



BY MATÍAS TORO, FEDERICO OLMEDO, AND ÉRIC TANTER

Gradual Differentially Private Programming

The tons of data generated—continuously—by individuals worldwide provides unprecedented informational opportunities, spanning from constructing recommendation systems, to models that automate repetitive and tedious tasks, to assisting public and private institutions in decision making. However, this data may contain sensitive information about individuals, such as medical, financial, or geolocation records, and its use may incur serious privacy breaches, even when taking measures to prevent the leakage of sensitive information.

To illustrate this issue, let us consider the Chilean Electoral Service (SERVEL), which publishes data removing all explicit identifiers of citizens, such as name and ID number—an anonymization technique known as *de-identification*. Despite these privatization efforts, we were able to identify several citizens by linking the voting records of the national referendum of October 2020 with other public data sources. Indeed, we could tell the party affiliation of these citizens and whether they voted or not—information deemed highly sensitive.¹⁵

Differential privacy to the rescue. In 2006, Cynthia Dwork put forward the first compelling notion of privacy-preserving data mining, together with basic mechanisms to realize it.⁶ Dubbed *differential privacy* (DP), this notion provides strong privacy guarantees, independent of the considered classes of attack and the auxiliary information available to the attacker (as leveraged in the SERVEL attack).

In general, a computation over a dataset is deemed differentially private if adding an extra individual to the dataset produces only a “negligible” variation of the computation outcome. This means that an attacker observing the computation outcome will be able to infer about the same information, whether any individual opts in or out of the dataset. Individuals will thus have no sensible reason to refrain from participating in such a data analysis. Figure 1 summarizes the main ingredients of the formal definition of DP.

Notably, we can achieve DP by *output perturbation*: if $f : D \rightarrow \mathbb{R}$ is a real-valued query, D denoting the dataset universe, we can construct a differentially private variant of f by adding statistical noise to its output. Moreover, this noise should be calibrated according to the *sensitivity* of f , defined as:

$$S(f) = \max_{d \sim d'} |f(d) - f(d')|$$

Intuitively, $S(f)$ captures how much f 's output can vary upon the addition of an extra individual to f 's input dataset. For example, if the query f counts the number of individuals in a dataset who are 40 years or older, then $S(f) = 1$ since adding an individual to the dataset can change the count by at most 1.

Figure 2 describes the *Laplacian mechanism*, which formalizes this privatization strategy. In general, the greater the sensitivity of query $f : D \rightarrow \mathbb{R}$, the more noise we must add to guarantee ϵ -DP, and, as a corollary, the less precise the private mechanism also becomes. Therefore, to

keep results as accurate as possible, while guaranteeing ϵ -DP, it is crucial to have bounds for the sensitivity of queries that are both *sound* and *as tight as possible*. This has proved to be a daunting task.

We have witnessed this phenomenon personally in a course on data privacy that we created at the University of Chile, where students have hardly been able to derive sensitivities for elementary queries. To illustrate the class of sensitivity analysis required, let us consider Figure 3a, which shows a simple program where privacy depends on a dynamically obtained input value: if `b` is `True`, the function is 1-sensitive, so the program is private, but if `b` is `False`, the added noise is insufficient to ensure privacy. In fact, this problem goes way over the heads of undergraduate students. Recently, Min Lyu et al. uncovered serious flaws in multiple DP algorithms published in recognized venues, with stringent peer-review processes.⁸ This calls for systematic and principled approaches for developing and verifying DP programs.

DP Programming

Several techniques have been proposed to achieve effective DP programming. Among them, techniques based on programming languages have gained a lot of attention. These can be classified into two main groups: those based on *type systems*^{2,3,7,11} and those based on *program logics*.^{4,5,12} Although the latter are more expressive, they are generally more challenging to automate and much more complex than type systems. In contrast, type systems are automatic and lightweight verification tools, which makes them the most used formal techniques in practice today. The increasing popularity of the Rust programming language,¹⁴ which uses ownership types to guarantee memory and thread safety, illustrates that programmers are willing to embrace complex typing when the benefits are substantial.

Figure 1. Basic notions of the differential privacy framework.

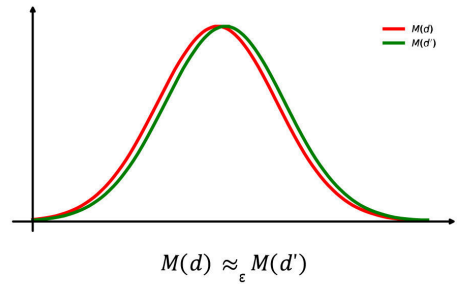
Data model. A *dataset* consists of a collection of records, where each record encodes the (possibly sensitive) information of an individual. Furthermore, a pair of datasets d and d' are said to be *adjacent*, written $d \sim d'$, if one is obtained from the other by adding the contribution (i.e. the record) of a single new individual.

Differential privacy. A randomized algorithm M over datasets is said to be ϵ -*differentially private* (ϵ -DP), for an $\epsilon \in \mathbb{R}_{\geq 0}$, if

$$d \sim d' \Rightarrow M(d) \approx_{\epsilon} M(d')$$

Here, the relation \approx_{ϵ} encodes a closeness condition between probability distributions, i.e. M 's outputs: the smaller the ϵ , the more similar distributions $M(d)$ and

$M(d')$ become (see figure below), and the stronger the privacy guarantees of M .



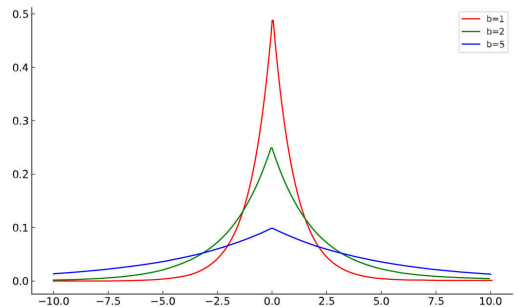
We dispense with the formal definition of \approx_{ϵ} here, because it depends on the precise variant of DP considered and it is irrelevant for our purposes.

Figure 2. Mechanism to achieve differential privacy for numeric queries.

Laplacian mechanism. Randomized algorithm

$$M(d) = f(d) + \text{Lap}\left(\frac{S(f)}{\epsilon}\right)$$

is an ϵ -DP version of numeric query $f : D \rightarrow \mathbb{R}$. Here, $\text{Lap}(b)$ represents a sample from the Laplacian distribution with parameter b — a symmetric distribution centered around 0, with exponential probability density function, illustrated on the right for different values of b .



Type systems assist programmers in automatically verifying a program *before its execution*, ensuring that certain classes of errors never occur at runtime. For example, a basic typechecker would prevent the program `1 + True` from executing. During compilation, a typechecker attempts to classify program expressions in terms of the type of values they produce, and if it cannot do so, it reports a static type error. When applied to DP, a typechecker can determine, through sensitivity and privacy annotations, whether a program is differentially private or not.

Fuzz. In 2010, the first language explicitly designed for reasoning about sensitivity and DP emerged: Fuzz.¹¹ The core idea revolves around the use of linear types to track sensitivity, making every well-typed program inherently differentially private.

Fuzz’s type system approximates sensitivity to the number of times variables are used in an expression. Generally, it offers two types for functions: $\mathbb{R} \rightarrow \mathbb{R}$, indicating unrestricted use of the argument (that is, ∞ -sensitive), and $\mathbb{R} \multimap \mathbb{R}$, denoting a function that uses its argument *at most* once (that is, 1-sensitive). For instance, $f(x) = x + 1$ can be typed as $\mathbb{R} \multimap \mathbb{R}$, but $g(x) = x + x$ only as $\mathbb{R} \rightarrow \mathbb{R}$, unless a technique known as *metric scaling* is used, which allows g to be typed as $\mathbb{R} \multimap \frac{1}{2}\mathbb{R}$.

Additionally, Fuzz introduces the type $\mathbb{O}\mathbb{R}$ for noisy computations, representing probability distributions over real numbers. For example, a function like `add_noise`, which adds Laplace-distributed noise with scaling parameter $1/\epsilon$ to a real number, has type $\mathbb{R} \multimap \mathbb{O}\mathbb{R}$. Suppose also a 1-sensitive function `over_40` that computes how many people in a dataset are 40 years or older (typed as $D \multimap \mathbb{R}$). Then we can build a differentially private version of this function as follows:

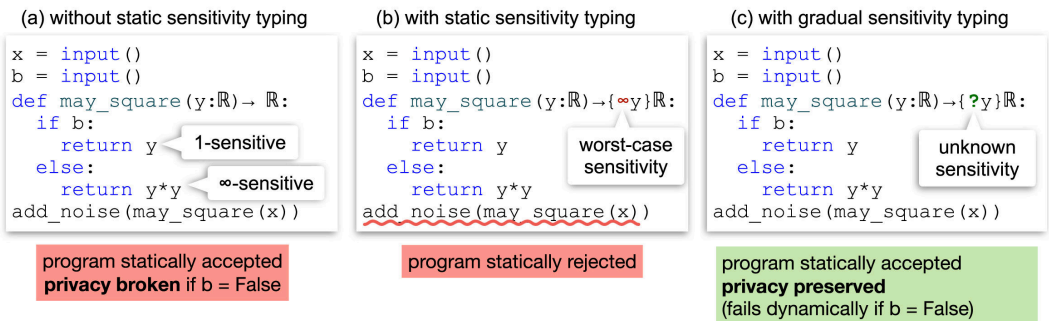
```
def over_40_DP(d):
    return add_noise(over_40(d))
```

The function `over_40_DP` is typed as $D \multimap \mathbb{O}\mathbb{R}$, and is formally guaranteed to be ϵ -DP.

Despite its contributions, Fuzz and some successors have limitations. In particular, Fuzz primarily supports the basic ϵ -DP variant, not accommodating more advanced versions, such as (ϵ, δ) -DP or Renyi-DP. Furthermore, some can be overly conservative in assessing program sensitivity, potentially requiring programmers to possess advanced knowledge about the sensitivity of their programs and scale types accordingly.

Jazz. In response to these limitations and to enhance the adoption of DP in programming languages, we developed the Jazz language.¹⁶ Jazz is inspired by Fuzz¹¹ and Duet,¹⁰ but offers more precise sensitivity analysis without sacrificing simplicity. It is designed to support advanced DP variants and higher-order programming.

Figure 3. Static and gradual sensitivity typing.



Like Duet, Jazz consists of two mutually defined sub-languages: one for sensitivity analysis and the other for privacy. The sensitivity sub-language allows fine-grained sensitivity information to be explicitly declared in types. For instance, the function $g(x) = x + x$ has the type $(x:\mathbb{R}) \rightarrow \{2x\}\mathbb{R}$, indicating a *latent sensitivity* of $2x$. In Jazz, the latent sensitivity of the function only matters whenever the function is applied. The type syntax also enables precise reasoning about multiple variables simultaneously. For instance, consider the following:

```
z = input()
def plus_z(y:ℝ)→{z+y}ℝ:
  def unused():
    z * y
  return z+y
```

Function `plus_z` is 1-sensitive in z and y (notation $z+y$), since `unused` is never applied. Jazz generalizes this idea of latent sensitivity accounting to other type constructors, such as data structures: the sensitivity of a computation that uses a tuple of values only considers the components that are effectively used, lazily, instead of eagerly accounting for the sensitivity of all the components. This inherently leads to more accurate analysis without resorting to complex scaling operators.

The privacy sub-language of Jazz allows reasoning about the privacy of individual variables. For instance, a function of type $(x:\mathbb{R} \cdot d) \rightarrow \{\varepsilon x\}\mathbb{R}$ satisfies ε -DP with respect to its argument x , if the *relational distance* of x is at most d . The relational distance measures how much an argument can vary in two separate executions, often setting $d = 1$ to recover the standard setting of two datasets that differ by only one individual. For instance, the previously introduced `add_noise` function is typed in Jazz as $(x:\mathbb{R} \cdot 1) \rightarrow \{\varepsilon x\}\mathbb{R}$. The type system guarantees that in programs such as `add_noise(y + z)`, the sum of the relational distances of y and z is at most 1.

Types are inherently conservative. Although we have seen the benefits offered by type systems like those of Fuzz, Duet, and Jazz, they provide a conservative approximation of program behavior and therefore can reject programs that may be private.

In Figure 3b, Jazz rejects the program statically because the `may_square` function can only be treated conservatively as an ∞ -sensitive function, the worst-case sensitivity corresponding to the `else` branch. This sensitivity conflicts with the `add_noise` function, which accepts arguments at a relational distance of 1. Note the program is rejected even though, when `b` is `True`, the `may_square` function in fact behaves as a 1-sensitive function! In addition to the issue of conservativity, sensitivity typing comes with a complexity overhead that can be daunting at first.

Gradual Typing Meets DP

In recent years, the adoption of type systems has been facilitated by a technique known as *gradual typing*, which supports the smooth integration of static and dynamic typechecking.¹³ The premise of gradual typing is that some type information can be unknown statically and that the typechecker works optimistically with respect to missing type information. While there are different ways to realize this vision, it has found its way into many industrial languages, such as Dart, TypeScript, Flow, Hack, Sorbet for Ruby, and Python 3 to some extent. The current success of TypeScript in the JavaScript community demonstrates that programmers who favor dynamically typed languages are willing to embrace types, if they can do it at their own pace, without being forced into an all-or-nothing commitment.¹⁴

We believe the strength and benefits of gradual typing are more attractive for advanced, complex typing disciplines. The research community, ourselves included, has been (and still is!) studying how to provide gradual versions of type systems for myriad properties, such as purity, structural invariants, object ownership, communication protocols, and so on. We are currently developing gradual approaches to sensitivity and DP, to ease the adoption of the techniques described earlier.

Gradual sensitivity. We developed the theory of gradual sensitivity typing, implemented in a prototype language called GSoul.¹ GSoul is close to the sensitivity sublanguage of Jazz, but also supports *imprecise* sensitivity information, either in the form of an interval (such as $[2, 8]x$ to denote a function that is at least 2-sensitive and at most 8-sensitive in its argument x) or a fully unknown sensitivity (such as $?x$ if the sensitivity with respect to x is not statically known, which is indeed equivalent to the interval $[0, \infty]x$). As a gradually typed language, GSoul treats imprecise information optimistically during typechecking, but backs such optimism with dynamic checks during program execution to ensure other statically stated invariants are not violated.

Figure 3c shows how we can use the unknown sensitivity to smoothly handle the statically unpredictable sensitivity of `may_square`. The program optimistically typechecks because it is *plausible* for `may_square` to be 1-sensitive in y . When run, if `b` happens to be `True`, execution proceeds without error. Conversely, if `b` happens to be `False`, then a runtime exception is raised when `add_noise` is applied, reporting that the argument violates the 1-sensitivity requirement.

In addition to making adoption of sensitivity typing smoother by permitting parts of sensitivity information to be statically and precisely specified, and parts left unknown or imprecise, gradual sensitivity typing also provides an alternative to handle recursive functions in an exact, albeit dynamically checked manner. This is especially useful when dealing with recursive functions on which the sensitivity with respect to an argument depends on the number of recursive calls (unless one adopts dependent types as in Gaboardi et al.⁷). In GSoul, it suffices to declare the sensitivity of a recursive function as unknown and let dynamic checking account for the actual sensitivity observed at runtime.

DP with gradual sensitivity. Gradual sensitivity opens at least two different ways to address DP programming. The approach described earlier consists in adding noise based on a fixed expected sensitivity: `add_noise` expects a 1-sensitive argument and adds noise accordingly. If the argument comes from a gradually typed computation that happens to violate this fixed requirement during execution, then a runtime error is raised. This approach has the benefit of ensuring a known accuracy of the result. On the downside, one could be adding too much noise, for instance if the argument happens to come from a computation that is insensitive, or whose sensitivity is strictly less than the expected one. While it is sound to do so, it unnecessarily sacrifices accuracy.

A dual approach consists of making a computation private by adding the amount of noise that corresponds to the observed sensitivity at runtime. This approach cannot fail dynamically and ensures that noise is appropriately calibrated to the actual sensitivity of the computation. However, its downside is that the resulting accuracy could be disastrous: if one erroneously implements a function as ∞ -sensitive while it was intended to have a finite sensitivity, the result accuracy will be annihilated by an excessive amount of noise. Here, a new concern emerges—that of reporting to users the actual amount of noise added to a final result, without endangering privacy.

Both approaches appear viable and useful, possibly even within the lifecycle of the same codebase. A practical language for gradual DP programming should ideally support both, letting users determine which best fits their needs.

Perspectives

In this brief overview, we have been deliberately oblivious to the theoretical aspects. Devising formal foundations for gradual DP programming is highly challenging. Additionally, much work remains to be done in terms of providing practical implementations, ideally integrated in mainstream languages routinely used for data analytics. For instance, a Python library integrated with a standard typechecker such as MyPy⁹ would go a long way in enabling privacy-preserving data analytics in the wild. Entities such as SERVEL could adopt a Pandas library enhanced with privacy annotations to generate and publish charts and statistics about national elections, while respecting the privacy of individuals. Gradual typing would ease the progressive and selective adoption and addition of privacy annotations in such systems, providing early feedback to developers along the way.

Acknowledgements

This work counts with the collaboration of Damián ÁRquez (Jazz and GSoul) as well as David Darais, Joseph P. Near, and Chike Abuah (Jazz).

References

1. ÁRquez, D., Toro, M., and Tanter, É. *Gradual sensitivity typing* (Aug. 2023); 10.48550/arXiv.2308.02018.
2. Azevedo de Amorim, A., Gaboardi, M., Hsu, J., and Katsumata, S. Probabilistic relational reasoning via metrics. In *Proceedings of Symp. Logic in Computer Science* (Jun. 2019), 1–19; 10.1109/LICS.2019.8785715.
3. Barthe, G. et al. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *Proceedings of Symp. Principles of Programming Languages* (Jan. 2015), 15–17; 10.1145/2676726.2677000.
4. Barthe, G. et al. Proving differential privacy via probabilistic couplings. In *Proceedings of Symp. Logic in Computer Science* (Jul. 2016), 749–758; 10.1145/2933575.2934554.
5. Barthe, G., Köpf, B., Olmedo, F., and Zanella-Béguelin, S. Probabilistic relational reasoning for differential privacy. *Trans. Programming Languages and Systems* (Nov. 2013), 1–49; 10.1145/2492061.
6. Dwork, C. Differential privacy. In *Proceedings of Intern. Colloquium on Automata, Languages, and Programming* (July 2006), 1–12; 10.1007/11787006_1.
7. Gaboardi, M. et al. Linear dependent types for differential privacy. In *Proceedings of Symp. Principles of Programming Languages* (Jan. 2013), 357–370; 10.1145/2429069.2429113.
8. Lyu, M. et al. Understanding the sparse vector technique for differential privacy. In *Proceedings VLDB Endow. IQ*, 6 (Feb.2017), 637–648; 10.14778/3055330.3055331.
9. MyPy; <https://mypy-lang.org/>.
10. Near, J.P. et al. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. In *Proceedings ACM Program. Lang* (2019), 172:1–172:30; 10.1145/3360598.
11. Reed, J. and Pierce, B.C. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of Intern. Conf. Functional Programming* (Sep. 2010), 157–168; doi: 10.1145/1863543.1863568.
12. Sato, T. et al. Approximate span liftings: compositional semantics for relaxations of differential privacy. In *Proceedings of Symp. Logic in Computer Science* (Jun. 2019), 1–14; 10.1109/LICS.2019.8785668.
13. Siek, J. and Taha, W. Gradual typing for functional languages. *Scheme and Functional Programming* (Sept. 2006), 81–92; <http://scheme2006.cs.uchicago.edu/13-siek.pdf>.
14. *The Top Programming Languages*; <https://octoverse.github.com/2022/top-programming-languages>.
15. Toro, M. Técnicas formales de privacidad de datos: ¿Está el Serval protegiendo nuestra privacidad? *Bits de Ciencia* 22, (June 2022); <https://www.dcc.uchile.cl/difusion/revista/22>.
16. Toro, M. et al. Contextual linear types for differential privacy. *Trans. Programming Languages and Systems* (2023), 8:1–8:69; 10.1145/3589207

Matías Toro is an assistant professor in the Computer Science Department of the University of Chile. **Federico Olmedo** is an assistant professor in the Computer Science Department of the University of Chile, and Young Researcher of the Millenium Institute on Foundational Research on Data (IMFD), Chile.

Éric Tanter is a professor in the Computer Science Department of the University of Chile and associate researcher of the Millenium Institute on Foundational Research on Data (IMFD), Chile.
