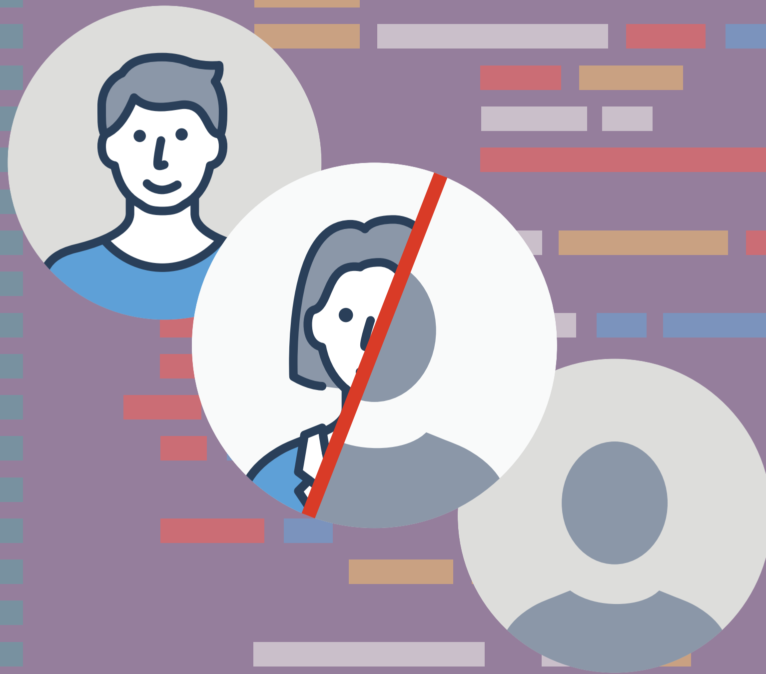
Mejorando la adopción de la privacidad diferencial en lenguajes de programación



DAMIÁN ARQUEZ

Estudiante de Doctorado en Computación de la Universidad de Chile. Magíster en Ciencias mención Computación por la Universidad de Chile. Líneas de investigación: lenguajes de programación, privacidad de datos.

darquez@dcc.uchile.cl

MATÍAS TORO

Profesor Asistente del Departamento de Ciencias de la Computación de la Universidad de Chile. Doctor en Ciencias mención Computación por la Universidad de Chile. Líneas de investigación: lenguajes de programación, privacidad de datos y desarrollo de software.

mtoro@dcc.uchile.cl

RESUMEN. Frente a la creciente cantidad de datos personales que se almacenan en bases de datos, la privacidad de los individuos se ha vuelto un tema de gran importancia. Una técnica que ha ganado mucha atención en los últimos años es la *privacidad diferencial*, basada en una definición formal de privacidad que garantiza que un atacante no pueda distinguir si un individuo participó o no en un conjunto de datos. A pesar de haber un interés en integrar privacidad diferencial en distintas áreas de la computación y estadística, no siempre es fácil hacerlo y mucho menos probar que un programa es, en efecto, diferencialmente privado. De hecho, existen muchos casos de publicaciones que, incluso siendo revisadas por expertos, fallan en implementar privacidad diferencial correctamente. Frente a esto, durante la última década, han existido esfuerzos por integrar privacidad diferencial en lenguajes de programación, con el fin de proveer garantías de privacidad de manera automática. En este artículo, presentamos Jazz, un lenguaje de programación que permite razonar de manera estática sobre privacidad diferencial. Además, abordamos el problema de la adopción de sistemas de tipos estáticos, proponiendo un sistema de tipos gradual que permita integrar privacidad diferencial de manera incremental en programas existentes. Nuestro primer paso en esta dirección es GSoul, un lenguaje de programación que permite razonar de manera gradual sobre sensibilidad de datos.

Esta investigación fue desarrollada por los autores del presente artículo, en colaboración con David Darais, Chike Abuah, Joseph P. Near, Federico Olmedo y Éric Tanter.

Existe una inherente tensión entre privacidad y precisión: no se puede lograr, simultáneamente, perfecta privacidad y precisión.

Los datos

La información almacenada en bases de datos crece cada día, especialmente debido a áreas emergentes como big data o machine learning. Todos estamos generando nuevos datos todo el tiempo: cuando salimos de compras, cuando escuchamos música o vemos películas, cuando vamos al médico, cuando usamos nuestra cuenta bancaria, etc. Estos datos proveen un sinfín de oportunidades: la personalización de servicios en Amazon o Netflix; construir modelos computacionales para automatizar tareas, como clasificadores de correos, o traductores de texto; apoyar el desarrollo científico; guiar la toma de decisiones públicas y mejorar su transparencia, etc. Los datos nunca habían tenido un rol tan importantes como ahora.

Privacidad

El uso de datos personales tiene también sus riesgos, ya que éstos podrían tener información sensible o confidencial de las personas que puede ser filtrada, por ejemplo, información médica, financiera, amorosa, de preferencias artísticas, etc. En otras palabras, datos privados que nos exponen y vulneran como personas. Esto puede sonar alejado de la realidad, pero a la fecha ya se han acumulado una considerable cantidad de casos emblemáticos en que la privacidad de individuos es violada, muchas veces sin querer o saber.

El primero es el caso de Netflix, que realizó en el 2007 una competencia abierta para mejorar su sistema de recomendación. El premio era un millón de dólares, y los participantes tenían acceso a un conjunto de datos de entrenamiento *anonimizado*, es decir, atributos identificadores como el nombre fueron eliminados. El problema se presentó luego de que al poco tiempo se público el conjunto de datos de Netflix de-anonimizado, al cruzar los datos con datos públicos de preferencias de películas de IMDB, se logró asociar nombres concretos con sus preferencias de Netflix.

Otro caso famoso fue en 1997, cuando William Weld, gobernador del estado de Massachusetts, aprobó la liberación de los registros médicos de funcionarios públicos, utilizando anonimización. Dos días después, Latanya Sweeney, en ese entonces una estudiante de doctorado del MIT, le envió un correo con todos sus registros médicos. ¿Cómo lo logró? Cruzó la información de registros médicos con el padrón electoral, usando el código postal, año de nacimiento y género como se ve en la Figura 1.

Casos como los anteriores, ha generado un gran interés en la comunidad científica respecto a técnicas que nos ayuden a preservar (y garantizar formalmente) la privacidad de los datos.

Técnicas de privacidad

Para prevenir violaciones a la privacidad existen dos grandes modelos. El primero y más conocido, es el basado en un conjunto predefinido de ataques. Decimos que la publicación de un conjunto de datos preserva privacidad si es inmune a ciertos ataques. La técnica que se usa para prevenir estos ataques es llamada anonimización pero, como vimos anteriormente, su efectividad está sujeta a la información auxiliar que pueda tener un atacante.

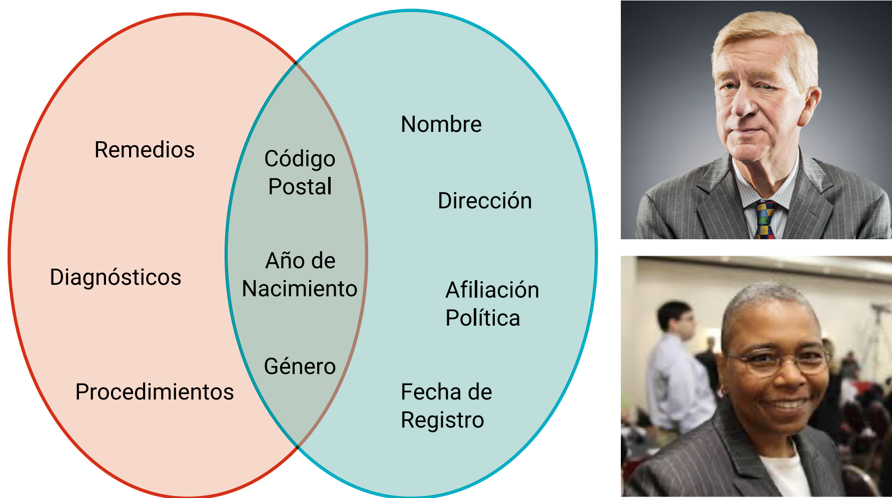


Figura 1. Latanya Sweeney cruzó información anonimizada de registros médicos con el padrón electoral utilizando el código postal, año de nacimiento y género, logrando asociar datos de medicamentos, diagnósticos y procedimientos médicos con el gobernador.

El segundo modelo está basado en el principio de la desinformación. Decimos que un conjunto de datos preserva la privacidad si al observarlo, un atacante gana muy poca información adicional respecto a la información que ya tenía. Aquí la técnica que se usa es llamada *privacidad diferencial* (DP, por sus siglas en inglés) [1], y es actualmente el enfoque más efectivo para lograr privacidad ya que es inmune a la información auxiliar que podría tener un atacante.

Privacidad diferencial

Privacidad diferencial es una técnica que garantiza privacidad a través de una definición formal basada en estadística. Se dice que una computación aleatoria es ϵ-DP, si para dos bases de datos que difieren en un individuo, los resultados de aplicar la computación aleatoria a ambas bases son “indistinguibles”. Formalmente, un mecanismo $\mathcal{M}$ es ϵ-DP si para cualquier par de bases de datos D y D' que difieran en un individuo, y para cualquier subconjunto S de los posibles resultados de $\mathcal{M}$, se cumple:

$$\frac{\Pr[\mathcal{M}(D) \in S]}{\Pr[\mathcal{M}(D') \in S]} \leq e^{\epsilon}$$

La técnica más común para implementar privacidad diferencial es agregar ruido aleatorio estadístico al resultado de una computación. El parámetro ϵ, a menudo llamado presupuesto de privacidad, controla el equilibrio entre privacidad y precisión: a menor ϵ, hay más ruido y por ende el cómputo es menos preciso pero más privado; análogamente, a mayor ϵ hay menos ruido y por ende el cómputo es más preciso pero menos privado. Existe una inherente tensión entre privacidad y precisión: no se puede lograr, simultáneamente, perfecta privacidad y precisión.

Sin embargo, no cualquier cantidad de ruido es adecuada: mucho ruido puede destruir completamente la utilidad de los datos. En el artículo seminal sobre DP [1], Cynthia Dwork probó que el ruido adecuado debe ser proporcional a la *sensibilidad* de la computación, que intuitivamente corresponde a cuánto puede la función magnificar la distancia entre dos entradas. Por ejemplo, una función que recibe y produce números reales es s-sensible si, para cualquier par de entradas x e y, se cumple:

$$|f(x) - f(y)| \leq s \cdot |x - y|$$

Si consideramos dos funciones $f(x) = x + 1$ y $g(x) = x + x$, utilizando la definición anterior, podemos deducir que f es 1-sensible y g es 2-sensible. Sabiendo la sensibilidad de una función, podemos crear una versión diferencialmente privada de esta agregando ruido aleatorio proporcional a su sensibilidad. Por ejemplo, $F(x) = f(x) + \mathrm{noise}(1)$ y $G(x) = g(x) + \mathrm{noise}(2)$ son versiones diferencialmente privadas de f y g, respectivamente. La forma más básica de agregar ruido es utilizando la distribución de Laplace, específicamente fijando $\mathrm{noise}(s) = \mathrm{laplace}(S/\epsilon)$.

La privacidad diferencial ha ganado mucha atención en organizaciones como Google [2], Apple [3], y el censo de Estados Unidos [4, 5]. Sin embargo, aún hay muchas preguntas y desafíos. Por ejemplo, ¿cuál es un valor de ϵ adecuado?, o ¿cómo podemos verificar que un computo más complejo realmente satisface ϵ-DP?

¿Cómo razonar acerca de DP?

Varios sistemas que realizan análisis de privacidad diferencial han demostrado ser efectivos [6–10], y en el campo del aprendizaje automático se han diseñado varios algoritmos utilizando privacidad diferencial [11–18].

Actualmente, existen dos principales enfoques basados en lenguajes de programación de propósito general para

razonar acerca de privacidad diferencial: aquellos basados en *sistemas de tipos* [19], y otros basados en *lógica de programas* [20, 21]. Aunque estos últimos son más expresivos, en general son más difíciles de automatizar y mucho mas complejos que los sistemas de tipos. Nuestro enfoque, en cambio, busca proveer herramientas automáticas y ligeras (*lightweight* en inglés) de verificación de privacidad diferencial, por lo que nos orientamos hacia sistemas de tipos.

Los sistemas de tipos ayudan a los programadores a verificar automáticamente un programa *antes de su ejecución*, asegurando que ciertas clases de errores nunca ocurran en tiempo de ejecución. Por ejemplo, un verificador de tipos básico prevendría que el programa `1 + True` llegase a ejecutarse, ya que no tiene sentido sumar un número con un Booleano. Durante la compilación de un programa, un verificador de tipos intenta clasificar las expresiones del programa en términos del tipo de valores que producen, y si no puede hacerlo, informa sobre un error de tipos estático. Uno de los principales beneficios de utilizar esta técnica es que los tipos son esenciales en el diseño de sistemas modulares. Aplicado a privacidad diferencial, nos permite razonar de manera composicional sobre esta disciplina.

Nuestro enfoque [...] busca proveer herramientas automáticas y ligeras [...] de verificación de privacidad diferencial, por lo que nos orientamos hacia sistemas de tipos.

Fuzz

En 2010, se publica el primer lenguaje diseñado precisamente para razonar sobre sensibilidad y privacidad diferencial: Fuzz [19]. La idea principal se funda en que el desarrollo de programas se lleve a cabo desde un principio en torno a la noción de privacidad y, por lo tanto, cualquier programa escrito en Fuzz es, por construcción, diferencialmente privado. Fuzz logra proveer esta garantía a través de un sistema de tipos que permite razonar sobre la sensibilidad de los programas y expresiones con ruido. Si un programa es válido en Fuzz, entonces no sólo sabremos los tipos de datos involucrados, sino que también podemos saber la sensibilidad de cada expresión.

Los autores de Fuzz diseñan el lenguaje con una idea clave: la sensibilidad es crucial para la privacidad diferencial. Por lo mismo, el sistema de tipos de Fuzz se inspira en una técnica llamada *tipos lineales* [22] para modelar un seguimiento de la sensibilidad de un programa. Dado el pequeño núcleo de funcionalidades de Fuzz, el sistema de tipos aproxima la sensibilidad al número de veces que una variable es usada en un programa.[1] Por ejemplo, en Fuzz existen dos formas de darle un tipo a una función: $\mathbb{R} \to \mathbb{R}$ describe una función que recibe un número real como argumento, retorna un número real, y utiliza sin restricciones su argumento, i.e. una función ∞-*sensible en su*

1 Esto no siempre es posible pues, por ejemplo, en la expresión $x * x$, x se utiliza 2 veces pero la sensibilidad es infinita. Existen otros mecanismos para razonar sobre la sensibilidad de tales operaciones, pero escapa del alcance de Fuzz.

La idea principal se funda en que el desarrollo de programas se lleve a cabo desde un principio en torno a la noción de privacidad y, por lo tanto, cualquier programa [con el tipo adecuado, corroborado por el verificador de tipos del lenguaje, resulte], por construcción, diferencialmente privado.

argumento; en contraste, $\mathbb{R} \multimap \mathbb{R}$ denota una función de reales a reales, pero que utiliza su argumento en un factor de 1, i.e. una función 1-*sensible en su argumento*. Por ende, la función $f(x) = x + 1$ puede no sólo tener tipo $\mathbb{R} \rightarrow \mathbb{R}$ sino que podemos ser más precisos y asignarle el tipo $\mathbb{R} \multimap \mathbb{R}$, estableciendo además su sensibilidad. Por otro lado, si quisiéramos asignarle el tipo $\mathbb{R} \multimap \mathbb{R}$ a la función $g(x) = x + x$, el verificador de tipos de Fuzz nos daría un error de tipos, ya que g es de hecho 2-sensible en su argumento.

Sumado al mecanismo para razonar sobre la sensibilidad de los programas, Fuzz también provee un tipo para computaciones ruidosas: $\bigcirc\mathbb{R}$ representa una distribución de probabilidad sobre reales. Por ejemplo, el tipo de una función `add_noise`, 1-sensitiva, y que recibe un real y produce un real añadiendo ruido según la distribución de Laplace con escala $1/\epsilon$, sería el siguiente:

```
add_noise: ℝ⊸○ℝ
```

Supongamos que tenemos la siguiente función, que recibe una base de datos y computa cuántas personas tienen más de 40 años:

```
over_40: db⊸ℝ
```

Podemos usarla para computar una versión ruidosa de la siguiente manera, en un lenguaje *à la* Python:

```
over_40_DP: db⊸○ℝ
def over_40_DP(d):
  add_noise(over_40(d))
```

De esta manera habremos creado una computación que *por construcción* es diferencialmente privada. De hecho, en Fuzz podemos demostrar matemáticamente que cualquier función aleatoria que tenga el tipo $db \multimap \bigcirc\mathbb{R}$ es ϵ-DP.

Limitaciones

La creación de Fuzz desencadenó una serie de investigaciones en torno a la privacidad diferencial y los sistemas de tipos [23]. Sin embargo, Fuzz tiene algunas limitaciones que lo hacen poco práctico para su uso en el mundo real. En primer lugar, Fuzz sólo soporta la variante más básica de privacidad diferencial, es decir, ϵ-DP. Otras variantes más avanzadas como (ϵ,δ)-DP, Renyí-DP, o *zero concentrated* DP no son soportadas. Además, Fuzz, y otros lenguajes derivados, en muchas ocasiones son por defecto demasiado conservadores sobre la sensibilidad de programas. Si bien esta limitación se puede mitigar a través de otros mecanismos del lenguaje, requiere que el programador tenga un conocimiento no trivial sobre la sensibilidad de su programa.

Jazz

Con el fin de hacernos cargo de estas limitaciones, y a su vez mejorar la adopción de privacidad diferencial en los lenguajes de programación, diseñamos el lenguaje Jazz [24]. Jazz es un lenguaje que sigue el espíritu de Fuzz, pero con un análisis de sensibilidad más preciso, sin sacrificar simplicidad. Además, Jazz está diseñado para soportar variantes avanzadas de DP en conjunto con programación de alto orden, donde funciones pueden tomar otras funciones como argumento.

Jazz está compuesto de dos sublenguajes mutuamente definidos: uno para razonar acerca de sensibilidad (Sax), y otro acerca de privacidad (λ_J).

El sublenguaje de sensibilidad. Sax permite expresar explícitamente la sensibilidad de cada variable de un programa directamente en los tipos. Por ejemplo, en Sax, la función $g(x) = x + x$ tiene tipo $(x : \mathbb{R}) \xrightarrow{2x} \mathbb{R}$, que describe a una función de reales a reales donde su *sensibilidad latente* es $2x$. Esto significa que si variamos el argumento (x) en 1, amplificaremos el resultado del cálculo en 2. La novedad radica en el término “latente”, que implica que esta sensibilidad sólo afecta al resultado final del programa cuando la expresión se aplica. Por ejemplo, considere la siguiente función:

```
def f(y):
  def g(x):
    x + y
  y + 1
```

En Sax, la sensibilidad de `f` es 1 respecto a y, dado que el cálculo final de `f` no depende de `g`. Otra característica innovadora de Sax es que permite razonar sobre varias variables simultáneamente. Por ejemplo, la función $h(x,y) = x + x + y$ tiene sensibilidad latente $2x + y$, notación para decir que es 2-sensible en x y 1-sensible en y.

Este concepto de sensibilidad latente también se aplica a los pares. Un tipo par tiene asociado una sensibilidad latente específica para cada componente, y, en consecuencia, al acceder a un componente es que se contribuye a la sensibilidad final. Por ejemplo, el par $(2{*}x,\ x)$ tiene sensibilidades latentes $2x$ y $1x$ en el primer y segundo componente respectivamente, y por

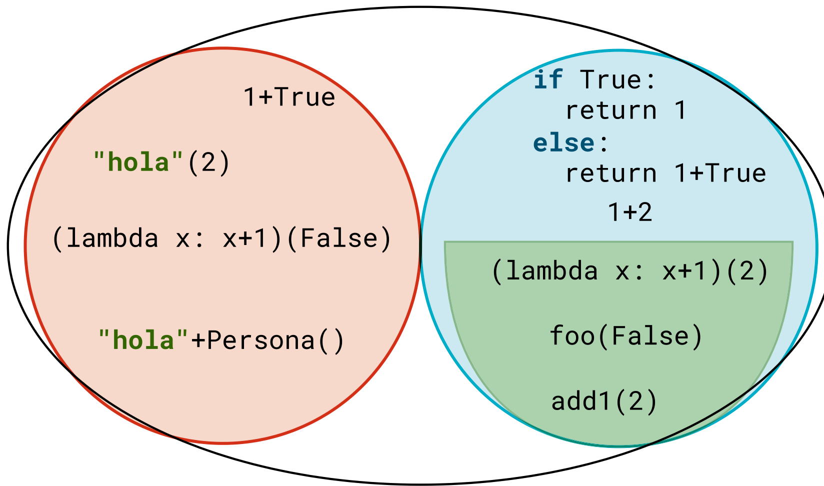


Figura 2. Los programas que funcionan correctamente se encuentran en azul, mientras que los que funcionan incorrectamente están en rojo. En verde se encuentran los programas aceptados por un sistema de tipos correcto, que aproxima de manera *conservadora* los programas que funcionan bien. La consecuencia de esto es que algunos programas que efectivamente funcionan bien, serán rechazados por el sistema de tipos (pero nunca aceptará un programa que definitivamente funcione mal).

consecuencia tiene tipo $\mathbb{R}^{2x} \times {}^{1x}\mathbb{R}$. Esto conduce a un análisis inherentemente más preciso, como se puede observar en el siguiente ejemplo:

```
def f(x):
  p = (2*x,x)
  if(b):
    3*p[0]
  else:
    2*(p[0] + p[1])
```

Primero, podemos notar que el cuerpo de la función es equivalente a escribir `6*x`, es decir, la función es 6-sensible en `x`. En lenguajes como Fuzz, el par `p` se clasifica como 2-sensible en x (que es el máximo de las sensibilidades de sus componentes). Luego, la expresión `if` toma, de manera pesimista, el valor máximo de las sensibilidades de ambas ramas. En este caso, la rama `else` es clasificada como 8-sensible en `x` $(8 = 2{\cdot}(2{+}2))$.[2] En contraste, en Jazz, gracias a los efectos latentes, la sensibilidad de la primera rama es $6x$ (de escalar por 3 la sensibilidad latente del primer component $2x$). De manera similar, la sensibilidad de la rama `else` es calculada como $6x$ también, al multiplicar por 2 la suma de las sensibilidades latentes $2x$ y $1x$. Como resultado, Jazz informa de manera precisa una sensibilidad de $6x$.

El sublenguaje de privacidad. Similar a Sax, λ_j permite razonar acerca de la privacidad de cada variable. Para ello, podemos utilizar *funciones privadas* como $(x:\mathbb{R}\cdot d) \xrightarrow{\epsilon x} \mathbb{R}$. Esta función representa una función de reales a reales, cuyo nivel de privacidad es ϵ-DP respecto a su argumento x, y la *distancia relacional* de dicho argumento puede ser como máximo d. La distancia relacional cuantifica el grado de variación que el argumento puede experimentar en dos ejecuciones diferentes, con un valor comúnmente definido como $d = 1$. Este valor viene de la definición de DP, donde se compara la ejecución del mecanismo aleatorio con dos bases de datos que difieren en 1 individuo. Para ejemplificar, podemos definir la función `add_noise` que introduce ruido aleatorio basado en la distribución de Laplace de la siguiente manera:

```
add_noise: (x:ℝ·1) -εx-> ℝ
def add_noise (x):
  x + Laplace(1/ϵ)
```

Posteriormente, el sistema de tipos garantiza que en situaciones como `add_noise(y+z)`, la distancia relacional máxima de cada variable no exceda el valor de 1.

Finalmente, establecimos la metateoría de Jazz, demostrando que programas bien tipados no fallan en ejecución, y que además satisfacen la formulación formal de privacidad diferencial, propiedades conocidas respectivamente como *type safety* y *type soundness*.

Hacia gradual DP

Aunque ya hemos visto las ventajas que nos pueden brindar los sistemas de tipos, como los de Fuzz o Jazz, estos por definición son una aproximación conservadora del comportamiento de programas, como se puede ver en la Figura 2. En general, es imposible diseñar un sistema de tipos que pueda distinguir con total certeza si un programa violará propiedades de sus tipos (esto es equivalente al *halting problem*). Esto significa que pueden rechazar programas que podrían funcionar correctamente, como por ejemplo:

2 En muchas ocaciones sí se puede lograr un análisis preciso en Fuzz, pero hay que utilizar mecanismos complejos como el escalamiento de métricas.

```
...
if True:
  return 1
else
  return 1+True
```

En programas más complejos, esto puede ser frustrante para desarrolladores, ya que pueden llevarse la impresión de que el sistema de tipos interfiere con la velocidad y experiencia de desarrollo, llevándolos a utilizar, en cambio, algún lenguaje *dinámicamente tipado* como Python, con menos garantías estáticas, pero más rápido para prototipado. Adicionalmente, un programador que quiera incorporar un sistema de tipos se podría enfrentar a una ardua transición, usualmente forzados a refactorizar código que ya funciona sólo para satisfacer al sistema de tipos.

El *tipado gradual* busca solucionar este tipo de problemas, soportando una transición fluida entre chequeo de tipos estático y dinámico. Consecuentemente, esta técnica permite a programadores dejar ciertas partes del código sin información de tipos o incluso con información de tipos parcial. Luego, el sistema de tipos hace lo mejor que puede (de manera optimista) durante la fase de compilación, y retrasa lo que no puede verificar inmediatamente en tiempo de ejecución. Durante la ejecución, si alguna garantía estática se viola, el programa falla con un error.

En vías de mejorar la adopción de conceptos de privacidad para programadores, nuestro principal objetivo a futuro consiste en el desarrollo de un lenguaje gradual que permita razonar sobre sensibilidad y privacidad. Por ejemplo, el siguiente programa no es aceptado por un sistema de tipos estático y sí debería serlo por un sistema de tipos gradual:

```
x = ...
b = ...
def foo(y):
  if b:
    return y
  else:
    return y*y
add_noise (foo(x))
```

Recordemos que `add_noise` sólo funciona si el argumento es a lo más 1-sensible. Un sistema de tipos estático, de manera conservadora, rechazaría este programa porque el cuerpo del condicional es ∞-sensible en `x` (se toma el máximo de ambas ramas). Con un sistema de tipos gradual, podemos declarar que el tipo de retorno de `foo` es desconocido. En este caso, si `b=True` entonces el programa debería correr con éxito. En cambio si `b=False` debería fallar con un error en ejecución. Ya hemos dado los primeros pasos al diseñar GSoul [25], el primer lenguaje gradual para razonar acerca de la sensibilidad de computaciones. Cómo extender GSoul para incluir privacidad diferencial es en estos momentos un tema activo de investigación y nuestro foco más inmediato. ■

REFERENCIAS

[1] Cynthia Dwork. Differential privacy. pages 1–12.

[2] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1054–1067.

[3] [n. d.]. Apple previews ios 10, the biggest ios release ever. https://www.apple.com/newsroom/2016/06/apple-previews-ios-10-biggest-io.

[4] Samuel Haney, Ashwin Machanavajjhala, John M. Abowd, Matthew Graham, Mark Kutzbach, and Lars Vilhuber. Utility cost of formal privacy for releasing national employer-employee statistics. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1339–1354.

[5] Ashwin Machanavajjhala, Daniel Kifer, John M. Abowd, Johannes Gehrke, and Lars Vilhuber. Privacy: Theory meets practice on the map. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 277–286.

[6] Noah M. Johnson, Joseph P. Near, and Dawn Song. Towards practical differential privacy for SQL queries. *PVLDB*, 11(5):526–539, 2018.

[7] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David E. Culler. GUPT: privacy preserving data analysis made easy. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 349–360.

[8] Arjun Narayan and Andreas Haeberlen. Djoin: Differentially private join queries over distributed databases. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 149–162.

[9] Davide Proserpio, Sharon Goldberg, and Frank McSherry. Calibrating data to sensitivity in private data analysis. *PVLDB*, 7(8):637–648, 2014.

[10] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 297–312.

[11] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 308–318.

[12] Raef Bassily, Adam D. Smith, and Abhradeep Thakurta. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 464–473.

[13] Kamalika Chaudhuri, Claire Monteleoni, and Anand D. Sarwate. Differentially private empirical risk minimization. *J. Mach. Learn. Res.*, 12:1069–1109, 2011.

[14] Arik Friedman, Shlomo Berkovsky, and Mohamed Ali Kâafar. A differential privacy framework for matrix factorization recommender systems. *User Model. User-Adapt. Interact.*, 26(5):425–458, 2016.

[15] Nicolas Papernot, Martín Abadi, Úlfar Erlingsson, Ian J. Goodfellow, and Kunal Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

[16] Shuang Song, Kamalika Chaudhuri, and Anand D. Sarwate. Stochastic gradient descent with differentially private updates. In *IEEE Global Conference on Signal and Information Processing, GlobalSIP 2013, Austin, TX, USA, December 3-5, 2013*, pages 245–248.

[17] Kunal Talwar, Abhradeep Thakurta, and Li Zhang. Nearly optimal private LASSO. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3025–3033.

[18] Xi Wu, Fengan Li, Arun Kumar, Kamalika Chaudhuri, Somesh Jha, and Jeffrey F. Naughton. Bolt-on differential privacy for scalable stochastic gradient descent-based analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1307–1322.

[19] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 157–168, Baltimore, Maryland, USA, September 2010. ACM Press.

[20] Gilles Barthe, George Danezis, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. Verified computational differential privacy with applications to smart metering. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 287–301.

[21] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. Relational reasoning via probabilistic coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 387–401.

[22] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[23] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[24] Matías Toro, David Darais, Chike Abuah, Joe Near, Damián Árquez, Federico Olmedo, and Éric Tanter. Contextual linear types for differential privacy. *ACM Transactions on Programming Languages and Systems*, 2023. To appear. https://arxiv.org/abs/2010.11342.

[25] Damián Árquez, Matías Toro, and Éric Tanter. Gradual sensitivity typing. *CoRR*, abs/2308.02018, 2023.