property of differential privacy—it allows e_2 to use the value computed by e_1 any number of times after paying for it once.

Privacy application $\mathtt{e}(\mathtt{e}, \ldots, \mathtt{e})$ applies a privacy function ($\mathtt{p}\lambda$, created in the sensitivity language) to a sequence of *1-sensitivity* arguments—the sensitivity is enforced by the typing rule. The type rule ($\multimap^*$-E) checks that the first term produces a privacy function and applies its privacy costs to function arguments which are restricted by the type system to be 1-sensitive. We use truncation in well-typed hypothesis for $e_1 \ldots e_n$ to encode the restriction that the argument must be 1-sensitive. This restriction is crucial for type soundness—arbitrary terms cannot be given tight privacy bounds statically due to the lack of a tight scaling operation in the model for (ϵ, δ)-differential privacy. The same is true for other advanced variants of differential privacy.

The `loop` expression is for loop iteration fixed to a statically known number of iterations. The syntax includes a list of variables ($<\mathtt{x}, \ldots, \mathtt{x}>$) to indicate which variables should be considered when calculating final privacy costs, as explained shortly. The typing rule (Loop) encodes advanced composition for (ϵ, δ)-differential privacy. e_1 is the δ' parameter to the advanced composition bound and e_2 is the number of loop iterations—each of these values must be statically known, which we encode with singleton types (a la DFuzz). Statically known values are fixed and their sensitivities do not appear in the resulting context. e_3 is the initial value passed to the loop, and for which no claim is made of privacy, indicated by truncation to infinity. e_4 is a loop body with free variables x_1 and x_2 which will be iterated e_2 times with the first variable bound to the iteration index, and the second variable bound to the loop state, where e_3 is used as the starting value. The loop body e_4 is checked in a privacy context $\Gamma_4 + \lceil\lfloor\Gamma'_4\rfloor\rceil_{\{x'^{\epsilon,\delta}_1,\ldots,x'_n\}}$, shorthand for $\lceil\lfloor\Gamma'_4\rfloor_{\{x'_1,\ldots,x'_n\}}\rceil^{\epsilon,\delta}$ where $\lfloor\Gamma'_4\rfloor_{\{x'_1,\ldots,x'_n\}}$ is a context restricted to only the variables $x'_1, \ldots, x'_n$. The ϵ, δ is an upper bound

on the privacy cost of the variables x_i' in the loop body, and the resulting privacy bound is restricted to only those variables. This allows variables for which the programmer is not interested in tracking privacy to appear in Γ_4 in the premise, and the rule's conclusion makes no claims about privacy for these variables. We make use of this feature in all of our examples programs.

The `gauss` expression is a *mechanism* of (ϵ, δ)-differential privacy; other mechanisms are used for other privacy variants. Like the loop expression, mechanism expressions take a list of variables to indicate which variables should be considered in the final privacy cost. The typing rule (Gauss) is similar in spirit to Loop: it takes parameters to the mechanism which must be statically known (encoded as singleton types), a list of variables to consider for the purposes of the resulting privacy bound, and a term $\{\texttt{e}\}$ for which there is a bound $\dot{r}$ on the sensitivity of free variables $x_1, \ldots, x_n$. The resulting privacy guarantee is that the term in brackets $\{\texttt{e}\}$ is ϵ, δ differentially private. Whereas `loop` and advanced composition consider a *privacy term* loop body with an upper bound on *privacy leakage*, `gauss` considers a *sensitivity term* body with an upper bound on its *sensitivity*.

5.1.6 Metatheory

We denote sensitivity language terms $e \in \text{exp}$ into total, functional, linear maps between metric spaces—the same model as the terminating fragment of Fuzz. Every term in our language terminates by design, which dramatically simplifies our models and proofs. This restriction poses no issues in implementing most differentially private machine learning algorithms, because such algorithms typically terminate in a statically determined number of loop iterations in order to achieve a particular privacy cost.

Types in Duet denote metric spaces, as in Fuzz. We notate metric spaces D,

their underlying carrier set $\| D \|$, and their distance metric $|x - y|_D$, or $|x - y|$ where D can be inferred from context. Sensitivity typing judgments $\Gamma \vdash e : \tau$ denote linear maps from a scaled cartesian product interpretation of Γ:

$$[\![\Gamma\{x_1 :_{s_1} \tau_1, \ldots, x_n :_{s_n} \tau_n\} \vdash \tau]\!] \triangleq !_{s_1}[\![\tau_1]\!] \otimes \cdots \otimes !_{s_n}[\![\tau_n]\!] \multimap [\![\tau]\!]$$

Although we do not make metric space scaling explicit in our syntax (for the purposes of effective type inference, a la DFuzz De Amorim et al. (2014)), scaling becomes apparent explicitly in our model. Privacy judgments $\Gamma \vdash e : \tau$ denote *probabilistic, privacy preserving maps* from an *unscaled* product interpretation of Γ:

$$[\![\Gamma\{x_1 :_{p_1} \tau_1, \ldots, x_n :_{p_n} \tau_n\} \vdash \tau]\!] \triangleq ([\![\tau_1]\!]@p_1, \ldots, [\![\tau_n]\!]@p_n) \multimap^* \| [\![\tau]\!] \|$$

The multi-arity (ϵ, δ)-differential-privacy-preserving map is defined:

$$\begin{aligned}
&(D_1@(\epsilon_1, \delta_1), \ldots, D_n@(\epsilon_n, \delta_n)) \multimap^* X \triangleq \\
&\quad \{\ f \in \| D_1 \| \times \cdots \times \| D_n \| \to \mathcal{D}(X) \\
&\quad \mid\ |x_i - y|_{D_i} \leq 1 \Rightarrow \Pr[f(x_1, \ldots, x_i, \ldots, x_n) = d] \\
&\quad \leq e^{\epsilon_i} \Pr[f(x_1, \ldots, y, \ldots, x_n) = d] + \delta_i\ \}
\end{aligned}$$

where $\mathcal{D}(X)$ is a distribution over elements in X.

We give a full semantic account of typing in the extended version of this paper Near et al. (2019a), as well as prove key type soundness lemmas, many of which appeal to well-known differential privacy proofs from the literature.

The final soundness theorem, proven by induction over typing derivations, is that the denotations for well-typed open terms e_s and e_p in well-typed environments γ_s and γ_p are contained in the denotation of their typing contexts $\Gamma_s \vdash \tau$ and $\Gamma_p \vdash \tau$.

Theorem 5.1.1.

1. *If* $\Gamma_p \vdash e_p : \tau$ *and* $\Gamma_p \vdash \gamma_p$ *then* $[\![e_p]\!]^{\gamma_p} \in [\![\Gamma_p \vdash \tau]\!]$

2. *If* $\Gamma_s \vdash e_s : \tau$ *and* $\Gamma_s \vdash \gamma_s$ *then* $[\![e_s]\!]^{\gamma_s} \in [\![\Gamma_s \vdash \tau]\!]$

A corollary is that any well-typed privacy lambda function satisfies (ϵ, δ)-differential privacy for each of its arguments w.r.t. that argument's privacy annotation used in typing.

Chapter 6

Machine Learning in DUET

6.1 Language Tools for Machine Learning

Machine learning algorithms typically operate over a training set of *samples*, and implementations of these algorithms often represent datasets using matrices. To express these algorithms, DUET includes a core matrix API which encodes sensitivity and privacy properties of matrix operations.

We add a matrix type $\mathbb{M}^c_\ell[m, n]\ \tau$, encode vectors as single-row matrices, and add typing rules for gradient computations that encode desirable properties. We also introduce a type for matrix indices $\texttt{idx}[n]$ for type-safe indexing. These new types are shown in Figure 6.2, along with sensitivity operations on matrices—encoded as library functions because their types can be encoded using existing connectives—and new matrix-level differential P privacy P mechanisms—encoded as primitive syntactic forms because their types *cannot* be expressed using existing type-level connectives.

In the matrix type $\mathbb{M}^c_\ell[m, n]\ \tau$, the m and n parameters refer to the number of rows and columns in the matrix, respectively. The ℓ parameter determines

the distance metric used for the matrix metric for the purposes of sensitivity analysis; the c parameter is used to specify a norm bound on each row of the matrix, which will be useful when applying gradient functions.

6.1.1 Distance Metrics for Matrices

Differentially private machine learning algorithms typically move from one distance metric on matrices and vectors to another as the algorithm progresses. For example, two input training datasets are neighbors if they differ on exactly one sample (i.e. one row of the matrix), but they may differ arbitrarily in that row. After computing a gradient, the algorithm may consider the $L2$ sensitivity of the resulting vector—i.e. two gradients g_1 and g_2 are neighbors if $\| g_1 - g_2 \|_2 \leq 1$. These are very different notions of distance—but the first is required by the definition of differential privacy, and the second is required as a condition on the input to the Gaussian mechanism.

The ℓ annotation on matrix types in Duet enables specifying the desired notion of distance between *rows*. The annotation is one of $L\infty$, $L1$, or $L2$; an annotation of $L\infty$, for example, means that the distance between two rows is equal to the $L\infty$ norm of the difference between the rows. The distance between two matrices is always equal to the sum of the distances between rows. The distance metric for the element datatype τ determines the distance between two corresponding elements, and the row metric ℓ specifies how to combine elementwise distances to determine the distance between two rows.

Figure 6.1 presents the complete set of distance metrics for matrices, as well as real numbers and the new domain data for elements of the $\mathbb{D}$ type, which is operationally a copy of $\mathbb{R}$ but with a discrete distance metric. Many combinations are possible, including the following common ones:

Ex. 1: $|X - X'|_{\mathbb{M}^{\mathtt{u}}_{\mathtt{L}\infty}[\mathtt{m},\mathtt{n}]\ \mathbb{D}} = \sum_i \max_j |X_{i,j} - X'_{i,j}|_{\mathbb{D}}$

Domain	*Carrier*: $X \in \text{set}$	*Metric*: $\lvert\_ - \_\rvert \in X \to X \to \mathbb{R} \uplus \{\infty\}$
real	$\mathbb{R}$	$\lvert r_1 - r_2\rvert \triangleq \lvert r_1 - r_2\rvert_{\mathbb{R}}$
data	$\mathbb{R}$	$\lvert r_1 - r_2\rvert \triangleq \begin{cases} 0 \text{ when } r_1 = r_2 \\ 1 \text{ when } r_1 \neq r_2 \end{cases}$
$\text{matrix}[n_1, n_2]_{L\infty}(D)$	$\mathbb{M}[n_1, n_2](\lVert\ D\ \rVert)$	$\lvert m_1 - m_2\rvert \triangleq \sum_i \max_j \lvert m_1[i,j] - m_2[i,j]\rvert_D$
$\text{matrix}[n_1, n_2]_{L1}(D)$	$\mathbb{M}[n_1, n_2](\lVert\ D\ \rVert)$	$\lvert m_1 - m_2\rvert \triangleq \sum_{i,j} \lvert m_1[i,j] - m_2[i,j]\rvert_D$
$\text{matrix}[n_1, n_2]_{L2}(D)$	$\mathbb{M}[n_1, n_2](\lVert\ D\ \rVert)$	$\lvert m_1 - m_2\rvert \triangleq \sum_i \sqrt{\sum_j \lvert m_1[i,j] - m_2[i,j]\rvert_D^2}$

Figure 6.1: Distance Metrics for Matrices

Distance is the *number of rows on which* X *and* X' *differ*; commonly used to describe neighboring input datasets.

Ex. 2: $|X - X'|_{\mathbb{M}^{\mathtt{U}}_{\mathtt{L1}}[\mathtt{m},\mathtt{n}]\ \mathbb{R}} = \sum_i \sum_j |X_{i,j} - X'_{i,j}|_{\mathbb{R}}$

Distance is the *sum of elementwise differences.*

Ex. 3: $|X - X'|_{\mathbb{M}^{\mathtt{U}}_{\mathtt{L2}}[\mathtt{m},\mathtt{n}]\ \mathbb{R}} = \sum_i \sqrt{\sum_j |X_{i,j} - X'_{i,j}|^2_{\mathbb{R}}}$

Distance is *sum of the* $L2$ *norm of the differences between corresponding rows.*

Ex. 4: $|X - X'|_{\mathbb{M}^{\mathtt{U}}_{\mathtt{L2}}[\mathtt{1},\mathtt{n}]\ \mathbb{R}} = \sqrt{\sum_j |X_{1,j} - X'_{1,j}|^2_{\mathbb{R}}}$

Represents a vector; distance is $L2$ sensitivity for vectors, as required by the Gaussian mechanism.

These distance metrics are used in the types of library functions which operate over matrices.

6.1.2 Matrix Operations

Figure 6.2 summarizes the matrix operations available in Duet's API. We focus on the non-standard operations which are designed specifically for sensitivity or privacy applications. For example, fr-sens allows converting between notions of distance between rows; when converting from $L2$ to $L1$, the distance between

$$\ell \in \text{norm} ::= \texttt{L1} \mid \texttt{L2} \mid \texttt{L}\infty \quad c \in \text{clip} ::= \ell \mid \texttt{U} \quad \tau \in \text{type} ::= \ldots \mid \mathbb{D} \mid \texttt{idx}[\texttt{n}] \mid \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{n},\texttt{n}]\ \tau$$

$$\begin{array}{rl}
\text{rows} : \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{m},\texttt{n}]\ \tau \multimap_0 \mathbb{N}[\texttt{m}] & \text{convert} : \mathbb{M}^{\ell}_{\ell'}[\texttt{m},\texttt{n}]\ \mathbb{D} \multimap_1 \mathbb{M}^{\texttt{U}}_{\ell}[\texttt{m},\texttt{n}]\ \mathbb{R} \\
\text{cols} : \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{m},\texttt{n}]\ \tau \multimap_0 \mathbb{N}[\texttt{n}] & \text{clip}^{\ell} : \mathbb{M}^{\texttt{c}}_{\ell'}[\texttt{m},\texttt{n}]\ \mathbb{D} \multimap_1 \mathbb{M}^{\ell}_{\ell'}[\texttt{m},\texttt{n}]\ \mathbb{D} \\
\text{discf} : (\tau \multimap_\infty \mathbb{R}) \multimap_1 \tau \multimap_1 \mathbb{D} & \text{fr-sens}^{\texttt{L}\infty} : \mathbb{M}^{\texttt{c}}_{\texttt{L}\infty}[\texttt{m},\texttt{n}]\ \tau \multimap_{\sqrt{\texttt{n}}} \mathbb{M}^{\texttt{c}}_{\texttt{L2}}[\texttt{m},\texttt{n}]\ \tau \\
\text{undisc} : \mathbb{D} \multimap_\infty \mathbb{R} & \text{fr-sens}^{\texttt{L2}} : \mathbb{M}^{\texttt{c}}_{\texttt{L2}}[\texttt{m},\texttt{n}]\ \tau \multimap_{\sqrt{\texttt{n}}} \mathbb{M}^{\texttt{c}}_{\texttt{L1}}[\texttt{m},\texttt{n}]\ \tau \\
\text{transpose} : \mathbb{M}^{\texttt{c}}_{\texttt{L1}}[\texttt{m},\texttt{n}]\ \tau \multimap_1 \mathbb{M}^{\texttt{U}}_{\texttt{L1}}[\texttt{n},\texttt{m}]\ \tau & \text{to-sens}^{\ell} : \mathbb{M}^{\texttt{c}}_{\texttt{L1}}[\texttt{m},\texttt{n}]\ \tau \multimap_1 \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{m},\texttt{n}]\ \tau
\end{array}$$

$$\begin{array}{rl}
\text{mcreate} : & \mathbb{N}[\texttt{m}] \multimap_0 \mathbb{N}[\texttt{n}] \multimap_0 (\texttt{idx}[\texttt{m}] \multimap_\infty \texttt{idx}[\texttt{n}] \multimap_\infty \tau) \multimap_{\texttt{mn}} \mathbb{M}^{\texttt{U}}_{\texttt{L1}}[\texttt{m},\texttt{n}]\ \tau \\
_ \,\#\, [_,_] : & \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{m},\texttt{n}]\ \tau \multimap_1 \texttt{idx}[\texttt{m}] \multimap_\infty \texttt{idx}[\texttt{n}] \multimap_\infty \tau \\
_ \,\#\, [_,_ \mapsto _] : & \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{m},\texttt{n}]\ \tau \multimap_1 \texttt{idx}[\texttt{m}] \multimap_\infty \texttt{idx}[\texttt{n}] \multimap_\infty \tau \multimap_1 \mathbb{M}^{\texttt{U}}_{\ell}[\texttt{m},\texttt{n}]\ \tau \\
\text{fld} : & (\tau_1 \multimap_{\texttt{s}_1} \tau_2 \multimap_{\texttt{s}_2} \tau_3) \multimap_{\texttt{mn}} \tau_2 \multimap_{\texttt{s}_2^{\texttt{mn}}} \mathbb{M}^{\texttt{c}}_{\texttt{L1}}[\texttt{m},\texttt{n}]\ \tau_1 \multimap_{\texttt{s}_1} \tau_2 \\
\text{map} : & (\tau_1 \multimap_{\texttt{s}} \tau_2) \multimap_{\texttt{mn}} \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{m},\texttt{n}]\ \tau_1 \multimap_{\texttt{s}} \mathbb{M}^{\texttt{U}}_{\ell}[\texttt{m},\texttt{n}]\ \tau_2 \\
\text{fld-row} : & (\tau_1 \multimap_{\texttt{s}_1} \tau_2 \multimap_{\texttt{s}_2} \tau_2) \multimap_{\texttt{m}} \tau_2 \multimap_{\texttt{s}_2^{\texttt{m}}} \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{m},\texttt{n}]\ \tau_1 \multimap_{\texttt{s}_1} \mathbb{M}^{\texttt{U}}_{\ell}[\texttt{m},\texttt{1}]\ \tau_2 \\
\text{map-row} : & (\mathbb{M}_{\ell_1^{\texttt{c}_1}}[\texttt{1},\texttt{n}_1]\ \tau_1 \multimap_{\texttt{s}} \mathbb{M}_{\ell_2^{\texttt{c}_2}}[\texttt{1},\texttt{n}_2]\ \tau_2) \multimap_{\texttt{m}} \mathbb{M}_{\ell_1^{\texttt{c}_1}}[\texttt{m},\texttt{n}_1]\ \tau_1 \multimap_{\texttt{s}} \mathbb{M}_{\ell_2^{\texttt{c}_2}}[\texttt{m},\texttt{n}_2]\ \tau_2 \\
\text{L}\ \nabla^{g}_{\ell}[_;_,_] : & \mathbb{M}^{\ell}_{\ell'}[\texttt{1},\texttt{n}]\ \mathbb{R} \multimap_\infty \mathbb{M}^{\ell}_{\ell''}[\texttt{1},\texttt{n}]\ \mathbb{D} \multimap_1 \mathbb{D} \multimap_1 \mathbb{M}^{\texttt{U}}_{\ell}[\texttt{1},\texttt{n}]\ \mathbb{R} \\
\text{U}\ \nabla[_;_,_] : & \mathbb{M}^{\ell}_{\ell'}[\texttt{1},\texttt{n}]\ \mathbb{R} \multimap_\infty \mathbb{M}^{\ell''}_{\texttt{L}\infty}[\texttt{1},\texttt{n}]\ \mathbb{D} \multimap_1 \mathbb{D} \multimap_1 \mathbb{M}^{\texttt{U}}_{\texttt{L}\infty}[\texttt{1},\texttt{n}]\ \mathbb{D}
\end{array}$$

$$\begin{array}{rl}
\text{above-threshold} : & (\mathbb{M}^{\texttt{c}}_{\ell}[\texttt{1},\texttt{n}]\ (\tau \multimap_1 \mathbb{R})@\infty,\ \mathbb{R}^+[\epsilon]@\texttt{0},\ \tau@\langle\epsilon,\texttt{0}\rangle,\ \mathbb{R}@\infty) \multimap^* \texttt{idx}[\texttt{n}] \\
\text{pfld-rows} : & (\mathbb{M}^{\texttt{c}_1}_{\texttt{L}\infty}[\texttt{m},\texttt{n}_1]\ \mathbb{D}@\langle\epsilon,\delta\rangle,\ \mathbb{M}^{\texttt{c}_2}_{\texttt{L}\infty}[\texttt{m},\texttt{n}_2]\ \mathbb{D}@\langle\epsilon,\delta\rangle, \\
& ((\mathbb{M}^{\texttt{c}_1}_{\texttt{L}\infty}[\texttt{1},\texttt{n}_1]\ \mathbb{D}@\langle\epsilon,\delta\rangle,\ \mathbb{M}^{\texttt{c}_2}_{\texttt{L}\infty}[\texttt{1},\texttt{n}_2]\ \mathbb{D}@\langle\epsilon,\delta\rangle,\ \mathbb{D}@\infty) \multimap^* \tau)@\infty, \\
& \tau@\infty \\
&) \multimap^* \tau \\
\text{sample} : & (\mathbb{N}[\texttt{m}_2]@\langle\texttt{0},\texttt{0}\rangle, \\
& \mathbb{M}^{\texttt{c}}_{\texttt{L}\infty}[\texttt{m}_1,\texttt{n}_1]\ \mathbb{D}@\langle 2\texttt{m}_2\epsilon_1/\texttt{m}_1, \texttt{m}_2\delta_1/\texttt{m}_1\rangle,\ \mathbb{M}^{\texttt{c}}_{\texttt{L}\infty}[\texttt{m}_1,\texttt{n}_2]\ \mathbb{D}@\langle 2\texttt{m}_2\epsilon_2/\texttt{m}_1, \texttt{m}_2\delta_2/\texttt{m}_1\rangle, \\
& ((\mathbb{M}^{\texttt{c}}_{\texttt{L}\infty}[\texttt{m}_2,\texttt{n}_1]\ \mathbb{D}@\langle\epsilon_1,\delta_1\rangle,\ \mathbb{M}^{\texttt{c}}_{\texttt{L}\infty}[\texttt{m}_2,\texttt{n}_2]\ \mathbb{D}@\langle\epsilon_2,\delta_2\rangle) \multimap^* \tau)@\infty \\
&) \multimap^* \tau
\end{array}$$

$\boxed{\Gamma \vdash e : \tau}$

MGauss

$$\frac{\Gamma_1 \vdash e_1 : \mathbb{R}^+[\texttt{r}] \qquad \Gamma_2 \vdash e_2 : \mathbb{R}^+[\epsilon] \qquad \Gamma_3 \vdash e_3 : \mathbb{R}^+[\delta] \qquad \Gamma_4 + \rceil\lfloor\Gamma_5\rfloor\lceil^{\dot{r}}_{\{x_1,\ldots,x_n\}} \vdash e_4 : \mathbb{M}^{\texttt{c}}_{\texttt{L2}}[\texttt{m},\texttt{n}]\ \mathbb{R}}{\rceil\Gamma_1 + \Gamma_2 + \Gamma_3\lceil^{0,0} + \rceil\Gamma_4\lceil^{\infty} + \rceil\Gamma_5\lceil^{\epsilon,\delta} \vdash \texttt{mgauss}[\texttt{e}_1,\texttt{e}_2,\texttt{e}_3]\ <\texttt{x}_1,\ldots,\texttt{x}_n>\ \{\texttt{e}_4\} : \mathbb{M}^{\texttt{U}}_{\texttt{L}\infty}[\texttt{m},\texttt{n}]\ \mathbb{R}}$$

Exponential

$$\frac{\Gamma_1 \vdash e_1 : \mathbb{R}^+[\texttt{r}] \qquad \Gamma_2 \vdash e_2 : \mathbb{R}^+[\epsilon] \qquad \Gamma_3 \vdash e_3 : \mathbb{M}^{\texttt{c}}_{\ell}[\texttt{1},\texttt{m}](\tau) \qquad \Gamma_4 + \rceil\lfloor\Gamma_5\rfloor\lceil^{\dot{r}}_{\{x_1,\ldots,x_n\}} \uplus \{x :_\infty \tau\} \vdash e_4 : \mathbb{R}}{\rceil\Gamma_1 + \Gamma_2\lceil^{0,0} + \rceil\Gamma_3 + \Gamma_4\lceil^{\infty} + \rceil\Gamma_5\lceil^{\epsilon,0} \vdash \texttt{exponential}[\texttt{e}_1,\texttt{e}_2]\ <\texttt{x}_1,\ldots,\texttt{x}_n>\ \texttt{e}_3\ \{\texttt{x} \Rightarrow \texttt{e}_4\} : \tau}$$

Figure 6.2: Matrix Typing Rules

two rows may increase by $\sqrt{n}$ (by Cauchy-Schwarz), so the corresponding version of fr-sens has a sensitivity annotation of $\sqrt{n}$.

undisc allows converting from discrete to standard reals, and is infinitely sensitive. discf allows converting an infinitely sensitive function which returns a real to a 1-sensitive function returning a discrete real; we can recover a 1-sensitive function from reals to discrete reals (disc : $\mathbb{R} \multimap_1 \mathbb{D}$) by applying `discf` to the identity function.

Pabove-threshold$P$ encodes the *Sparse Vector Technique* Dwork et al. (2014a), discussed in the extended version of this paper Near et al. (2019a). Ppfld-rows$P$ encodes parallel composition of privacy mechanisms, and is discussed in Section 7.1.5. Psample$P$ performs random subsampling with privacy amplification, and is discussed in Section 7.1.4.

Gradients are computed using L $\nabla^g_{M\ell M}[\_;\_,\_]$ and U $\nabla[\_;\_,\_]$. The first represents an ℓ-Lipschitz gradient (typical in convex optimization problems like logistic regression) like the gradient function introduced in Section 5.1.2; it is a 1-sensitive function which produces a matrix of real numbers. The second represents a gradient *without* a known Lipschitz constant (typical in non-convex optimization problems, including training neural networks); it produces a matrix of discrete reals. We demonstrate applications of both in Section 7.1.

In order to produce a matrix with sensitivity bound L2, $L\ \nabla^g_{M\mathtt{L2}M}$ requires input of type $\mathrm{M}\mathbb{M}^{\mathtt{L2}}_{\ell}[\mathtt{m},\mathtt{n}]\ \mathbb{D}M$ for any ℓ. We obtain such a matrix by *clipping*, a common operation in differentially private machine learning. Clipping scales each row of a matrix to ensure its c norm (for $c \in \{L\infty, L1, L2\}$) is less than 1:

$$\mathrm{clip}^c\ x_i \triangleq \begin{cases} \frac{x_i}{\|x_i\|_c} & \mathit{if} \quad \|\ x_i\ \|_c > 1 \\ x_i & \mathit{if} \quad \|\ x_i\ \|_c \leq 1 \end{cases}$$

The clipping process is encoded in Duet as clip (Figure 6.2), which introduces

a new bound on the c norm of its output.

6.1.3 Vector-Valued Privacy Mechanisms

Both the Laplace and Gaussian mechanisms are capable of operating directly over vectors; the Laplace mechanism adds noise calibrated to the $L1$ sensitivity of the vector, while the Gaussian mechanism uses its $L2$ sensitivity. With the addition of matrices to Duet, we can introduce typing rules for these vector-valued mechanisms, using single-row matrices to represent vectors. We present the typing rule for MGauss in Figure 6.2; the rule for MLaplace is similar. We also introduce a typing rule for the exponential mechanism, which picks one element out of an input vector based on a sensitive scoring function (Figure 6.2, rule Exponential).

Chapter 7

Case Studies in DUET

7.1 Case Studies

In this section, we demonstrate the use of DUET to express and verify a number of different algorithms for differentially private machine learning.

There are four basic approaches to differentially private convex optimization: input perturbation Chaudhuri et al. (2011), objective perturbation Chaudhuri et al. (2011), gradient perturbation Song et al. (2013); Bassily et al. (2014b), and output perturbation Chaudhuri et al. (2011); Wu et al. (2017). Of these, the latter three are known to provide competitive accuracy, and the latter two (gradient perturbation and output perturbation) are the most widely used; our first two case studies verify these two techniques. Our third case study verifies the noisy Frank-Wolfe algorithm Talwar et al. (2015), a variant of gradient perturbation especially suited to high-dimensional datasets.

Our next three case studies demonstrate the use of DUET to verify commonly-used variations on the above algorithms, including various kinds of minibatching and a gradient clipping approach used in deep learning. Our final three case stud-

ies explore techniques for preprocessing input datasets so that the preconditions of the above algorithms are satisfied.

In Section 7.1.6, we discuss the use of DUET to combine all of these components—many of which leverage *different* variants of differential privacy—to build a complete machine learning system. Our case studies are summarized in the following table.

Technique	Ref.	§	Privacy Concept
Optimization Algorithms			
Noisy Gradient Descent	Bassily et al. (2014b)	7.1.1	Composition
Gradient Descent w/ Output Perturbation	Wu et al. (2017)	7.1.2	Parallel comp. (sens.)
Noisy Frank-Wolfe	Talwar et al. (2015)	7.1.3	Exponential mechanism
Variations on Gradient Descent			
Minibatching	Bassily et al. (2014b)	7.1.4	Ampl. by sampling
Parallel-composition minibatching	—	7.1.5	Parallel composition
Gradient clipping	Abadi et al. (2016)	†	Sensitivity bounds
Preprocessing & Deployment			
Hyperparameter tuning	Chaudhuri and Vinterbo (2013)	†	Exponential mechanism
Adaptive clipping	—	†	Sparse Vector Technique
Z-Score normalization	skl (2019)	†	Composition
Combining All of the Above		7.1.6	Composition

7.1.1 Noisy Gradient Descent

We begin with a fully-worked version of the differentially-private gradient descent algorithm from Section 5.1.2. This algorithm was first proposed by Song et al. Song et al. (2013) and later refined by Bassily et al. Bassily et al. (2014b). Gradient descent is a simple but effective training algorithm in machine learning, and has been applied in a wide range of contexts, from simple linear models to deep neural networks. The program below implements noisy gradient descent in DUET (without minibatching, though we will extend it with minibatching in Section 7.1.4). It performs k iterations of gradient descent, starting from an initial guess θ_0 consisting of all zeros. At each iteration, the algorithm computes a noisy gradient using noisy-grad, scales the gradient by the *learning rate* η, and

† these case studies appear in this paper Near et al. (2019a).

subtracts the result from the current model θ to arrive at the updated model.

```
noisy-grad(θ, X, y, ϵ, δ) ≜
  let s = ℝ[1.0]/real (rows X) in
  let z = zeros (cols X) in
  let gs = mmap-row (sλ Xᵢ yᵢ ⇒
              L ∇^LR_L2[θ; Xᵢ, yᵢ]) X y in
  let g = fld-row (sλ x₁ x₂ ⇒ x₁ + x₂) z gs in
  let gₛ = map (sλ x ⇒ s · x) g in
  mgauss[s, ϵ, δ] <X, y> {gₛ}
```

```
zeros(n) ≜ mcreate_L∞ 1 n (sλ i j ⇒ 0.0)

noisy-gradient-descent(X, y, k, η, ϵ, δ) ≜
  let X₁ = box (mclip^L2 X) in
  let θ₀ = zeros (cols X₁) in
  loop[δ'] k on θ₀ <X₁, y> {t, θ ⇒
      gₚ ← noisy-grad θ (unbox X₁) y ϵ δ ;
      return θ − η·gₚ   }
```

Under (ϵ, δ)-differential privacy, DUET derives a total privacy cost of $(2\epsilon\sqrt{2k\log(1/\delta')}, k\delta + \delta')$-differential privacy for this implementation, which matches the total cost manually proven by Bassily et al. Bassily et al. (2014b). DUET can also derive a total cost for other privacy variants: the same program satisfies $k\rho$-zCDP, or $(\alpha, k\epsilon)$-RDP.

7.1.2 Output Perturbation Gradient Descent

An alternative to gradient perturbation is *output perturbation*—adding noise to the final trained model, rather than during the training process. Wu et al. Wu et al. (2017) present a competitive algorithm based on this idea, which works by bounding the *total sensitivity* (rather than privacy) of the iterative gradient descent process. Their algorithm leverages *parallel composition* for sensitivity: it divides the dataset into small chunks called *minibatches*, and each iteration of the algorithm processes one minibatch. A single pass over all minibatches (and thus, the whole dataset) is often called an *epoch*. If the dataset has size m and each minibatch is of size b, then each epoch comprises m/b iterations of the training algorithm. This approach to minibatching is often used (without privacy) in deep learning. The sensitivity of a complete epoch in this technique is just $1/b$.

We encode parallel composition for sensitivity in DUET using the `mfold-row` function, defined in Section 6.1, whose type matches that of `foldl` for lists in the FUZZ type system Reed and Pierce (2010). `mfold-row` considers each row to be a "minibatch" of size 1, but is easily extended to consider multiple rows at a time (as in our encoding below). DUET derives a sensitivity bound of k/b for the training process, and a total privacy cost of (ϵ, δ)-differential privacy, matching the manual analysis of Wu et al. Wu et al. (2017).

```
gd-output-perturbation(xs, ys, k, η, ϵ, δ) ≜
  let m₀ = zeros (cols X) in
  let c = box (mclip^L2 xs) in
  let s = real k/real b in
  mgauss[s, ϵ, δ] <xs, ys> {
    loop k on m₀ { a, θ ⇒
      mfold-row b, θ, unbox c, ys { θ, xb, yb ⇒
        let g = ∇^LR_L2 [θ ; xb, yb] in
          θ-η·g } } }
```

7.1.3 Noisy Frank-Wolfe

We next consider a variation on gradient perturbation called the private Frank-Wolfe algorithm Talwar et al. (2015). This algorithm has dimension-independent utility, making it useful for high-dimensional datasets. In each iteration, the algorithm takes a step of fixed size in a *single* dimension, using the exponential mechanism to choose the best direction based

```
frank-wolfe X y k ϵ δ ≜
  let X₁ = clip-matrix_L∞ X in
  let d = cols X in
  let θ₀ = zeros d in
  let idxs = mcreate_L∞[1,2·d]{i,j ⇒
              ⟨j mod d,sign(j-d)⟩} in
  ZCDP [δ] { loop k on θ₀ {t, θ ⇒
    let μ = 1.0/((real t) + 2.0) in
    let g = L ∇^LR_L∞[θ; X₁, y] in
    ⟨i, s⟩ ← EPS_DP {
      exponential[1/rows X₁, ϵ] idxs {⟨i, s⟩ ⇒
        s·g # [0, i]} ;}
    let g_p = (zeros d)# [0, i ↦ s·100] in
    return ((1.0 − μ)·θ) + (μ·g_p) } }
```

on the gradient. The sensitivity of each update is therefore dependent on the L_∞ norm of each sample, rather than the L_2 norm.

Our implementation uses the exponential mechanism to select the direction in which the gradient has its maximum value, then updates θ in only the selected dimension. To get the right sensitivity, we compute the gradient with $L\ \nabla^{LR}_{\mathtt{L}\infty}$, which requires an $\mathtt{L}\infty$ norm bound on its input and ensures bounded $L\infty$ sensitivity.

We mix several variants of differential privacy in this implementation. Each use of the exponential mechanism provides ϵ-differential privacy; each iteration of the loop satisfies $\frac{1}{2}\epsilon^2$-zCDP, and the whole algorithm satisfies $(\frac{1}{2}\epsilon^2 + 2\sqrt{\frac{1}{2}\epsilon^2 \log(1/\delta)}, \delta)$-differential privacy. The use of zCDP for composition is an improvement over the manual analysis of Talwar et al. Talwar et al. (2015), which used advanced composition.

7.1.4 Minibatching

An alternative form of minibatching to the one discussed in Section 7.1.2 is to randomly sample a subset of of the data in each iteration. Bassily et al. Bassily et al. (2014b) present an algorithm for differentially private stochastic gradient descent based on this idea: their approach samples a single random example from the training to compute the gradient in each iteration, and leverages the idea of *privacy amplification* to improve privacy cost. The privacy amplification lemma states that if mechanism $\mathcal{M}(D)$ provides (ϵ, δ)-differential privacy for the datset D of size n, then running $\mathcal{M}$ on uniformly random γn entries of D (for $\gamma \leq 1$) provides $(2\gamma\epsilon, \gamma\delta)$-differential privacy Bassily et al. (2014b); Wang et al. (2018) (this bound is loose, but used here for readability).

We encode the privacy amplification lemma in Duet using the `sample` construct defined in Section 6.1. Similar privacy amplification lemmas exist for RDP Wang et al. (2018) and tCDP Bun et al. (2018), but not for zCDP. We can use sampling with privacy amplification to implement minibatching SGD in Duet. Under (ϵ, δ)-differential privacy with privacy amplification, Duet derives a total privacy cost of $(4(b/m)\epsilon\sqrt{2k\log(1/\delta')}, (b/m)k\delta + \delta')$-differential privacy for this algorithm, which is equivalent to the manual proof of Bassily et al. Bassily et al. (2014b).

```
minibatch-gradient-descent X y k b η ϵ δ ≜
  let X₁ = clip-matrix X in
  loop [δ] k on zeros (cols X₁) <X₁, y> {t, θ ⇒
    sample b on X₁, y {X₁′, y′ ⇒
      gₚ ← noisy-grad θ X₁′ y′ ϵ δ ; return θ-η·gₚ }}
```

7.1.5 Parallel-Composition Minibatching

As a final form of minibatching, we consider extending the parallel composition approach used by Wu et al. Wu et al. (2017) for *sensitivity* to parallel composition of *privacy mechanisms* for minibatching in the gradient perturbation approach from Section 7.1.1. Since the minibatches are disjoint in this approach, we can leverage the parallel composition property for privacy mechanisms (McSherry McSherry (2009), Theorem 4; Dwork & Lei Dwork and Lei (2009), Corollary 20), which states that running an (ϵ, δ)-differentially private mechanism k times on k disjoint subsets of a database yields (ϵ, δ)-differential privacy. We encode this concept in Duet using the `pfld-rows` construct defined in Section 6.1. The arguments to `pfld-rows` include the dataset and a function representing an (ϵ, δ)-differentially private mechanism, and `pfld-rows` ensures (ϵ, δ)-differential privacy for the dataset. This version considers minibatches of size 1, and is easily extended to consider other sizes. We can use `pfld-rows` to implement

epoch-based minibatching with gradient perturbation, even for privacy variants like zCDP which do not admit sampling:

```
epoch b ρ η ≜
 pλ xs ys θ ⇒
  let s = ℝ⁺[1.0]/real b in
  g ← mgauss[s, ρ] <xs, ys> {∇^LR [θ ; xs, ys] } ;
  return θ-η·g
```

```
epoch-minibatch-GD X y ρ η k b ≜
 let m₀ = zeros (cols xs) in
 loop k on m₀ <X, y> {a, θ ⇒
  pfld-rows(b, θ, mclip^L2 X, y, epoch b ρ η)
 }
```

This algorithm is similar in concept to the output perturbation approach of Wu et al. Wu et al. (2017), but leverages parallel composition of privacy mechanisms for gradient perturbation instead, and has not been previously published. The algorithm runs k epochs with a batch size of b, for a total of kb iterations. Duet derives a privacy cost of $k\rho$-zCDP for the algorithm.

7.1.6 Composing Privacy Variants to Build Complete Learning Systems

Putting together the pieces we have described to build real machine learning systems that preserve differential privacy often requires mixing privacy variants in order to obtain optimal results. We can use Duet's ability to mix variants of differential privacy to combine components in a way that optimizes the use of the privacy budget. We demonstrate this ability with an example that performs several data-dependent analyses as pre-processing steps before training a model. Our example uses

```
adaptiveClippingGradientDescent xs ys k ϵ δ ηs bs ≜
 means ← colMeans(xs, ϵ, δ, bs);
 scales ← EPS_DP { colScaleParams(xs, ϵ, bs, means) };
 let xsₙ = box (normalize xs means scales) in
 η ← pick_η(unbox xsₙ, ys, k, ϵ, δ, ηs);
 ZCDP[δ]{noisyGradientDescentZCDP(b·(unbox xsₙ), ys, k, η, ϵ, δ)}
```

DUET's ability to mix variants to compose z-score normalization (using both pure ϵ and (ϵ, δ)-differential privacy), hyperparameter tuning (with (ϵ, δ)-differential privacy), and gradient descent (with zCDP), returning a total (ϵ, δ) privacy cost.

Chapter 8

Evaluation of DUET

8.1 Implementation & Evaluation

This section describes our implementation of DUET, and our empirical evaluation of DUET's ability to produce accurate differentially private models. Our results demonstrate that the state-of-the-art privacy bounds derivable by DUET can result in huge gains in accuracy for a given level of privacy.

8.1.1 Implementation & Typechecking Performance

We have implemented a prototype of DUET in Haskell that includes type inference of privacy bounds, and an interpreter that runs on all examples described in this paper. We do not implement Hindley-Milner-style constraint-based

Technique	LOC	Time (ms)
Noisy G.D.	23	0.51ms
G.D. + Output Pert.	25	0.39ms
Noisy Frank-Wolfe	31	0.59ms
Minibatching	26	0.51ms
Parallel minibatching	42	0.65ms
Gradient clipping	21	0.40ms
Hyperparameter tuning	125	3.87ms
Adaptive clipping	68	1.01ms
Z-Score normalization	104	1.51ms

Figure 8.1: Summary of Typechecking Performance on Case Study Programs

type inference of quantified types; our type inference is syntax-directed and limited to construction of privacy bounds as symbolic formulas over input variables. Our implementation of type inference roughly follows the bottom-up approach of DFuzz's implementation De Amorim et al. (2014). Type checking requires solving constraints over symbolic expressions containing log and square root operations. Prior work (DFuzz and HOARe^2) uses an SMT solver during typechecking to check validity of these constraints, but SMT solvers typically do not support operators like log and square root, and struggle in the presence of non-linear formulas. Because of these limitations, we implement a custom solver for inequalities over symbolic real expressions instead of relying on support from off-the-shelf solvers. Our custom solver is based on a simple decidable (but incomplete) theory which supports log and square root operations, and a more general subset of non-linear (polynomial) formulas than typical SMT theories.

The Duet typechecker demonstrates very practical performance. Figure 8.1 summarizes the number of lines of code and typechecking time for each of our case study programs; even medium-size programs with many functions typecheck in just a few milliseconds.

8.1.2 Evaluation of Private Gradient Descent and Private Frank-Wolfe

We also study the accuracy of the models produced by the Duet implementations of private gradient descent and private Frank-Wolfe in Section 7.1. We evaluate both algorithms on 4 datasets. Details about the datasets can be found in Figure 8.2.

We ran both algorithms on each dataset with per-iteration $\epsilon_i \in \{0.0001, 0.001, 0.01, 0.1\}$ and then used DUET to derive the corresponding total privacy cost. We fixed $\delta = \frac{1}{n^2}$, where n is the size of the dataset. For private gradient descent, we set $\eta = 1.0$, and for private Frank-Wolfe we set the size of each corner $c = 100$.

We randomly shuffled each dataset, then chose 80% of the dataset as training data and reserved 20% for testing. We ran each training algorithm 5 times on the training data, and take the average testing error over all 5, to account for the randomness in the training process.

We present the results in Figure 8.3. Both algorithms are capable of generating accurate models at reasonable values of ϵ. Note that all three models in the results provide *exactly the same privacy guarantee* for a given value of ϵ, yet their accuracies vary significantly—demonstrating the advantages of recently developed variants of differential privacy.

Dataset	**Samples**	**Dim.**
Synthetic	10,000	20
Adult	45,220	104
KDDCup99	70,000	114
Facebook	40,949	54

Figure 8.2: Dataset Used in Accuracy Evaluation

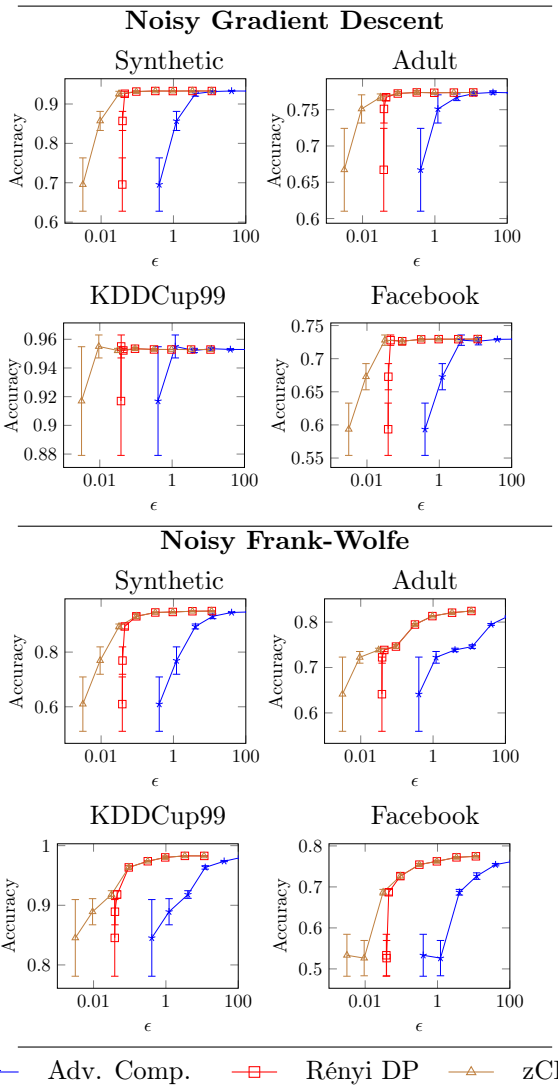

Figure 8.3: Accuracy Results for Noisy Gradient Descent (Top) and Noisy Frank-Wolfe (Bottom).

Part III

DDuo: General-Purpose Dynamic Analysis for Differential Privacy

Chapter 9

Introduction & Contributions

Differential privacy enables general statistical analysis of data with formal guarantees of privacy protection at the individual level. Tools that assist data analysts with utilizing differential privacy have frequently taken the form of programming languages and libraries. However, many existing programming languages designed for compositional verification of differential privacy impose significant burden on the programmer (in the form of complex type annotations). Supplementary library support for privacy analysis built on top of existing general-purpose languages has been more usable, but incapable of pervasive end-to-end enforcement of sensitivity analysis and privacy composition.

We introduce DDuo, a *dynamic analysis* for enforcing differential privacy. DDuo is usable by non-experts: its analysis is automatic and it requires no additional type annotations. DDuo can be implemented *as a library* for existing programming languages; we present a reference implementation in Python which features moderate runtime overheads on realistic workloads. We include support for several data types, distance metrics and operations which are commonly used in modern machine learning programs. We also provide initial support for

tracking the sensitivity of data transformations in popular Python libraries for data analysis.

We formalize the novel core of the DDuo system and prove it sound for sensitivity analysis via a logical relation for metric preservation. We also illustrate DDuo's usability and flexibility through various case studies which implement state-of-the-art machine learning algorithms.

Both static and dynamic tools have been developed to help non-experts write differentially private programs. Many of the static tools take the form of statically-typed programming languages, where correct privacy analysis is built into the soundness of the type system. However, existing language-oriented tools for compositional verification of differential privacy impose significant burden on the programmer (in the form of additional type annotations) (Reed and Pierce, 2010; Gaboardi et al., 2013b; Near et al., 2019b; de Amorim et al., 2019; Zhang et al., 2019a; Winograd-Cort et al., 2017; Barthe et al., 2019, 2012, 2013, 2016b; Sato et al., 2019; Albarghouthi and Hsu, 2018; Zhang and Kifer, 2017; Wang et al., 2019; Bichsel et al., 2018; Ding et al., 2018; Wang et al., 2020) (see Chapter 25 for a longer discussion).

The best-known dynamic tool is PINQ (McSherry, 2009), a dynamic analysis for sensitivity and privacy. It features an extensible system which allows non-experts in differential privacy to execute SQL-like queries against relational databases. However, PINQ comes with several restrictions that limit its applicability. For example, PINQ's expressiveness is limited to a subset of the SQL language for relational databases. Methods in PINQ are assumed to be side-effect free, which is necessary to preserve their privacy guarantee.

We introduce DDuo, a dynamic analysis for enforcing differential privacy. DDuo is usable by non-experts: its analysis is automatic and it requires no additional type annotations. DDuo can be implemented *as a library* for existing

programming languages; we present a reference implementation in Python. Our goal in this work is to answer the following four questions, based on the limitations of PINQ:

- Can a PINQ-style dynamic analysis extend to base types in the programming language, to allow its use pervasively?
- Is the analysis sound in the presence of side effects?
- Can we use this style of analysis for complex algorithms like differentially private gradient descent?
- Can we extend the privacy analysis beyond pure ϵ-differential privacy?

We answer all four questions in the affirmative, building on PINQ in the following ways:

- DDuo provides a dynamic analysis for base types in a general purpose language (Python). DDuo supports general language operations, such as mapping arbitrary functions over lists, and tracks the sensitivity (stability) and privacy throughout.
- Methods in DDuo are not required to be side-effect free and allow programmers to mutate references inside functions which manipulate sensitive values.
- DDuo supports various notions of sensitivity and arbitrary distance metrics (including L_1 and L_2 distance).
- DDuo is capable of leveraging advanced privacy variants such as (ϵ, δ) and Rényi differential privacy.

Privacy analysis is reliant on *sensitivity* analysis, which determines the scale of noise an analyst must add to values in order to achieve any level of privacy. Dynamic analysis for differential privacy is thus a dual challenge:

9.1 Dynamic sensitivity analysis.

Program sensitivity is a (hyper)property quantified over two runs of a program with related inputs (sources). A major challenge for dynamic sensitivity analysis is the ability to bound sensitivity, ensuring that the metric preservation property is satisfied, by only observing *a single run* of the program. In addition, an analysis which is performed on a specific input to the program must generalize to future possible arbitrary inputs.

The key insight to our solution is attaching sensitivity environments and distance metric information to *values* rather than variables. Our approach provides a sound upper bound on global sensitivity even in the presence of side effects, conditionals, and higher-order functions. We present a proof using a *step-indexed logical relation* which shows that our sensitivity analysis is sound.

9.2 Dynamic privacy analysis.

To implement a dynamic privacy analysis, we leverage prior work on privacy filters and odometers (Rogers et al., 2016). This work, originally designed for the adaptive choice of privacy parameters, can also be used as part of a dynamic analysis for privacy analysis. We view each application of a privacy mechanism (e.g. the Laplace mechanism) as a *global privacy effect* on total privacy cost, and use privacy filters and odometers to track total privacy cost.

We implemented these features in a Python prototype of DDuo via object proxies and other pythonic idioms. We implement several case studies to showcase these features and demonstrate the usage of DDuo in practice. We also provide integrations with several popular Python libraries for data and privacy analysis.

9.3 Contributions.

In summary, DDuo makes the following contributions:

- We introduce DDuo, a *dynamic* analysis for enforcing differential privacy, and a reference implementation as a Python library [1].
- We formalize a subset of DDuo in a core language model, and prove the soundness of DDuo's dynamic sensitivity analysis (as encoded in the model) using a step-indexed logical relation.
- We present several case studies demonstrating the use of DDuo to build practical, verified Python implementations of complex differentially private algorithms.

[1] The reference implementation is available here: https://github.com/uvm-plaid/dduo-python

Chapter 10

Overview of DDuo

DDuo is a *dynamic* analysis for enforcing differential privacy. Our approach does not require static analysis of programs, and allows DDuo to be implemented as a library for programming languages like Python. DDuo's dynamic analysis has complete access to run-time information, so it does not require the programmer to write any additional type annotations—in many cases, DDuo can verify differential privacy for essentially unmodified Python programs (see the case studies in Chapter 15). As a Python library, DDuo is easily integrated with popular libraries like Pandas and NumPy.

Challenges of dynamic analysis for differential privacy. Differential privacy is an example of a *hyperproperty*—a property that relates two executions of a program on two different inputs. Verifying a hyperproperty via dynamic analysis is challenging, because in this setting, only one execution of the program is observable. Control flow is particularly difficult: only one branch of a conditional is visible to a dynamic analysis, for example, while a static analysis can examine both branches. This is the key challenge of DDuo—to build a sound dynamic analysis for a hyperproperty.

Advantages and disadvantages of dynamic privacy analysis Dynamic privacy analysis of general purpose programs is valuable due to its usability and practicality for a wide variety of programmers. DDuo takes a lightweight approach based on implementation around existing general-purpose languages such as Python. A static analyzer for all of Python is unrealistic; and it would be very inconvenient to learn an entirely new language with a confusing/unfamiliar type system. Tools such as Python, Pandas, and NumPy are incredibly popular, and a runtime privacy monitor built around these tools is much more convenient. However there are also certain disadvantages to the dynamic approach: it is impossible to obtain a privacy bound without running the program, and there may be instrumentation overhead cost during runtime.

10.1 Threat model.

We assume an "honest but fallible" programmer—that is, the programmer *intends* to produce a differentially private program, but may unintentionally introduce bugs. We assume that the programmer is *not* intentionally attempting to subvert DDuo's enforcement approach. Our reference implementation is embedded in Python, an inherently dynamic language with run-time features like reflection. In this setting, a malicious programmer or privacy-violating third-party libraries can bypass our dynamic monitor and extract sensitive information directly. We allow several common side-effects such as reference mutation, printing, reading/writing files, etc. Note that printing/writing sensitive values in DDuo will reveal the type of the value, but not the actual value. Data-independent exceptions can be safely used in our system, however our model must explicitly avoid data-dependent exceptions such as division-by-zero errors. Terminated programs can be rerun safely (while consuming the privacy budget) because our analysis is independent of any sensitive information (our metatheory implies that

sensitivity of a value is itself not sensitive). We also do not address side-channels, including execution time. Like existing enforcement approaches (PINQ, OpenDP, Diffprivlib), DDuo is intended as a tool to help well-intentioned programmers produce correct differentially private algorithms.

Soundness of the analysis. We formalize our dynamic sensitivity analysis and prove its soundness in Chapter 14. Our formalization includes the most challenging features of the dynamic setting—conditionals and side effects—and provides evidence that our Python implementation will be effective in catching privacy bugs in real programs. DDuo relies on existing work on privacy filters and odometers (discussed in Chapter 13), whose soundness has been previously established, for tracking privacy cost.

Chapter 11

DDuo by Example

This chapter introduces the DDuo system via examples written using our reference Python implementation.

Data Sources. Data sources are wrappers around sensitive data that enable tracking of privacy information in the DDuo python library. Each data source is associated with an identifying string, such as the name of the input file the data was read from. Data sources can be created manually by attaching an identifying string (such as a filename) to a raw value (such as a vector). Or, data sources be created automatically upon loading data through DDuo's custom-wrapped third party APIs, such as pandas. Note that our API can be easily modified to account for initial sensitivities greater than 1 when users have multiple datapoints in the input data.

```
from dduo import pandas as pd
df = pd.read_csv("data.csv")
df
```

```
Sensitive(<'DataFrame'>, {data.csv ↦ 1}, L∞)
```

A `Sensitive` value is returned. `Sensitive` values represent sensitive information that cannot be viewed by the analyst. When a `Sensitive` value is printed out, the analyst sees (1) the type of the value, (2) its *sensitivity environment*, and (3) its *distance metric*. The latter two components are described next. The analyst is *prevented* from viewing the value itself.

11.1 Sensitivity & distance metrics.

Function sensitivity is a scalar value which represents how much a change in a function's input will change the function's output. For example, the binary addition function $f(x, y) = x + y$ is 1-sensitive in both x and y, because changing either input by n will change the sum by n. The function $f(x) = x + x$, on the other hand, is 2-sensitive in its argument x, because changing x by n changes the function's output by $2n$. Sensitivity is key to differential privacy because it is directly proportional to the amount of noise we must add to the output of a function to make it private.

A sensitivity environment is a mapping of program variables to their sensitivities. For example, in the program $f(x) = x + x$, the sensitivity ennvironment is $x : 2$. In the program $f(x, y) = x + y$, the sensitivity environment is $x : 1, y : 1$. Note that the identity function $f(x) = x$ that does nothing with its input and simply returns it as output has a sensitivity environment of $x : 1, y : 1$

DDuo tracks the sensitivity of a value to changes in the program's inputs using a *sensitivity environment* mapping input data sources to sensitivities. Our example program returned a `Sensitive` value with a sensitivity environment of $\{data.csv \mapsto 1\}$, indicating that the underlying value is 1-sensitive in the data contained in *data.csv*. The DDuo library tracks and updates the sensitivity

environments of `Sensitive` objects as operations are applied to them. For example, adding a constant value to the elements of the DataFrame results in no change to the sensitivity environment.

```
df + 5 # no change to sensitivity environment
```

```
Sensitive(<'DataFrame'>, {data.csv ↦ 1}, L∞)
```

Adding the DataFrame to *itself* doubles the sensitivity, in the same way as the function $f(x) = x + x$.

```
df + df  # doubles the sensitivity
```

```
Sensitive(<'DataFrame'>, {data.csv ↦ 2}, L∞)
```

Finally, multiplying the DataFrame by a constant scales the sensitivity, and multiplying the DataFrame by *itself* results in *infinite* sensitivity.

```
( df * 5, df * df)
```

```
( Sensitive(<'DataFrame'>, {data.csv ↦ 5},  L∞),
  Sensitive(<'DataFrame'>, {data.csv ↦ ∞}, L∞) )
```

The *distance metric* component of a `Sensitive` value describes *how* to measure sensitivity. For simple numeric functions like $f(x) = x + x$, the distance between two possible inputs x and x' is simply $|x - x'|$ (this is called the *cartesian metric*). For more complicated data structures (e.g. DataFrames), calculating the distance between two values is more involved. The L_∞ metric used in our example calculates the distance between two DataFrames by measuring how many rows are different (this is one standard way of defining "neighboring

databases" in differential privacy). DDuo's handling of distance metrics is detailed in Section 12.2.

11.2 Privacy.

DDuo also tracks the *privacy* of computations. To achieve differential privacy, programs add noise to sensitive values. The Laplace mechanism described earlier is one basic mechanism for achieving differential privacy by adding noise drawn from the Laplace distribution (DDuo provides a number of basic mechanisms, including the Gaussian mechanism). The following expression counts the number of rows in our example DataFrame and uses the Laplace mechanism to achieve ϵ-differential privacy, for $\epsilon = 1.0$.

```
dduo.laplace(df.shape[0], ϵ=1.0)
```

```
9.963971319623278
```

The result is a *regular Python value*—the analyst is free to view it, write it to a file, or do further computation on it. Once the correct amount of noise has been added, the principle of *post-processing* applies, and so DDuo no longer needs to track the sensitivity or privacy cost of operations on the value.

When the Laplace mechanism is used multiple times, their privacy costs *compose* (i.e. the ϵs "add up" as described earlier). DDuo tracks *total* privacy cost using objects called *privacy odometers* (Rogers et al., 2016). The analyst can interact with a privacy odometer object to learn the total privacy cost of a complex computation.

```
with dduo.EpsOdometer() as odo:
  _ = dduo.laplace(df.shape[0], ϵ = 1.0)
  _ = dduo.laplace(df.shape[0], ϵ = 1.0)
  print(odo)
```

```
Odometer_ϵ({data.csv ↦ 2.0})
```

Printing the odometer's value allows the analyst to view the privacy cost of the program with respect to each of the data sources used in the computation. In this example, two differentially private approximations of the number of rows in the dataframe *df* are computed, each with a privacy cost of $\epsilon = 1.0$. The total privacy cost of running the program is therefore $2{\cdot}\epsilon = 2.0$.

DDuo also allows the analyst to place upper bounds on total privacy cost (i.e. a privacy *budget*) using privacy *filters* (Rogers et al., 2016). Privacy odometers and filters are discussed in detail in Chaper 13.

Chapter 12

Dynamic Sensitivity Tracking

DDuo implements a *dynamic sensitivity analysis* by wrapping values in `Sensitive` objects and calculating sensitivities as operations are performed on these objects. Type systems for sensitivity (Reed and Pierce, 2010; Gaboardi et al., 2013b) construct a sensitivity environment for each program expression; in the static analysis setting, a sensitivity environment records the expression's sensitivity with respect to each of the variables currently in scope.

DDuo attaches sensitivity environments to *values* at runtime: each `Sensitive` object holds both a value and its sensitivity environment. As described earlier, DDuo's sensitivity environments record a value's sensitivity with respect to each of the program's data sources.

Formally, the sensitivity of a single-argument function f in its input is defined as:

$$\text{sens}(f) \triangleq \text{argmax}_{x,y}\left(\frac{d(f(x), f(y))}{d(x, y)}\right)$$

Where d is a *distance metric* over the values x and y could take (distance metrics are discussed in Section 12.2). Thus, a sensitivity environment $\{a \mapsto 1\}$ means

that if the value of the program input a changes by n, then the value of $f(a)$ will change by at most n.

12.1 Bounding the Sensitivity of Operations

Operations on `Sensitive` objects are defined to perform the same operation on the underlying values, and *also* construct a new sensitivity environment for the operation's result. For example, DDuo's `__add__` operation sums both the underlying values *and* their sensitivity environments:

```
def __add__(self, other):
    assert self.metric == other.metric
    return dduo.Sensitive(self.value + other.value,
                          self.senv + other.senv,
                          self.metric)
```

The sum of two sensitivity environments is defined as the element-wise sum of their items. For example:

$$\{a \mapsto 2, b \mapsto 1\} + \{b \mapsto 3, c \mapsto 5\} = \{a \mapsto 2, b \mapsto 4, c \mapsto 5\}$$

The DDuo library provides sensitivity-aware versions of Python's basic numeric operations (formalized in Chapter 14). We have also defined sensitivity-aware versions of commonly-used library functions, including the Pandas functions used in Chapter **??**, and subsets of NumPy and Scikit-learn.

12.2 Distance Metrics

At the core of the concept of sensitivity is the notion of distance: how far apart we consider two information sources to be from each other. For scalar values, the following two distance metrics are often used:

- Cartesian (absolute difference) metric: $d(x, y) = |x - y|$
- Discrete metric: $d(x, y) = 0 \ \textit{if} \ x = y; 1 \ \textit{otherwise}$

For more complex structures—like lists and dataframes—we can use distance metrics on *vectors*. Two commonly-used metrics for vectors x and y of equal length are:

- $L_1(d_i)$ metric: $d(x, y) = \sum_{x_i, y_i \in x, y} d_i(x_i, y_i)$
- $L_2(d_i)$ metric: $d(x, y) = \sqrt{\sum_{x_i, y_i \in x, y} d_i(x_i, y_i)^2}$

Both metrics are parameterized by d_i, a metric for the vector's elements. In addition to these two, we use the shorthand L_∞ to mean $L_1(d)$, where d is the cartesian metric defined above. The L_∞ metric works for any space with equality (e.g. strings), and measures the *number of elements where x and y differ*.

The definition of differential privacy is parameterized by a distance metric that is intended to capture the idea of two inputs that *differ in one individual's data*. Database-oriented algorithms typically assume that each individual contributes exactly one row to the database, and use the L_∞ metric to define neighboring databases (as we did in Chapter **??**).

Distance metrics can be manipulated manually through operations such as clipping, a technique commonly employed in differentially private machine learning. DDuo tracks distance metrics for `Sensitive` information, which can allow for automatic conservation of the privacy budget while providing more accurate query analysis.

Lists and arrays are compared by one of the L_1, L_2, or L_∞ distance metrics. The choice of distance metric is important when defining sensitivity and thus privacy. For example, the Laplace mechanism can only be used with the L_1 metric, while the Gaussian mechanism can be used with either L_1 or L_2.

12.3 Conditionals & Side Effects

Conditionals and other branching structures are challenging for any sensitivity analysis, but they present a particular challenge for our dynamic analysis. Consider the following conditional:

```
if df.shape[0] == 10:
  return df.shape[0]
else:
  return df.shape[0] * 10000
```

Here, the two branches have *different* sensitivities (the *else* branch is 10,000 times more sensitive in its data sources than the *then* branch). Static sensitivity analyses handle this situation by taking the maximum of the two branches' sensitivities (i.e. they assume the worst-case branch is executed), but this approach is not possible in our dynamic analysis.

In addition, special care must be taken when a sensitive value appears in the guard position (as in our example). Static analyses typically scale the branches' sensitivity by the sensitivity of the guard; in practice, this approach results in *infinite sensitivity for conditionals with a sensitive guard.*

To retain soundness in our dynamic analysis, DDuo requires that *conditional guards contain no sensitive values.* A run-time error is thrown if DDuo finds a sensitive value in the guard position (as in our example above). Disallowing sensitive guards makes it possible to ignore branches that are not executed: the guard's value remains the same under neighboring program inputs, so the program follows the same branch for neighboring executions. This approach does not limit the set of useful programs we can write, since conditionals with sensitive guards yield infinite sensitivities even under a precise static analysis.

Since DDuo attaches sensitivity environments to *values* (instead of variables),

the use of side effects does not affect the soundness of the analysis. When a program variable is updated to reference a new value, that value's sensitivity environment remains attached. DDuo handles many common side-effect-based patterns used in Python this way; for example, DDuo correctly infers that the following program results in the variable `total` holding a value that is 20 times more sensitive than `df.shape[0]`.

```
total = 0
for i in range(20):
  total = total + df.shape[0]
```

For side effects, our dynamic analysis is more capable than type-based static analysis, due to the additional challenges arising in the static setting (e.g. aliasing). We have formalized the way DDuo handles side effects and conditionals, and proved the soundness of our sensitivity analysis; our formalization appears in Chapter 14.

Chapter 13

Dynamic Privacy Tracking

DDuo tracks privacy cost *dynamically*, at runtime. Dynamic privacy tracking is challenging because the dynamic analysis has no visibility into code that is *not executed*. For example, consider the following conditional:

```
if dduo.gauss(ϵ=1.0, δ=1e-5, x) > 5:
  print(dduo.gauss(ϵ=1.0, δ=1e-5, y))
else:
  print(dduo.gauss(ϵ=100000000000.0, δ=1e-5, y))
```

The executed branch of this conditional depends on the result of the first call to `dduo.gauss`, which is non-deterministic. The two branches use different privacy parameters for the remaining calls to `dduo.gauss`; in other words, the privacy parameter for the second use of the Gaussian mechanism is chosen *adaptively*, based on the results of the first use. Sequential composition theorems for differential privacy (Dwork et al., 2014b) are typically stated in terms of *fixed* (i.e. non-adaptive) privacy parameters, and do not apply if the privacy parameters are chosen adaptively.

A static analysis of this program will consider *both* branches, and most

analyses will produce an upper bound on the program's privacy cost by combining the two (i.e. taking the maximum of the two ϵ values). This approach avoids the issue of adaptively-chosen privacy parameters.

A dynamic analysis, by contrast, *cannot* consider both branches, and must bound privacy cost by analyzing *only* the branch that is executed. Sequential composition does not apply directly when privacy parameters are chosen adaptively, so ignoring the non-executed branch in a dynamic analysis of privacy would be *unsound*.

13.1 Privacy Filters & Odometers

Privacy *filters* and *odometers* were originally developed by Rogers et al. (Rogers et al., 2016) specifically to address the setting in which privacy parameters are selected adaptively. Winograd-Cort et al. (Winograd-Cort et al., 2017) used privacy filters and odometers as part of the Adaptive Fuzz framework, which integrates both dynamic analysis (for composing privacy mechanisms) and static analysis (for bounding the cost of individual mechanisms). Recently, Feldman and Zrnic (Feldman and Zrnic, 2020) developed filters and odometers for Rényi differential privacy (Mironov, 2017b).

Privacy odometers can be used to obtain a running upper bound on total privacy cost at any point in the sequence of adaptive mechanisms, and to obtain an overall total at the end of the sequence. A function $\texttt{COMP}_{\delta_g} : \mathbb{R}^{2k}_{\geq 0} \to \mathbb{R} \cup \{\infty\}$ is called a *valid privacy odometer* (Rogers et al., 2016) for a sequence of mechanisms $\mathcal{M}_1, \ldots, \mathcal{M}_k$ if for all (adaptively-chosen) settings of $(\epsilon_1, \delta_1), \ldots, (\epsilon_k, \delta_k)$ for the individual mechanisms in the sequence, their composition satisfies $(\texttt{COMP}_{\delta_g}(\cdot), \delta_g)$-differential privacy. In other words, $\texttt{COMP}_{\delta_g}(\cdot)$ returns a value for ϵ that upper-bounds the privacy cost of the adaptive sequence of mechanisms. A valid privacy odometer for sequential composition in (ϵ, δ)-differential privacy can be defined

as follows (Rogers et al. (Rogers et al., 2016), Theorem 3.6):

$$\texttt{COMP}_{\delta_g}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k) = \begin{cases} \infty & \textit{if } \sum_{i=1}^{k} \delta_i > \delta_g \\ \sum_{i=1}^{k} \epsilon_i & \textit{otherwise} \end{cases}$$

Privacy filters allow the analyst to place an upper bound (ϵ_g, δ_g) on the desired privacy cost, and halt the computation immediately if the bound is violated. A function $\texttt{COMP}_{\epsilon_g,\delta_g} : \mathbb{R}^{2k}_{\geq 0} \rightarrow \{\texttt{HALT}, \texttt{CONT}\}$ is called a *valid privacy filter* (Rogers et al., 2016) for a sequence of mechanisms $\mathcal{M}_1, \ldots, \mathcal{M}_k$ if for all (adaptively-chosen) settings of $(\epsilon_1, \delta_1), \ldots, (\epsilon_k, \delta_k)$ for the individual mechanisms in the sequence, $\texttt{COMP}_{\epsilon_g,\delta_g}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k)$ outputs $\texttt{CONT}$ only if the sequence satisfies (ϵ_g, δ_g)-differential privacy (otherwise, it outputs $\texttt{HALT}$ for the first mechanism in the sequence that violates the privacy cost bound). A valid privacy filter for sequential composition in (ϵ, δ)-differential privacy can be defined as follows (Rogers et al. (Rogers et al., 2016), Theorem 3.6):

$$\texttt{COMP}_{\epsilon_g,\delta_g}(\epsilon_1, \delta_1, \ldots, \epsilon_k, \delta_k) = \begin{cases} \texttt{HALT} & \textit{if } \sum_{i=1}^{k} \delta_i > \delta_g \textit{ or } \sum_{i=1}^{k} \epsilon_i > \epsilon_g \\ \texttt{CONT} & \textit{otherwise} \end{cases}$$

It is clear from these definitions that the odometer and filter for sequential composition under (ϵ, δ)-differential privacy yield the same bounds on privacy loss as the standard theorem for sequential composition (Dwork et al., 2014b) (i.e. there is no "cost" to picking the privacy parameters adaptively).

Rogers et al. (Rogers et al., 2016) also define filters and odometers for *advanced composition* under (ϵ, δ)-differential privacy ((Rogers et al., 2016), §5 and §6); in

this case, there *is* a cost. In exchange for the ability to set privacy parameters adaptively, filters and odometers for advanced composition have slightly worse constants than the standard advanced composition theorem (Dwork et al., 2014b) (but are asymptotically the same).

13.2 Filters & Odometers in DDuo

DDuo's API allows the programmer to explicitly create privacy odometers and filters, and make them active for a specific part of the program (using Python's *with* syntax). When an odometer is active, it records a running total of the total privacy cost, and it can be queried to return this information to the programmer.

```
with dduo.EdOdometer(max_delta = 10e-5) as odo:
  _ = dduo.gauss(df.shape[0], ϵ = 1.0, δ = 10e-6)
  _ = dduo.gauss(df.shape[0], ϵ = 1.0, δ = 10e-6)
  print(odo)
```

```
Odometer_(ϵ, δ)({data.csv ↦ (2.0, 20^−6)})
```

When a filter is active, it tracks the privacy cost for individual mechanisms, and halts the program if the filter's upper bound on privacy cost is violated.

```
with dduo.EdFilter(ϵ = 1.0, δ = 10e-6) as odo:
  print('1:', dduo.gauss(df.shape[0], ϵ=1.0, δ=10e-6))
  print('2:', dduo.gauss(df.shape[0], ϵ=1.0, δ=10e-6))
```

```
1: 10.5627
Traceback (most recent call last):
  ...
  dduo.PrivacyFilterException
```

In addition to odometers and filters for sequential composition under (ϵ, δ)-differential privacy (such as `EdFilter` and `EdOdometer`), DDuo provides odometers and filters for advanced composition (`AdvEdFilter` and `AdvEdOdometer`) and Rényi differential privacy (`RenyiFilter` and `RenyiOdometer`, which follow the results of Feldman and Zrnic (Feldman and Zrnic, 2020)).

13.3 Loops and Composition

Iterative algorithms can be built in DDuo using Python's standard looping constructs, and DDuo's privacy odometers and filters take care of ensuring the correct form of composition. Parallel composition is also available—via functional mapping. Advanced composition can be achieved via special advanced composition filters and odometers exposed in the DDuo API. For example, the following simple loop runs the Laplace mechanism 20 times, and its total privacy cost is reflected by the odometer:

```
with dduo.EpsOdometer() as odo:
  for i in range(20):
    dduo.laplace(df.shape[0], ϵ = 1.0)
  print(odo)
```

```
Odometer_ϵ({data.csv ↦ 20.0})
```

To use advanced composition instead of sequential composition, we simply replace the odometer with a different one:

```
with dduo.AdvEdOdometer() as odo:
  for i in range(20):
    dduo.gauss(df.shape[0], ϵ = 0.01, δ = 0.001)
```

13.4 Mixing Variants of Differential Privacy

The DDuo library includes support for pure ϵ-differential privacy, (ϵ, δ)-differential privacy, and Rényi differential privacy (RDP). Programs may use all three variants together, convert between them, and compose mechanisms from each.

We demonstrate execution of a query while switched to the Rényi differential privacy variant using pythonic "with" syntax blocks. For programs that make extensive use of composition, this approach yields significant improvements in privacy cost. For example, the following program uses the Gaussian mechanism 200 times, using Rényi differential privacy for sequential composition; the total privacy cost is automatically converted into an (ϵ, δ)-differential privacy cost after the loop finishes.

```
with dduo.EdOdometer(max_delta = 1e-4) as odo:
  with dduo.RenyiDP(1e-5):
    for x in range(200):
      noisy_count = dduo.renyi_gauss(α = 10,
        ϵ=0.2, df.shape[0])
  print(odo)
```

```
Odometer_(ϵ, δ)({data.csv ↦ (41.28, 1^−5)})
```

Chapter 14

Formal Description of Sensitivity Analysis

In DDuo we implement a novel dynamic analysis for *function sensitivity*, which is a relational (hyper)property quantified over two runs of the program with arbitrary but related inputs. In particular, our analysis computes function sensitivity—a two-run property—after only observing *one* execution of the program. Only observing one execution poses challenges to the design of the analysis, and significant challenges to the proof, all of which we overcome. To overcome this challenge in the design of the analysis, we first disallow branching control flow which depends on any sensitive inputs; this ensures that any two runs of the program being considered for the purposes of privacy will take the same branch observed by the dynamic analysis. Second, we disallow sensitive input-dependent arguments to the "scalar" side of multiplication; this ensures that the dynamic analysis' use of that argument in analysis results is identical for any two runs of the program being considered for the purposes of privacy. Our dynamic analysis for function sensitivity is *sound*—meaning that the true

sensitivity of a program is guaranteed to be equal or less than the sensitivity reported by DDuo's dynamic monitor—and we support this claim with a detailed proof.

14.1 Formalism Approach.

We formalize the correctness of our dynamic analysis for function sensitivity using a *step-indexed big-step semantics* to describe the dynamic analysis, a *step-indexed logical relation* to describe the meaning of function sensitivity, and a proof by induction and case analysis on program syntax to show that dynamic analysis results soundly predict function sensitivity. A *step-indexed* relation is a relation $\mathcal{R} \in A \to B \to \text{prop}$ whose definition is stratified by a natural number index n, so for each level n there is a new relation $\mathcal{R}_n$. Typically, the relation $\mathcal{R}_0$ is defined $\mathcal{R}_0(x, y) \triangleq \text{true}$, and the final relation of interest is $\hat{\mathcal{R}} \triangleq \bigcap_n \mathcal{R}_n$, i.e., $\hat{\mathcal{R}}(x, y) \iff \forall n.\ \mathcal{R}_n(x, y)$. Step-indexing is typically used—as we do in our formalism—when the definition of a relation would be not well founded in its absence. The most common reason a relation definition might be not well-founded is the use of self-reference without any decreasing measure. When a decreasing measure exists, self-reference leads to well-founded recursion, however when a decreasing measure does not exist, self-reference is not well-founded. When using step-indexing, self-reference is allowed in the definition of $\mathcal{R}_n$, but only for the relation at strictly lower levels, so $\mathcal{R}_{n'}$ when $n' < n$; this is well-founded because the index n becomes a decreasing measure for the self-reference. In this way, step-indexing enables self-reference without any existing decreasing measure by introducing a new decreasing measure, and maintains well-foundedness of the relation definition.

A *logical* relation is one where the definition of relation on function values (or types) is extensional, essentially saying "when given related inputs, the function

produces related outputs". This definition is self-referrential and not well-founded, and among common reasons to introduce step-indexing in programming language proofs. As the relation $\mathcal{R}$ is stratified with a step-index to $\mathcal{R}_n$, so must the definition of the semantics, so for a big-step relation $e \Downarrow v$ (relating an expression e to its final value v after evaluation) we stratify as $e \Downarrow_n v$. Also, because the definition of a logical relation *decrements* the step-index for the case of function values, we *increment* the step-index in the semantic case for function application. These techniques are standard from prior work (Ahmed, 2006), and we merely summarize the key ideas here to give background to our reader.

14.2 Formal Definition of Dynamic Analysis.

We model language features for arithmetic operations ($e \odot e$), conditionals ($\texttt{if0}(e)\{e\}\{e\}$), pairs ($\langle e, e\rangle$ and $\pi_i(e)$), functions ($\lambda x.\ e$ and $e(e)$) and references ($\texttt{ref}(e)$, $!e$ and $e \leftarrow e$); the full language is shown in Figure 22.1. There is one base value: $r@^{\Sigma}_{m}$ for a real number result r tagged with dynamic analysis information Σ—the sensitivity analysis for the expression which evaluated to r—and m—the metric associated with the resulting value r. The sensitivity analysis Σ—also called a *sensitivity environment*—is a map from sensitive sources $o \in \text{source}$ to how sensitive the result is w.r.t. that source. Our formalism includes two base metrics $m \in \text{metric}$: `diff` and `disc` for absolute difference ($|x - y|$) and discrete distance (0 if $x = y$ and 1 otherwise) respectively—and two derived metrics: $\top$ and $\bot$ for the smallest metric larger than each base metric and largest metric smaller than each base metric, respectively. Each metric is commonly used when implementing differentially private algorithms. Pair values ($\langle v, v\rangle$), closure values ($\langle \lambda x.\ e \mid \rho\rangle$) and reference values ($\ell$) do not contain dynamic analysis information.

Our dynamic analysis is described formally as a big-step relation $\rho \vdash \ulcorner \sigma, e \urcorner \Downarrow_n$

$\ulcorner \sigma, v \urcorner$ where $\rho \in \text{var} \rightharpoonup \text{value}$ is the lexical environment mapping lexical variables to values, $\sigma \in \text{loc} \rightharpoonup \text{value}$ is the dynamic environment (i.e., the heap, or store) mapping dynamically allocated references to values, e is the expression being executed, and v is the resulting runtime value which also includes dynamic analysis information. We write gray box corners around the "input" configuration $\ulcorner \sigma, e \urcorner$ and the "output" configuration $\ulcorner \sigma, v \urcorner$ to aid readability. The index n is for step-indexing, and tracks the number of function applications which occurred in the process of evaluation. We show the full definition of the dynamic analysis in Figure 22.5.

Consider the following example:

$$\{x \mapsto 21@^{\{o \mapsto 1\}}_{\texttt{diff}}\} \vdash \varnothing, (x + x) \Downarrow_0 42@^{\{o \mapsto 2\}}_{\texttt{diff}}$$

This relation corresponds to a scenario where the program to evaluate and analyze is $x + x$, the variable x represents a sensitive source value o, we want to track sensitivity w.r.t. the absolute difference metric, and the initial value for x is 21. This information is encoded in an initial environment $\rho = \{x \mapsto 21@^{\{o \mapsto 1\}}_{\texttt{diff}}\}$. The result value is 42, and the resulting analysis reports that e is 2-sensitive in the source o w.r.t. the absolute difference metric. This analysis information is encoded in the return value $42@^{\{o \mapsto 2\}}_{\texttt{diff}}$. Because no function applications occur during evaluation, the step index n is 0.

14.3 Formal Definition of Function Sensitivity.

Function sensitivity is encoded through multiple relation definitions:

1. $\rho_1, \sigma_1, e_1 \sim^{\Sigma}_n \rho_2, \sigma_2, e_2$ holds when the input triples ρ_1, σ_1, e_1 and ρ_2, σ_2, e_2 evaluate to output stores and values which are related by Σ. Note this definition decrements the step-index n, and is the constant relation when

$b \in \mathbb{B}$ $\quad n \in \mathbb{N}$ $\quad i \in \mathbb{Z}$ $\quad r \in \mathbb{R}$ $\quad x \in \text{var}$

$o \in \text{source}$		sensitive sources	$q \in \hat{\mathbb{R}}$	$::= r \mid \infty$		ext. reals
$\ell \in \text{loc}$		reference locations	$\odot \in \text{binop}$	$::= + \mid \ltimes \mid \rtimes$		operations
$e \in \text{expr}$	$::= x$	variables	$m \in \text{metric}$	$::=$ `diff`		absolute difference
	$\mid r$	real numbers		$\mid$ `disc`		discrete
	$\mid e \odot e$	arith. operations		$\mid \bot$		bot metric
	$\mid$ `if0`$(e)\{e\}\{e\}$	cond. branching		$\mid \top$		top metric
	$\mid \langle e, e \rangle$	pair creation	$v \in \text{value}$	$::= r@^{\Sigma}_{m}$		tagged base value
	$\mid \pi_i(e)$	pair access		$\mid \langle v, v \rangle$		pair
	$\mid \lambda x.\ e$	function creation		$\mid \langle \lambda x.\ e \mid \rho \rangle$		function (closure)
	$\mid e(e)$	function application		$\mid \ell$		location (pointer)
	$\mid$ `ref`(e)	reference creation	$\rho \in \text{env}$	$\triangleq \text{var} \rightharpoonup \text{value}$		value environment
	$\mid\ !e$	reference read	$\sigma \in \text{store}$	$\triangleq \text{loc} \rightharpoonup \text{value}$		mutable store
	$\mid\ e \leftarrow e$	reference write	$\Sigma \in \text{senv}$	$\triangleq \text{source} \rightharpoonup \hat{\mathbb{R}}$		sens. environment

---

$$\rho_1, \sigma_1, e_1 \ \sim^{\Sigma}_{0}\ \rho_2, \sigma_2, e_2 \iff^{\triangle} \text{true}$$

$$\rho_1, \sigma_1, e_1 \sim^{\Sigma}_{n+1} \rho_2, \sigma_2, e_2 \iff^{\triangle} \forall n_1 \le n, n_2, \sigma_1', \sigma_2', v_1, v_2.$$
$$\rho_1 \vdash \ulcorner \sigma_1, e_1 \urcorner \Downarrow_{n_1} \ulcorner \sigma_1', v_1 \urcorner \ \wedge\ \rho_2 \vdash \ulcorner \sigma_2, e_2 \urcorner \Downarrow_{n_2} \ulcorner \sigma_2', v_2 \urcorner$$
$$\Rightarrow n_1 = n_2 \ \wedge\ \sigma_1' \sim^{\Sigma}_{n-n_1} \sigma_2' \ \wedge\ v_1 \sim^{\Sigma}_{n-n_1} v_2$$

$\boxed{\rho, \sigma, e \sim^{\Sigma}_{n} \rho, \sigma, e}$

$\boxed{r \sim^{r}_{m} r}$

$$r_1 \sim^{r}_{\texttt{diff}} r_2 \iff^{\triangle} |r_1 - r_2| \le r \qquad r_1 \sim^{r}_{\bot} r_2 \iff^{\triangle} r_1 \sim^{r}_{\texttt{diff}} r_2 \ \wedge\ r_1 \sim^{r}_{\texttt{disc}} r_2$$

$$r_1 \sim^{r}_{\texttt{disc}} r_2 \iff^{\triangle} \begin{cases} 0 \le r & \text{if} \quad r_1 = r_2 \\ 1 \le r & \text{if} \quad r_1 \neq r_2 \end{cases} \qquad r_1 \sim^{r}_{\top} r_2 \iff^{\triangle} r_1 \sim^{r}_{\texttt{diff}} r_2 \ \vee\ r_1 \sim^{r}_{\texttt{disc}} r_2$$

$\boxed{v \sim^{\Sigma}_{n} v}$

$$r_1@_{m_1^{\Sigma_1}} \sim^{\Sigma}_{n} r_2@_{m_2^{\Sigma_2}} \iff^{\triangle} \Sigma_1 = \Sigma_2 \ \wedge\ m_1 = m_2 \ \wedge\ r_1 \sim_{m_1^{\Sigma \cdot \Sigma_1}} r_2$$

$$\langle v_{11}, v_{12} \rangle \sim^{\Sigma}_{n} \langle v_{21}, v_{22} \rangle \iff^{\triangle} v_{11} \sim^{\Sigma}_{n} v_{21} \ \wedge\ v_{12} \sim^{\Sigma}_{n} v_{22}$$

$$\langle \lambda x.\ e_1 \mid \rho_1 \rangle \sim^{\Sigma}_{n} \langle \lambda x.\ e_2 \mid \rho_2 \rangle \iff^{\triangle} \forall n' \le n, v_1, v_2, \sigma_1, \sigma_2.\ \sigma_1 \sim^{\Sigma}_{n'} \sigma_2 \ \wedge\ v_1 \sim^{\Sigma}_{n'} v_2$$
$$\Rightarrow \sigma_1, \{x \mapsto v_1\} \uplus \rho_1, e_1 \sim^{\Sigma}_{n'} \sigma_2, \{x \mapsto v_2\} \uplus \rho_2, e_2$$

$$\ell_1 \sim^{\Sigma}_{n} \ell_2 \iff^{\triangle} \ell_1 = \ell_2$$

$$\rho_1 \sim^{\Sigma}_{n} \rho_2 \iff^{\triangle} \forall x \in (\text{dom}(\rho_1) \cup \text{dom}(\rho_2)).\ \rho_1(x) \sim^{\Sigma}_{n} \rho_2(x)$$
$$\sigma_1 \sim^{\Sigma}_{n} \sigma_2 \iff^{\triangle} \forall \ell \in (\text{dom}(\sigma_1) \cup \text{dom}(\sigma_2)).\ \sigma_1(\ell) \sim^{\Sigma}_{n} \sigma_2(\ell)$$

$\boxed{\rho \sim^{\Sigma}_{n} \rho \quad \sigma \sim^{\Sigma}_{n} \sigma}$

Figure 14.1: Formal Syntax & Step-indexed Logical Relation.

$n = 0$.

2. $r_1 \sim_m^r r_2$ holds when the difference between real numbers r_1 and r_2 w.r.t. metric m is less than r.

3. $v_1 \sim_n^\Sigma v_2$ holds when values v_1 and v_2 are related for initial distance Σ and step-index n. The definition is by case analysis on the syntactic category for values, such as:

   (a) The relation on base values $r_1@_{m_1^{\Sigma_1}} \sim_n^\Sigma r_2@_{m_2^{\Sigma_2}}$ holds when Σ_1, Σ_2, m_1 and m_2 are pairwise equal, and when r_1 and r_2 are related by $\Sigma \cdot \Sigma_1$, where Σ is the initial distances between each input source o, and Σ_1 is how much r_1 and r_2 are allowed to differ as a linear function of input distances Σ, and where this function is applied via vector dot product $\cdot$.

   (b) The relation on pair values $\langle v_{11}, v_{12} \rangle \sim_m^\Sigma \langle v_{21}, v_{22} \rangle$ holds when each element of the pair are pairwise related.

   (c) The relation on function values $\langle \lambda x.\ e_1 \mid \rho_1 \rangle \sim_n^\Sigma \langle \lambda x.\ e_2 \mid \rho_2 \rangle$ holds when each closure returns related output configurations when evaluated with related inputs.

   (d) The relation on locations $\ell_1 \sim_n^\Sigma \ell_2$ holds when the two locations are equal.

4. $\rho_1 \sim_n^\Sigma \rho_2$ holds when lexical environments ρ_1 and ρ_2 map all variables to related values.

5. $\sigma_1 \sim_n^\Sigma \sigma_2$ holds when dynamic environments σ_1 and σ_2 map all locations to related values.

Note that the definitions of $\rho_1, \sigma_1, e_1 \sim_n^\Sigma \rho_2, \sigma_2, e_2$ and $v \sim_n^\Sigma v$ are mutually recursive, but are well founded due to the decrement of the step index in the

former relation. We show the full definition of these relations in Figure 22.6.

The function sensitivity of an expression is encoded first as a statement about expressions respecting relatedness, that is, returning related outputs when given related inputs, i.e., (assuming no use of the store) if $\rho_1 \sim_n^\Sigma \rho_2$ and $\rho_1 \vdash \varnothing, e \Downarrow_{n_1} \varnothing, v_1$ and $\rho_2 \vdash \varnothing, e \Downarrow_{n_2} \varnothing, v_2$ then $n_1 = n_2$ and $v_1 \sim_{n-n_1^\Sigma} v_2$. When instantiated to base types, we have: if $\rho_1 \sim_n^\Sigma \rho_2$ and $\rho_1 \vdash \varnothing, e \Downarrow_{n_1} \varnothing, r_1 @_{m_1^{\Sigma_1}}$ and $\rho_2 \vdash \varnothing, e \Downarrow_{n_2} \varnothing, r_2 @_{m_2^{\Sigma_2}}$ then $n_1 = n_2$, $\Sigma_1 = \Sigma_2$, $m_1 = m_2$ and $r_1 \sim_{m_1^{\Sigma \cdot \Sigma_1}} r_2$. The fully general form of this property is called *metric preservation*, which is the main property we prove in our formal development.

14.4 Metric Preservation.

Metric preservation states that when given related initial configurations and evaluation outputs, then those outputs are related. Outputs include result values, as well as dynamic analysis results, and the relationship that holds demonstrates the soundness of the analysis results.

Theorem 14.4.1 (Metric Preservation)**.**

If:	$\rho_1 \sim_n^\Sigma \rho_2$	*(H1)*
And:	$\sigma_1 \sim_n^\Sigma \sigma_2$	*(H2)*
Then:	$\rho_1, \sigma_1, e \sim_n^\Sigma \rho_2, \sigma_2, e$	

That is, either $n = 0$ *or* $n = n' + 1$ *and...*

$\rceil\_\lceil^r \in \hat{\mathbb{R}} \to \hat{\mathbb{R}}$ $\quad \rceil r' \lceil^r \triangleq \begin{cases} 0 & \mathit{if} \quad r' = 0 \\ r & \mathit{if} \quad r' \neq 0 \end{cases}$ $\quad \mathrm{alloc}(\mathcal{L}) \notin \mathcal{L} \in \wp(\mathrm{loc})\,\mathbf{Z} =$

$\rceil\_\lceil^r \in \mathrm{senv} \to \mathrm{sens}$ $\quad \rceil\Sigma\lceil^r(o) \triangleq \rceil\Sigma(o)\lceil^r$

$\{o \mapsto 0\}$

$$\boxed{\rho \vdash \ulcorner \sigma, e \urcorner \Downarrow_n \ulcorner \sigma, v \urcorner}$$

VAR
$$\frac{}{\rho \vdash \ulcorner \sigma, x \urcorner \Downarrow_0 \ulcorner \sigma, \rho(x) \urcorner}$$

REAL
$$\frac{}{\rho \vdash \ulcorner \sigma, r \urcorner \Downarrow_0 \ulcorner \sigma, r@^{\mathbf{Z}}_{\mathtt{disc}} \urcorner}$$

FUN
$$\frac{}{\rho \vdash \ulcorner \sigma, \lambda x.\ e \urcorner \Downarrow_0 \ulcorner \sigma, \langle \lambda x.\ e \mid \rho \rangle \urcorner}$$

PLUS
$$\frac{\rho \vdash \ulcorner \sigma\ , e_1 \urcorner \Downarrow_{n_1} \ulcorner \sigma'\ , r_1@_{m_1^{\Sigma_1}} \urcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \urcorner \Downarrow_{n_2} \ulcorner \sigma'', r_2@_{m_2^{\Sigma_2}} \urcorner}{\rho \vdash \ulcorner \sigma, e_1 + e_2 \urcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', (r_1 + r_2)@_{m_1^{\Sigma_1+\Sigma_2} \sqcup m_2} \urcorner}$$

TIMES-L
$$\frac{\rho \vdash \ulcorner \sigma\ , e_1 \urcorner \Downarrow_{n_1} \ulcorner \sigma'\ , r_1@_{m_1^{\mathbf{Z}}} \urcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \urcorner \Downarrow_{n_2} \ulcorner \sigma'', r_2@_{m_2^{\Sigma_2}} \urcorner}{\rho \vdash \ulcorner \sigma, e_1 \ltimes e_2 \urcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', (r_1 \times r_2)@_{m_2^{r_1\Sigma_2}} \urcorner}$$

TIMES-R
$$\frac{\rho \vdash \ulcorner \sigma\ , e_1 \urcorner \Downarrow_{n_1} \ulcorner \sigma'\ , r_1@_{m_1^{\Sigma_1}} \urcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \urcorner \Downarrow_{n_2} \ulcorner \sigma'', r_2@_{m_2^{\mathbf{Z}}} \urcorner}{\rho \vdash \ulcorner \sigma, e_1 \rtimes e_2 \urcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', (r_1 \times r_2)@_{m_1^{r_2\Sigma_1}} \urcorner}$$

IFZ-T
$$\frac{\rho \vdash \ulcorner \sigma\ , e_1 \urcorner \Downarrow_{n_1} \ulcorner \sigma'\ , r_1\ @_{m_1^{\mathbf{Z}}} \urcorner \qquad \rho \vdash \ulcorner \sigma', e_2 \urcorner \Downarrow_{n_2} \ulcorner \sigma'', v_2 \urcorner \qquad r_1 = 0}{\rho \vdash \ulcorner \sigma, \mathtt{if0}(e_1)\{e_2\}\{e_3\} \urcorner \Downarrow_{n_1+n_2} \ulcorner \sigma'', v_2 \urcorner}$$

Figure 14.2: Formal Big-step, Step-indexed Semantics and Metafunctions.

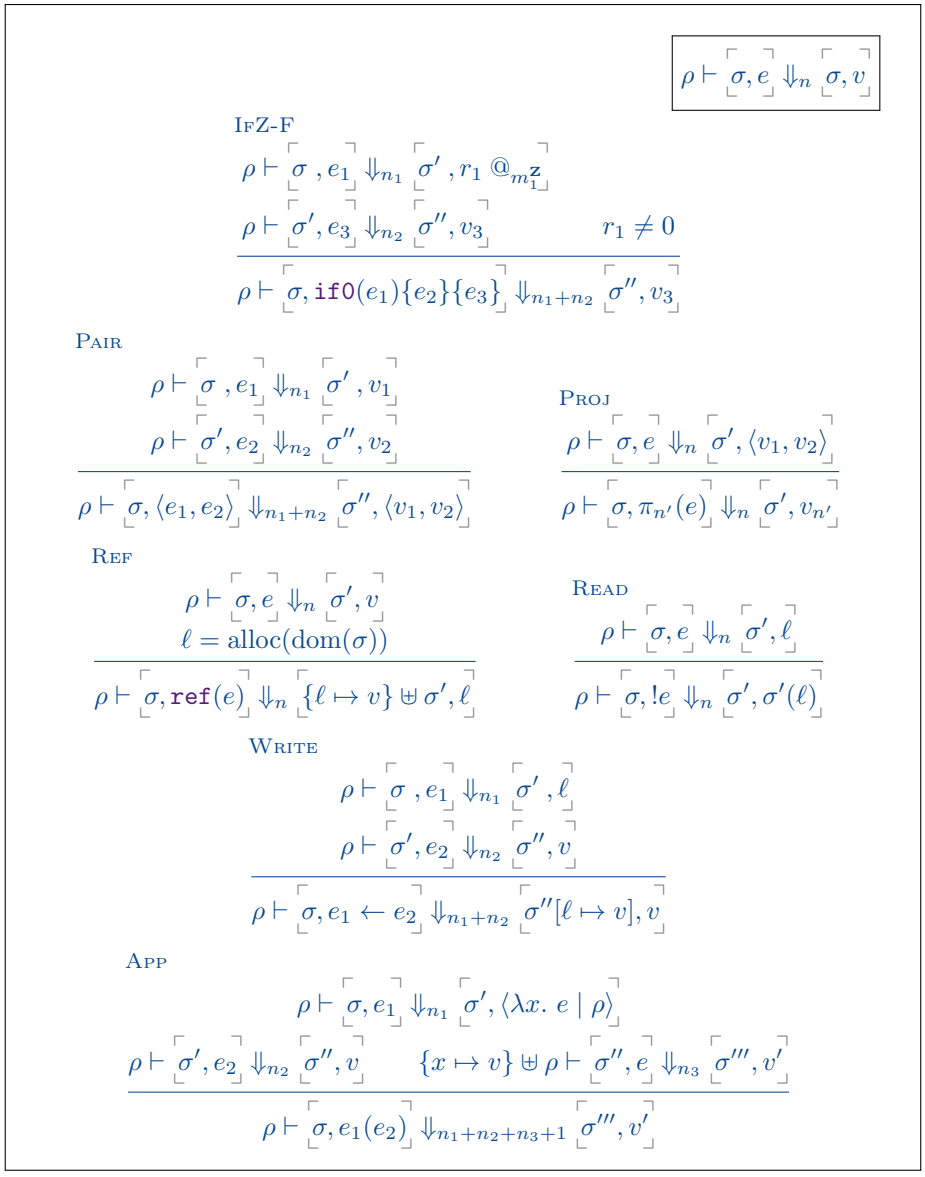

Figure 14.3: Formal Big-step, Step-indexed Semantics and Metafunctions.

If:	$n_1 \leq n$	*(H3)*
And:	$\rho_1 \vdash \ulcorner \sigma_1, e \urcorner \Downarrow_{n_1} \ulcorner \sigma_1', v_1 \urcorner$	*(H4)*
And:	$\rho_2 \vdash \ulcorner \sigma_2, e \urcorner \Downarrow_{n_2} \ulcorner \sigma_2', v_2 \urcorner$	*(H5)*
Then:	$n_1 = n_2$	*(C1)*
And:	$\sigma_1' \sim_{n-n_1^\Sigma} \sigma_2'$	*(C2)*
And:	$v_1 \sim_{n-n_1^\Sigma} v_2$	*(C3)*

Proof. See detailed proof in Chapter 16. □

14.5 Instantiating Metric Preservation.

Metric preservation is not enough on its own to demonstrate sound dynamic analysis of function sensitivity. Suppose we execute the dynamic analysis on program e with initial environment ρ, yielding a final store σ, base value r, sensitivity environment Σ, metric m and step-index n as a result:

$$\rho \vdash \varnothing, e \Downarrow_n \sigma, r@_m^\Sigma$$

To know the sensitivity of e is to know a bound on two *arbitrary* runs of e, that is, using two arbitrary environments ρ_1 and ρ_2. Does Σ tell us this? Remarkably, it does, with one small condition: ρ_1 and ρ_2 must agree with ρ on all non-sensitive values. This is not actually limiting: a non-sensitive value is essentially auxiliary information; they are constants and fixed for the purposes of sensitivity and privacy.

We can encode the relationship that environments ρ and ρ_1 agree on all non-sensitive values as $\rho \sim^{\Sigma'} \rho_1$ for any Σ', and we allow for environments ρ and ρ_1 to differ on any sensitive value while agreeing on non-sensitive values as $\rho \sim^{\{o \mapsto \infty\}} \rho_1$. Under such an assumption, Σ and m are sound dynamic analysis results for two arbitrary runs of e, i.e., under environments ρ_1 and ρ_2, so long as

one of those environments agrees with ρ—the environment used to compute the dynamic analysis. We encode this property formally as the following corollary to metric preservation:

Corollary 14.5.0.1 (Sound Dynamic Analysis for Sensitivity)**.**

If: $n_1 < n$, $n_2 < n$ *and* $n_3 < n$ *(H1)*

And: $\rho \sim_n^{\{o \mapsto \infty\}} \rho_1$ *(H2)*

And: $\rho \vdash \ulcorner \varnothing, e \urcorner \Downarrow_{n_1} \ulcorner \sigma, r @ m^{\Sigma} \urcorner$ *(H3)*

And: $\rho_1 \sim_n^{\Sigma'} \rho_2$ *(H4)*

And: $\rho_1 \vdash \ulcorner \varnothing, e \urcorner \Downarrow_{n_2} \ulcorner \sigma_1, r_1 @ m_1^{\Sigma_1} \urcorner$ *(H5)*

And: $\rho_2 \vdash \ulcorner \varnothing, e \urcorner \Downarrow_{n_3} \ulcorner \sigma_2, r_2 @ m_2^{\Sigma_2} \urcorner$ *(H6)*

Then: $r_1 \sim_{m_1}^{\Sigma' \cdot \Sigma} r_2$ *(C1)*

Proof.

By Metric Preservation, *(H2)*, *(H1)*, *(H3)* and *(H5)* we have $\Sigma_1 = \Sigma$ and $m_1 = m$. By Metric Preservation, *(H4)*, *(H1)*, *(H5)* and *(H6)* we have proved the goal *(C1)*. □

Note that the final results are related using Σ—the analysis result derived from an execution under ρ—while r_1 and r_2 are derived from executions under unrelated (modulo auxiliary information) environments ρ_1 and ρ_2.

In simpler terms, this corollary shows that even though the dynamic analysis only sees one particular execution of the program, it is accurate in describing the sensitivity of the program—even though the notion of sensitivity considers two arbitrary runs of the program, including those whose inputs differ entirely from those used in the dynamic analysis.

Chapter 15

Implementation & Case Studies

We have developed a reference implementation of DDuo as a Python library, using the approaches described in Chapters **??**, 12, and 13.

A major goal in the design of DDuo is seamless integration with other libraries. Our reference implementation provides initial support for NumPy, Pandas, and Sklearn. DDuo provides hooks for tracking both sensitivity and privacy, to simplify integrating with additional libraries.

We present case studies which focus on demonstrating DDuo's (1) similarity to regular Python code, (2) applicability to complex algorithms, (3) easy integration with existing libraries. Although our approach is automatic, DDuo is able to compute privacy leakage bounds that match those of bespoke privacy-preserving algorithms.

We introduce new *adaptive* variants of algorithms that stop early when possible to conserve privacy budget. These variants cannot be verified by prior work using purely static analyses, because their privacy parameters are chosen

Name	Type			Conditions
laplace	$(\epsilon : \mathbb{R}, value : \mathbb{R})$	$\to \mathbb{R}$	*where*	$priv(\text{laplace}(\epsilon, value)) \triangleq \epsilon$
gauss	$(\epsilon : \mathbb{R}, \delta : \mathbb{R}, value : \mathbb{R})$	$\to \mathbb{R}$	*where*	$priv(\text{gauss}(\epsilon, \delta, value)) \triangleq (\epsilon, \delta)$
ed_odo	$(f : A \to B, in : A)$	$\to (out : B, (\epsilon, \delta))$	*where*	$priv(\text{ed\_odo}(f, in)) \triangleq priv(f(in))$
renyi_odo	$(f : A \to B, in : A)$	$\to (out : B, (\alpha, \epsilon))$	*where*	$priv(\text{renyi\_odo}(f, in)) \triangleq priv(f(in))$
ed_filter	$(f : A \to B, in : A, (\epsilon, \delta))$	$\to (out : B)$	*where*	$(\epsilon, \delta) \geq priv(f(in))$
renyi_filter	$(f : A \to B, in : A, (\alpha, \epsilon))$	$\to (out : B)$	*where*	$(\alpha, \epsilon) \geq priv(f(in))$
conv_renyi	$(f : A \to B, in : A, \delta : \mathbb{R})$	$\to (out : B)$	*where*	$priv(f(\ldots)) = (\alpha, \epsilon)$
			and	$\text{conv}(\alpha, \epsilon, \delta) = (\epsilon, \delta)$
svt	$(\epsilon : \mathbb{R}, qs : [A \to B], data : [A], t : \mathbb{R})$	$\to \mathbb{N}$	*where*	for q in qs, $sens(q) = 1$
			and	$priv(svt(\epsilon, qs, data, t)) \triangleq \epsilon$
exp	$(\epsilon : \mathbb{R}, q : A \to B, data : [A])$	$\to \mathbb{N}$	*where*	$priv(exp(\epsilon, q, data)) \triangleq \epsilon$
map	$(f : A \to B, in : [A])$	$\to [B]$	*where*	$\text{sens}(\text{map}(f, \_)) \triangleq \text{sens}(f(\_))$

priv: denotes the privacy leakage of a program given by dynamic analysis; *sens*: denotes the sensitivity of a program given by dynamic analysis; *conv*: represents the conversion equation from renyi to approximate differential privacy; ϵ,δ,and α are always assumed to be of type $\mathbb{R}$. Types are written as follows: the $\to$ symbol is used to seperate the domain and range of a function, either of which may be given as an atomic type such as a natural number ($\mathbb{N}$), or as a tuple which is a comma-seperated list of types surrounded by parentheses, or as a symbol (A) indicating parametric polymorphism (generics). In some cases, types may also be accompanied with a placeholder name ($\epsilon : \mathbb{R}$) for further qualification in the *where* clause.

Figure 15.1: Core API Methods

adaptively.

15.1 Run-time overhead.

Run-time overhead is a key concern in DDuo's instrumentation for dynamic analysis. Fortunately, experiments on our case studies suggest that the overhead of DDuo's analysis is generally low. Table 15.1 presents the run-time performance overhead of DDuo's analysis as a percentage *increase* of total runtime. The worst overhead time observed in our case studies was less than 60%.

In certain rare cases, DDuo's overhead can be much higher. For example, mapping the function `lambda x: x + 1` over a list of 1 million numbers takes 160x longer under DDuo than in standard Python. The overhead in this case comes from a combination of factors: first, DDuo's `map` function, itself implemented in Python, is much slower than Python's built-in `map` operator; second, DDuo's `map` function requires the creation of a new `Sensitive` object for each element of the list—a slow operation in Python.

Fortunately, the same strategies for producing high-performance Python

Technique	Ref.	Libraries Used	Overhead
Noisy Gradient Descent	(Bassily et al., 2014b)	NumPy	6.42%
Multiplicative Weights (MWEM)	(Hardt et al., 2012)	Pandas	14.90%
Private Naive Bayes Classification	(Vaidya et al., 2013)	DiffPrivLib	12.44%
Private Logistic Regression	(Chaudhuri and Monteleoni, 2008)	DiffPrivLib	56.33%

Table 15.1: List of case studies included with the DDuo implementation.

code *without* privacy also help reduce DDuo's overhead. Python's performance characteristics have prompted the development of higher-performance libraries like NumPy and Pandas, which essentially provide data-processing combinators that programmers compose. By providing sensitivity annotations for these libraries, we can re-use these high-performance implementations and avoid creating extra objects. As a result, none of our case studies demonstrates the worst-case performance overhead described above.

15.2 Case study: gradient descent with NumPy.

Our first case study (Figure 23.3) is a simple machine learning algorithm based on (Bassily et al., 2014b) implemented directly with DDuo-instrumented NumPy primitives.

Given a dataset X which is a list of feature vectors representing training examples, and a vector y which classifies each element of X in a finite set, gradient descent is the process of computing a model (a linear set of weights) which most accurately classifies a new, never seen before training example, based on our pre-existing evidence represented by the model.

Gradient descent works by first specifying a loss function that computes the effectiveness of a model in classifying a given dataset according to its known labels. The algorithm then iteratively computes a model that minimizes the loss function, by calculating the gradient of the loss function and moving the model in the opposite direction of the gradient.

One method of ensuring privacy in gradient descent involves adding noise to the gradient calculation, which is the only part of the process that is exposed to

the private training data. In order to add noise to the gradient, it is convenient to bound its sensitivity via clipping to some L2 norm. In this example, clipping occurs in the `gradient_sum` function before summation.

The original implementation of this algorithm (Bassily et al., 2014b) was based on the advanced composition theorem. Advanced composition improves on sequential composition by providing much tighter privacy bounds over several iterations, but requires the analyst to fix the number of iterations up front, regardless of how many iterations the gradient descent algorithm actually takes to converge to minimal error.

We present a modified version based on *adaptive* Renyi differential privacy which provides not only a tighter analysis of the privacy leakage over several iterations, but also allows the analyst to halt computation adaptively (conserving the remaining privacy budget) once a certain level of model accuracy has been reached, or loss has been minimized. We introduce random noise to the accuracy calculation because it is a computation on the sensitive input training dataset in this case.

15.3 MWEM with Pandas

The MWEM algorithm (Hardt et al., 2012) constructs a differentially private synthetic dataset that approximates a real dataset. MWEM produces competitive privacy bounds by utilizing a combination of the exponential mechanism, Laplacian/Gaussian noise, and the multiplicative weights update rule. The algorithm uses these mechanisms iteratively, providing a tight analysis of privacy leakage via composition.

The inputs to the MWEM are as follows: some uniform or random distribution over a domain (syn_data), some sensitive dataset (age_counts), a query workload, a number of iterations i, and a privacy budget ϵ.

```
def dp_gradient_descent(iterations, alpha, eps):
  eps_i = eps/iterations
  theta = np.zeros(X.shape[1])
  with dduo.RenyiFilter(alpha,eps_max):
    with dduo.RenyiOdometer((alpha, eps)) as odo:
      noisy_count = dduo.renyi_gauss(α=alpha,
        ϵ=eps,X.shape[0])
        priv_acc = 0
        acc_diff = 1
      while acc_diff > 0.05:
        grad_sum = gradient_sum(theta,
          X_train, y_train, sensitivity)
        noisy_grad_sum = dduo.gauss_vec(grad_sum,
          α=alpha,ϵ=eps_i)
        noisy_avg_grad = noisy_grad_sum/noisy_count
        theta = np.subtract(theta, noisy_avg_grad)
        priv_acc_curr = dduo.renyi_gauss(alpha,
          eps_acc, accuracy(theta))
        acc_diff =  priv_acc_curr - priv_acc
        priv_acc = priv_acc_curr
      print(odo)
    return theta

theta = dp_gradient_descent(iterations,
  α=alpha, ϵ=epsilon)
acc = dduo.renyi_gauss(alpha,
  eps_acc, accuracy(theta))
print(f"final accuracy: {acc}")
```

```
Odometer_(α, ϵ)({data.csv ↦ (10.0, 2.40)})
final accuracy: 0.753
```

Figure 15.2: Gradient Descent with NumPy

The algorithm works by, at each iteration:

- privately selecting a query from the query workload (using the exponential mechanism) whose result on the synthetic dataset greatly differs from the real dataset

```
for t in range(i):
  q = exponential(q_workload,score_fn,eps/(2*i))
  ...
```

- and then privately using the query result on the real dataset to adjust the synthetic dataset towards the truth using the multiplicative weights update rule

```
for t in range(i):
  ...
  syn_data = mwem_step(q, age_counts, syn_data)
```

We present a modified, adaptive MWEM algorithm (Figure 23.4) which privately halts execution if the error of the synthetic dataset reaches an acceptably low level before the entire privacy budget is exceeded, conserving the remainder of the budget for other private analyses.

15.4 DiffPrivLib

DiffPrivLib is library for experimenting with analytics and machine learning with differential privacy in Python by IBM. It provides a comprehensive suite of differentially private mechanisms, tools, and machine learning models.

```
def mwem_step(query, real_data, syn_data):
  lower, upper = query
  sm = [v for k, v in syn_data.items()]
  total = np.sum(sm)
  q_ans = range_query(real_data, lower, upper)
  real = dduo.renyi_gauss(α=alpha,ϵ=eps,q_ans,sens)
  syn = range_query(syn_data, lower, upper)
  l = [(k, mwem_update(k, x, lower, upper,
        real, syn, total))
      for k, x in syn_data.items()]
  return dict(l)

with dduo.RenyiFilter(alpha,20.0):
  with dduo.RenyiOdometer((alpha,2.0)) as odo:
    while stable < stability_thresh:
      e = err(age_counts,curr_syn)
      curr_noisy_err=dduo.renyi_gauss(α=alpha,ϵ=1.0,e)
      if (curr_noisy_err < thresh):
        stable += step
      else:
        stable = 0
      for t in range(iterations):
        q = exponential(q_workload,score_fn,eps/(2*i))
        curr_syn = mwem_step(q,age_counts,curr_syn)

acc = dduo.renyi_gauss(alpha, eps_acc,
  accuracy(age_counts,curr_syn))
print(f"final accuracy: {acc}")
```

```
Odometer_(α,ϵ)({data.csv ↦ (10.0, 0.5)})
final accuracy: 0.703
```

Figure 15.3: MWEM with Pandas

While DiffPrivLib provides several mechanisms, models and tools for developing private applications, as well as a basic privacy accountant, it lacks the ability to perform a tight privacy analysis in the context of more sophisticated forms of composition with dynamic and adaptive privacy tracking. Via integration with DDuo we are able to gain these abilities with minimal changes to library and program code.

Figure 15.4 shows an example of a modified DiffPrivLib program: a private naive Bayes classifier run on the standard iris dataset. The original program has been modified with DDuo hooks to detect sensitivity violations and track privacy cost.

We also present a DDuo instrumented example of differentially private logistic regression with DiffPrivLib (Figure 15.5).

Both of these programs have been modified to perform *adaptively* private clipping. Over several iterations, clipping parameters are gradually modified to optimize model accuracy. This form of control flow on probabilistic values is only sound following the adaptive composition strategies that DDuo provides. In order to preserve the privacy budget, such hyperparameter optimization procedures should normally be run on artificial datasets based on domain knowledge.

The changes required for the integration with the DiffPrivLib library consist of 15 lines of DDuo instrumentation code.

```
from dduo import sklearn as sk
from dduo import DiffPrivLib as dpl
with dduo.AdvEdOdometer() as odo:
  while noisy_acc < thresh or iters < max_iters:
    prev_bounds = bounds
    bounds = update_bounds(bounds)
    clf = dpl.GNB(bounds=bounds, epsilon=epsilon)
    clf.fit(X_train, y_train)
    prev_acc = noisy_acc
    accuracy = dpl.score(y_test, clf.predict(X_test))
    noisy_acc = dduo.gauss(epsilon_acc,delta,accuracy)
    if noisy_acc < prev_acc:
      bounds = prev_bounds
    iters += 1
dduo.print_privacy_cost()
```

```
Odometer_(ϵ, δ)({data.csv ↦ (0.82, 0.0035)})
```

Figure 15.4: DiffPrivLib: Naive Bayes Classification

```
from dduo import sklearn as sk
from dduo import DiffPrivLib as dpl
with dduo.AdvEdOdometer() as odo:
  while noisy_acc < thresh or norm > 0.0:
    dp_clf = dpl.LogisticRegression(epsilon=epsilon,
      data_norm = norm)
    dp_clf.fit(X_train, y_train)
    accuracy = dp_clf.score(X_test, y_test)
    noisy_acc = dduo.gauss(epsilon_acc,delta,accuracy)
    norm -= step
dduo.print_privacy_cost()
```

```
Odometer_(ϵ, δ)({data.csv ↦ (0.53, 0.0015)})
```

Figure 15.5: DiffPrivLib: Logistic Regression

Chapter 16

Lemmas, Theorems & Proofs

Lemma 16.0.1 (Plus Respects)**.** *If* $r_1 \sim_m^r r_2$ *Then* $(r_1 + r_3) \sim_m^r (r_2 + r_3)$.

Proof. By unfolding definitions and simple arithmetic. □

Lemma 16.0.2 (Times Respects)**.** *If* $r_1 \sim_m^r r_2$ *then* $(r_1 \times r_3) \sim_m^{r_3 r} (r_2 \times r_3)$.

Proof. By unfolding definitions and simple arithmetic. □

Lemma 16.0.3 (Triangle)**.** *If* $r_1 \sim_{m_A^{r_A}} r_2$ *and* $r_2 \sim_{m_B^{r_B}} r_3$ *then* $r_1 \sim_{m_A^{r_A + r_B} \sqcup m_B} r_3$.

Proof. By unfolding definitions, simple arithmetic, and the standard triangle inequality property for real numbers. □

Lemma 16.0.4 (Real Metric Weakening)**.** *If* $r_1 \sim_m^r r_2$, $r \leq r'$ *and* $m \sqsubseteq m'$ *then* $r_1 \sim_{m'}^{r'} r_2$.

Proof. By unfolding definitions and simple arithmetic. □

Lemma 16.0.5 (Step-index Weakening)**.** *For* $n' \leq n$*: (1): If* $\rho_1 \sim_n^\Sigma \rho_2$ *then* $\rho_1 \sim_{n'}^\Sigma \rho_2$*; (2): If* $\sigma_1 \sim_n^\Sigma \sigma_2$ *then* $\sigma_1 \sim_{n'}^\Sigma \sigma_2$*; and (3): If* $\rho_1, \sigma_1, e_1 \sim_n^\Sigma \rho_2, \sigma_2, e_2$ *then* $\rho_1, \sigma_1, e_1 \sim_{n'}^\Sigma \rho_2, \sigma_2, e_2$.

Proof. By mutual induction on n for all three properties, and additionally case analysis on e_1 and e_2 for property *(3)*. □

Theorem 16.0.6 (Metric Preservation)**.**

If: $\rho_1 \sim_n^{\Sigma} \rho_2$ *(H1)*

And: $\sigma_1 \sim_n^{\Sigma} \sigma_2$ *(H2)*

Then: $\rho_1, \sigma_1, e \sim_n^{\Sigma} \rho_2, \sigma_2, e$

That is, either $n = 0$ *or* $n = n' + 1$ *and...*

If: $n_1 \leq n$ *(H3)*

And: $\rho_1 \vdash \ulcorner \sigma_1, e \urcorner \Downarrow_{n_1} \ulcorner \sigma_1', v_1 \urcorner$ *(H4)*

And: $\rho_2 \vdash \ulcorner \sigma_2, e \urcorner \Downarrow_{n_2} \ulcorner \sigma_2', v_2 \urcorner$ *(H5)*

Then: $n_1 = n_2$ *(C1)*

And: $\sigma_1' \sim_{n - n_1^{\Sigma}} \sigma_2'$ *(C2)*

And: $v_1 \sim_{n - n_1^{\Sigma}} v_2$ *(C3)*

Proof. See detailed proof later in this section. □

Corollary 16.0.6.1 (Sound Dynamic Analysis for Sensitivity)**.**

If: $n_1 < n$, $n_2 < n$ *and* $n_3 < n$ *(H1)*

And: $\rho \sim_n^{\{o \mapsto \infty\}} \rho_1$ *(H2)*

And: $\rho \vdash \ulcorner \varnothing, e \urcorner \Downarrow_{n_1} \ulcorner \sigma, r @_m^{\Sigma} \urcorner$ *(H3)*

And: $\rho_1 \sim_n^{\Sigma'} \rho_2$ *(H4)*

And: $\rho_1 \vdash \ulcorner \varnothing, e \urcorner \Downarrow_{n_2} \ulcorner \sigma_1, r_1 @_{m_1^{\Sigma_1}} \urcorner$ *(H5)*

And: $\rho_2 \vdash \ulcorner \varnothing, e \urcorner \Downarrow_{n_3} \ulcorner \sigma_2, r_2 @_{m_2^{\Sigma_2}} \urcorner$ *(H6)*

Then: $r_1 \sim_{m_1}^{\Sigma' \cdot \Sigma} r_2$ *(C1)*

Proof.

By Metric Preservation, *(H2)*, *(H1)*, *(H3)* and *(H5)* we have $\Sigma_1 = \Sigma$ and $m_1 = m$.

By Metric Preservation, *(H4)*, *(H1)*, *(H5)* and *(H6)* we have proved the goal *(C1)*.

□

Proof of Metric Preservation.

By strong induction on n and case analysis on e:

- **Case** $n = 0$: Trivial by definition.

- **Case** $n = n + 1$ **and** $e = x$:

  By inversion on *(H4)* and *(H5)* we have: $n_1 = n_2 = 0$, $\sigma_1' = \sigma_1$, $\sigma_2' = \sigma_2$, $v_1 = \rho_1(x)$ and $v_2 = \rho_2(x)$. To show: *(C1)*: $0 = 0$; *(C2)*: $\sigma_1 \sim_n^{\Sigma} \sigma_2$; and *(C3)*: $\rho_1(x) \sim_n^{\Sigma} \rho_2(x)$. *(C1)* is trivial. *(C2)* is immediate by *(H2)*. *(C3)* is immediate by *(H1)*.

- **Case** $n = n + 1$ **and** $e = r$:

  By inversion on *(H4)* and *(H5)* we have $n_1 = n_2 = 0$, $v_1 = v_2 = r@_{\mathtt{disc}}^{\mathbf{Z}}$, $\sigma_1' = \sigma_1$ and $\sigma_2' = \sigma_2$. To show: *(C1)*: $0 = 0$; *(C2)*: $\sigma_1 \sim_n^{\Sigma} \sigma_2$; and *(C3)*: $r \sim_{\mathtt{disc}}^{0} r$. *(C1)* is trivial. *(C2)* is immediate by *(H2)*. *(C3)* is immediate by definition of relation $\sim_m^r$.

- **Case** $n = n + 1$ **and** $e = e_1 + e_2$:

  By inversion on *(H4)* and *(H5)* we have:

$\rho_1 \vdash \lceil \sigma_1, e_1 \rfloor \Downarrow_{n_{11}} \lceil \sigma_1', r_{11}@_{m_{11}^{\Sigma_{11}}} \rfloor$ *(H4.1)*

$\rho_1 \vdash \lceil \sigma_1', e_2 \rfloor \Downarrow_{n_{12}} \lceil \sigma_1'', r_{12}@_{m_{12}^{\Sigma_{12}}} \rfloor$ *(H4.2)*

$\rho_2 \vdash \lceil \sigma_2, e_1 \rfloor \Downarrow_{n_{21}} \lceil \sigma_2', r_{21}@_{m_{21}^{\Sigma_{21}}} \rfloor$ *(H5.1)*

$\rho_2 \vdash \lceil \sigma_2', e_2 \rfloor \Downarrow_{n_{22}} \lceil \sigma_2'', r_{22}@_{m_{22}^{\Sigma_{22}}} \rfloor$ *(H5.2)*

and we also have: $n_1 = n_{11} + n_{12}$, $n_2 = n_{21} + n_{22}$, $\sigma_1' = \sigma_1''$, $\sigma_2' = \sigma_2''$, $v_1 = (r_{11}+r_{12})@_{m_{11}^{\Sigma_{11}+\Sigma_{12}} \sqcup m_{12}}$ and $v_2 = (r_{21}+r_{22})@_{m_{21}^{\Sigma_{21}+\Sigma_{22}} \sqcup m_{22}}$. By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_{11} = n_{21}$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n_{11}^{\Sigma}} \sigma_2'$ *(IH.1.C2)*; and $r_{11}@_{m_{11}^{\Sigma_{11}}} \sim_{n-n_{11}^{\Sigma}} r_{21}@_{m_{21}^{\Sigma_{21}}}$ *(IH.1.C3)*. By unfolding the definition in *(IH.1.C3)*, we have: $\Sigma_{11} = \Sigma_{21}$ *(IH.1.C3.1)*; $m_{11} = m_{21}$ *(IH.1.C3.2)*; and $r_{11} \sim_{m_{11}^{\Sigma \cdot \Sigma_{11}}} r_{21}$ *(IH.1.C3.3)*. Note the following facts: $n_{12} \leq n - n_{11}$ *(F1)*; and $\rho_1 \sim_{n-n_{11}^{\Sigma}}$ *(F2)*. *(F1)* follows from *(H3)* and $n_1 = n_{11} + n_{12}$. *(F2)* follows from *(H1)* and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), *(F2)*, *(IH.1.C2)*, *(F1)*, *(H4.2)* and *(H5.2)* we have: $n_{12} = n_{22}$ *(IH.2.C1)*; $\sigma_1'' \sim_{n-n_{11}^{\Sigma}-n_{12}} \sigma_2''$ *(IH.2.C2)*; and $r_{12}@_{m_{12}^{\Sigma_{12}}} \sim_{n_{11}^{\Sigma}-n_{12}} r_{22}@_{m_{22}^{\Sigma_{22}}}$ *(IH.2.C3)*. By unfolding the definition in *(IH.2.C3)*, we have: $\Sigma_{12} = \Sigma_{22}$ *(IH.2.C3.1)*; $m_{12} = m_{22}$ *(IH.2.C3.2)*; and $r_{12} \sim_{m_{12}^{\Sigma \cdot \Sigma_{12}}} r_{22}$ *(IH.2.C3.3)*. To show: *(C1)*: $n_{11} + n_{12} = n_{21} + n_{22}$; *(C2)*: $\sigma_1'' \sim_{n-n_{11}^{\Sigma}-n_{12}} \sigma_2''$; and *(C3)*: $(r_{11} + r_{12})@_{m_{11}^{\Sigma_{11}+\Sigma_{12}} \sqcup m_{12}} \sim_{n-n_{11}^{\Sigma}-n_{12}} (r_{21} + r_{22})@_{m_{21}^{\Sigma_{21}+\Sigma_{22}} \sqcup m_{22}}$. *(C1)* is immediate from *(IH.1.C1)* and *(IH.2.C1)*. *(C2)* is immediate from *(IH.2.C2)*. To show *(C3)* we must show: *(C3.1)*: $\Sigma_{11}+\Sigma_{12} = \Sigma_{21}+\Sigma_{22}$; *(C3.2)*: $m_{11} \sqcup m_{12} = m_{21} \sqcup m_{22}$; and *(C3.3)*: $(r_{11} + r_{12}) \sim_{m_{11}^{\Sigma \cdot (\Sigma_{11}+\Sigma_{12})} \sqcup m_{12}} (r_{21} + r_{22})$. *(C3.1)* is immediate from *(IH.1.C3.1)* and *(IH.2.C3.1)*. *(C3.2)* is immediate from *(IH.1.C3.2)* and *(IH.2.C3.2)*. *(C3.3)* holds as follows: By Plus Respects, *(IH.1.C3.3)* and *(IH.2.C3.3)*: $(r_{11}+r_{12}) \sim_{m_{11}^{\Sigma \cdot \Sigma_{11}}} (r_{21}+r_{12}) \sim_{m_{12}^{\Sigma \cdot \Sigma_{12}}} (r_{21}+r_{22})$. By Triangle: $(r_{11} + r_{12}) \sim_{m_{11}^{\Sigma \cdot \Sigma_{11}+\Sigma \cdot \Sigma_{12}} \sqcup m_{12}} (r_{21} + r_{22})$. By basic algebra: $(r_{11} + r_{12}) \sim_{m_{11}^{\Sigma \cdot (\Sigma_{11}+\Sigma_{12})} \sqcup m_{12}} (r_{21} + r_{22})$ We have shown the goal.

- **Case** $n = n + 1$ **and** $e = e_1 \ltimes e_2$:

By inversion on *(H4)* and *(H5)* we have:

$\rho_1 \vdash \ulcorner \sigma_1, e_1 \urcorner \Downarrow_{n_{11}} \ulcorner \sigma_1', r_{11}@_{m_{11}^{\mathbf{z}}} \urcorner$ *(H4.1)*

$\rho_1 \vdash \ulcorner \sigma_1', e_2 \urcorner \Downarrow_{n_{12}} \ulcorner \sigma_1'', r_{12}@_{m_{12}^{\mathbf{z}}} \urcorner$ *(H4.2)*

$\rho_2 \vdash \ulcorner \sigma_2, e_1 \urcorner \Downarrow_{n_{21}} \ulcorner \sigma_2', r_{21}@_{m_{21}^{\Sigma_1}} \urcorner$ *(H5.1)*

$\rho_2 \vdash \ulcorner \sigma_2', e_2 \urcorner \Downarrow_{n_{22}} \ulcorner \sigma_2'', r_{22}@_{m_{22}^{\Sigma_2}} \urcorner$ *(H5.2)*

and we also have: $n_1 = n_{11} + n_{12}$, $n_2 = n_{21} + n_{22}$, $\sigma_1' = \sigma_1''$, $\sigma_2' = \sigma_2''$, $v_1 = (r_{11} \times r_{12})@_{m_{12}^{r_{11}\Sigma_{12}}}$ and $v_2 = (r_{21} \times r_{22})@_{m_{22}^{r_{21}\Sigma_{22}}}$. By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_{11} = n_{21}$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n_{11}^{\Sigma}} \sigma_2'$ *(IH.1.C2)*; and $r_{11}@_{m_{11}^{\Sigma_{11}}} \sim_{n-n_{11}^{\Sigma}-n_{12}} r_{21}@_{m_{21}^{\Sigma_{21}}}$ *(IH.1.C3)*. By unfolding the definition in *(IH.1.C3)*, we have: $\Sigma_{11} = \Sigma_{21}$ *(IH.1.C3.1)*; $m_{11} = m_{21}$ *(IH.1.C3.2)*; and $r_{11} \sim_{m_{11}^{0}} r_{21}$ *(IH.1.C3.3)*. As a consequence of *(IH.1.C3)*, we have: $r_{11} = r_{21}$. Note the following facts: $n_{12} \leq n - n_{11}$ *(F1)*; and $\rho_1 \sim_{n-n_{11}^{\Sigma}}$ *(F2)*. *(F1)* follows from *(H3)* and $n_1 = n_{11} + n_{12}$. *(F2)* follows from *(H1)* and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), *(F2)*, *(IH.1.C2)*, *(F1)*, *(H4.2)* and *(H5.2)* we have: $n_{12} = n_{22}$ *(IH.2.C1)*; $\sigma_1'' \sim_{n-n_{11}^{\Sigma}-n_{12}} \sigma_2''$ *(IH.2.C2)*; and $r_{12}@_{m_{12}^{\Sigma_{12}}} \sim_{n_{11}^{\Sigma}-n_{12}} r_{22}@_{m_{22}^{\Sigma_{22}}}$ *(IH.2.C3)*. By unfolding the definition in *(IH.2.C3)*, we have: $\Sigma_{12} = \Sigma_{22}$ *(IH.2.C3.1)*; $m_{12} = m_{22}$ *(IH.2.C3.2)*; and $r_{12} \sim_{m_{12}^{\Sigma \cdot \Sigma_{12}}} r_{22}$ *(IH.2.C3.3)*. To show: *(C1)*: $n_{11} + n_{12} = n_{21} + n_{22}$; *(C2)*: $\sigma_1'' \sim_{n-n_{11}^{\Sigma}-n_{12}} \sigma_2''$; and *(C3)*: $(r_{11} \times r_{12})@_{m_{12}^{r_{11}\Sigma_{12}}} \sim_{n-n_{11}^{\Sigma}-n_{12}} (r_{21} \times r_{22})@_{m_{22}^{(r_{21}\Sigma_{22})}}$. *(C1)* is immediate from *(IH.1.C1)* and *(IH.2.C1)*. *(C2)* is immediate from *(IH.2.C2)*. To show *(C3)* we must show: *(C3.1)*: $r_{11}\Sigma_{12} = r_{21}\Sigma_{22}$; *(C3.2)*: $m_{12} = m_{22}$; and *(C3.3)*: $(r_{11} \times r_{12}) \sim_{m_{12}^{\Sigma \cdot r_{11}\Sigma_{12}}} (r_{21} \times r_{22})$. *(C3)* holds as follows: *(C3.1)* is immediate from $r_{11} = r_{21}$ and *(IH.2.C3.1)*. *(C3.2)* is immediate from *(IH.2.C3.2)*. *(C3.3)* holds as follows: By Times Respects, $r_{11} = r_{21}$ and *(IH.2.C3.3)*: $(r_{11} \times r_{12}) = (r_{21} \times r_{12}) \sim_{m_{21}^{r_{11}(\Sigma \cdot \Sigma_{11})}} (r_{21} \times r_{22})$. By basic algebra: $(r_{11} \times r_{12}) = (r_{21} \times r_{12}) \sim_{m_{21}^{\Sigma \cdot r_{11}\Sigma_{11}}} (r_{21} \times r_{22})$. We have shown the goal.

- **Case** $n = n + 1$ **and** $e = e_1 \rtimes e_2$: *Analogous to previous case.*

- **Case** $n = n + 1$ **and** $e = \texttt{if0}(e_1)\{e_2\}\{e_3\}$:

  By inversion on *(H4)* and *(H5)* we have 4 subcases, each which induce:

  $\rho_1 \vdash \lceil \sigma_1, e_1 \rfloor \Downarrow_{n_{11}} \lceil \sigma_1', r_1@_{m_1}\mathbf{z} \rfloor$ *(H4.1)*

  $\rho_2 \vdash \lceil \sigma_2, e_1 \rfloor \Downarrow_{n_{21}} \lceil \sigma_2', r_2@_{m_2}\mathbf{z} \rfloor$ *(H5.1)*

  By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_{11} = n_{21}$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n_{11}^{\Sigma}} \sigma_2'$ *(IH.1.C2)*; and $r_1 \sim_{m_1^0 \sqcap m_2} r_2$ *(IH.1.C3)*. As a consequence of *(IH.1.C3)*, we have: $r_1 = r_2$. The 4 subcases initially are for: *(1)*: $r_1 = 0$ and $r_2 = 0$; *(2)*: $r_1 = 0$ and $r_2 \neq 0$; *(3)*: $r_1 \neq 0$ and $r_2 = 0$; and *(4)*: $r_1 \neq 0$ and $r_2 \neq 0$. However 2 are absurd given $r_1 = r_2$, so these 4 subcases collapse to 2:

  - **Subcase** $r_1 = r_2 = 0$:

    From prior inversion on *(H4)* and *(H5)* and fact $r_1 = r_2 = 0$, we also have:

    $\rho_1 \vdash \lceil \sigma_1', e_2 \rfloor \Downarrow_{n_{12}} \lceil \sigma_1'', v_1 \rfloor$ *(H4.2)*

    $\rho_2 \vdash \lceil \sigma_2', e_2 \rfloor \Downarrow_{n_{22}} \lceil \sigma_2'', v_2 \rfloor$ *(H5.2)*

    and we also have: $n_1 = n_{11} + n_{12}$, $n_2 = n_{21} + n_{22}$, $\sigma_1' = \sigma_1''$, $\sigma_2' = \sigma_2''$, $v_1 = v_1$ and $v_2 = v_2$. We continue reasoning in a generic way outside this subcase...

  - **Subcase** $r_1 \neq 0$ and $r_2 \neq 0$:

    From prior inversion on *(H4)* and *(H5)* and fact $r_1 = r_2 \neq 0$, we also have:

    $\rho_1 \vdash \lceil \sigma_1', e_3 \rfloor \Downarrow_{n_{12}} \lceil \sigma_1'', v_1 \rfloor$ *(H4.2)*

    $\rho_2 \vdash \lceil \sigma_2', e_3 \rfloor \Downarrow_{n_{22}} \lceil \sigma_2'', v_2 \rfloor$ *(H5.2)*

    and we also have: $n_1 = n_{11} + n_{12}$ $n_2 = n_{21} + n_{22}$ $\sigma_1' = \sigma_1''$ $\sigma_2' = \sigma_2''$ $v_1 = v_1$ $v_2 = v_2$ We continue reasoning in a generic way outside this subcase...

  Note the following facts: $n_{12} \leq n - n_{11}$ *(F1)*; and $\rho_1 \sim_{n-n_{11}^{\Sigma}}$ *(F2)*. *(F1)* follows from *(H3)* and $n_1 = n_{11} + n_{12}$. *(F2)* follows from *(H1)* and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), *(F2)*, *(IH.1.C2)*, *(F1)*, *(H4.2)* and *(H5.2)* we have: $n_{21} = n_{22}$ *(IH.2.C1)*; $\sigma_1'' \sim_{n-n_{11}^{\Sigma}-n_{21}} \sigma_2''$ *(IH.2.C2)*;

and $v_1 \sim_{n-n^{\Sigma}_{11}-n_{21}} v_2$ *(IH.2.C3)*. To show: *(C1)*: $n_{11}+n_{12} = n_{21}+n_{22}$; *(C2)*: $\sigma_1'' \sim_{n-n^{\Sigma}_{11}-n_{12}} \sigma_2''$; and *(C3)*: $v_1 \sim_{n-n^{\Sigma}_{11}-n_{12}} v_2$. *(C1)* is immediate from *(IH.1.C1)* and *(IH.2.C1)*. *(C2)* is immediate from *(IH.2.C2)*. *(C3)* is immediate from *(IH.2.C3)*

- **Case** $n = n+1$ **and** $e = \langle e_1, e_2 \rangle$:

  By inversion on *(H4)* and *(H5)* we have:

  $\rho_1 \vdash \lceil \sigma_1, e_1 \rfloor \Downarrow_{n_{11}} \lceil \sigma_1', v_{11} \rfloor$ *(H4.1)*
  $\rho_1 \vdash \lceil \sigma_1', e_2 \rfloor \Downarrow_{n_{12}} \lceil \sigma_1'', v_{12} \rfloor$ *(H4.2)*
  $\rho_2 \vdash \lceil \sigma_2, e_1 \rfloor \Downarrow_{n_{21}} \lceil \sigma_2', v_{21} \rfloor$ *(H5.1)*
  $\rho_2 \vdash \lceil \sigma_2', e_2 \rfloor \Downarrow_{n_{22}} \lceil \sigma_2'', v_{22} \rfloor$ *(H5.2)*

  and we also have: $n_1 = n_{11}+n_{12}$, $n_2 = n_{21}+n_{22}$, $\sigma_1' = \sigma_1''$, $\sigma_2' = \sigma_2''$, $v_1 = \langle v_{11}, v_{12} \rangle$ and $v_2 = \langle v_{21}, v_{22} \rangle$. By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_{11} = n_{21}$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n^{\Sigma}_{11}} \sigma_2'$ *(IH.1.C2)*; and $v_{11} \sim_{n-n^{\Sigma}_{11}} v_{21}$ *(IH.1.C3)*. Note the following facts: $n_{12} \leq n - n_{11}$ *(F1)*; and $\rho_1 \sim_{n-n^{\Sigma}_{11}}$ *(F2)*. *(F1)* follows from *(H3)* and $n_1 = n_{11}+n_{12}$. *(F2)* follows from *(H1)* and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), *(F2)*, *(IH.1.C2)*, *(F1)*, *(H4.2)* and *(H5.2)* we have: $n_{12} = n_{22}$ *(IH.2.C1)*; $\sigma_1'' \sim_{n-n^{\Sigma}_{11}-n_{12}} \sigma_2''$ *(IH.2.C2)*; and $v_{12} \sim_{n-n^{\Sigma}_{11}-n_{12}} v_{22}$ *(IH.2.C3)*. To show: *(C1)*: $n_{11}+n_{12} = n_{21}+n_{22}$; *(C2)*: $\sigma_1'' \sim_{n-n^{\Sigma}_{11}-n_{12}} \sigma_2''$; and *(C3)*: $\langle v_{11}, v_{12} \rangle \sim_{n-n^{\Sigma}_{11}-n_{12}} \langle v_{21}, v_{22} \rangle$. *(C1)* is immediate from *(IH.1.C1)* and *(IH.2.C1)*. *(C2)* is immediate from *(IH.2.C2)*. *(C3)* is immediate from *(IH.1.C3)* and *(IH.2.C3)*.

- **Case** $n = n+1$ **and** $e = \pi_i(e)$:

  By inversion on *(H4)* and *(H5)* we have:

  $\rho_1 \vdash \lceil \sigma_1, e \rfloor \Downarrow_{n_1} \lceil \sigma_1', \langle v_{11}, v_{12} \rangle \rfloor$ *(H4.1)*
  $\rho_2 \vdash \lceil \sigma_2, e \rfloor \Downarrow_{n_2} \lceil \sigma_2', \langle v_{21}, v_{22} \rangle \rfloor$ *(H5.1)*

and we also have: $n_1 = n_1$, $n_2 = n_2$, $\sigma_1' = \sigma_1'$, $\sigma_2' = \sigma_2'$, $v_1 = v_{1i}$ and $v_2 = v_{2i}$. By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_1 = n_2$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n_1^\Sigma} \sigma_2'$ *(IH.1.C2)*; and $\langle v_{11}, v_{12}\rangle \sim_{n-n_1^\Sigma} \langle v_{21}, v_{22}\rangle$ *(IH.1.C3)*. To show: *(C1)*: $n_1 = n_2$; *(C2)*: $\sigma_1' \sim_{n-n_1^\Sigma} \sigma_2'$; and *(C3)*: $v_{1n'} \sim_{n-n_1^\Sigma} v_{2n'}$. *(C1)* is immediate from *(IH.1.C1)*. *(C2)* is immediate from *(IH.1.C2)*. *(C3)* is immediate from *(IH.1.C3)*.

- **Case** $n = n{+}1$ **and** $e = \lambda x.\ e$: By inversion on *(H4)* and *(H5)* we have: $n_1 = 0$, $n_2 = 0$, $\sigma_1' = \sigma_1$, $\sigma_2' = \sigma_2$, $v_1 = \langle \lambda x.\ e \mid \rho_1\rangle$ and $v_2 = \langle \lambda x.\ e \mid \rho_2\rangle$. To show: *(C1)*: $0 = 0$; *(C2)*: $\sigma_1 \sim_n^\Sigma \sigma_2$; and *(C3)*: $\langle \lambda x.\ e \mid \rho_1\rangle \sim_n^\Sigma \langle \lambda x.\ e \mid \rho_2\rangle$. *(C1)* is trivial. *(C2)* is immediate from *(H2)*. *(C3)* holds as follows: Unfolding the definition, we must show: *(C3)*: $\forall n' \le n, v_1, v_2, \sigma_1', \sigma_2'.\ \sigma_1' \sim_{n'}^\Sigma \sigma_2' \ \wedge\ v_1 \sim_{n'}^\Sigma v_2. \Rightarrow \sigma_1', \{x \mapsto v_1\} \uplus \rho_1, e \sim_{n'}^\Sigma \sigma_2', \{x \mapsto v_2\} \uplus \rho_2, e$
To show *(C3)*, we assume: $\sigma_1' \sim_{n'}^\Sigma \sigma_2'$ *(C3.H1)*; and $v_1 \sim_{n'}^\Sigma v_2$ *(C3.H2)*. And we must show: *(C3.1)*: $\sigma_1', \{x \mapsto v_1\} \uplus \rho_1, e \sim_{n'}^\Sigma \sigma_2', \{x \mapsto v_2\} \uplus \rho_2, e$. Note the following facts: $\rho_1 \sim_{n'}^\Sigma \rho_1$ *(F1)*; and $\{x \mapsto v_1\} \uplus \rho_1 \sim_{n'}^\Sigma \{x \mapsto v_2\} \uplus \rho_2$ *(F2)*. *(F1)* holds from *H1* and Step-index Weakening.1. *(F2)* holds from *(F1)*, *(C3.H2)* and the definition of $\rho \sim_{n'}^\Sigma \rho$. *(C3.1)* then holds by IH ($n = n'$ decreasing), $F2$ and $C3.H1$.

- **Case** $n = n + 1$ **and** $e = e_1(e_2)$:

By inversion on *(H4)* and *(H5)* we have:

$$\rho_1 \vdash \lceil \sigma_1, e_1 \rfloor \Downarrow_{n_{11}} \lceil \sigma_1', \langle \lambda x.\ e_1' \mid \rho_1'\rangle \rfloor \quad \textit{(H4.1)}$$
$$\rho_1 \vdash \lceil \sigma_1', e_2 \rfloor \Downarrow_{n_{12}} \lceil \sigma_1'', v_1 \rfloor \quad \textit{(H4.2)}$$
$$\{x \mapsto v_1\} \uplus \rho_1' \vdash \lceil \sigma_2'', e_1' \rfloor \Downarrow_{n_{13}} \lceil \sigma_1''', v_1' \rfloor \quad \textit{(H4.3)}$$
$$\rho_2 \vdash \lceil \sigma_2, e_1 \rfloor \Downarrow_{n_{21}} \lceil \sigma_2', \langle \lambda x.\ e_2' \mid \rho_2'\rangle \rfloor \quad \textit{(H5.1)}$$
$$\rho_2 \vdash \lceil \sigma_2', e_2 \rfloor \Downarrow_{n_{22}} \lceil \sigma_2'', v_2 \rfloor \quad \textit{(H5.2)}$$
$$\{x \mapsto v_2\} \uplus \rho_2' \vdash \lceil \sigma_2'', e_2' \rfloor \Downarrow_{n_{23}} \lceil \sigma_2''', v_2' \rfloor \quad \textit{(H5.3)}$$

and we also have: $n_1 = n_{11} + n_{12} + n_{13} + 1$, $n_2 = n_{21} + n_{22} + n_{23} + 1$, $\sigma_1' = \sigma_1'''$, $\sigma_2' = \sigma_2'''$, $v_1 = v_1'$ and $v_2 = v_2'$. By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_{11} = n_{21}$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n_{11}^{\Sigma}} \sigma_2'$ *(IH.1.C2)*; and $\langle \lambda x.\ e_1' \mid \rho_1' \sim_{n-n_{11}^{\Sigma}} v_{21}$ *(IH.1.C3)*. Note the following facts: $n_{12} \leq n - n_{11}$ *(F1)*; and $\rho_1 \sim_{n-n_{11}^{\Sigma}}$ *(F2)*. *(F1)* follows from *(H3)* and $n_1 = n_{11}+n_{12}+n_{13}+1$. *(F2)* follows from *(H1)* and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), *(H2)*, *(IH.1.C2)*, *(F1)*, *(H4.2)* and *(H5.2)* we have: $n_{12} = n_{22}$ *(IH.2.C1)*; $\sigma_1'' \sim_{n-n_{11}^{\Sigma}-n_{12}} \sigma_2''$ *(IH.2.C2)*; and $v_1 \sim_{n-n_{11}^{\Sigma}-n_{12}} v_2$ *(IH.2.C3)*. Note the following facts, each of which follow from *(H3)* and $n_1 = n_{11}+n_{12}+n_{13}+1$: $n_{13} \leq n-n_{11}-n_{12}$ *(F3)*; and $n-n_{11}-n_{12}-n_{13} > 0$ *(F4)*. Also note the following facts which follow from *(IH.1.C3)*, *(IH.2.C2)*, *(IH.2.C3)*, *(F3)* and *(F4)*: $n_{13} = n_{23}$ *(F4.C1)*; $\sigma_1''' \sim_{n-n_{11}^{\Sigma}-n_{12}-n_{13}-1} \sigma_2'''$ *(F4.C2)*; and $v_1' \sim_{n-n_{11}^{\sigma}-n_{12}-n_{13}-1} v_2'$ *(F4.C3)*. To show: *(C1)*: $n_{11} + n_{12} + n_{13} + 1 = n_{21} + n_{22} + n_{23} + 1$; *(C2)*: $\sigma_1''' \sim_{n-n_{11}^{\Sigma}-n_{12}-n_{13}-1} \sigma_2'''$; and *(C3)*: $v_1' \sim_{n-n_{11}^{\Sigma}-n_{12}-n_{13}-1} v_2'$. *(C1)* is immediate from *(IH.1.C1)*, *(IH.2.C1)* and *(F4.C1)*. *(C2)* is immediate from *(F4.C2)* and Step-index Weakening.2. *(C3)* is immediate from *(F4.C3)* and Step-index Weakening.3.

- **Case** $n = n + 1$ **and** $e = \texttt{ref}(e)$:

  By inversion on *(H4)* and *(H5)* we have:

  $$\rho_1 \vdash \ulcorner \sigma_1, e \urcorner \Downarrow_{n_1} \ulcorner \sigma_1', v_1 \urcorner \quad \textit{(H4.1)}$$
  $$\ell_1 = \text{alloc}(\text{dom}(\sigma_1')) \quad \textit{(H4.2)}$$
  $$\rho_2 \vdash \ulcorner \sigma_2, e \urcorner \Downarrow_{n_2} \ulcorner \sigma_2', v_2 \urcorner \quad \textit{(H5.1)}$$
  $$\ell_2 = \text{alloc}(\text{dom}(\sigma_2')) \quad \textit{(H4.2)}$$

  and we also have: $n_1 = n_1$, $n_2 = n_2$, $\sigma_1' = \{\ell \mapsto v_1\} \uplus \sigma_1'$, $\sigma_2' = \{\ell \mapsto v_2\} \uplus \sigma_2'$, $v_1 = \ell_1$ and $v_2 = \ell_2$. By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_1 = n_2$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n_1^{\Sigma}} \sigma_2'$ *(IH.1.C2)*; and $v_1 \sim_{n-n_1^{\Sigma}} v_2$ *(IH.1.C3)*. Because $\sigma_1 \sim_{n-n_1^{\Sigma}} \sigma_2'$, we know $\text{dom}(\sigma_1) = \text{dom}(\sigma_2)$ and therefore

$\ell_1 = \ell_2$. To show: *(C1)*: $n_1 = n_2$; *(C2)*: $\{\ell \mapsto v_1\} \uplus \sigma_1' \sim_{n-n_1^\Sigma} \{\ell \mapsto v_2\} \uplus \sigma_2'$; and *(C3)*: $\ell_1 \sim_{n-n_1^\Sigma} \ell_2$. *(C1)* is immediate from *(IH.1.C1)*. *(C2)* is immediate from *(IH.1.C2)*, *(IH.1.C3)* and the definition of $\sigma \sim_n^\Sigma \sigma$. *(C3)* is immediate by definition of $v \sim_n^\Sigma v$ and $\ell_1 = \ell_2$.

- **Case** $n = n + 1$ **and** $e = !e$:

  By inversion on *(H4)* and *(H5)* we have:

  $\rho_1 \vdash \lceil \sigma_1, e \rceil \Downarrow_{n_1} \lceil \sigma_1', \ell_1 \rceil$ *(H4.1)*

  $\rho_2 \vdash \lceil \sigma_2, e \rceil \Downarrow_{n_2} \lceil \sigma_2', \ell_2 \rceil$ *(H5.1)*

  and we also have: $n_1 = n_1$, $n_2 = n_2$, $\sigma_1' = \sigma_1'$, $\sigma_2' = \sigma_2'$, $v_1 = \sigma_1'(\ell_1)$ and $v_2 = \sigma_2'(\ell_2)$. By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_1 = n_2$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n_1^\Sigma} \sigma_2'$ *(IH.1.C2)*; and $\ell_1 \sim_{n-n_1^\Sigma} \ell_2$ *(IH.1.C3)*. Because $\ell_1 \sim_{n-n_1^\Sigma} \ell_2$, we know $\ell_1 = \ell_2$ by definition of $v \sim_n^\Sigma v$. To show: *(C1)*: $n_1 = n_2$; *(C2)*: $\sigma_1' \sim_{n-n_1^\Sigma} \sigma_2'$; and *(C3)*: $\sigma_1'(\ell_1) \sim_{n-n_1^\Sigma} \sigma_2'(\ell_2)$. *(C1)* is immediate from *(IH.1.C1)*. *(C2)* is immediate from *(IH.1.C2)*. *(C3)* is immediate from *(IH.1.C2)* and $\ell_1 = \ell_2$.

- **Case** $n = n + 1$ **and** $e = e_1 \leftarrow e_2$:

  By inversion on *(H4)* and *(H5)* we have:

  $\rho_1 \vdash \lceil \sigma_1, e_1 \rceil \Downarrow_{n_{11}} \lceil \sigma_1', \ell_1 \rceil$ *(H4.1)*

  $\rho_1 \vdash \lceil \sigma_1', e_2 \rceil \Downarrow_{n_{12}} \lceil \sigma_1'', \ell_2 \rceil$ *(H4.2)*

  $\rho_2 \vdash \lceil \sigma_2, e_1 \rceil \Downarrow_{n_{21}} \lceil \sigma_2', v_1 \rceil$ *(H5.1)*

  $\rho_2 \vdash \lceil \sigma_2', e_2 \rceil \Downarrow_{n_{22}} \lceil \sigma_2'', v_2 \rceil$ *(H5.2)*

  and we also have: $n_1 = n_{11} + n_{12}$, $n_2 = n_{21} + n_{22}$, $\sigma_1' = \sigma_1''$, $\sigma_2' = \sigma_2''$, $v_1 = v_1$ and $v_2 = v_2$. By IH ($n = n$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H5.1)* we have: $n_{11} = n_{21}$ *(IH.1.C1)*; $\sigma_1' \sim_{n-n_{11}^\Sigma} \sigma_2'$ *(IH.1.C2)*; and $\ell_1 \sim_{n-n_{11}^\Sigma} \ell_2$ *(IH.1.C3)*. Because $\ell_1 \sim_{n-n_{11}^\Sigma} \ell_2$ we know $\ell_1 = \ell_2$. Note the following facts: $n_{12} \leq n - n_{11}$ *(F1)*; and $\rho_1 \sim_{n-n_{11}^\Sigma}$ *(F2)*. *(F1)* follows

from *(H3)* and $n_1 = n_{11} + n_{12}$. *(F2)* follows from *(H1)* and Step-index Weakening.1. By IH ($n = n - n_{11}$ decreasing), *(F2)*, *(IH.1.C2)*, *(F1)*, *(H4.2)* and *(H5.2)* we have: $n_{12} = n_{22}$ *(IH.2.C1)*; $\sigma_1'' \sim_{n-n_{11}^{\Sigma}-n_{12}} \sigma_2''$ *(IH.2.C2)*; and $v_1 \sim_{n-n_{11}^{\Sigma}-n_{12}} v_2$ *(IH.2.C3)*. To show: *(C1)*: $n_{11} + n_{12} = n_{21} + n_{22}$; *(C2)*: $\sigma_1''[\ell \mapsto v_1] \sim_{n-n_{11}^{\Sigma}-n_{12}} \sigma_2''[\ell \mapsto v_2]$; and *(C3)*: $v_1 \sim_{n-n_{11}^{\Sigma}-n_{12}} v_2$. *(C1)* is immediate from *(IH.1.C1)* and *(IH.2.C1)*. *(C2)* is immediate from *(IH.2.C2)* and *(IH.2.C3)*. *(C3)* is immediate from *(IH.2.C3)*.

□

Part IV

Solo: A Lightweight Static Analysis for Differential Privacy

Chapter 17

Introduction and Contributions

All current approaches for statically enforcing differential privacy in higher order languages make use of either linear or relational refinement types. A barrier to adoption for these approaches is the lack of support for expressing these "fancy types" in mainstream programming languages. For example, no mainstream language supports relational refinement types, and although Rust and modern versions of Haskell both employ some linear typing techniques, they are inadequate for embedding enforcement of differential privacy, which requires "full" linear types a la Girard/Reynolds. We propose a new type system that enforces differential privacy, avoids the use of linear and relational refinement types, and can be easily embedded in mainstream richly typed programming languages such as Scala, OCaml and Haskell. We demonstrate such an embedding in Haskell, demonstrate its expressiveness on case studies, and prove that our type-based enforcement of differential privacy is sound.

Recent work has made significant progress towards techniques for static

verification of differentially private programs. Existing techniques typically define novel programming languages that incorporate specialized static type systems (linear types (Reed and Pierce, 2010; Near et al., 2019b), relational types (Barthe et al., 2015), dependent types (Gaboardi et al., 2013b), etc.). However, there remains a major challenge in bringing these techniques to practice: the specialized features they rely on do not exist in mainstream programming languages.

We introduce Solo, a novel type system for static verification of differential privacy, and present a reference implementation as a Haskell library. Solo is similar to Fuzz (Reed and Pierce, 2010) and its descendants in expressive power, but Solo does not rely on linear types and can be implemented entirely in Haskell with no additional language extensions. In particular, Solo's sensitivity and privacy tracking mechanisms are compatible with higher-order functions, and leverage Haskell's type inference system to minimize the need for additional type annotations.

In differential privacy, the *sensitivity* of a computation determines how much noise must be added to its result to achieve differential privacy. Fuzz-like languages track sensitivity relative to program variables, using a linear typing discipline. The key innovation in Solo is to track sensitivity relative to a set of global *data sources* instead, which eliminates the need for linear types. Compared to prior work on static verification of differential privacy, our system can be embedded in existing programming languages without support for linear types, and supports advanced variants of differential privacy like (ϵ, δ)-differential privacy and Rényi differential privacy.

We describe our approach using the Haskell implementation of Solo, and demonstrate its use to verify differential privacy for practical algorithms in four case studies. We formalize a subset of Solo's sensitivity analysis and prove

metric preservation, the soundness property for this analysis.

In summary, we make the following contributions:

- We introduce Solo, a novel type system for the static verification of differential privacy without linear types.
- We present a reference implementation of Solo as a Haskell library, which retains support for type inference and does not require additional language extensions.
- We formalize a subset of Solo's type system and prove its soundness.
- We demonstrate the applicability of the Solo library in four case studies.

Type Systems for Differential Privacy. The first static approach for verifying differential privacy in the context of higher-order programming constructs was Fuzz (Reed and Pierce, 2010). Fuzz uses linear types to verify both sensitivity and privacy properties of programs, even in the context of higher-order functions. Conceptual descendents of Fuzz include DFuzz (Gaboardi et al., 2013b), Adaptive Fuzz (Winograd-Cort et al., 2017), Fuzzi (Zhang et al., 2019a), Duet (Near et al., 2019b), and the system due to Azevedo de Amorim et al. (de Amorim et al., 2019). Approaches based on linear types combine a high degree of automation with support for higher-order programming, but require the host language to support linear types, so none has yet been implemented in a mainstream programming language. See Chapter 25 for a complete discussion of related work.

Chapter 18

Overview of Solo

Solo is a static analysis for differential privacy, which can be implemented as a library in Haskell. Its analysis is completely static, and it does not impose any runtime overhead. Solo requires special type annotations, but in many cases these types can be inferred, and typechecking is aided by the flexibilty of parametric polymorphism in Haskell. Solo retains many of the strengths of linear typing approaches to differential privacy, while taking a light-weight approach capable of being embedded in mainstream functional languages. Specifically, Solo:

1. is capable of sensitivity analysis for general-purpose programs in the context of higher order programming.

2. implements a privacy verification approach with separate privacy cost analysis for multiple program inputs using ideas from Duet.

3. leverages type-level dependency on values via Haskell singleton types, allowing verification of private programs with types that reference symbolic parameters

4. features verification of several recent variants of differential privacy including (ϵ, δ) and Rényi differential privacy.

However, Solo is not intended for the verification of low-level privacy mechanisms such as the core mechanisms described previously, the exponential mechanism (Dwork et al., 2014b), or the sparse vector technique (Dwork et al., 2014b).

18.1 Threat Model.

The threat model for Solo is "honest but fallible"—that is, we assume the programmer *intends* to write a differentially private program, but may make mistakes. Solo is intended as a tool to help the programmer implement correct differentially private programs in this context. Our approach implements a sound analysis for sensitivity and privacy, but its embedding in a larger system (Haskell) may result in weak points that a malicious programmer could exploit to subvert Solo's guarantees (unsoundness in Haskell's type system, for example). The Solo library can be used with Safe Haskell (Terei et al., 2012) to address this issue; Solo exports only a set of safe primitives which are designed to enforce privacy preserving invariants that adhere to our metatheory. However, Solo's protection against malicious programmers are only as strong as the guarantees made by Safe Haskell. Our guarantees against malicious programmers are therefore similar to those provided by language-based information flow control libraries that also utilize Safe Haskell (e.g. (Russo et al., 2008)).

18.2 Soundness.

We formalize our privacy analysis in terms of a metric preservation metatheory and prove its soundness in Chapter 22 via a step-indexed logical relation w.r.t. a

step-indexed big-step semantics relation. A consequence of metric preservation is that well-typed pure functions are semantically *sensitive* functions, and that well-typed monadic functions are semantically *differentially private* functions. Our model includes two variants of pair and list type connectives—one sensitive and the other non-sensitive—as well as recursive functions.

Chapter 19

SOLO by Example

This section introduces the usage of SOLO based on code examples written in our Haskell reference implementation.

19.1 Data Sources.

Our approach makes use of the idea that a static privacy analysis of a program can be centered around a set of sensitive *data sources* which the analyst wants to preserve privacy for. A data source may be represented by some identifier such as a string value, which represents some sensitive program input such as raw numeric data, a file or an IO stream. SOLO's data sources are inspired by ideas from static taint analysis—we "taint" the program's data sources with sensitivity annotations that are tracked and modified throughout type-checking.

To enable the static privacy analysis of a program, we track privacy information for data sources at the type-level. Because our analysis is based on the information flow of distinguished data source values throughout a program, we are able to perform a fully static analysis without precise tracking of variable usage within functions, and without a specialized linear type system. In SOLO,

data sources are created for sensitive inputs external to the program. For example, the `readDoubleFromIO` function reads in a sensitive double value from the user:

```
readDoubleFromIO :: ∀ m o. IO (SDouble m '[ '(o, 1) ])
```

Sensitive values (whose types are prefixed with "S" such as `SDouble`) represent base values that have been tagged with sensitivity tracking information—specifically, a *sensitivity environment* σ. `readDoubleFromIO` instantiates the sensitivity environment to `'[ '(o, 1) ]`, which indicates that values with this type are 1-sensitive with respect to the input read from the sensitive source. The distance metric identifier `m` specifies the metric used to measure distance (as described in Definition 2.2.1).

Sensitive values (like `SDouble`) are encapsulated in order to restrict their usage to only privacy preserving operations. The constructors of sensitive data types are hidden, and they are manipulated solely through trusted primitive operations provided in our implementation.

19.2 Sensitivity Tracking.

We can operate on `SDouble` values with an attached sensitivity environments using specialized operators provided by Solo. For example, Solo's `<+>` function adds two `SDouble` values, and has the following type:

```
(<+>) :: SDouble 'Diff s₁ -> SDouble 'Diff s₂ -> SDouble 'Diff (Plus s₁ s₂)
```

The type of the `<+>` function indicates that it adds the sensitivity environments of its arguments together (`(Plus a b)`), just like the typing rule for addition in Fuzz. The distance metric `Diff` is described in Chapter 20. We can use `<+>` to define a doubling function, as below. The type signature can be left off, and

will be inferred automatically by Haskell. The type signature of `dbl` describes its sensitivity—it is 2-sensitive in its argument.

```
dbl :: SDouble 'Diff senv -> SDouble 'Diff (Plus senv senv)
dbl x = x <+> x
```

19.3 Privacy.

We can use the Laplace and Gaussian mechanisms introduced in Part 2 to add noise to sensitive values and satisfy differential privacy. Solo tracks the total privacy cost of multiple uses of these mechanisms using a *privacy monad*, which is similar to the one used in Fuzz and related systems. Solo implements privacy monads for several different privacy variants, with conversion operations between them. These monads are described in detail in Chapter 21. The following function takes a `SDouble` as input, doubles it, and adds noise:

```
privacyFunc :: SDouble 'Diff '[ '(o, 1) ] -> EpsPrivacyMonad '[ '(o, 2) ] Double
privacyFunc x = laplace @2 Proxy x
```

The type `EpsPrivacyMonad '[ '(o, 2) ] Double` indicates that the function satisfies ϵ-differential privacy for $\epsilon = 2$. As in the previous example, Haskell is able to infer the type if the annotation is left off.

Chapter 20

Sensitivity Analysis

Prior type-based analyses for sensitivity analysis (Reed and Pierce, 2010; Gaboardi et al., 2013b; Winograd-Cort et al., 2017; Near et al., 2019b) focus on *function sensitivity* with respect to *program variables*. Solo's type system, in contrast, associates sensitivity with *base types* (not functions), and these sensitivities are determined with respect to *data sources* (not program variables). This difference represents a significant departure from previous systems, and is the key innovation that enables embedding Solo's type system in a language (like Haskell) without linear types. Figure 20.1 presents the types for the sensitivity analysis in the Solo system. The rest of this section describes types in Solo and how they can be used to describe the sensitivity of a program. We describe the privacy analysis in Chapter 21, and we formalize both analyses in Chapter 22.

20.1 Types, Metrics, and Environments

This section describes Figure 20.1 in detail. We begin with sources (written o), environments (Σ), metrics (m and w), types (τ), and sensitive types (σ).

```
import qualified GHC.TypeLits as TL

-- Sources & Sensitivity Environments (§20.1)
type Source = TL.Symbol                                  -- sensitive data sources
data Sensitivity = InfSens | NatSens TL.Nat              -- sensitivity values
type SEnv = [(Source, Sensitivity)]                      -- sensitivity environments

-- Distance Metrics (§20.1)
data NMetric = Diff | Disc                               -- distance metrics
SDouble :: NMetric -> SEnv -> *                          -- sensitive doubles

-- Pairs (§20.3.1)
data CMetric = L1 | L2 | LInf                            -- compound type metrics
SPair    :: CMetric -> (SEnv -> *) -> (SEnv -> *)        -- sensitive pairs
                    -> SEnv -> *
L1Pair   = SPair L1                                      -- ⊗-pairs in Fuzz
L2Pair   = SPair L2                                      -- Not in Fuzz
LInfPair = SPair LInf                                    -- &-pairs in Fuzz

-- Lists (§20.3.1)
SList    :: CMetric -> (SEnv -> *) -> SEnv -> *          -- sensitive lists
L1List   = SList L1                                      -- τ list in Fuzz
L2List   = SList L2                                      -- Not in Fuzz
LInfList = SList LInf                                    -- τ alist in Fuzz
```

Figure 20.1: Sensitivity Types in Solo.

Sources & Environments. *Sensitive data sources* are placeholders for sources of sensitive data external to the program (e.g. the filename of a file full of sensitive data that has been read in using `readSensitiveFile`). These placeholders are represented in Solo using type-level symbols. Solo tracks sensitivity *relative* to data sources (i.e. Solo assumes that data sources have an "absolute sensitivity" of 1). In Solo, like in Fuzz, *sensitivities* can be either a number or ∞. In Solo, numeric sensitivities are represented using type-level natural numbers. A *sensitivity environment* `SEnv` is an association list of data sources and their sensitivities, and corresponds to the same concept in Fuzz.

20.2 Distance Metrics & Metric-Carrying Types.

Interpreting sensitivity requires describing how to measure distances between values; different metrics for this measurement produce different privacy properties. Solo provides support for several distance metrics including those commonly used in differentially private algorithms. The *base metrics* listed in Figure 20.1 (`BMetric`) are distance metrics for base types. The *sensitive base types* (`SBase`) are metric-carrying base types (i.e. every sensitive type must have a distance metric). For example, the type of a sensitive `Double` would be `SBase Double m` , where `m` is a metric. The metrics for base types in Solo are:

- `Diff` , the *absolute difference metric*, is defined as the absolute value of the difference between two values: $d(x, y) = |x - y|$.
- `Disc` , the *discrete metric*, is 0 if its arguments are equivalent, and 1 if its arguments are distinct: $d(x, y) = 0 \ \textit{if} \ x = y; 1 \ \textit{otherwise}$

Thus the types `SBase Double Diff` and `SBase Double Disc` mean very different things when interpreting sensitivity. The distance between two values v_1, v_2 : `SBase Double Diff` is $|v_1 - v_2|$, but the distance between two values v_3, v_4 : `SBase Double Disc` is at most 1 (when $v_3 \neq v_4$).

Both of these metrics are useful in writing differentially private programs; basic mechanisms for differential privacy (like the Laplace mechanism) typically require their inputs to use the `Diff` metric, while the distance between program inputs is often described using the `Disc` metric. For example, we might consider a "database" of real numbers, each contributed by one individual; two neighboring databases in this setting will differ in exactly one of those numbers, but the change to the number itself may be unbounded. In this case, each number in the database would have the type `SBase Double Disc` . Fuzz fixes the distance metric for numbers to be the absolute difference metric; Duet provides two separate

types for real numbers, each with its own distance metric.

20.3 Types.

A sensitive type in Solo carries both a metric and a sensitivity environment (e.g. `SBase` has kind `* -> BMetric -> SEnv -> *`). Thus, sensitivities are associated with *values*, rather than with *program variables* (as in Fuzz). For example, the type `SDouble '[ '("sensitive_input", 1) ] 'Diff` from Chapter 19 is the type of a double value that is 1-sensitive with respect to the data source *input* under the absolute difference metric. Adding such a value to itself results in the type `SDouble '[ '("sensitive_input", 2) ] 'Diff` —encoding the fact that the sensitivity has doubled. In Fuzz, the same information is encoded by the sensitivities recorded in the context; but with respect to program variables rather than data sources. Note that it is not possible to attach a sensitivity environment to a function type—*only* the metric-carrying sensitive types may have associated sensitivity environments. Solo does not provide a "sensitive function" type connective (like Fuzz's $\multimap$); in Solo, function sensitivity must be stated in terms of the sensitivity of the function's arguments with respect to the program's data sources (more in Section 20.4).

20.3.1 Pairs and Lists

The Fuzz system contains two connectives for pairs, $\otimes$ and $\&$, which differ in their metrics. The distance between two $\otimes$ pairs is the sum of the distances between their elements, while the distance between two $\&$ pairs is the maximum of distances between their elements. Solo provides a single pair type, `SPair`, that can express both types by specifying a *compound metric* `CMetric`.

20.3.2 Compound Metrics

In Solo, metrics for compound types are derived from standard vector-space distance metrics. For example, a sensitive pair has the type `SPair w` where `w` is one of the compound metrics in Figure 20.1:

- `L1` , the L_1 (or *Manhattan*) distance, is the sum of the distances between corresponding elements: $d(x, y) = \sum_{x_i \in x, y_i \in y} d_i(x_i, y_i)$.
- `L2` , the L_2 (or *Euclidian*) distance, is the root of the sum of squares of distances between corresponding elements: $d(x, y) = \sqrt{\sum_{x_i \in x, y_i \in y} d_i(x_i, y_i)^2}$
- `LInf` , the L_∞ distance, is the maximum of the distances between corresponding elements: $d(x, y) = \max_{x_i \in x, y_i \in y} d_i(x_i, y_i)$

Thus we can represent Fuzz's $\otimes$ pairs in Solo using the `SPair L1` type constructor, and Fuzz's $\&$ pairs using `SPair LInf` . We can construct pairs from sensitive values using the following two functions:

```
makeL1Pair :: a m s1 -> b m s2 -> SPair L1 a b (Plus s1 s2)     -- Fuzz's ⊗-pair
makeLInfPair :: a m s1 -> b m s2 -> SPair LInf a b (Join s1 s2) -- Fuzz's &-pair
```

Here, the `Plus` operator for sensitivity environments performs elementwise addition on sensitivities, and the `Join` operator performs elementwise maximum.

20.3.3 Lists

Fuzz defines the list type τ `list`, and gives types to standard operators over lists reflecting their sensitivities. In Solo, we define the `SList` type to represent sensitive lists. Sensitive lists in Solo carry a metric, in the same way as sensitive pairs, and can only contain metric-carrying types. The type of a sensitive list of doubles with the L_1 distance metric, for example, is `SList L1 SDouble` ; this type corresponds to Fuzz's $\otimes$-lists. The type `SList LInf SDouble` corresponds

to Fuzz's &-lists. Fuzz does not provide the equivalent of `SList L2 SDouble`, which uses the L_2 distance metric.

The distance metrics available in Solo are useful for writing practical differentially private programs. For example, we might want to sum up a list of sensitive numbers drawn from a database. The typical definition of neighboring databases tells us that the distance between two such lists is equal to the number of elements which differ—and those elements may differ by any amount. As a result, their sums may also differ by any amount, and the sensitivity of the computation is unbounded. To address this problem, differentially private programs often *clip* (or "top-code") the input data, which enforces an upper bound on input values and results in bounded sensitivity. We can implement this process in a Solo program:

```
db   :: L1List (SDouble Disc) '[ '( "input_db", 1 ) ]
clip :: L1List (SDouble Disc) senv -> L1List (SDouble Diff) senv
sum  :: L1List (SDouble Diff) senv -> SDouble Diff senv

summationFunction :: L1List (SDouble Disc) senv -> SDouble Diff senv
summationFunction = sum . clip

summationResult :: SDouble Diff '[ '( "input_db", 1 ) ]
summationResult = summationFunction db
```

Here, the `clip` function limits each element of the list to lie between 0 and 1, which allows changing the metric on the underlying `SDouble` from the discrete metric to the absolute difference metric (which is the metric required by the `sum` function). Without the use of `clip` in `summationFunction`, the metrics would not match, and the program would not be well-typed.

Solo's sensitive list types are less powerful than Fuzz's, but serve the same purpose in analyzing programs. In Solo, it is possible to give types to recursive

functions over lists (like `map`), and we do so in Section 20.4. However, it is not possible to *implement* these functions using Solo's types, since the structure of a sensitive list is opaque to programs written using the Solo library. Hence `map` is provided as a trusted primitive with sound typing.

20.4 Function Sensitivity & Higher-Order Functions

In Fuzz, an s-sensitive function is given the type $\tau_1 \multimap_s \tau_2$. Solo does not have sensitive function types, but we have already seen examples of the approach used in Solo to bound function sensitivity: we write function types that are polymorphic over sensitivity environments. In general, we can recover the notion of an s-sensitive function in Solo by writing a Haskell function type that scales the sensitivity environment of its input by a scalar s:

```
-- An s-sensitive function
s_sensitive :: SDouble senv m -> SDouble (ScaleSens senv s) m
```

Here, `ScaleSens` is implemented as a type family that scales the sensitivity environment `senv` by `s`: for each mapping $o \mapsto s_1$ in `senv`, the scaled sensitivity environment contains the mapping $o \mapsto s \cdot s_1$. The common case of a 1-sensitive (or linear) function can be represented by keeping the input's sensitivity environment unchanged (as in `clip` and `sum` in the previous section):

```
-- A 1-sensitive function
one_sensitive :: SDouble senv m -> SDouble senv m
```

20.5 Sensitive Higher-Order Operations

An important goal in the design of Solo is support for sensitivity analysis for higher-order, general-purpose programs. For example, prior systems such as Fuzz and Duet encode the type for the higher-order map function as follows:

$$\text{map} : (\tau_1 \multimap_s \tau_2) \multimap_\infty \text{list } \tau_1 \multimap_s \text{list } \tau_2$$

This map function describes a computation that accepts as inputs: an s-sensitive unary function from values of type τ_1 to values of type τ_2 (map is allowed to apply this function an unlimited number of times), and a list of values of type τ_1. map returns a list of values of type τ_2 which is s-sensitive in the former list. We can give an equivalent type to `map` in Solo as follows, by explicitly scaling the appropriate sensitivity environments using type-level arithmetic:

```
map :: ∀ m s s1 a b.(∀ s'. a s' -> b (s*s')) -> SList m a s1 -> SList m b (s*s1)
```

20.6 Polymorphism for Sensitive Function Types.

Special care is needed for functions that close over sensitive values, especially in the context of higher-order functions like `map`. Consider the following example:

```
dangerousMap :: SDouble m1 s1 -> SList m2 (SDouble m1) s2 -> _
dangerousMap x ls =
  let f y = x
  in map f ls
```

Note that `f` is *not* a function that is s-sensitive with respect to its input—instead, it is s_1-sensitive with respect to the closed-over value of `x`. This use of `map` is dangerous, because it may apply `f` many times, creating duplicate

copies of `x` without accounting for the sensitivity effect of this operation. Fuzz assigns an infinite sensitivity for `x` in this program.

Solo rejects this program as not well-typed. The type of `f` is equivalent to `SDouble m1 s3 -> SDouble m1 s1`, but `map` requires it to have the type `(∀ s'. a s' -> b (s * s'))` —and these two types do not unify. Specifically, the scope of the sensitivity environment `s'` is limited to `f`'s type—but in the situation above, the environment `s1` comes from *outside* of that scope.

This use of parametric polymorphism to limit the ability of higher-order functions to close over sensitive values is key to our ability to support this kind of programming. Without it, we would not be able to give a type for `map` that ensures soundness of the sensitivity analysis. The use of parametric polymorphism to aid in information flow analysis has been previously noted (Bowman and Ahmed, 2015).

Chapter 21

Privacy Analysis

The goal of static privacy analysis is to check that (1) the program adds the correct amount of noise for the sensitivity of underlying computations (i.e. that core mechanisms are used correctly), and (2) the program composes privacy-preserving computations correctly (i.e. the total privacy cost of the program is correct, according to differential privacy's composition properties). A well-typed program should satisfy both conditions. As described earlier, sensitivity analysis often supports privacy analysis, especially in systems based on linear types.

Previous work has taken several approaches to static privacy analysis; we provide a summary in the next section. Solo provides a *privacy monad* that encodes privacy as an effect. As in our sensitivity analysis, the primary difference between Solo and previous work is that our privacy monad tracks privacy cost with respect to data sources, rather than program variables. This distinction allows the implementation of Solo's privacy monad in Haskell, and additionally enables our approach to describe variants of differential privacy without linear group privacy (e.g. (ϵ, δ)-differential privacy).

21.1 Existing Approaches for Privacy Analysis

The Fuzz language pioneered static verification of ϵ-differential privacy, using a linear type system to track sensitivity of data transformations. In this approach, the linear function space can be interpreted as a space of ϵ-differentially private functions by lifting into the probability monad. However, more advanced variants of differential privacy such as (ϵ, δ) differential privacy do not satisfy the restrictions placed on the interpretation of the linear function space in this approach, and Fuzz cannot be easily extended to support these variants. Azevedo de Amorim et al. (de Amorim et al., 2019) provide an extension discussion of this challenge.

More recently, Lobo-Vesga et al. in DPella present an approach in Haskell which tracks sensitivity via data types which are indexed with their *accumulated stability* i.e. sensitivity. Typically in privacy analysis we consider sensitivity to be a property of functions, however as they show, we can also represent sensitivity via the arguments to these functions. Their approach represents private computations via a monad value and monadic operations, similar to the approach in Fuzz. However, in the absence of true linear types, their approach relies on dynamic taint analysis and runtime symbolic execution.

The technique of separating sensitivity composition from privacy composition has been seen before, subsequent to Fuzz, in order to facilitate (ϵ, δ) differential privacy. Azevedo de Amorim et al. (de Amorim et al., 2019) introduce a *path construction* technique which performs a *parameterized comonadic lifting* of a metric space layer à la Fuzz to a separate relational space layer for (ϵ, δ) differential privacy. The Duet system (Near et al., 2019b) uses a dual type system, with dedicated systems for sensitive composition and privacy composition. In principle, this follows a combined effect/co-effect system approach (Petricek, 2017), where one type system tracks the co-effect (in this case sensitivity) and

another tracks the effect which is randomness due to privacy.

Our approach embodies the spirit of DUET and simulates coeffectful program behavior by embedding the co-effect (i.e. the entire sensitivity environment) as an index in comonadic base data types. We then track privacy composition via a special monadic type as an effect. As in DUET, the core privacy mechanisms such as Laplace and Gauss police the boundary between the two. Due to the nature of our co-effect oriented approach in which we track the full sensitivity context, our solution can be embedded in Haskell completely statically, without the need for runtime dynamic symbolic execution. We are also able to verify advanced privacy variants such as (ϵ, δ) and state-of-the-art composition theorems such as advanced composition and the moments accountant via a family of higher-order primitives.

21.1.1 Monads & Effect Systems

Effect systems are known for providing more detailed static type information than possible with monadic typing. They are the topic of a variety of research on enhancing monadic types with program effect information, in order to provider stricter static guarantees. Orchard et al (Orchard et al., 2014), following up on initial work by Wadler and Thiemann (Wadler and Thiemann, 2003), provide a denotational semantics which unify effect systems with a monadic-style semantics as an *parametric effect monad*, establishing an isomorphism between indices of the denotations and the effect annotations of traditional effect systems. They present a formulation of parametric effect monads which generalize monads to include annotation of an effect with a strict monoidal structure. Below typing

rules of the general parametric effect monad are shown:

$$\textsc{Bind}\quad \frac{f : a \to M\ r\ b \qquad g : b \to M\ s\ c}{\lambda x.\ f\ x >>= g : a \to M\ (r \otimes s)\ c} \qquad \textsc{Return}\quad \frac{}{return : a \to M\ \varnothing\ a}$$

These typing rules describe a formulation of parametric effect monads M which accept an effect index as their first argument. This effect index of some arbitrary type E is a monoid $(E, \otimes, \varnothing)$.

21.2 Solo's Privacy Monad

Solo defines *privacy environments* in the same way as sensitivity environments; instead of tracking a sensitivity with respect to each of the program's data sources, however, a privacy environment tracks a *privacy cost* associated with each data source. Privacy environments for pure ϵ-differential privacy are defined as follows:

```
-- Privacy Environments
data EpsPrivacyCost = InfEps | EpsCost TLRat     -- values for ϵ
type EpsPrivEnv = [(Source, EpsPrivacyCost)]     -- privacy environments
```

`TLRat` is a type-level encoding of positive rational numbers by a pair of the numerator and denominator as natural numbers in GCD-reduced form.

The *sequential composition* theorem for differential privacy (Theorem 2.4.1) says that when sequencing ϵ-differentially private computations, we can add up their privacy costs. This theorem provides the basis for the definition of a privacy monad. We observe that our privacy environments have a monoidal structure `(EpsPrivEnv, EpsSeqComp, '[])` , where `EpsSeqComp` is a type family implementing the sequential composition theorem. We derive a privacy monad which is indexed by our privacy environments, in the same style as a notion of effectful monads

or *parametric effect monads* given separately by Orchard (Orchard et al., 2014; Orchard and Petricek, 2014) and Katsumata (Katsumata, 2014). Computations of type `PrivacyMonad` are constructing via these core functions:

```
-- Privacy Monad for ε-differential privacy
return :: a -> EpsPrivacyMonad '[] a
(>>=) :: EpsPrivacyMonad p₁ a -> (a -> EpsPrivacyMonad p₂ b)
  -> EpsPrivacyMonad (EpsSeqComp p₁ p₂) b
```

The `return` operation accepts some value and embeds it in the `PrivacyMonad` without causing any side-effects. The `(>>=)` (`bind`) operation allows us to sequence private computations using differential privacy's sequential composition property, encoded here as the type family `EpsSeqComp`. The implementation of `EpsSeqComp` performs elementwise summation of two privacy environments. In the computation `f>>=g` we execute the private computation `f` for some polymorphic privacy cost p_1, pass its result to the private computation `g`, and output the result of `g` at a total privacy cost of the degradation of the p_1 and p_2 privacy environments combined according to sequential composition. Note that while `PrivacyMonad` is not a regular monad in Haskell (due to the extra index in its type) we may still make use of `do`-notation in our examples by using Haskell's `RebindableSyntax` language extension.

Note that `return` in Solo's privacy monad is very different from the same operator in Fuzz. The typing rule for `return` in Fuzz scales the sensitivities in the context by ∞—reflecting the idea that `return`'s argument is revealed with no added noise, incurring infinite privacy cost. However, this definition of `return` does not satisfy the monad laws; for example, in Fuzz:

$$\texttt{return}\ x \gg\!= \texttt{laplace} \quad \neq \quad \texttt{laplace}\ x$$

The `return` operator in Solo attaches an empty privacy environment to the

returned value, and does satisfy the monad laws. If a sensitive value is given as the argument to `return`, then *it remains sensitive*, rather than being revealed (as in Fuzz)—so there is no need to assign the value an infinite privacy cost. This approach is not feasible in Fuzz because privacy costs are associated with program variables rather than with values. We can recover Fuzz's `return` behavior (revealing a value without noise, and scaling its privacy cost by infinity) using a `reveal` function with the following type:

```
reveal :: SDouble m senv -> EpsPrivacyMonad (ScaleToInfinity senv) Double
```

21.2.1 Core Privacy Mechanisms.

We can define core privacy mechanisms like the Laplace mechanism (described in Part 2), which satisfies ϵ-differential privacy:

```
laplace     :: Proxy ϵ -> SDouble s Diff
  -> EpsPrivacyMonad (TruncateSens ϵ s) Double

listLaplace :: Proxy ϵ -> L1List (SDouble Diff) s
  -> EpsPrivacyMonad (TruncateSens ϵ s) [Double]
```

The first argument to `laplace` is the privacy parameter ϵ (as a type-level natural). The second argument is the value we would like to add noise to; it must be a sensitive number with the `Diff` metric. The function's result is a regular Haskell `Double`, in the privacy monad. The `TruncateSens` type family transforms a sensitivity environment into a privacy environment by replacing each sensitivity with the privacy parameter ϵ. The function's implementation follows the definition of the Laplace mechanism; it determines the scale of the noise to add using the maximum sensitivity in the sensitivity environment `s` and the privacy parameter ϵ.

The `listLaplace` function implements the *vector-valued Laplace mechanism*, which adds noise to each element of a vector based on the vector's L_1 sensitivity. Its argument is required to be a `L1List` of sensitive doubles with the `Diff` metric, and its output is a list of Haskell doubles in the privacy monad.

As a simple example, the following function adds noise to its input twice, once with $\epsilon = 2$ and once with $\epsilon = 3$, for a total privacy cost of $\epsilon = 5$. If the type annotation is left off, Haskell infers this type.

```
addNoiseTwice :: TL.KnownNat (MaxSens s)
  => SDouble s Diff
  -> EpsPrivacyMonad (Plus (TruncateSens 2 s) (TruncateSens 3 s)) Double
addNoiseTwice x = do
  y₁ <- laplace @2 Proxy x
  y₂ <- laplace @3 Proxy x
  return $ y₁ + y₂
```

21.2.2 (ϵ, δ)-Differential Privacy & Advanced Composition

The *advanced composition theorem* for differential privacy (Dwork et al., 2014b) provides tighter bounds on the privacy cost of iterative algorithms, but requires the use of (ϵ, δ)-differential privacy.

Theorem 21.2.1 (Advanced composition)**.** *For $0 < \epsilon' < 1$ and $\delta' > 0$, the class of (ϵ, δ)-differentially private mechanisms satisfies $(\epsilon', k\delta + \delta')$-differential privacy under k-fold adaptive composition for:*

$$\epsilon' = 2\epsilon\sqrt{2k\ln(1/\delta')}$$

To support advanced composition in Solo, we first define privacy environments and a privacy monad for (ϵ, δ)-differential privacy as follows:

```
-- Privacy Environments & Monad for (ε, δ)-differential privacy
data EDPrivacyCost = InfED | EDCost TLReal TLReal
type EDEnv = [(TL.Symbol, EDPrivacyCost)]
return :: a -> EDPrivacyMonad '[] a
(>>=) :: EDPrivacyMonad p₁ a -> (a -> EDPrivacyMonad p₂ b)
  -> EDPrivacyMonad (EDSeqComp p₁ p₂) b
```

where `EDSeqComp` is a type family that implements sequential composition for (ϵ, δ)-differential privacy (Theorem 2.4.1) via elementwise summation of both ϵ and δ values. Rational numbers were sufficient to represent privacy costs in pure ϵ-differential privacy, but we use a type-level representation of real numbers (`TLReal`) for (ϵ, δ)-differential privacy. For advanced composition, we will need operations like square root and natural logarithm. Haskell avoids supporting doubles at the type level, because equality for doubles does not interact well with the notion of equality required for typing. We therefore implement `TLReal` by building type-level expressions that represent real-valued computations, and interpret those expressions using Haskell's standard double type at the value level.

We can now write the type of a looping combinator primitive that leverages advanced composition:

```
advloop :: NatS k -> a -> (a -> EDPrivacyMonad p a)
  -> EDPrivacyMonad (AdvComp k δ′ p) a
```

The looping combinator `advloop` is designed to run an (ϵ, δ)-differentially private mechanism k times, and satisfies $(2\epsilon\sqrt{2k\ln(1/\delta')}, \delta' + k\delta)$-differential privacy—which is significantly lower than the standard composition theorem when k is large. The first argument `k` is the statically known number of iterations. The type family `AdvComp` is a helper to statically compute the appropriate total privacy cost given the privacy parameters of the private function passed as the

penultimate parameter to the primitive which satisfies ϵ, δ-differential privacy. `AdvComp` builds a type-level expression containing square roots and logarithms, as described earlier.

The Gaussian Mechanism. The Gaussian mechanism (described in Part 2) adds Gaussian noise instead of Laplace noise, and ensures (ϵ, δ)-differential privacy (with $\delta > 0$). The primary advantage of the Gaussian mechanism is in the vector setting: the Gaussian mechanism uses L_2 sensitivity, which is typically much lower than the L_1 sensitivity used by the Laplace mechanism. This requirement is reflected in the type of the Gaussian mechanism in Solo:

```
gauss     :: Proxy ϵ -> Proxy δ -> SDouble s Diff
                        -> EDPrivacyMonad (TruncateSensED ϵ δ s) Double
listGauss :: Proxy ϵ -> Proxy δ -> L2List (SDouble Diff) s
                        -> EDPrivacyMonad (TruncateSensED ϵ δ s) [Double]
```

21.2.3 Additional Variants & Converting Between Variants

Solo provides a type class of privacy monads instantiated for each supported variant of differential privacy. For each privacy variant, the corresponding privacy monad is indexed with a privacy environment that tracks the appropriate privacy parameters, and the bind operation enforces the appropriate form of sequential composition. Conversion operations are provided between variants to enable variant-mixing in programs. For example, the following function converts an ϵ-differentially private computation into an (ϵ, δ)-differentially private one, setting $\delta = 0$.

```
-- variant conversion function
conv_eps_to_ed :: EpsPrivacyMonad p1 a -> EDPrivacyMonad (ConvEpstoED p1) a
```

```
-- interactive conversion example
x = conv_eps_to_ed applied_3
:t x
-- x :: EDPrivacyMonad '[ '("sensitive_input",5,1)] Double
```

Solo currently supports ϵ-differential privacy, (ϵ, δ)-differential privacy, and Rényi differential privacy (RDP) (Mironov, 2017b). Conversions are possible from ϵ-DP to (ϵ, δ)-DP and RDP, and from RDP to (ϵ, δ)-DP. Conversions are not possible from (ϵ, δ)-DP to ϵ-DP or RDP.

Chapter 22

Formalism

In Solo, we implement a novel static analysis for function sensitivity and differential privacy. Our approach can be seen as a type-and-effect system, which may be embedded in statically typed functional languages with support for monads and type-level arithmetic.

22.1 Program Syntax.

Figure 22.1 shows a core subset of the syntax for our analysis system. Our language model includes arithmetic operations ($e \odot e$), pairs ($\langle e, e \rangle$ and $\pi_i(e)$), conditionals ($\texttt{if0}(e)\{e\}\{e\}$), and functions ($\lambda_x x.\ e$ and $e(e)$). Types τ presented in the formalism include: base numeric types `real`, singleton numeric types with a known runtime value at compile-time $\texttt{real}[r]$, booleans `bool`, functions $\tau \to \tau$, pairs $\tau \times \tau$, and the privacy monad $\bigcirc_\Sigma(\tau)$. Regular types τ are accompanied by sensitive types σ which are essentially regular types annotated with static sensitivity analysis information Σ—which is the sensitivity analysis (or sensitivity environment) for the expression which was typed as τ. Senstitive types shown in our formalism include sensitive numeric types `sreal`, sensititve pairs $\sigma \otimes \sigma$, and

sensitive lists $\texttt{slist}(\sigma)$. A metric-carrying singleton numeric type is unnecessary since its value is fixed and cannot vary. Σ—the *sensitivity/privacy environment*—is defined as a mapping from sensitive sources $o \in \text{source}$ to scalar values which represent the sensitivity/privacy of the resulting value with respect to that source.

Types/values with standard treatment are not shown in our formalism, but included in our implementation with both regular and metric-carrying versions, include vectors and matrices which have known dimensions at compile-time via singleton natural number indices. Single natural numbers are also used to execute loops with statically known number of iterations and to help contruct sensitivity and privacy quantities.

Figure 22.5 shows a core subset of the standard dynamic semantics that accompanies the syntax for our analysis system.

22.2 Typing Rules.

Figure 22.3 shows typing rules in our system used to reason about the sensitivity of computations. Sensitivity environment composition $\Sigma_1 \sqcup \Sigma_2$, and sensitivity environment scaling $\texttt{s}(\Sigma)$ are defined as seen in prior work (Reed and Pierce, 2010; Near et al., 2019b).

22.2.1 Type Soundness.

The property of type soundness in our system is defined (as in prior work) as the *metric preservation* theorem. Essentially, metric preservation dictates a maximum variation which is possible when a sensitive open term is closed over by two distinct but related sensitive closure environments. This means that given related initial well-typed configurations, we expect the outputs to be related by some level of variation. Specifically: given two well-typed environments which are

$b \in \mathbb{B}$ $\quad r \in \mathbb{R}$ $\quad \dot{r} \in \dot{\mathbb{R}} ::= r \mid \infty$ $\quad x, z \in \text{var}$ $\quad o \in \text{source}$

$\Sigma \in \text{spenv}$	$\triangleq$	$\text{source} \rightharpoonup \dot{\mathbb{R}}$	sensitivity/privacy environment
$\tau \in \text{type}$	$::=$	$\texttt{bool} \mid \texttt{real} \mid \texttt{real}[r]$	base and singleton types
	$\mid$	$\tau \times \tau \mid \texttt{list}(\tau) \mid \tau \to \tau$	connectives
	$\mid$	$\bigcirc_\Sigma(\tau) \mid \sigma@\Sigma$	privacy monad and sensitive types
$\sigma \in \text{stype}$	$::=$	$\texttt{sreal} \mid \sigma \otimes \sigma \mid \texttt{slist}(\sigma)$	sensitive types
$\odot \in \text{binop}$	$::=$	$+ \mid \times \mid \rtimes$	operations
$e \in \text{expr}$	$::=$	$x \mid b \mid r \mid \texttt{sing}(r)$	variables and literals
	$\mid$	$e \odot e \mid \texttt{if}(e)\{e\}\{e\}$	binary operations and conditionals
	$\mid$	$\langle e, e \rangle \mid \pi_i(e)$	pair creation and access
	$\mid$	$[] \mid e :: e$	list creation
	$\mid$	$\texttt{case}(e)\{[].e\}\{x :: x.e\}$	list destruction
	$\mid$	$\lambda_x x.\ e \mid e(e)$	recursive functions
	$\mid$	$\texttt{reveal}(e) \mid \texttt{laplace}[e, e](e)$	privacy operations
	$\mid$	$\texttt{return}(e) \mid x \leftarrow e\ ;\ e$	privacy monad
	$\mid$	$\hat{\langle} e, e \hat{\rangle} \mid \hat{\pi}_i(e)$	sensitive pair creation and access
	$\mid$	$\hat{[]} \mid e \mathbin{\hat{::}} e$	sensitive list creation
	$\mid$	$\texttt{case}(e)\{\hat{[]}.e\}\{x \mathbin{\hat{::}} x.e\}$	sensitive list destruction
$\gamma \in \text{venv}$	$\triangleq$	$\text{var} \rightharpoonup \text{value}$	evaluation environment
$\rho \in \text{ddist}$	$\triangleq$	$\left\{ f \in \text{value} \to \mathbb{R} \mid \sum_v f(v) = 1 \right\}$	discrete distributions (PMF)
$v \in \text{value}$	$::=$	$b \mid r$	literals
	$\mid$	$\langle v, v \rangle$	pairs
	$\mid$	$[] \mid v :: v$	lists
	$\mid$	$\langle \lambda_x x.\ e \mid \gamma \rangle$	recursive closures
	$\mid$	ρ	distributions of values

Figure 22.1: Syntax for types, expressions and values. = sensitivity sources, types and expressions unique to Solo.

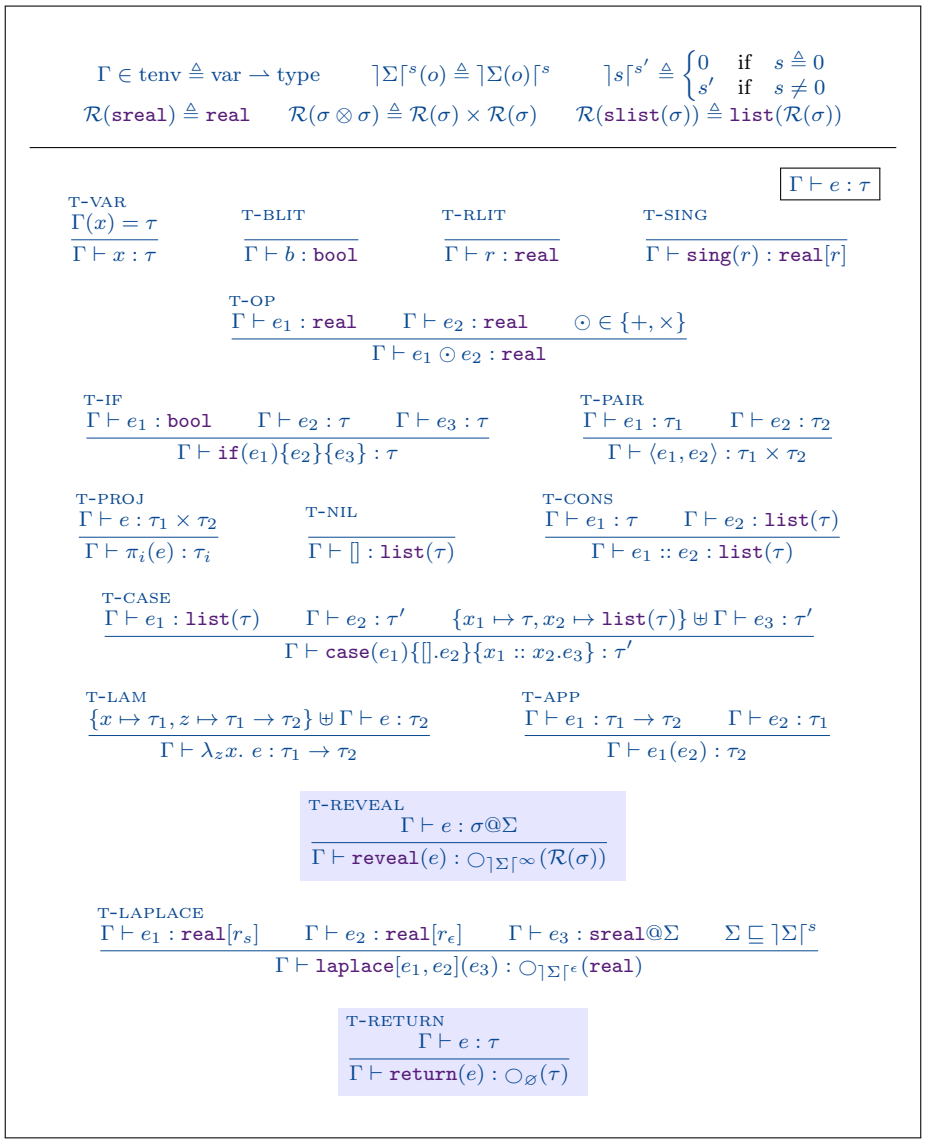

Figure 22.2: The type system. ■ = type rules unique to Solo.

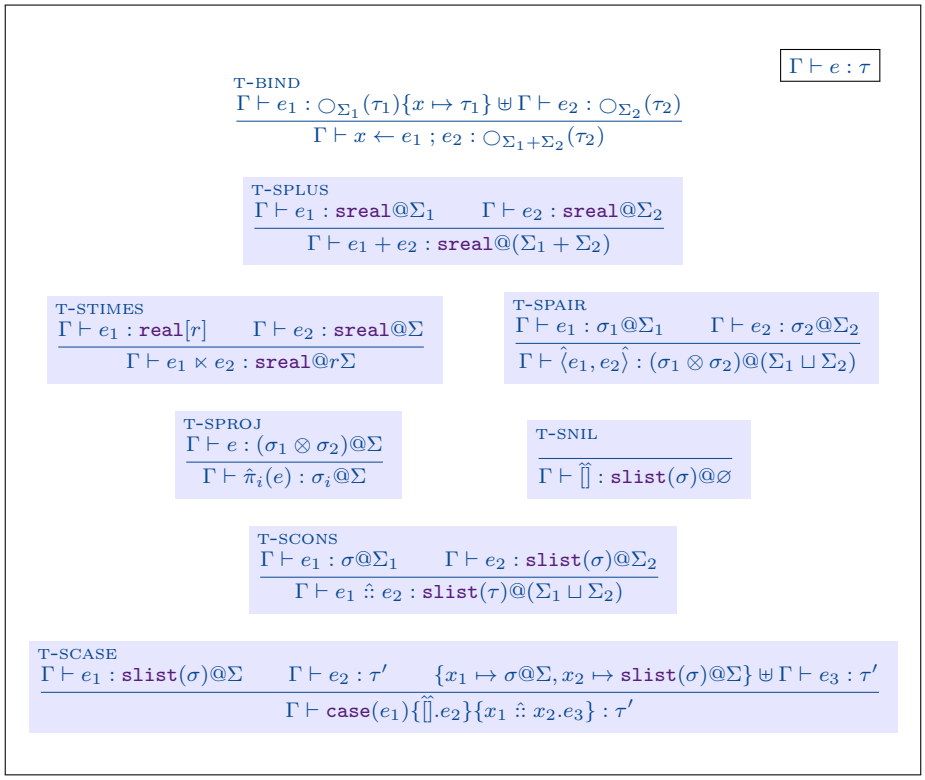

Figure 22.3: The type system. ■ = type rules unique to Solo.

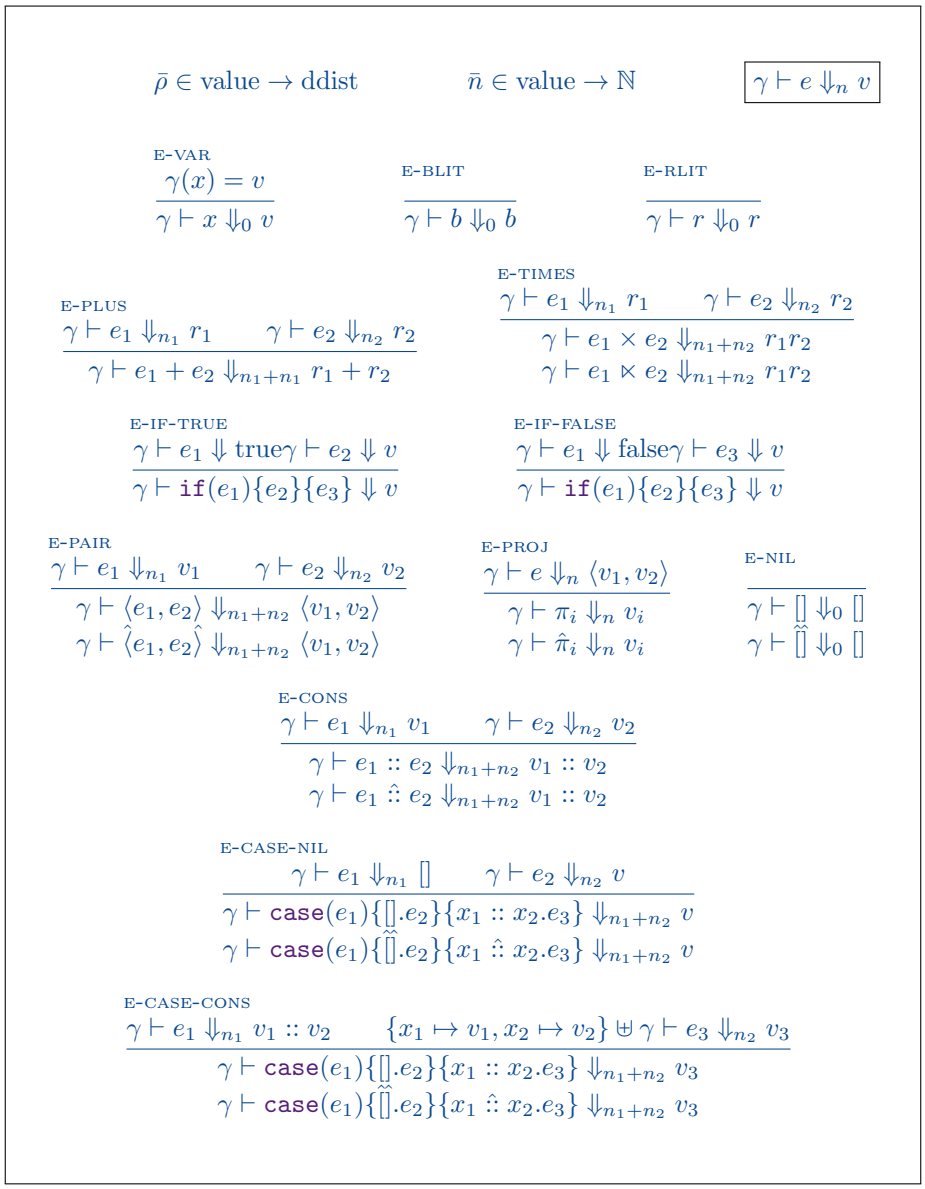

Figure 22.4: Step-indexed big-step evaluation semantics.

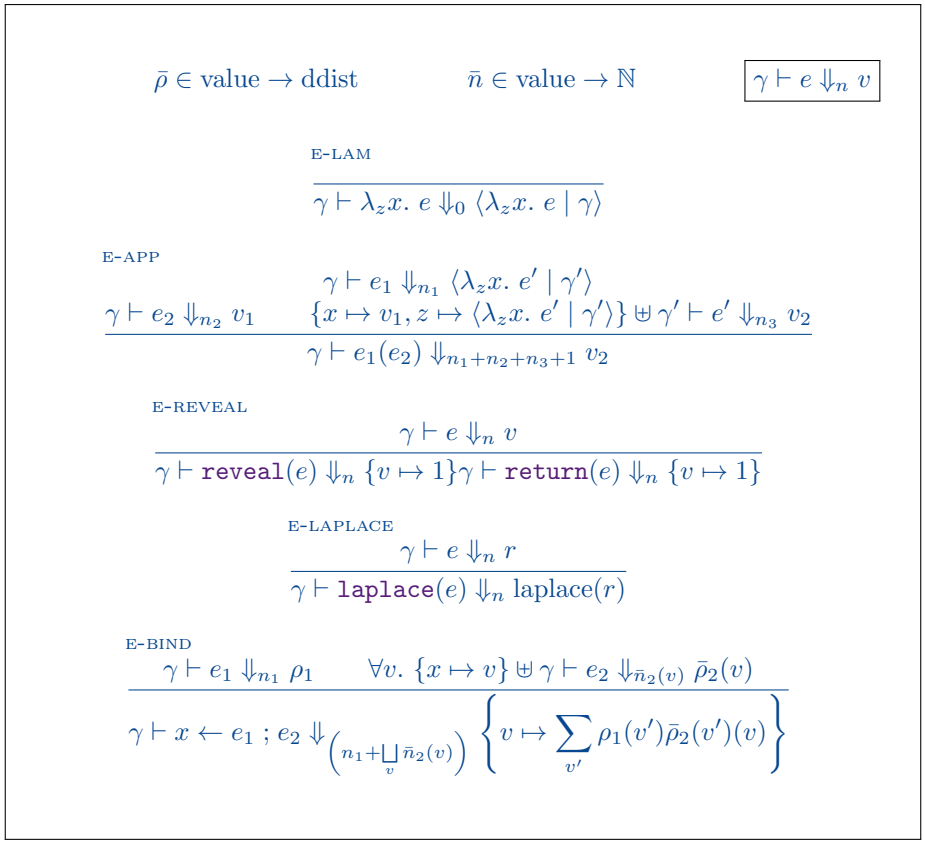

Figure 22.5: Step-indexed big-step evaluation semantics.

related by the logical relation (values may be apart by distance Σ, for n steps), and a well typed term, then each evaluation of that term in each environment is related by the relation, that is, when one side terminates in $< n$ steps to a value, the other side will deterministically terminate to a related value.

Theorem 22.2.1 (Metric Preservation)**.**

$$\begin{array}{rll} \textit{If:} & \gamma_1 \sim \gamma_2 \in \mathcal{G}_n^{\Sigma}[\![\Gamma]\!] & \textit{(H1)} \\ \textit{And:} & \Gamma \vdash e : \tau & \textit{(H2)} \\ \textit{Then:} & \gamma_1, e \sim \gamma_2, e \in \mathcal{E}_n^{\Sigma}[\![\tau]\!] & \end{array}$$

That is, either $n = 0$*, or* $n = n' + 1$ *and...*

$$\begin{array}{rll} \textit{If:} & n'' \leq n & \textit{(H3)} \\ \textit{And:} & \gamma_1 \vdash e \Downarrow_{n''} v_1 & \textit{(H4)} \\ \textit{Then:} & \exists! v_2.\ \gamma_2 \vdash e \Downarrow_{n''} v_2 & \textit{(C1)} \\ \textit{And:} & v_1 \sim v_2 \in \mathcal{V}_{n-n''}^{\Sigma}[\![\tau]\!] & \textit{(C2)} \end{array}$$

Similar to prior work, in order to state and prove the metric preservation theorem, we define the notion of function sensitivity as a (step-indexed) logical relation. Figure 22.6 shows the step-indexed logical relation used to define function sensitivity. We briefly describe the logical relations seen in this figure below:

1. Two real numbers are related $r_1 \sim^r r_2$ at type $\mathbb{R}$ and distance r when the absolute difference between real numbers r_1 and r_2 is less than r.

2. Two values are related $v_1 \sim v_2$ in $\mathcal{V}_{\Sigma}[\![\tau]\!]$ when v_1 and v_2 are related at type τ for initial distance Σ. We may define relatedness for the syntactic category of values via case analysis as follows:

(a) Base numeric values are related $r_1 \sim^{\Sigma} r_2$ at type $\mathbb{R}$ in $\mathcal{V}_{\Sigma_1}[\![\tau]\!]$ when r_1 and r_2 are related by $\Sigma \cdot \Sigma_1$, where Σ is the initial distances between each input source o, and Σ_1 describes how much these values may wiggle as function arguments i.e. the maximum permitted argument variation. $\cdot$ is defined as the vector dot product.

(b) Function values $\langle \lambda x.\ e_1 \mid \gamma_1 \rangle \sim \langle \lambda x.\ e_2 \mid \gamma_2 \rangle$ are related at type $(\tau \to \tau)$ in $\mathcal{V}_{\Sigma}[\![\tau]\!]$ when given related inputs, they produce related computations.

(c) Pair values $\langle v_{11}, v_{12} \rangle \sim \langle v_{21}, v_{22} \rangle$ are related at type $\langle \tau, \tau \rangle$ in $\mathcal{V}_{\Sigma}[\![\tau]\!]$ when they are elementwise related.

(d) $\gamma_1, e_1 \sim \gamma_2, e_2$ are related at type τ and distance Σ in $\mathcal{E}_{\Sigma}[\![\tau]\!]$ when the input doubles γ_1, e_1 and γ_2, e_2 evaluate to output values which are related by Σ.

3. Two value environments $\gamma_1 \sim \gamma_2$ are related at type environment Γ and sensitivity environment Σ in $\mathcal{G}_{\Sigma}[\![\Gamma]\!]$ if value environments γ_1 and γ_2 both map each variable in the type environment Γ to related values at a matching type at distance Σ.

The proofs appear in Chapter 24.

$$\gamma_1, e_1 \sim \gamma_2, e_2 \in \mathcal{E}^\Sigma_n[\![\tau]\!] \overset{\Delta}{\iff} n = 0 \implies \text{true} \qquad \boxed{\gamma, e \sim \gamma, e \in \mathcal{E}^\Sigma_n[\![\tau]\!]}$$
$$\wedge\, n = n' + 1 \implies \forall n'' \le n', v_1.\ \gamma_1 \vdash e_1 \Downarrow_{n''} v_1$$
$$\Rightarrow \exists! v_2.\ \gamma_2 \vdash e_2 \Downarrow_{n''} v_2 \ \wedge\ v_1 \sim v_2 \in \mathcal{V}^\Sigma_{n'-n''}[\![\tau]\!]$$

$$r_1 \sim^r r_2 \overset{\Delta}{\iff} |r_1 - r_2| \le r \qquad \boxed{r \sim^r r}$$

$$\begin{array}{rll}
b_1 \sim b_2 \in \mathcal{V}^\Sigma_n[\![\texttt{bool}]\!] & \overset{\Delta}{\iff} & b_1 = b_2 \qquad \boxed{v \sim v \in \mathcal{V}^\Sigma_n[\![\tau]\!]} \\
r_1 \sim r_2 \in \mathcal{V}^\Sigma_n[\![\texttt{real}]\!] & \overset{\Delta}{\iff} & r_1 = r_2 \\
r_1 \sim r_2 \in \mathcal{V}^\Sigma_n[\![\texttt{real}[r]]\!] & \overset{\Delta}{\iff} & r_1 = r_2 = r \\
r_1 \sim r_2 \in \mathcal{V}^\Sigma_n[\![\texttt{sreal}@\Sigma']\!] & \overset{\Delta}{\iff} & r_1 \sim^{\Sigma\cdot\Sigma'} r_2 \\
\langle v_{11}, v_{12}\rangle \sim \langle v_{21}, v_{22}\rangle \in \mathcal{V}^\Sigma_n[\![\tau_1 \times \tau_2]\!] & \overset{\Delta}{\iff} & v_{11} \sim v_{21} \in \mathcal{V}^\Sigma_n[\![\tau_1]\!] \\
 & & \wedge\, v_{12} \sim v_{22} \in \mathcal{V}^\Sigma_n[\![\tau_2]\!] \\
\langle v_{11}, v_{12}\rangle \sim \langle v_{21}, v_{22}\rangle \in \mathcal{V}^\Sigma_n[\![(\sigma_1 \otimes \sigma_2)@\Sigma']\!] & \overset{\Delta}{\iff} & v_{11} \sim v_{21} \in \mathcal{V}^\Sigma_n[\![\sigma_1@\Sigma']\!] \\
 & & \wedge\, v_{12} \sim v_{22} \in \mathcal{V}^\Sigma_n[\![\sigma_2@\Sigma']\!] \\
v_{11} :: v_{12} \sim v_{21} :: v_{22} \in \mathcal{V}^\Sigma_n[\![\texttt{list}(\tau)]\!] & \overset{\Delta}{\iff} & v_{11} \sim v_{21} \in \mathcal{V}^\Sigma_n[\![\tau]\!] \\
 & & \wedge\, v_{12} \sim v_{22} \in \mathcal{V}^\Sigma_n[\![\texttt{list}(\tau)]\!] \\
v_{11} \hat{::} v_{12} \sim v_{21} \hat{::} v_{22} \in \mathcal{V}^\Sigma_n[\![\texttt{slist}(\sigma)@\Sigma']\!] & \overset{\Delta}{\iff} & v_{11} \sim v_{21} \in \mathcal{V}^\Sigma_n[\![\sigma@\Sigma']\!] \\
 & & \wedge\, v_{12} \sim v_{22} \in \mathcal{V}^\Sigma_n[\![\texttt{slist}(\sigma)@\Sigma']\!] \\
\langle \lambda_z x.e_1 | \gamma_1\rangle \sim \langle \lambda_z x.e_2 | \gamma_2\rangle \in \mathcal{V}^\Sigma_n[\![\tau_1 \to \tau_2]\!] & \overset{\Delta}{\iff} & \forall n' \le n, v_1, v_2.\ v_1 \sim v_2 \in \mathcal{V}^\Sigma_{n'}[\![\tau_1]\!] \\
 & & \Rightarrow \{x \mapsto v_1, z \mapsto \langle \lambda_z x.e_1 \mid \gamma_1\rangle\} \uplus \gamma_1, e_1 \\
 & & \sim \{x \mapsto v_2, z \mapsto \langle \lambda_z x.e_2 \mid \gamma_2\rangle\} \uplus \gamma_2, e_2 \\
 & & \in \mathcal{E}^\Sigma_{n'}[\![\tau_2]\!] \\
\rho_1 \sim \rho_2 \in \mathcal{V}^\Sigma_n[\![\bigcirc_{\Sigma'}(\tau)]\!] & \overset{\Delta}{\iff} & \forall v.\ \rho_1(v) \le e^{|\lceil \Sigma \rceil^1 \times \Sigma'|_L \infty} \rho_2(v)
\end{array}$$

$$\gamma_1 \sim \gamma_2 \in \mathcal{G}^\Sigma_n[\![\Gamma]\!] \overset{\Delta}{\iff} \forall x \in \mathrm{dom}(\gamma_1 \cup \gamma_2).\ \gamma_1(x) \sim \gamma_2(x) \in \mathcal{V}^\Sigma_n[\![\tau]\!] \qquad \boxed{\gamma \sim \gamma \in \mathcal{G}^\Sigma_n[\![\Gamma]\!]}$$

Figure 22.6: Step-indexed Logical Relation.

Chapter 23

Implementation & Case Studies

Additional tools. For our case studies, we introduce sensitive matrices `SMatrix σ m r c a`, sensitive key-value mappings (dictionaries) `SDict σ m a b`, and sensitive sets `SSet σ a`, as well as sound primitive operations over these values. `r c` are matrix dimensions, `a b` are type parameters representing the contents of the compound types. σ represents the sensitivity environments as usual, and `m` represents the distance metric. We provide the usual primitives over these types seen in prior work (Reed and Pierce, 2010; Near et al., 2019b). Sets are assumed to use the Hamming metric, while matrices and key-value maps use the standard compound metrics discussed earlier: `L1 | L2 | LInf`. Recall that `NatS` is a type for singleton naturals and `natS @ 5` creates a singleton for the value `5`.

23.1 k-means clustering

We present a case study based on the privacy-preserving implementation of the k-means clustering algorithm seen originally in Blum et al, as well as in the presentation of the Fuzz language. The goal of the k-means clustering algorithm is to iteratively find a set of k clusters to which n datapoints can be partitioned, where each datapoint belongs to the cluster with the nearest *center* or *centroid* to it.

The algorithm operates by beginning from an initial guess at the list of cluster centroids which it iteratively improves on. A single iteration consists of grouping each datapoint with the centroid it is closest to, then recalculating the mean of each group to initialize the next round's list of centroids.

The `assign` function is responsible for pairing each initial datapoint with the index of the centroid it is closest to in the initial centroid list. The `partition` function then groups the set of datapoints into a list of sets, where each set represents a cluster. The rest of the algorithm proceeds to compute the private new center of each cluster. Given that our datapoints are two-dimensional, `totx` and `toty` sum the `x` and `y` coordinates of each cluster of datapoint. After we compute the size of each cluster, the `avg` function calculates the new mean of each cluster with the three-element tuple of coordinate and size data zipped together for each cluster.

The Haskell typechecker can infer the privacy cost of one iteration of the k-means algorithm as 3ϵ.

23.2 Cumulative Distribution Function

Our next case study implements the private `cdf` function as seen in DFuzz (McSherry and Mahajan, 2010; Gaboardi et al., 2013b). Given a database of

```
type Pt = (Double, Double)
-- helpers
assign :: [Pt] -> SSet σ Pt -> SSet σ (Pt,Integer)
ppartition :: SSet σ (Pt,Integer) -> SList σ m SSet (Set Pt)
totx :: SSet β Pt -> SDouble β 'AbsoluteM
toty :: SSet β Pt -> SDouble β 'AbsoluteM
size :: SSet β Pt -> SDouble β 'AbsoluteM
avg :: ((Double, Double), Double) ->  (Double, Double)

-- kmeans: 3ε-private
iterate :: ∀ m σ. (TL.KnownNat (MaxSens σ)) => SSet σ Pt -> [Pt] -> _
iterate b ms = do
  let b' = ppartition (assign ms b)
  tx <- vector_laplace @1 Proxy $ map₀ totx b'
  ty <- vector_laplace @1 Proxy $ map₀ toty b'
  t <- vector_laplace @1 Proxy $ map₀ size b'
  let stats = zip (zip tx ty) t
  return $ (map avg stats)
```

Figure 23.1: kmeans

numeric records, and a set of buckets associated with cutoff values, the `cdf` function privately partitions each record to its respective bucket. As in DFuzz, this case study demonstrates the ability of Solo to verify privacy costs which depend on a program input, in this case the symbolic number of buckets `m`. However, our approach to achieve this feature relies on singleton types in Haskell, and does not require a *true* dependent type system.

23.3 Gradient Descent

We now present a case study (Figure 23.3) based on a simple machine learning algorithm (Bassily et al., 2014b) which performs gradient descent.

As inputs, the `gd` algorithm accepts a list of feature vectors `xs` representing sensitive user data, a set of corresponding classifier labels `ys`, a number of iterations to run `k` and the desired privacy cost per iteration ϵ. Gradient descent also requires a loss function which describes the accuracy of the current model in predicting the correct classification of user examples. The algorithm

```
cdf :: ∀ m o s ϵ. (TL.KnownNat m,TL.KnownNat ϵ) =>
     NatS m
  -> NatS ϵ
  -> Matrix m 1 Double -- buckets
  -> SSet σ Double -- db
  -> EpsPrivacyMonad (ScalePriv m (TruncateSens ϵ σ)) [Double]
cdf m t buckets db = do
  let f :: Double -> SSet σ Double
          -> _
      f = \x -> \db1 ->
        let (lt,gt) = bag_split (\k -> k < x) db1 in
          (laplace @ϵ Proxy (natS @5) $ (bag_size lt), db)
      z = mloop₁ m buckets db f $ return []
  z
```

Figure 23.2: CDF

works by moving the current model in the opposite direction of the gradient of the loss function. In order to preserve privacy for this algorithm, we may introduce noise at the point where user data is exposed: the gradient calculation. The let-bound function `f` in the `gd` algorithm contains the workload of a single iteration of the program: in which we perform the gradient calculation and introduce noise using the vector-based Laplacian mechanism.

23.4 Multiplicative-Weights Exponential Mechanism

Our final case study, the MWEM algorithm (Hardt et al., 2012), builds a differentially private synthetic dataset which approximates some sensitive real dataset with some level of accuracy. The algorithm combines usage of the Exponential Mechanism, Laplacian noise, and the multiplicative-weights update rule to construct a noisy synthetic dataset over several iterations with competitive privacy leakage bounds via composition.

```
-- sequential composition privacy loop over a matrix
mloop :: NatS k
  -> SMatrix σ LInf 1 n SDouble
  -> (SMatrix σ LInf 1 n SDouble ->
      EpsPrivacyMonad (TruncateSens ϵ σ) (Matrix 1 n Double))
  -> EpsPrivacyMonad (ScalePriv k (TruncateSens ϵ σ)) (Matrix 1 n Double)

-- gradient descent algorithm
gd :: NatS k
  -> NatS ϵ
  -> SMatrix σ LInf m n SDouble
  -> SMatrix σ LInf m 1 SDouble
  -> EpsPrivacyMonad (ScalePriv k (TruncateSens ϵ σ)) (Matrix 1 n Double)
gd k t xs ys = do
  let m0 = matrix (sn32 @ 1) (sn32 @ n) $ \ i j -> 0
      cxs = mclip xs (natS @ 1)
  let f :: SMatrix σ1 LInf 1 n SDouble
    -> EpsPrivacyMonad (TruncateSens ϵ σ1) (Matrix 1 n Double)
      f = \θ -> let g = mlaplace @ϵ Proxy (natS @5) $ xgradient θ cxs ys
      in msubM (return θ) g
      z = mloop @(TruncateNat t 1) k (sourceM $ xbp m0) f
  z
```

Figure 23.3: Gradient Descent

`mwem` (Figure 23.4) takes the following inputs: a number of iterations `k`, a privacy cost ϵ to be used by the exponential mechanism and Laplace, `real_data` the sensitive information dataset, a query workload `queries` over the sensitive dataset, and lastly `syn_data` which represents a uniform or random distribution over the domain of the real dataset.

Each iteration, the `mwem` algorithm selects a query from the query workload privately using the exponential mechanism. The query selected is selected by virtue of a scoring function which determines that the result of the query on the synthetic dataset greatly differs from its result on the real dataset (more so than other queries in the workload, with some amount of error). The algorithm updates the synthetic dataset using the multiplicative weights update rule, based on the query result on the real dataset with some noise added. This process continues over several iterations until the synthetic dataset reaches some some level of accuracy relative to the real dataset.

```
-- exponential mechanism
expmech :: [(Double,Double)]
  -> NatS ϵ
  -> SDict σ LInf SDouble SDouble
  -> EpsPrivacyMonad (TruncateSens ϵ σ) Int

-- exponential mechanism + laplace loop
expnloop :: NatS k
  -> NatS ϵ
  -> [(Double,Double)]
  -> SDict σ LInf SDouble SDouble
  -> Map.Map Double Double
  -> EpsPrivacyMonad (ScalePriv (2 TL.* k) (TruncateSens ϵ σ)) (Map.Map Double Double)

-- multiplicative-weights exponential mechanism
mwem :: NatS k
  -> NatS ϵ
  -> [(Double,Double)]
  -> SDict σ LInf SDouble SDouble
  -> Map Double Double
  -> EpsPrivacyMonad (ScalePriv (2 TL.* k) (TruncateSens ϵ σ)) (Map Double Double)
mwem k ϵ queries real_data syn_data =
  expnloop k ϵ queries real_data syn_data
```

Figure 23.4: Multiplicative Weights Exponential Mechanism.

Chapter 24

Lemmas, Theorems & Proofs

Lemma 24.0.1 (Plus Respects)**.**

If $r_1 \sim^r r_2$ *then* $r_1 + r_3 \sim^r r_2 + r_3$.

Proof.

By $|r_1 - r_2| \leq r \implies |(r_1 + r_3) - (r_2 + r_3)| \leq r$. □

Lemma 24.0.2 (Times Respects)**.**

If $r_1 \sim^r r_2$ *then* $r_3 r_1 \sim^{r_3 r} r_3 r_2$.

Proof.

By $|r_1 - r_2| \leq r \implies |r_3 r_1 - r_3 r_2| \leq r$. □

Lemma 24.0.3 (Triangle)**.**

If $r_1 \sim^{r_A} r_2$ *and* $r_2 \sim^{r_B} r_3$ *then* $r_1 \sim^{r_A + r_B} r_3$.

Proof.

By the classic triangle inequality lemma for real numbers. □

Lemma 24.0.4 (Step-index Weakening)**.**

For $n' \leq n$*: (1) If* $\gamma_1 \sim \gamma_2 \in \mathcal{G}^{\Sigma}_{n}[\![\Gamma]\!]$ *then* $\gamma_1 \sim \gamma_2 \in \mathcal{G}^{\Sigma}_{n'}[\![\Gamma]\!]$*; and (2) If*

$v_1 \sim v_2 \in \mathcal{V}_n^\Sigma[\![\tau]\!]$ *then* $v_1 \sim v_2 \in \mathcal{V}_{n'}^\Sigma[\![\tau]\!]$*; and (3) If* $\gamma_1, e_1 \sim \gamma_2, e_2 \in \mathcal{E}_n^\Sigma[\![\tau]\!]$ *then* $\gamma_1, e_1 \sim \gamma_2, e_2 \in \mathcal{E}_n^\Sigma[\![\tau]\!]$.

Proof.

By induction on n mutually for all properties; case analysis on v_1 and v_2 for property (2), and case analysis on e_1 and e_2 for property (3). □

Theorem 24.0.5 (Metric Preservation)**.**

If:	$\gamma_1 \sim \gamma_2 \in \mathcal{G}_n^\Sigma[\![\Gamma]\!]$	*(H1)*
And:	$\Gamma \vdash e : \tau$	*(H2)*
Then:	$\gamma_1, e \sim \gamma_2, e \in \mathcal{E}_n^\Sigma[\![\tau]\!]$	

That is, either $n = 0$*, or* $n = n' + 1$ *and...*

If:	$n'' \leq n$	*(H3)*
And:	$\gamma_1 \vdash e \Downarrow_{n''} v_1$	*(H4)*
Then:	$\exists! v_2.\ \gamma_2 \vdash e \Downarrow_{n''} v_2$	*(C1)*
And:	$v_1 \sim v_2 \in \mathcal{V}_{n-n''}^\Sigma[\![\tau]\!]$	*(C2)*

Proof.

By strong induction on n and case analysis on e and τ:

- **Case** $n = 0$: Trivial by definition.

- **Case** $n = n' + 1$ **and** $e = x$:

  By inversion on *(H4)* we have: $n' = 0$ and $v_1 = \gamma_1(x)$. Instantiate $v_2 = \gamma_2(x)$ in the conclusion. To show: *(C1)*: $\gamma_2 \vdash x \Downarrow_0 \gamma_2(x)$ unique; and *(C2)*: $\gamma_1(x) \sim \gamma_2(x) \in \mathcal{V}_{n'}^\Sigma[\![\tau]\!]$. *(C1)* is by E-VAR application and inversion. *(C2)* is by *(H1)* and Step-index Weakening.

- **Case** $n = n' + 1$ **and** $e = r$ **and** $\tau = \texttt{real}$:

  By inversion on *(H4)* we have: $n' = 0$ and $v_1 = r$. Instantiate $v_2 = r$ in the conclusion. To show: *(C1)*: $\gamma_2 \vdash r \Downarrow_0 r$ unique; and *(C2)*: $r = r$. *(C1)* is by E-REAL application and inversion. *(C2)* is trivial.

- **Case** $n = n' + 1$ **and** $e = r$ **and** $\tau = \texttt{sreal@}\varnothing$:

  By inversion on *(H4)* we have: $n' = 0$ and $v_1 = r$. Instantiate $v_2 = r$ in the conclusion. To show: *(C1)*: $\gamma_2 \vdash r \Downarrow_0 r$ unique; and *(C2)*: $r \sim^0 r$. *(C1)* is by E-SREAL application and inversion. *(C2)* is immediate by $|r - r| = 0 \leq 0$.

- **Case** $n = n' + 1$ **and** $e = \text{sing}(r)$ **and** $\tau = \texttt{real}[r]$:

  By inversion on *(H4)* we have: $n' = 0$ and $v_1 = r$. Instantiate $v_2 = r$ in the conclusion. To show: *(C1)*: $\gamma_2 \vdash \text{sing}(r) \Downarrow_0 r$ unique; and *(C2)*: $r = r$. *(C1)* is by E-SING application and inversion. *(C2)* is immediate.

- **Case** $n = n' + 1$ **and** $e = e_1 + e_2$ **and** $\tau = \texttt{real}$:

  By inversion on *(H4)*:

  $\gamma_1 \vdash e_1 \Downarrow_{n_1} r_{11}$ *(H4.1)*

  $\gamma_1 \vdash e_2 \Downarrow_{n_2} r_{12}$ *(H4.2)*

  and we also have: $n' = n_1 + n_2$, $v_1 = r_{11} + r_{12}$ and By IH ($n = n_i$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H4.2)* we have: $\gamma_2 \vdash e_1 \Downarrow_{n_1} r_{21}$ (unique) *(IH.C1.1)*; $\gamma_2 \vdash e_2 \Downarrow_{n_2} r_{22}$ (unique) *(IH.C1.2)*; $r_{11} = r_{21}$ *(IH.C2.1)* ; and $r_{12} = r_{22}$ *(IH.C2.2)* . Instantiate $v_2 = r_{21} + r_{22}$. To show: *(C1)*: $\gamma_2 \vdash e_1 + e_2 \Downarrow_{n_1+n_2} r_{21} + r_{22}$ (unique); and *(C2)*: $r_{11} + r_{12} = r_{21} + r_{22}$. *(C1)* is by $(IH.C1.1)$, $(IH.C1.2)$, and E-PLUS application and inversion. *(C2)* is by *(IH.C2.1)* and *(IH.C2.2)*.

- **Case** $n = n' + 1$ **and** $e = e_1 + e_2$ **and** $\tau = \texttt{sreal@}\Sigma'$:

  By inversion on *(H2)* and *(H4)* we have:

$\Gamma \vdash e_1 : \texttt{sreal@}\Sigma_1$ *(H2.1)*

$\Gamma \vdash e_2 : \texttt{sreal@}\Sigma_2$ *(H2.2)*

$\gamma_1 \vdash e_1 \Downarrow_{n_1} r_{11}$ *(H4.1)*

$\gamma_1 \vdash e_2 \Downarrow_{n_2} r_{12}$ *(H4.2)*

and we also have: $\Sigma' = \Sigma_1 + \Sigma_2$, $n' = n_1 + n_2$, $v_1 = r_{11} + r_{12}$ and By IH ($n = n_i$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H4.2)* we have: $\gamma_2 \vdash e_1 \Downarrow_{n_1} r_{21}$ (unique) *(IH.C1.1)*; $\gamma_2 \vdash e_2 \Downarrow_{n_2} r_{22}$ (unique) *(IH.C1.2)*; $r_{11} \sim^{\Sigma \cdot \Sigma_1} r_{21}$ *(IH.C2.1)*; and $r_{21} \sim^{\Sigma \cdot \Sigma_2} r_{22}$ *(IH.C2.2)*. Instantiate $v_2 = r_{21} + r_{22}$. To show: *(C1)*: $\gamma_2 \vdash e_1 + e_2 \Downarrow_{n_1+n_2} r_{21} + r_{22}$ (unique); and *(C2)*: $r_{11} + r_{12} \sim^{\Sigma \cdot (\Sigma_1 + \Sigma_2)} r_{21} + r_{22}$. *(C1)* is by *(IH.C1.1)*, *(IH.C1.2)*, and E-PLUS application and inversion. *(C2)* is by *(IH.C2.1)*, *(IH.C2.2)*, Plus Respects and Triangle.

- **Case** $n = n' + 1$ **and** $e = e_1 \ltimes e_2$ **and** either $\tau = \texttt{real}$ **or** $\tau = \texttt{sreal@}\Sigma'$:

  Similar to previous two cases, using Times Respects instead of Plus Respects.

- **Case** $n = n' + 1$ **and** $e = \texttt{if0}(e_1)\{e_2\}\{e_3\}$:

  By inversion on *(H4)* we have 2 subcases, each which induce:

  $\gamma_1 \vdash e_1 \Downarrow_{n_1} b_1$ *(H4.1)*

  By IH ($n = n_1$ decreasing), *(H1)*, *(H2)*, *(H3)* and *(H4.1)* we have: $\gamma_2 \vdash e_1 \Downarrow_{n_1} b_2$ (unique) *(IH.1.C1)*; and $b_1 = b_2$ *(IH.1.C2)*.

  - **Subcase** $b_1 = b_2 = \text{true}$:

    From prior inversion on *(H4)* we also have:

    $\gamma_1 \vdash e_2 \Downarrow_{n_2} v_1$ *(H4.2)*

    By IH ($n = n_2$ decreasing), *(H1)*, *(H2)*, *(H3)* and *(H4.2)* we have: $\gamma_2 \vdash e_2 \Downarrow_{n_2} v_2$ (unique) *(IH.2.C1)*; and $v_1 \sim v_2 \in \mathcal{V}_{n_1^{\Sigma}+n_2}[\![\tau]\!]$ *(IH.2.C2)*. Instantiate $v_2 = v_2$. To show: *(C1)*: $\gamma_2 \vdash \texttt{if}(e_1)\{e_2\}\{e_3\} \Downarrow_{n_1+n_2} v_2$; and

(C2): $v_1 \sim v_2 \in \mathcal{V}_{n_1^\Sigma + n_2}[\![\tau]\!]$. *(C1)* is by *(IH.1.C1)*, *(IH.2.C2)* and E-IF-TRUE application and inversion. *(C2)* is by *(IH.2.C2)*.

- **Subcase** $b_1 = b_2 = \text{false}$:

  Analogous to case $b_1 = b_2 = \text{true}$.

- **Case** $n = n' + 1$ **and** either $e = \langle e_1, e_2 \rangle$ **and** $\tau = \tau_1 \times \tau_2$ **or** $e = \hat{\langle} e_1, e_2 \hat{\rangle}$ **and** $\tau = (\sigma_1 \otimes \sigma_2)@\Sigma'$:

  Analogous to cases for $e = e_1 + e_2$ where $\tau = \texttt{real}$ or $\tau = \texttt{sreal}@\Sigma'$, and instead of appealing to Triangle, appealing to the definition of the logical relation.

- **Case** $n = n' + 1$ **and** either $e = \pi_i(e)$ **or** $e = \hat{\pi}_i(e)$):

  Analogous to cases for $e = e_1 + e_2$ where $\tau = \texttt{real}$ or $\tau = \texttt{sreal}@\Sigma'$, and instead of appealing to Triangle, appealing to the definition of the logical relation.

- **Case** $n = n' + 1$ **and** either $e = e_1 :: e_2$ **and** $\tau = \texttt{list}(\tau)$ **or** $e = e_1 \hat{::} e_2$ **and** $\tau = \texttt{slist}(\sigma)@\Sigma'$:

  Analogous to cases for $e = e_1 + e_2$ where $\tau = \texttt{real}$ or $\tau = \texttt{sreal}@\Sigma'$, and instead of appealing to Triangle, appealing to the definition of the logical relation.

- **Case** $n = n' + 1$ **and** either $e = \texttt{case}(e_1)\{[].e_2\}\{x_1 :: x_2.e_3\}$ **or** $e = \texttt{case}(e_1)\{\hat{[]}.e_2\}\{x_1 \hat{::} x_2.e_3\}$:

  Analogous to cases for $e = \texttt{if}(e_1)\{e_2\}\{e_3\}$.

- **Case** $n = n' + 1$ **and** $e = \lambda_z x.\ e$ **and** $\tau = \tau_1 \to \tau_2$: By inversion on *(H4)* we have: $n' = 0$, and $v_1 = \langle \lambda x_z.\ e \mid \gamma_1 \rangle$. Instantiate $v_2 = \langle \lambda_z x.\ e \mid \gamma_2 \rangle$. To show: *(C1)*: $\gamma_2 \vdash \lambda_z x.\ e \Downarrow \langle \lambda_z x.\ e \mid \gamma_2 \rangle$ unique; and *(C2)*: $\langle \lambda_z x.\ e \mid \gamma_1 \rangle \sim \langle \lambda_z x.\ e \mid \gamma_2 \rangle \in \mathcal{V}^\Sigma_{n'}[\![\tau_1 \to \tau_2]\!]$. Unfolding the definition, we must

show: $\forall n'' \leq n', v_1, v_2, .\ v_1 \sim v_2 \in \mathcal{V}^{\Sigma}_{n''}[\![\tau_1]\!] \Rightarrow \{x \mapsto v_1, z \mapsto \langle \lambda_z x.\ e \mid \gamma_1 \rangle\} \uplus \gamma_1, e \sim \{x \mapsto v_2, z \mapsto \langle \lambda_z x.\ e \mid \gamma_2 \rangle\} \uplus \gamma_2, e \in \mathcal{E}^{\Sigma}_{n''}[\![\tau_2]\!]$. To show, we assume: $v_1 \sim v_2 \in \mathcal{V}^{\Sigma}_{n'}[\![\tau_1]\!]$ *(C2.H1)*. Note the following facts: $\gamma_1 \sim \gamma_2 \in \mathcal{G}^{\Sigma}_{n'}[\![\Gamma]\!]$ *(F1)*; and $\{x \mapsto v_1\} \uplus \gamma_1 \sim \{x \mapsto v_2\} \uplus \gamma_2 \in \mathcal{G}^{\Sigma}_{n'}[\![\{x \mapsto \tau_1, z \mapsto \tau_1 \to \tau_2\} \uplus \Gamma]\!]$ *(F2)*. *(F1)* holds from *H1* and Step-index Weakening.1. *(F2)* holds from *(F1)*, *(C2.H1)* and the definition of $\gamma \sim \gamma \in \mathcal{G}^{\Sigma}_{n}[\![\Gamma]\!]$. Conclusion holds by IH ($n = n'$ decreasing), $F2$ and $C2.H1$.

- **Case** $n = n' + 1$ **and** $e = e_1(e_2)$:

  By inversion on *(H4)* we have:

$$\gamma_1 \vdash e_1 \Downarrow_{n_1} \langle \lambda_z x.\ e_1' \mid \gamma_1' \rangle \quad \textit{(H4.1)}$$

$$\gamma_1 \vdash e_2 \Downarrow_{n_2} v_1 \quad \textit{(H4.2)}$$

$$\{x \mapsto v_1, z \mapsto \langle \lambda_z x.\ e_1' \mid \gamma_1' \rangle\} \uplus \gamma_1' \vdash e_1' \Downarrow_{n_3} v_1' \quad \textit{(H4.3)}$$

  and we also have: $n' = n_1 + n_2 + n_3 + 1$, and $v_1 = v_1'$. By IH ($n = n'$ decreasing), *(H1)*, *(H2)*, *(H3)*, *(H4.1)* and *(H4.2)* we have: $\gamma_2 \vdash e_1 \Downarrow_{n_1} \langle \lambda_z x.\ e_2' \mid \gamma_2' \rangle$ *(IH.1.C1)*, $\gamma_2 \vdash e_2 \Downarrow_{n_2} v_2$ *(IH.2.C1)*, $\langle \lambda_z x.\ e_1' \mid \gamma_1' \rangle \sim \langle \lambda_z x.\ e_2' \mid \gamma_2' \rangle \in \mathcal{V}_{n'-n_1^{\Sigma}}[\![\tau_1 \to \tau_2]\!]$ *(IH.1.C2)*, and $v_1 \sim v_2 \in \mathcal{V}_{n'-n_2^{\Sigma}}[\![\tau_1]\!]$ *(IH.2.C2)*. Note the following facts: $n_3 \leq n' - n_1 - n_2$ *(F1)*; $\gamma_1 \sim \gamma_2 \in \mathcal{G}_{n-n_1^{\Sigma}-n_2}[\![\Gamma]\!]$ *(F2)*; and $v_1 \sim v_2 \in \mathcal{V}_{n-n_1^{\Sigma}-n_2}[\![\tau_2]\!]$ *(F3)*. *(F1)* follows from *(H3)* and $n' = n_1 + n_2 + n_3 + 1$. *(F2)* and *(F3)* follow from *(H1)*, *(IH.2.C2)* and Step-index Weakening. By IH ($n = n' - n_1 - n_2$ decreasing), *(H2)*, *(IH.1.C2)*, *(IH.2.C2)*, *(F1)*, *(F2)*, *(F3)* and *(H4.3)* we have: $\{x \mapsto v_2, z \mapsto \langle \lambda_z x.\ e_2' \mid \gamma_2' \rangle\} \uplus \gamma_2' \vdash e_2' \Downarrow_{n_3} v_2'$ *((IH.3.C1))* and $v_1' \sim v_2' \in \mathcal{V}_{n-n_1^{\Sigma}-n_2-n_3}[\![\tau_2]\!]$ *(IH.3.C2)*. Instantiate $v_2 = v_2'$. To show: *(C1)*: $\gamma_2 \vdash e_1(e_2) \Downarrow_{n_1+n_2+n_3+1} v_2'$ (unique); and *(C2)*: $v_1' \sim v_2' \in \mathcal{V}_{n-n_{11}^{\Sigma}-n_{12}-n_{13}-1}[\![\tau_2]\!]$. *(C1)* is immediate from *(IH.1.C1)*, *(IH.2.C1)* and *(IH.3.C1)*. *(C2)* is immediate from *(IH.3.C2)* and Step-index Weakening.2.

- **Case** $n = n' + 1$ **and** either $e = \texttt{reveal}(e')$ **or** $e = \texttt{return}(e')$ **or** $e = \texttt{laplace}[e_1, e_2](e_3)$ **or** $e = x \leftarrow e_1\ ;\ e_2$:

Follows from inductive hypothesis, post processing (for `laplace`) and sequential composition (for $x \leftarrow e_1 \ ; \ e_2$) theorems from the differential privacy literature (Dwork et al., 2014a).

□

Part V

Related Work & Conclusion

Chapter 25

Related Work

25.1 Languages for Static Verification of Differential Privacy.

25.1.1 Linear Types.

Fuzz was the first language and type system designed to verify differential privacy costs of a program, and did so by modeling sensitivity using linear types (Reed and Pierce, 2010). DFuzz extended Fuzz with dependent types and automation aided by SMT solvers (Gaboardi et al., 2013b).

As described by Azevedo de Amorim et al. (de Amorim et al., 2018), encoding (ϵ, δ)-differential privacy in linear type systems like Fuzz is particularly challenging because these systems place restrictions on the interpretation of the linear function space, and (ϵ, δ)-differential privacy does not satisfy these restrictions. In particular, using Fuzz requires that the desired notion of privacy can be recovered from an instantiation of function sensitivity for an appropriately defined metric on probabilistic functions. No such metric can be defined for

(ϵ, δ)-differential privacy, preventing a straightforward interpretation of linear functions as (ϵ, δ)-differentially private functions.

The DUET language extends FUZZ with support for advanced variants of differential privacy such as (ϵ, δ)-differential privacy (Near et al., 2019b). Adaptive Fuzz embeds a static sensitivity analysis within a dynamic privacy analysis using privacy odometers and filters (Winograd-Cort et al., 2017).

25.1.2 Indexed Monadic Types

In their work, Azevedo de Amorim et al. (de Amorim et al., 2018) define a *path construction* to encode non-linear scaling via an indexed probability monad, which can be used to extend FUZZ with support for arbitrary relational properties (including (ϵ, δ)-differential privacy). However, this approach (1) internalizes the use of *group privacy* (Dwork et al., 2014a) which in many cases provides sub-optimal bounds on privacy cost–and (2) is unable to provide privacy bounds for more than one input to a function–a useful capability of the original FUZZ language, and a necessary feature to obtain optimal privacy bounds for multi-argument functions. Note, however, that all the novel approaches proposed in this dissertation provide optimal privacy bounds for multi-argument functions.

25.1.3 Program Logics

Program logics such as APRHL (Barthe et al., 2012, 2013) are very flexible and expressive but difficult to automate. FUZZI introduces a type system equivalent to FUZZ which is then enriched with program logics (Zhang et al., 2019a) and is more amenable to automation.

At a high level, Fuzzi (Zhang et al., 2019b) has a similar aim to DUET: supporting differential privacy for general-purpose programs and supporting recent variants of differential privacy. DUET is designed primarily as a fully-

automated type system with a rich set of primitives for vector-based and higher-order programming; low-level mechanisms in DUET are opaque and trusted. On the other hand, Fuzzi is designed for general-purpose programming, low-level mechanism implementation, and their combination; however, to achieve this, Fuzzi has less support for higher-order programming and automation in typechecking.

25.1.4 Higher-order Relational Type Systems

Following the initial work on linear typing for differential privacy (Reed and Pierce, 2010), a parallel line of work (Barthe et al., 2015, 2016a) leverages *relational refinement types* aided by SMT solvers in order to support type-level dependency of privacy parameters (à la DFuzz (Gaboardi et al., 2013a)) in addition to more powerful variants of differential privacy such as (ϵ, δ)-differential privacy. These approaches support (ϵ, δ)-differential privacy, but did not support usable type inference until a recently proposed heuristic bi-directional type system (Çiçek et al., 2018). Although a direct case study of bidirectional type inference for relational refinement types has not yet been applied to differential privacy, the possibility of such a system appears promising.

The overall technique for supporting (ϵ, δ)-differential privacy in these relational refinement type systems is similar to (and predates) Azevedo de Amorim et al.–privacy cost is tracked through an "effect" type, embodied by an indexed monad. It is this "effect"-based treatment of privacy cost that fundamentally limits these type system to not support multi-arity functions, resulting in non-optimal privacy bounds for some programs.

25.1.5 Relational Type Systems with Randomness Alignments & Probabilistic Couplings.

Yet another approach is LightDP which uses a light-weight relational type system and randomness alignments to verify (ϵ, δ)-differential privacy bounds of first-order imperative programs (Zhang and Kifer, 2017), and is suitable for verifying low-level implementations of differentially private mechanisms. Essentially, aligning randomness involves the creation of an injective function from the randomness in the execution under a database D_1 into the randomness in the execution under an adjacent database D_2, so that both executions generate the same output. A notable achievement of this work is a lightweight, automated verification of the Sparse Vector Technique (Dwork et al., 2014a) (SVT). However, LightDP is not suitable for sensitivity analysis, an important component of differentially-private algorithm design. Note that all the novel approaches proposed in this dissertation are capable of sensitivity analysis, but are incapable of verified implementation of the Sparse Vector Technique.

Differential privacy mechanisms often require knowledge of (or place restrictions on) function sensitivity of arguments to the mechanism. In principle, a language like Fuzz could be combined with LightDP to fully verify both an application which uses SVT, as well as the implementation of SVT itself.

Barthe et al introduce an approach for proving differential privacy using a generalization of probabilistic couplings. They present several case studies in the apRHL$^+$ (Barthe et al., 2016b) language which extends program logics with approximate couplings. The technique of aligning randomness is also used in the coupling method.

Albarghouthi and Hsu (Albarghouthi and Hsu, 2018) use an alternative approach based on randomness alignments as well as approximate couplings.

25.2 Dynamic Enforcement of Differential Privacy.

The first approach for dynamic enforcement of differential privacy was PINQ (McSherry, 2009). Since then several works have been based on PINQ, such as Featherweight PINQ (Ebadi and Sands, 2015) which models PINQ formally and proves that any programs which use its simplified PINQ API are differentially private. ProPer (Ebadi et al., 2015) is a system (based on PINQ) designed to maintain a privacy budget for each individual in a database system, and operates by silently dropping records from queries when their privacy budget is exceeded. UniTrax (Munz et al., 2018) follows up on ProPer: this system allows per-user budgets but gets around the issue of silently dropping records by tracking queries against an abstract database as opposed to the actual database records. These approaches are limited to an embedded DSL for expressing relational database queries, and do not support general purpose programming.

A number of programming frameworks for differential privacy have been developed as libraries for existing programming languages. DPella (Lobo-Vesga et al., 2020) is a Haskell library that provides static bounds on the accuracy of differentially private programs. Diffprivlib (Holohan et al., 2019) (for Python) and Google's library (Wilson et al., 2020) (for several languages) provide differentially private algorithms, but do not track sensitivity or privacy as these algorithms are composed. ϵktelo (Zhang et al., 2018) executes programmer-specified *plans* that encode differentially private algorithms using framework-supplied building blocks.

25.2.1 Dynamic Information Flow Control.

Our approach to dynamic enforcement of differential privacy can be seen as similar to work on dynamic information flow control (IFC) and taint analysis (Austin and Flanagan, 2009). The sensitivities that we attach to values are comparable to IFC labels. However, dynamic IFC typically allows the programmer to branch on sensitive information and handles implicit flows dynamically.

25.2.2 Dynamic Testing for Differential Privacy.

A recent line of work (Bichsel et al., 2018; Ding et al., 2018; Wang et al., 2020; Wilson et al., 2020) has resulted in approaches for *testing* differentially private programs. These approaches generate a series of neighboring inputs, run the program many times on the neighboring inputs, and raise an alarm if a counterexample is found. These approaches do not require type annotations, but do require running the program many times. Static or dynamic analysis is preferable to testing because it is more efficient and generates a proof of privacy.

25.3 Security as a Library/Language Extension

Li et al (Peng Li and Zdancewic, 2006) present an embedded security sublanguage in Haskell using the arrows combinator interface. Russo et al introduce a monadic library for light-weight information flow security in Haskell (Russo et al., 2008). Crockett et al propose a domain specific language for safe homomorphic encryption in Haskell (Crockett et al., 2018). Safe Haskell (Terei et al., 2012) is a Haskell language extension which implements various security policies as monads. Parker et al (Parker et al., 2019) introduce a Haskell framework for enforcing information flow control policies in database-oriented web applications. DPella (Lobo-Vesga et al., 2020) is a programming framework in Haskell that

performs privacy and accuracy bound tracking of data analysis programs.

Chapter 26

Conclusion

Differential privacy has become the standard for protecting the privacy of individuals with formal guarantees of plausible deniability. In this dissertation we have proposed new techniques for language-based analysis of differential privacy of programs in a variety of contexts spanning static and dynamic analysis. Our approach towards differential privacy analysis makes use of ideas from linear type systems and static/dynamic taint analysis. We have shown a series of works that, in short, demonstrated the following key ideas:

- A pure linear typing discipline is sufficient to perform accurate analysis of differential privacy, even for its advanced variants.
- Prescriptive and descriptive dynamic analysis of differential privacy is possible with low overhead for general-purpose programming languages.
- Mainstream statically typed languages can be made to perform differential privacy analysis as part of their standard typechecking process, without any runtime execution or information.
- Language-based analysis of differential privacy can be practical and conve-

nient in any given scenario.

It is our hope that the usability and strong guarantees of the works contained in this dissertation will inspire data analysts, technology corporations, researchers, and students in computer science to continue to build a community and culture of verifiable data privacy.

Chapter 27

Bibliography

2019. scikit-learn: Standardization, or mean removal and variance scaling. https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler

Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 308–318. https://doi.org/10.1145/2976749.2978318

John M. Abowd. 2018. The U.S. Census Bureau Adopts Differential Privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 2867. https://doi.org/10.1145/3219819.3226070

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 69–83.

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

Thomas H. Austin and C. Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *PLAS '09*.

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *POPL*, Vol. 52. ACM, 545–556.

Andrew Barber. 1996. *Dual Intuitionistic Linear Logic*. Technical Report ECS-LFCS-96-347. University of Edinburgh.

Gilles Barthe, Thomas Espitau, Justin Hsu, Tetsuya Sato, and Pierre-Yves Strub. 2019. Relational $\star\star$\star-Liftings for Differential Privacy. *Logical Methods in Computer Science* 15, 4 (2019). https://lmcs.episciences.org/5989

Gilles Barthe, Gian Pietro Farina, Marco Gaboardi, Emilio Jesus Gallego Arias, Andy Gordon, Justin Hsu, and Pierre-Yves Strub. 2016a. Differentially Private Bayesian Programming. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 68–79. https://doi.org/10.1145/2976749.2978371

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In *POPL*. ACM, 55–68.

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016b. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 749–758. https://doi.org/10.1145/2933575.2934554

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 97–110. https://doi.org/10.1145/2103656.2103670

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3, Article 9 (Nov. 2013), 49 pages. https://doi.org/10.1145/2492061

Raef Bassily, Adam Smith, and Abhradeep Thakurta. 2014a. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*. IEEE, 464–473.

Raef Bassily, Adam Smith, and Abhradeep Thakurta. 2014b. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*. IEEE, 464–473.

Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. 2018. Dp-finder: Finding differential privacy violations by sampling and optimization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 508–524.

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/2784731.2784733

Mark Bun, Cynthia Dwork, Guy N Rothblum, and Thomas Steinke. 2018. Composable and versatile privacy via truncated CDP. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. 74–86.

Mark Bun and Thomas Steinke. 2016. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of Cryptography Conference*. Springer, 635–658.

K. Chaudhuri and C. Monteleoni. 2008. Privacy-preserving logistic regression. In *NIPS*.

Kamalika Chaudhuri, Claire Monteleoni, and Anand D Sarwate. 2011. Differentially private empirical risk minimization. *Journal of Machine Learning Research* 12, Mar (2011), 1069–1109.

Kamalika Chaudhuri and Staal A Vinterbo. 2013. A stability-based validation procedure for differentially private machine learning. In *Advances in Neural Information Processing Systems*. 2652–2660.

Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2018. Bidirectional Type Checking for Relational Properties. *CoRR* abs/1812.05067 (2018). arXiv:1812.05067 http://arxiv.org/abs/1812.05067

Eric Crockett, Chris Peikert, and Chad Sharp. 2018. ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1020–1037. https://doi.org/10.1145/3243734.3243828

Arthur Azevedo De Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type Checking. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. ACM, 5.

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2018. Metric Semantics for Probabilistic Relational Reasoning. *CoRR* abs/1807.05091 (2018). arXiv:1807.05091 http://arxiv.org/abs/1807.05091

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Probabilistic Relational Reasoning via Metrics. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–19.

Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting violations of differential privacy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 475–489.

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

Cynthia Dwork and Jing Lei. 2009. Differential privacy and robust statistics.. In *STOC*, Vol. 9. 371–380.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006a. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proceedings of the Third Conference on Theory of Cryptography* (New York, NY) *(TCC'06)*. Springer-Verlag, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006b. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

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

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

Hamid Ebadi and David Sands. 2015. Featherweight PINQ. arXiv:1505.02642 [cs.PL]

Hamid Ebadi, David Sands, and Gerardo Schneider. 2015. Differential Privacy: Now it's Getting Personal. *ACM SIGPLAN Notices* 50. https://doi.org/10.1145/2676726.2677005

Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1054–1067. https://doi.org/10.1145/2660267.2660348

Georgina Evans and Gary King. 2021. Statistically Valid Inferences from Differentially Private Data Releases, with Application to the Facebook URLs Dataset.

Georgina Evans, Gary King, Margaret Schwenzfeier, and Abhradeep Thakurta. 2021. Statistically Valid Inferences from Privacy Protected Data.

Vitaly Feldman and Tijana Zrnic. 2020. Individual Privacy Accounting via a Renyi Filter. *arXiv preprint arXiv:2008.11193* (2020).

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013a. Linear dependent types for differential privacy. In *POPL*, Vol. 48. ACM, 357–370.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013b. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 357–370.

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50, 1 (Jan. 1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Miguel Guevara. 2019. Enabling developers and organizations to use differential privacy. https://developers.googleblog.com/2019/09/enabling-developers-and-organizations.html

Miguel Guevara, Mirac Vuslat Basaran, Sasha Kulankhina, and Badih Ghazi. 2020. Expanding our Differential Privacy Library. https://opensource.googleblog.com/2020/06/expanding-our-differential-privacy.html

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*. 2339–2347.

Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. 2019. Diffprivlib: the IBM differential privacy library. *arXiv preprint arXiv:1907.02444* (2019).

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 633–645. https://doi.org/10.1145/2535838.2535846

Daniel Kifer, Solomon Messing, Aaron Roth, Abhradeep Thakurta, and Danfeng Zhang. 2020. Guidelines for Implementing and Auditing Differentially Private Systems. arXiv:2002.04049 [cs.CR]

Gary King and Nathaniel Persily. 2020. Unprecedented Facebook URLs Dataset now Available for Academic Research through Social Science One. https://socialscience.one/blog/unprecedented-facebook-urls-dataset-now-available-research-through-social-science-one

Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2020. A Programming Framework for Differential Privacy with Accuracy Concentration Bounds. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 411–428.

Frank McSherry and Ratul Mahajan. 2010. Differentially-Private Network Trace Analysis. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/1851182.1851199

Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/1559845.1559850

Ilya Mironov. 2017a. Rényi differential privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 263–275.

Ilya Mironov. 2017b. Rényi Differential Privacy. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 263–275. https://doi.org/10.1109/CSF.2017.11

Reinhard Munz, Fabienne Eigner, Matteo Maffei, Paul Francis, and Deepak Garg. 2018. UniTraX: Protecting Data Privacy with Discoverable Biases. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 278–299.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (June 2008), 49 pages. https://doi.org/10.1145/1352582.1352591

Chaya Nayak. 2020. New privacy-protected Facebook data for independent research on social media's impact on democracy. https://research.fb.com/blog/2020/02/new-privacy-protected-facebook-data-for-independent-research-on-social-medias-impact-on-democracy/

Joseph P Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, et al. 2019b. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019a. Duet: An Expressive Higher-order Language and Linear Type System for Statically Enforcing Differential Privacy. *CoRR* abs/1909.02481 (2019). https://arxiv.org/abs/1909.02481

Dominic Orchard and Tomas Petricek. 2014. Embedding Effect Systems in Haskell. *SIGPLAN Not.* 49, 12 (Sept. 2014), 13–24. https://doi.org/10.1145/2775050.2633368

D. Orchard, Tomas Petricek, and A. Mycroft. 2014. The semantic marriage of monads and effects. *ArXiv* abs/1401.5391 (2014).

James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information Flow Security for Multi-Tier Web Applications. *Proc. ACM Program. Lang.* 3, POPL, Article 75 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290388

Peng Li and S. Zdancewic. 2006. Encoding information flow in Haskell. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 12 pp.–16. https://doi.org/10.1109/CSFW.2006.13

Tomas Petricek. 2017. *Context-aware programming languages*. Ph.D. Dissertation. University of Cambridge.

Jason Reed and Benjamin C Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 157–168.

Ryan M. Rogers, Salil P. Vadhan, Aaron Roth, and Jonathan Ullman. 2016. Privacy Odometers and Filters: Pay-as-you-Go Composition. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 1921–1929. http://papers.nips.cc/paper/6170-privacy-odometers-and-filters-pay-as-you-go-composition

Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-Weight Information-Flow Security in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1411286.1411289

T. Sato, G. Barthe, M. Gaboardi, J. Hsu, and S. Katsumata. 2019. Approximate Span Liftings: Compositional Semantics for Relaxations of Differential Privacy. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–14. https://doi.org/10.1109/LICS.2019.8785668

Shuang Song, Kamalika Chaudhuri, and Anand D Sarwate. 2013. Stochastic gradient descent with differentially private updates. In *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*. IEEE, 245–248.

Kunal Talwar, Abhradeep Guha Thakurta, and Li Zhang. 2015. Nearly optimal private lasso. In *Advances in Neural Information Processing Systems*. 3025–3033.

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 137–148. https://doi.org/10.1145/2364506.2364524

J. Vaidya, B. Shafiq, A. Basu, and Y. Hong. 2013. Differentially Private Naive Bayes Classification. In *2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, Vol. 1. 571–576. https://doi.org/10.1109/WI-IAT.2013.80

Philip Wadler and Peter Thiemann. 2003. The Marriage of Effects and Monads. *ACM Trans. Comput. Logic* 4, 1 (Jan. 2003), 1–32. https://doi.org/10.1145/601775.601776

Yu-Xiang Wang, Borja Balle, and Shiva Kasiviswanathan. 2018. Subsampled Rényi Differential Privacy and Analytical Moments Accountant. *CoRR* abs/1808.00087 (2018). arXiv:1808.00087 http://arxiv.org/abs/1808.00087

Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng Zhang. 2020. CheckDP: An Automated and Integrated Approach for Proving Differential Privacy or Finding Precise Counterexamples. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 919–938.

Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving differential privacy with shadow execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 655–669.

Royce J. Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2019. Differentially Private SQL with Bounded User Contribution. *CoRR* abs/1909.01917 (2019). arXiv:1909.01917 http://arxiv.org/abs/1909.01917

Royce J Wilson, Celia Yuxing Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially Private SQL with Bounded User Contribution. *Proceedings on Privacy Enhancing Technologies* 2020, 2 (2020).

Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A framework for adaptive differential privacy. *Proc. ACM Program. Lang.* 1, ICFP (2017), 10:1–10:29. https://doi.org/10.1145/3110254

Xi Wu, Fengan Li, Arun Kumar, Kamalika Chaudhuri, Somesh Jha, and Jeffrey Naughton. 2017. Bolt-on Differential Privacy for Scalable Stochastic Gradient Descent-based Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 1307–1322. https://doi.org/10.1145/3035918.3064047

Danfeng Zhang and Daniel Kifer. 2017. LightDP: towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 888–901.

Dan Zhang, Ryan McKenna, Ios Kotsogiannis, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. 2018. Ektelo: A framework for defining differentially-private computations. In *Proceedings of the 2018 International Conference on Management of Data*. 115–130.

Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C Pierce, and Aaron Roth. 2019a. Fuzzi: A three-level logic for differential privacy. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–28.

Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019b. Fuzzi: A Three-Level Logic for Differential Privacy. Accepted for publication in PACMPL / ICFP 2019.