10

A Framework for Adaptive Differential Privacy

DANIEL WINOGRAD-CORT, University of Pennsylvania, USA
ANDREAS HAEBERLEN, University of Pennsylvania, USA
AARON ROTH, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

Differential privacy is a widely studied theory for analyzing sensitive data with a strong privacy guarantee—any change in an individual's data can have only a small statistical effect on the result—and a growing number of programming languages now support differentially private data analysis. A common shortcoming of these languages is poor support for *adaptivity*. In practice, a data analyst rarely wants to run just one function over a sensitive database, nor even a predetermined sequence of functions with fixed privacy parameters; rather, she wants to engage in an interaction where, at each step, both the choice of the next function and its privacy parameters are informed by the results of prior functions. Existing languages support this scenario using a *simple composition theorem*, which often gives rather loose bounds on the actual privacy cost of composite functions, substantially reducing how much computation can be performed within a given privacy budget. The theory of differential privacy includes other theorems with much better bounds, but these have not yet been incorporated into programming languages.

We propose a novel framework for adaptive composition that is elegant, practical, and implementable. It consists of a reformulation based on typed functional programming of *privacy filters*, together with a concrete realization of this framework in the design and implementation of a new language, called *Adaptive Fuzz*. Adaptive Fuzz transplants the core static type system of Fuzz to the adaptive setting by wrapping the Fuzz typechecker and runtime system in an outer *adaptive layer*, allowing Fuzz programs to be conveniently constructed and typechecked on the fly. We describe an interpreter for Adaptive Fuzz and report results from two case studies demonstrating its effectiveness for implementing common statistical algorithms over real data sets.

CCS Concepts: • **Theory of computation** → **Theory of database privacy and security**; *Operational semantics*; • **Software and its engineering** → **Semantics**;

Additional Key Words and Phrases: Differential Privacy, Adaptivity, Privacy Filter, Fuzz, Case Study

ACM Reference Format:
Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A Framework for Adaptive Differential Privacy. *Proc. ACM Program. Lang.* 1, ICFP, Article 10 (September 2017), 29 pages.
https://doi.org/10.1145/3110254

1 INTRODUCTION

Computing with sensitive data always involves a compromise between releasing useful results on one hand, and protecting the privacy of individuals on the other. Among the numerous proposals for negotiating this inherent tension, *differential privacy* [Dwork et al. 2006b], with its firm statistical foundation and minimal assumptions on attacker knowledge, is appealingly resilient to the vulnerabilities that have plagued other mechanisms [Mills 2006; Narayanan and Shmatikov 2008;

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2017 Copyright held by the owner/author(s).
2475-1421/2017/9-ART10
https://doi.org/10.1145/3110254

Singel 2009, etc.]. It has become a topic of intense interest in the algorithms community [Dwork and Roth 2014], with applications ranging from data mining [McSherry and Mironov 2009] and medical research [Johnson and Shmatikov 2013] to collecting browser statistics in a deployed system at Google [Erlingsson et al. 2014]. Because the theory is somewhat subtle and typically requires expert training to put into practice correctly, researchers have created a number of tools to assist non-experts in using it [Barthe et al. 2012; Gaboardi et al. aper; Haeberlen et al. 2011; McSherry 2009; Narayan and Haeberlen 2012; Reed and Pierce 2010; Roy et al. 2010]. Many take the form of programming languages.

The most common presentation of differential privacy divides the world into a *curator* who is responsible for protecting the privacy of some sensitive *database*, and an *analyst* who wants to extract information from this database by asking the curator to evaluate probabilistic functions against it and return their (sampled) results. Each of these functions must be (ε, δ)-*differentially private*, where the parameters ε and δ quantify the amount of privacy lost when running this function.[1] We refer to these functions as *pieces*, to emphasize the fact that the analyst generally wants to combine many of them while performing some larger data analysis. (Coining the new term "piece" also avoids misunderstandings that may arise from existing terms like "query," which have divergent meanings in various communities.)

For each new piece proposed by the analyst, the curator must decide whether the *aggregate* privacy loss resulting from answering this piece together with the already-released results of previous pieces will exceed a global *privacy budget* $(\varepsilon_g, \delta_g)$ associated with the database. Formally, the curator needs to check that the *composite piece* formed from all previous pieces plus this one is $(\varepsilon_g, \delta_g)$-differentially private. The differential privacy literature provides many tools to help carry out this check, ranging from simple [Dwork et al. 2006a,b] to sophisticated [Dwork et al. 2010; Kairouz et al. 2015; Murtagh and Vadhan 2016], each with its own strengths and weaknesses.

More sophisticated tools often allow a user to run many more pieces within the same budget than would be possible with simpler tools. However, until recently, most advanced tools did not support *parameter adaptive* choice of pieces—in other words, the ε and δ parameters of the individual pieces could not depend on the results from previous pieces and instead had to be fixed up front. This explains why programming languages for differential privacy have only used the simplest tools until now. However, the recent introduction of a new set of theoretical tools for parameter adaptivity called *privacy filters* [Rogers et al. 2016] opens the door to programming languages for differential privacy that can execute vastly more pieces with the same budget.

Example. Suppose a curator has assembled a database of census data for a million people, each represented as a record of 146 features. He sets the total privacy budget to $(\varepsilon_g, \delta_g) = (2^{-1}, 2^{-30})$. The ε parameter indicates that he considers it acceptable that an adversary, after seeing the output, will have at most a $e^{2^{-1}} = \sqrt{e} \approx 1.65\times$ advantage over random guessing[2] in determining whether the analysis was run with or without a given individual's specific row of data, and the δ parameter essentially bounds the chance that the output gives a larger advantage to about one in a billion. The analyst would like to use this data to generate a predictor for one of these features—whether the person makes more than \$50,000 annually—based on the values of the other 145.

[1] Because our work focuses on cases where $\delta > 0$, it lives in the realm of *approximate* differential privacy, which provides a somewhat weaker guarantee than the $\delta = 0$ *pure* differential privacy.

[2] If the probability that an individual is in the dataset is p, then, before observing the outcome, no method an adversary might use to guess the truth will be correct with probability $\geq p$. Differential privacy promises that, even after the adversary observes the outcome, no method he might employ will correctly guess the truth with probability greater than $\approx 1.65 \cdot p$.

The analyst suspects that a good predictor for this feature can be made by finding some set of linear relationships between features, so she decides to use a differentially private gradient-descent algorithm, which can generate a linear polynomial predictor by iteratively deducing better coefficients, asymptotically approaching a best fit. Each round of the algorithm requires an $(\varepsilon, 0)$-piece for each of the features in the database (where ε can be "tuned" based on desired accuracy and size of the database—the optimal choice of ε in practice is an empirical question [Hsu et al. 2014], requiring running experiments of the sort that we describe in §5). For her particular needs, the analyst chooses $\varepsilon = 2^{-11}$, and she would like to continue performing rounds of the algorithm until the per-round gain in predictor accuracy falls below a certain threshold.

The most basic tool for analyzing a sequence of pieces is the *simple composition theorem*, which reports the aggregate cost of the sequence as the sum of the costs of the pieces; if the curator uses this, he will permit exactly 1024 pieces (because $2^{-11} \times 1024 = 2^{-1}$), allowing the analyst only 7 full rounds of her gradient descent algorithm (in aggregate, about a $(.495, 0)$-piece, since $145 \times 7 \times 2^{-11} \approx 0.495$). This is likely not enough, and because the curator must discard the database once its budget is exhausted, it means any further computation—by *any* analyst, not just this one—will require performing another census!

Another option for the curator is to use the *advanced composition theorem*, which uses the δ portion of the budget (2^{-30} in this case) to allow the ε cost of a sequence of pieces to grow much more slowly. Using this theorem, the curator could allow about 24,000 pieces to be run, which would permit well over 150 rounds of gradient descent, but it can only be used if the analyst declares how many pieces she intends to run before she starts running them. In this case, she would only be allowed to run those 150 rounds if she decided to use her entire budget on gradient descent before seeing any of its results, which means that even if the algorithm converged to a result in many fewer rounds, the rest of the budget would be wasted. Estimating how many pieces she will need in advance is tricky, considering she will not know when to stop until she starts seeing intermediate results from the algorithm, and while overestimating wastes the budget, underestimating means incurring a high budget cost to continue. For this reason, most languages do not even bother to support advanced composition tools. (Psi [Gaboardi et al. aper] is a notable exception, which we discuss in §7.)

To solve this problem, the curator can use privacy filters, which allow pieces to be *adaptively composed*, meaning that subsequent pieces can be progressively checked against the budget instead of needing to be declared up front. Indeed, if the curator uses the *advanced privacy filter*, the analyst can perform over 10,000 $(2^{-11}, 0)$-pieces, and, when she decides to stop, any remaining budget is preserved for use on future pieces that may have different privacy parameters. This freedom comes with some cost in ε over the advanced composition theorem (10,000 pieces instead of 24,000 in this example), because the best known theoretical analysis of adaptive privacy filters involves higher constant factors, but it still represents more than a ten-fold improvement compared to simple composition.

Contributions. We present the formal foundation, design, implementation, and evaluation of the first differential privacy language to support advanced adaptive composition: *Adaptive Fuzz*. To begin, we propose a new semantic presentation of adaptive composition. The existing model of Dwork et al. [2010] is phrased in terms of total but not necessarily computable functions, whereas in reality the curator, the analyst, and the pieces proposed by the analyst are all computable functions that may sometimes diverge. Our presentation also cleanly distinguishes the roles of curator and analyst by re-casting Dwork et al. [2010]'s model in the setting of typed functional programming over streams.

Next, we sketch the design of a new programming language that implements this semantics. This language, Adaptive Fuzz, supports two different *modes* of programming. Functions in the *data mode* represent individual pieces or their sub-functions; they are given direct access to the privacy-sensitive database, and must therefore be proven to be differentially private. The *adaptive mode* is used to assemble and combine individual data-mode pieces; successive pieces are constructed in light of the results from executing earlier pieces. This mode-based design allows pieces to be constructed adaptively and then proven private in between rounds of adaptivity by using a static system in a *piecewise static* way.

We illustrate the programming style supported by Adaptive Fuzz by showing implementations of two common statistical algorithms—gradient descent and stagewise regression—and measuring the results of executing them on a million-entry database of U.S. census data. These case studies demonstrate the gains that advanced privacy tools have over simple ones. In general, we see effective gains of ten times as many pieces over simple tools (up to 40× in some cases), meaning that Adaptive Fuzz programs can achieve the same accuracy with a significantly smaller budget than the same algorithms coded in other differentially private languages, or they can achieve higher accuracy with the same budget.

In summary, our main contributions are:

- We offer a novel semantic presentation of adaptive differential privacy, transposing the *privacy filters* of Rogers *et al.* to the world of typed functional programs over streams (§3). Our presentation cleanly separates the roles of curator and analyst and enriches the original total-functions-only account to deal with the possibility of divergent subcomputations.
- On the basis of this semantic account, we design an adaptively private programming language, Adaptive Fuzz, using the static type system of Fuzz together with an outer "adaptive layer" in which pieces are adaptively constructed and typechecked on the fly (§4). We prove that Adaptive Fuzz programs fit the requirements of §3 and are thus guaranteed to be differentially private.
- We demonstrate that Adaptive Fuzz achieves better privacy budget performance than languages based on simple composition by implementing and measuring two common statistical algorithms, gradient descent and stagewise regression (§5).

We conclude with discussions of limitations (§6), related work (§7), and future work (§8). This short version of the paper omits proofs and some extended figures and data. An extended version can be found on our web pages.

2 BACKGROUND

Differential Privacy is a notion of algorithmic stability introduced by Dwork et al. [2006b]. This section recapitulates key definitions and theorems from the literature presented in their original form, modulo trivial notational changes (Dwork and Roth [2014] provide a more thorough introduction); in particular, "functions" in this section should be understood as arbitrary total functions, rather than computable partial functions.

Probabilities. A *probability distribution* π over X (written $\pi : \bigcirc X$) is a function f from reals on the interval $[0, 1]$ to values of type X where the probability of x is the integral of its indicator function: $\int_0^1 \mathbf{1}_x(f\ r)\ dr$. (Here $\mathbf{1}_x$, the *indicator function* for x, returns 1 when its argument is x and 0 otherwise. Thus, the integral indicates how much of the domain $[0, 1]$ causes f to output x.)[3] We

[3] A simpler approach, such as taken by Reed and Pierce [2010], is to represent probability distributions as lists of pairs (p_i, v_i) where the probability of the distribution taking the value v_i is p_i. However, this representation has the unintended effect that, during evaluation, if one possible output of a distribution diverges, then the entire distribution diverges as well.

write $\Pr_{x \sim \pi}[x \in E]$ for the probability of event E when x is sampled according to π. A *randomized function* is a function that yields a probability distribution. Probability distributions form a monad

$$\begin{aligned} \pi \gg= f &= \lambda r.\ \mathbf{let}\ (r_1, r_2) = \mathit{split}\ r\ \mathbf{in}\ f\ (\pi\ r_1)\ r_2 \\ \mathit{return}\ v &= \lambda r.\ v \end{aligned}$$

where *split* is a surjective function that generates two real numbers from one in such a way that, if its inputs are randomly distributed, then its outputs are too (e.g., by constructing the first one using the odd digits of its input and the second using the even digits). We use this notation for defining operations involving distributions.

As an example, the distribution representing a fair coin could be written as

$$\mathit{fairCoin} \;=\; \lambda r.\ \mathbf{if}\ r > 0.5\ \mathbf{then}\ \text{“Heads”}\ \mathbf{else}\ \text{“Tails”}$$

and *fairCoin* would have the type $\bigcirc \mathit{String}$.

Differential Privacy. Informally, differential privacy requires that small changes to the input of an algorithm induce only small changes to the distribution of its outputs. Formally, these small changes are measured using *metrics* $d_X : X \times X \to \mathbb{R}_{\geq 0}$ for both the input and outcome spaces. The input space represents the sensitive data and is typically a multiset of “rows” of data, each representing a single individual. Its metric is hamming distance, where the distance between two multisets is the number of rows that need to be added or removed to turn one into the other. The output space is typically $\mathbb{R}$ with its usual metric. For all types X, two values $x, x' : X$ are *neighbors* if $d_X(x, x') \leq 1$.

Definition 2.1. *A randomized function* $\phi : D \to \bigcirc X$ *is* (ε, δ)-differentially private *if, for every pair of neighboring data sets* $d, d' \in D$ *and for every event* $S \subseteq X$,

$$\Pr_{o \sim \phi(d)}[o \in S] \;\leq\; e^{\varepsilon} \cdot \Pr_{o \sim \phi(d')}[o \in S] + \delta.$$

That is, for any set S of possible outputs, the probability that $\phi(d)$ will yield an output in S differs from the probability that $\phi(d')$ will do so by at most a multiplicative factor of e^{ε} with an additive offset of δ. This guarantee also means roughly that either e^{ε} is a multiplicative bound on the probability difference (with probability $(1 - \delta)$), or the difference may be arbitrarily large (with probability δ).[4] Thus, as long as the probability of an event is non-zero, then as ε grows toward infinity, the allowable difference in the probabilities grows as well; of course, if the event has zero probability in d, then its probability in d' is entirely constrained by δ. We usually use the shorter name *piece* or (ε, δ)*-piece* to refer to an (ε, δ)-differentially private algorithm.

To keep his database entirely private, a curator might think to choose $\varepsilon = \delta = 0$; this would prevent any chance of leaking private data to the analyst, but it also restricts the analyst from learning *anything* about the database. Instead, a typical curator must compromise: he does not want the analyst learning an arbitrary amount very often, so he restricts δ to be very small, e.g. 2^{-30}, but he allows ε to be large enough to permit useful results, e.g. 2^{-1}.

Although our approach (distributions as functions) means that there is no unique representation for a distribution, it permits enough laziness to prevent partial distributions from diverging.

[4] This description is “rough” because it actually describes a property called “pointwise indistinguishability” rather than differential privacy. Precisely, a function ϕ is (ε, δ)*-pointwise indistinguishable* over its input if, with probability $1 - \delta$, it satisfies $\log(\Pr[\phi(d) = x]/\Pr[\phi(d') = x]) \leq \varepsilon$. This easily implies (ε, δ)-differential privacy, but the reverse direction is true only up to a loss in parameters (as Kasiviswanathan and Smith [2014] show, (ε, δ)-differential privacy implies $(2\varepsilon, \delta/(\varepsilon \cdot e^{\varepsilon}))$-pointwise indistinguishability). Thus, our description is roughly valid because the *notion* of pointwise indistinguishability carries over to differential privacy.

Composition. Differential privacy is robust to both *post-processing* and *composition*, properties that support modular programming. Informally, *post-processing*—database-independent computation over the outcome of a differentially private computation—does not degrade privacy.

Lemma 2.2 (Robustness to Post-processing, Dwork et al. 2006b). *Let $\phi : D \to \bigcirc X_1$ be any (ε, δ)-piece, and let $f : X_1 \to \bigcirc X_2$ be an arbitrary randomized function. Then*

$$\lambda d.\ \phi(d) \gg= f : D \to \bigcirc X_2$$

is (ε, δ)-differentially private.

The *composition* of multiple pieces is itself always a piece (i.e., the composite is also differentially private), with well-behaved degradation of the privacy parameters. Indeed, a number of different "composition theorems" can be found in the literature, each specifying a different way of calculating a bound on the privacy loss from a composite piece. The most commonly used are *simple composition* and *advanced composition.*

Definition 2.3. *A sequence of k algorithms $\phi_0, \ldots, \phi_{k-1}$ (written $\overline{\phi}^k$) with types $\phi_i : D \times X^i \to \bigcirc X$, is a* composable sequence. *The k-*fold composition *of $\overline{\phi}^k$ is the algorithm $\phi : D \to \bigcirc(X^k)$ defined as $\phi(d) = loop\ 0\ nil$, where*

$$loop\ i\ \sigma \quad = \quad \begin{cases} return\ \sigma & \text{if } i \geq k \\ \phi_i(d, \sigma) \gg= (\lambda x.\ loop\ (i+1)\ (\sigma :: x)) & \text{if } i < k, \end{cases}$$

nil is the empty sequence, and $\sigma :: x$ and $x :: \sigma$ are notations for attaching an element x to the end or the beginning of a sequence σ respectively. We will use the notation $\overline{o}^i$ to represent sequences of realized outcomes $(o_0, \ldots, o_{i-1}) \in X^i$ *produced by the first i pieces in the sequence.*

Note that we force each ϕ_i to have the same output type X rather than letting them be polymorphic. We could generalize slightly and define composition with arbitrary output types, but since all of the types must be fixed in advance, this generalization is not actually very useful. We therefore assume all the output types are the same (e.g., a large sum type).

Theorem 2.4 (Simple Composition, Dwork et al. 2006a,b). *Let $\overline{\phi}^k$ be a composable sequence such that each ϕ_i, once provided with the X^i portion of its argument (which we write as $\phi_i(\cdot, \overline{o}^i)$ and think of as having the type $D \to \bigcirc X$), is $(\varepsilon_i, \delta_i)$-differentially private for every choice of realized outcome vector $\overline{o}^i$ (that is, regardless of what outcomes have been produced by the functions $\phi_0 \ldots \phi_{i-1}$). Then the k-fold composition of $\overline{\phi}^k$ is (ε, δ)-differentially private for $\varepsilon = \sum_i \varepsilon_i$ and $\delta = \sum_i \delta_i$.*

The simple composition theorem bounds the privacy cost of a sequence of pieces by the sum of their individual costs, pessimistically assuming every piece will behave like its worst case. However, if we treat each piece as producing an independently random result, we can instead consider their *average* privacy loss. Now ε grows only by the square root of the number of pieces being composed, but a small amount of δ must be added to account for the low probability that all the pieces indeed behave in a worst-case way. This argument is captured by the following "advanced composition" theorem:

Theorem 2.5 (Advanced Composition, Dwork et al. 2010). *Let $\overline{\phi}^k$ be a composable sequence such that each $\phi_i(\cdot, \overline{o}^i)$ is (ε, δ)-differentially private for every choice of realized outcome vector $\overline{o}^i$. Then, for all $\delta' > 0$, the k-fold composition of $\overline{\phi}^k$ is $(\varepsilon', k\delta + \delta')$-differentially private, where*

$$\varepsilon' \quad = \quad k \cdot \varepsilon \cdot (e^{\varepsilon} - 1) + \varepsilon \cdot \sqrt{2k \cdot \ln(1/\delta')}.$$

Advanced composition often yields a significantly better bound, even for very small choices of δ' (as long as k is large enough and ε is small enough). For instance, suppose the analyst needs to run $k = 2000$ $(10^{-4}, 0)$-pieces. Simple composition tells us that the composite is a $(0.2, 0)$-piece, whereas advanced composition, choosing $\delta' = 2^{-30}$, shows it is a $(0.029, 2^{-30})$-piece—a nearly 7× improvement. Thus, if the curator permits this tiny δ, the analyst can run her algorithm using much less budget.

Privacy Filters. The requirement in Theorem 2.5 that each ϕ_i be (ε, δ)-differentially private for the *same* ε and δ may seem artificial. Indeed, with a little work the theorem can be generalized so that each ϕ_i has different $(\varepsilon_i, \delta_i)$ parameters; however, this is less useful than it might seem, because these parameters must all be chosen before seeing any of the intermediate results. In fact, the theorem actually fails to hold in the parameter-adaptive case, when these parameters can themselves be chosen as functions of the outcome of previously run pieces $\phi_0, \ldots, \phi_{i-1}$ [Rogers et al. 2016]. Indeed, even the *definition* of differential privacy becomes problematic when the ε_i and δ_i parameters are not fixed constants: if ε_i and δ_i are functions of the previous realized outcomes of the mechanism, then they cannot appear outside of the probability operator, as they do in Definition 2.1, because there they are undefined.

What we need is a more general definition of differential privacy that makes sense in the parameter-adaptive case and that coincides with the standard definition when the parameters are fixed. The idea for this, due to Rogers et al. [2016], is to build the sequence of pieces directly into the definition and then, instead of looking simply at the final outcome distribution, look at the difference in outcome probabilities for each piece.

Definition 2.6 (Privacy Loss with Adaptive Parameters, Rogers et al. 2016). *Let $\overline{\phi}^k$ be a composable sequence whose k-fold composition is ϕ, let ε_i and δ_i be functions of type $X^i \to \mathbb{R}$, and let each $\phi_i(\cdot, \overline{o}^i)$ be $(\varepsilon_i(\overline{o}^i), \delta_i(\overline{o}^i))$-differentially private for every choice of realized outcome vector $\overline{o}^i$. For every pair of neighboring databases d and d', define the* realized privacy loss *of ϕ over realized output vectors to be*

$$L_\phi(\overline{o}^k, d, d') \quad = \quad \ln\left(\frac{\prod_{i=0}^{k-1} \Pr_{x\sim\phi_i(d,\,\overline{o}^i)}[x = o_i]}{\prod_{i=0}^{k-1} \Pr_{x\sim\phi_i(d',\overline{o}^i)}[x = o_i]}\right).$$

The definition of realized loss bears a strong resemblance to the definition of differential privacy. To see this, consider the case where $k = 1$ (and obviously $\overline{\phi}^1 = \phi$):

$$L(o, d, d') \quad = \quad \ln\left(\frac{\Pr_{x\sim\phi(d)}[x = o]}{\Pr_{x\sim\phi(d')}[x = o]}\right).$$

Exponentiating both sides and multiplying by the denominator yields an equation similar to the one in Definition 2.1 (with $\varepsilon = L_\phi(o, d, d')$, $\delta = 0$, and S the singleton set containing o). Indeed, if it can be proven that if $\Pr_{o\sim\phi(d)}[L_\phi(o, d, d') > \varepsilon] \le \delta$, then it follows that ϕ is (ε, δ)-differentially private. This reasoning can be expanded to the $k > 1$ non-adaptive parameter setting, where all the $\varepsilon_i(\overline{o}^i)$ and $\delta_i(\overline{o}^i)$ are constants that do not vary with $\overline{o}^i$. In this case, ϕ is (ε, δ)-differentially private (up to a small loss in the parameters—Kasiviswanathan and Smith [2014] provide an exact statement) if $\Pr_{\overline{o}^k\sim\phi(d)}[L_\phi(\overline{o}^k, d, d') > \varepsilon] \le \delta$.

To extend this idea to the parameter adaptive case, Rogers et al. [2016] define *privacy filters*, which capture one of the main roles of composition theorems: providing a rule that allows the curator to halt the analyst's computation before the privacy budget is exceeded. Formally, a *valid privacy filter* provides a way to halt the computation, with probability $1 - \delta$, before the realized privacy loss exceeds ε.

Definition 2.7 (Valid Privacy Filter, Rogers et al. 2016). *Given a privacy budget* $\varepsilon_g, \delta_g \geq 0$, *a function* $f : (\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0})^* \longrightarrow \{\textsc{Cont}, \textsc{Halt}\}$ *is an* $(\varepsilon_g, \delta_g)$-valid privacy filter *if, for every* k*-fold composition* ϕ *of a composable sequence* $\overline{\phi}^k$ *and for every pair of neighboring databases* d *and* d'*,*

$$\Pr_{\overline{o}^k \sim \phi(d)} \left[\begin{array}{c} L_\phi(\overline{o}^k, d, d') > \varepsilon_g \;\; \textit{and} \;\; f[ed_0, \cdots, ed_{k-1}] = \textsc{Cont} \\ \textit{where } ed_i = (\varepsilon_i(\overline{o}^i), \delta_i(\overline{o}^i)) \end{array} \right] \quad \leq \quad \delta_g.$$

We can use a valid privacy filter to compose parameter-adaptive pieces in such a way that, if running some piece ever exceeds the budget, then no subsequent piece will run. That is, rather than define exactly the privacy cost for a fixed number of pieces as conventional composition theorems do, we simply define a valid composition of parameter-adaptive pieces to be the composition of as many of those pieces as can be run before the filter returns Halt. This is captured in the following definition.

Definition 2.8 (Filtered Composition). *Given an* $(\varepsilon_g, \delta_g)$*-valid privacy filter* f*, the* filtered composition $\phi : D \longrightarrow \bigcirc(X^*)$ *of a composable sequence* $\overline{\phi}^k$ *is defined as* $\phi(d) = \textit{loop}\ 0\ \textit{nil}\ \textit{nil}$*, where*

$$\textit{loop}\ i\ \sigma\ ed \quad = \quad \begin{cases} \textit{return}\ \sigma & \text{if } i \geq k \\ \begin{array}{l} \textbf{let}\ ed' = ed :: (\varepsilon_i(\sigma), \delta_i(\sigma))\ \textbf{in} \\ \textbf{match}\ f\ ed'\ \textbf{with} \\ \mid \textsc{Halt} \longrightarrow \textit{return}\ \sigma \\ \mid \textsc{Cont} \longrightarrow \phi_i(d, \sigma) \gg= \\ \quad (\lambda x.\ \textit{loop}\ (i+1)\ (\sigma :: x)\ ed') \end{array} & \text{if } i < k. \end{cases}$$

The i argument to *loop* counts up to k, σ represents the sequence of results produced so far, and *ed* is the sequence of (ε, δ) pairs seen so far.

Choosing the privacy parameters of the next piece gives us an extra freedom of not needing to worry about how many pieces to run; that is, parameter adaptivity allows us to, depending on any intermediate result, set all future parameters to zero, effectively terminating all future querying. Indeed, this is exactly what we made use of in our introductory example.

A simple result connecting privacy filters to adaptive-parameter differential privacy is the following:

Lemma 2.9 (Filtered Composition is Differentially Private, Rogers et al. 2016). *If* f *is an* $(\varepsilon_g, \delta_g)$*-valid privacy filter, then the filtered composition of a composable sequence* $\overline{\phi}^k$ *is* $(\varepsilon_g, \delta_g)$*-differentially private.*

Simple composition (Theorem 2.4) has a straightforward analog as a privacy filter.

Theorem 2.10 (Simple Filter for Adaptive Parameters, Rogers et al. 2016). *For all global privacy budgets* $\varepsilon_g, \delta_g \geq 0$*, the function*

$$f\,[(\varepsilon_0, \delta_0), \cdots, (\varepsilon_{k-1}, \delta_{k-1})] \quad = \quad \begin{cases} \textsc{Halt} & \textit{if } \sum_{i=0}^{k-1} \varepsilon_i > \varepsilon_g \textit{ or } \sum_{i=0}^{k-1} \delta_i > \delta_g \\ \textsc{Cont} & \textit{otherwise} \end{cases}$$

is an $(\varepsilon_g, \delta_g)$*-valid privacy filter.*

The analog of advanced composition (Theorem 2.5) has a more complex form.

Theorem 2.11 (Advanced Filter for Adaptive Parameters, Rogers et al. 2016). *For a fixed privacy budget* $\varepsilon_g, \delta_g > 0$*, the function*

$$f\,[(\varepsilon_0, \delta_0), \cdots, (\varepsilon_{k-1}, \delta_{k-1})] \quad = \quad \begin{cases} \textsc{Halt} & \textit{if } \sum_{i=0}^{k-1} \delta_i > \frac{\delta_g}{2} \textit{ or } \mathcal{K} > \varepsilon_g \\ \textsc{Cont} & \textit{otherwise} \end{cases}$$

where

$$\mathcal{K} \;=\; \sum_{j=0}^{k-1} \varepsilon_j \left(\frac{e^{\varepsilon_j}-1}{2}\right) + \sqrt{2\left(\sum_{i=0}^{k-1}\varepsilon_i^2 + \frac{\varepsilon_g^2}{28.04\cdot\ln(1/\delta_g)}\right)\left(1+\frac{1}{2}\ln\left(\frac{28.04\cdot\ln(1/\delta_g)\sum_{i=0}^{k-1}\varepsilon_i^2}{\varepsilon_g^2}+1\right)\right)\ln(2/\delta_g)}$$

is an $(\varepsilon_g, \delta_g)$-valid privacy filter.[5]

$\mathcal{K}$ looks a bit intimidating at first (and second) glance, but it shares a common structure with ε' from Theorem 2.5. The first term is the sum of the ε values (which, if they were all the same, would be $k\varepsilon$) times an $e^{\varepsilon} - 1$ term. Then, instead of ε outside of the square root, $\mathcal{K}$ has a $k \cdot \varepsilon^2$ term inside. The next term in the root is one plus the log of an ε^2 term divided by an ε_g^2 term, which together behave as a scaling factor, and the last term is just a log of $1/\delta$ term. Indeed, although this definition is more complicated and has worse constants, it stems from the same core principles as Theorem 2.5.

3 SEMANTICS OF ADAPTIVE COMPOSITION

We next describe a more "PL-oriented" presentation of adaptive differential privacy with filtered composition, using the language of typed functional programming. This alternative presentation cleanly distinguishes the roles of curator and analyst, using the familiar concept of *streams* to connect them in an intuitive framework. This framework handles all communication between the two actors, allowing them to be defined simply as ordinary, independent functions, with no need for, say, linear types (to prevent the curator from being called more than once in the same state), built-in state (for either actor to recall information from earlier), or explicit random sampling (because the curator cannot release a whole distribution to the analyst without fear that she will sample it more than once).

Our semantics also addresses a number of smaller nits present in the previous section's design, such as: the number of algorithms to compose together being required up front, disallowing divergence but allowing uncomputable functions, and halting filtered composition entirely the first time the curator rejects an analyst's proposed piece.

We begin with some low-level components, next curators and analysts, and then the framework that connects them. Finally, we describe how using the framework guarantees differential privacy.

In this section, all functions are computable and strict; they are also total unless indicated in the type (where $X_\perp$ includes both divergence and proper values of type X).

Domain. As in §2, we fix a type X to be the common output type of every piece. Simplistically, we might take X to be $\mathbb{R}_{\geq 0}$, but more realistically, X will be some large union type. (In §4 it will include all Adaptive Fuzz values, both atomic and structured, as long as they do not contain functions.)

Pieces. As in §2, a *piece* is conceptually just a differentially private function. To represent this so that the curator is able to verify its privacy claim, the function is stored as *code*, together with claimed ε and δ bounds on its privacy cost; when compiled, this code should yield an (ε, δ)-differentially private function with type $D \to \bigcirc X_\perp$ (note that $\perp$ binds more tightly than $\bigcirc$, meaning that here we are describing a distribution over potentially diverging values). Formally, the type of pieces is

$$P = \text{Code} \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}.$$

The two numbers represent ε and δ, and given a piece p, we will write p_ε and p_δ to access them.

To verify and compile pieces, we use a *verifier*, of type

$$V = P \to \{\textsc{VerRej} \mid \textsc{VerAcc} : (D \to \bigcirc X_\perp)\},$$

[5] $\mathcal{K}$ is sufficient but is not known to be a tight bound. Rogers et al. [2016] discuss this in more detail.

where $\{\textsc{Name}_1 : \ldots \mid \textsc{Name}_2 : \ldots\}$ is syntactic sugar for a binary sum type where we can name (and later pattern match on) the two sides. A valid verifier is one that returns $\textsc{VerAcc}\ \phi$ only when it can prove that the code of the provided piece can be compiled into an appropriately typed function ϕ that is differentially private with the given piece's privacy parameters; it should return $\textsc{VerRej}$ otherwise.

Filters. Filters remain unchanged from the previous section except for a few syntactic differences:

$$F = (\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0})\ \mathsf{list} \rightarrow \{\textsc{FilRej} \mid \textsc{FilAcc}\}$$

In the previous section, we used the notation τ^k to refer to sequences of a particular length. In an effort to avoid complicating our type system with dependent types, we use only arbitrary length sequences, and we use the familiar notation τ list instead of τ^* to refer to them (the *nil*, ::, and $[\cdots]$ notation stays the same).

Curators and Analysts. To clarify the interplay between curators and analysts, we model their behaviors as *streams*. In particular, curators are modeled as functions on pieces. If an input piece meets the curator's constraints (that is, it satisfies a valid verifier and its privacy parameters are accepted by a valid filter), then the result is a distribution over outputs; otherwise, the piece is rejected. If it accepts the piece, it also returns a new curator (with an appropriately updated privacy budget).

$$C = P \rightarrow \left\{\textsc{CurRej} \mid \textsc{CurAcc} : (\bigcirc X_{\perp} \times C)\right\}$$

The analyst represents the (generally untrusted) user who wants to compute over the curator's private data. At each point, the analyst has the option to either end the interaction or propose a piece, after which she is able to react to the curator's response. We model this back-and-forth with a *continuation* that is able to deal with either a sampled result from the curator or $\textsc{None}$, if the curator rejected her piece. In both cases, the analyst's continuation specifies what she will do on the next round of interaction (which may include diverging while thinking about what to do next).

$$A = \left\{\textsc{AnDone} \mid \textsc{AnNext} : P \times \left(\{\textsc{Some} : X \mid \textsc{None}\} \rightarrow A_{\perp}\right)\right\}$$

Framework. The last piece of the puzzle is a *framework* that links an analyst together with a curator, allowing the two to communicate in the expected way: the analyst proposes a piece, which the framework passes on to the curator; the curator uses this to produce a new curator along with a distribution of results (or $\textsc{CurRej}$); the framework uses this distribution and the analyst's continuation to produce a new analyst; and so on *ad infinitum* until either one of the pieces diverges or the analyst decides to halt or diverges herself. After each "round" we come back to where we started, except that the analyst has presumably learned something and the curator has subtracted the cost of the previous piece from his budget. An example interaction can be seen in Figure 1.

One subtlety of the curator-analyst interaction is that, if an analyst's proposed piece is rejected by the curator, the analyst should still have an opportunity to propose a new one. Because the analyst itself is deterministic, and getting rejected by the curator is independent of the sensitive database, it is a deterministic process to find the next piece that is acceptable. Therefore, the framework can find the analyst's next good piece (i.e. the next value that the curator will accept) using the function adv_A defined in Figure 2. Then, the main function of the framework, to act as a communication conduit between curator and analyst, can be defined as in Figure 3.

The *run* function takes a curator generator (a function from databases to curators), an analyst, and a database, and it produces a probabilistic *stream* of results—that is, each time we poke it by applying it to a trivial unit argument, it returns either $\textsc{Done}$ or else a *distribution* over pairs of an

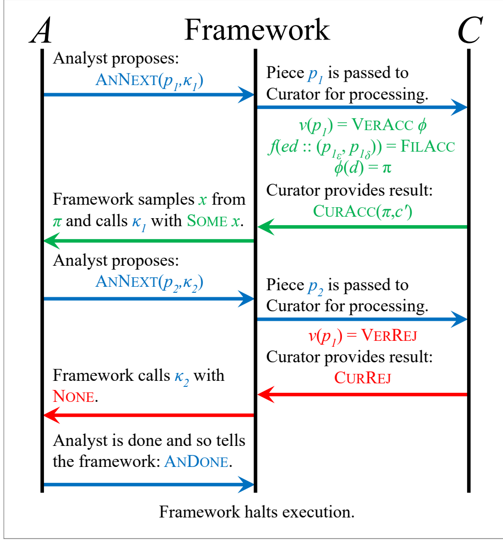

Fig. 1. A sample interaction between analyst and curator

$$
\begin{aligned}
&adv_A : C \to A \to A_\perp\\
&adv_A = \lambda c.\ \lambda a.\\
&\quad \textbf{match } a \textbf{ with}\\
&\quad \mid \text{ANDONE} \to a\\
&\quad \mid \text{ANNEXT } (p, \kappa) \to \textbf{match } c\ p \textbf{ with}\\
&\qquad \mid \text{CURREJ} \to adv_A\ c\ (\kappa\ \text{NONE})\\
&\qquad \mid \text{CURACC } _ \to a
\end{aligned}
$$

Fig. 2. The adv_A function

$$
\begin{aligned}
&run : (D \to C) \to A \to D \to S\\
&run = \lambda g.\ \lambda a.\ \lambda d.\ run'\ (g\ d)\ a\\
&run' = \lambda c.\ \lambda a.\ \lambda ().\\
&\quad \textbf{let } a' = adv_A\ c\ a \textbf{ in}\\
&\qquad \textbf{match } a' \textbf{ with}\\
&\qquad \mid \text{ANDONE} \to \text{DONE}\\
&\qquad \mid \text{ANNEXT } (p, \kappa) \to \textbf{match } c\ p \textbf{ with}\\
&\qquad\quad \mid \text{CURREJ} \to \perp\\
&\qquad\quad \mid \text{CURACC } (\pi, c') \to \text{NEXT } (\pi \gg= \lambda x.\\
&\qquad\qquad (x, run'\ c'\ (\kappa\ (\text{SOME } x))))
\end{aligned}
$$

Fig. 3. The *run* function

X value and a new stream (or $\perp$). Formally, the stream type is defined as

$$S = 1 \to \{\text{DONE} \mid \text{NEXT} : \bigcirc(X \times S)_\perp\}.$$

The database is provided as a separate argument to *run* so that we can talk about the differential privacy of *run* applied to its first two arguments. The first step of *run* is to supply the database to the curator generator and call *run′*. Then, the next acceptable piece by the analyst is found (or she is done, in which case the stream is capped off with a DONE value), and it is provided to the curator. Because the definition of adv_A guarantees that the curator will accept it, the rejecting case is left undefined. In the accepting case, the distribution is used to create a NEXT value.

Correctness. If *run* is called with a generator that yields well-behaved curators and (importantly) any analyst whatsoever, it yields a filtered composition. The proof of this fact has to deal with two issues. First, strictly speaking, the standard definition of differential privacy from §2 does not actually apply to streams—it requires the output to be a distribution of values while streams have a nested distribution structure—so we need to generalize it. We propose the following definition, which meshes cleanly with the existing theory of adaptive composition.

$$
\begin{aligned}
s &::= \mathbb{R}_{\geq 0} \mid \infty \mid e \\
\tau &::= \alpha \mid b \mid 1 \mid \tau \multimap [s]\ \tau \mid \tau + \tau \mid (\tau \times \tau) \mid \mu\alpha.\,\tau \mid \bigcirc\tau \\
e &::= x \mid c \mid op\ e \ldots e \mid \textbf{fun}\ (x : [s]\ \tau)\ \{e\} \mid e\ e \mid x = e;\ e \mid (e, e) \mid \textbf{let}\ (x, x)\ = e;\ e \\
&\quad \mid \textbf{inl}\ e \mid \textbf{inr}\ e \mid \textbf{case}\ e\ \textbf{of}\ \{\textbf{inl}(x) \Rightarrow e \mid \textbf{inr}(x) \Rightarrow e\} \mid \textbf{sample}\ x = e;\ e \mid \textbf{return}\ e \\
&\quad \mid \textbf{fold}_\tau\ e \mid \textbf{unfold}\ e \mid \textbf{forall}\ t.e \mid e\ [t] \mid \textbf{runfuzz}\ [t]\ e\ e
\end{aligned}
$$

Fig. 4. Adaptive Fuzz core syntax (in examples, we will use some standard syntactic sugar)

Definition 3.1. *A function $f : D \to S$ is said to be (ε, δ)-*differentially private *if, for all $k \in \mathbb{N}$, the function $f_k\ d = unroll_k\ 0\ nil\ (f\ d)$ is (ε, δ)-differentially private in the standard sense, where*

$$
\begin{aligned}
&unroll_k : \mathbb{N} \to X\ \text{list} \to S \to \bigcirc(X\ \text{list})_\perp \\
&unroll_k = \lambda i.\ \lambda\sigma.\ \lambda s. \\
&\quad \textbf{if}\ i \geq k\ \textbf{then}\ return\ \sigma \\
&\quad \textbf{else match}\ s\ ()\ \textbf{with} \\
&\qquad \mid \textsc{Done} \to return\ \sigma \\
&\qquad \mid \textsc{Next}\ \pi \to \pi \gg= \lambda(x, s').\ unroll_k\ (i+1)\ (\sigma :: x)\ s'.
\end{aligned}
$$

Intuitively, if every finite prefix of a stream can be produced in a differentially private way, then the stream can too.

Second, we need a "well-behaved curator generator." We use this one:

$$
\begin{aligned}
&mkC : V \to F \to (\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0})\ \text{list} \to D \to C \\
&mkC = \lambda v.\ \lambda f.\ \lambda ed.\ \lambda d.\ \lambda p. \\
&\quad \textbf{match}\ v\ p\ \textbf{with} \\
&\quad \mid \textsc{VerRej} \to \textsc{CurRej} \\
&\quad \mid \textsc{VerAcc}\ \phi \to \textbf{let}\ ed' = ed :: (p_\varepsilon, p_\delta)\ \textbf{in} \\
&\qquad \textbf{match}\ f\ ed'\ \textbf{with} \\
&\qquad \mid \textsc{FilRej} \to \textsc{CurRej} \\
&\qquad \mid \textsc{FilAcc} \to \textsc{CurAcc}\ (\phi\ d,\ mkC\ v\ f\ ed'\ d)
\end{aligned}
$$

The *mkC* function accepts a verifier, a filter, a list of (ε, δ) pairs representing the costs of prior pieces, and a database. It then accepts and runs a piece only if it satisfies both the verifier and the filter, in which case the "next" curator is the same but with an updated history of piece costs.

Theorem 3.2. *If a is an analyst, v is a valid verifier, and f is an $(\varepsilon_g, \delta_g)$-valid privacy filter, then the function run $(mkC\ v\ f\ nil)\ a$ is $(\varepsilon_g, \delta_g)$-differentially private.*

The full proof can be found in the extended version.

4 LANGUAGE DESIGN

The semantics in §3 is abstracted on three parameters: a privacy filter, a verifier, and an analyst. In our Adaptive Fuzz implementation, the first two are fixed—the filter is built from the functions in Theorems 2.10 and 2.11, and the verifier is the typechecker for pieces that we sketch later in this section—while the analyst is a program written in Adaptive Fuzz.

An Adaptive Fuzz program starts running in the so-called *adaptive mode*, where it has no direct access to the private database and, accordingly, no need to worry about privacy. At some point, it may need to run a subcomputation over the database—a piece; this requires switching into the *data mode*, where the program is given access to the database but where its behavior is constrained by a

stringent typechecker, which ensures that the database is handled in a differentially private way. If the typechecker accepts the piece and there is enough global budget remaining to cover its privacy cost, then the database is provided as an argument to the piece and the runtime system samples from the resulting distribution to yield a concrete value. This value is globally published, meaning that the analyst, now back in the adaptive mode, can use it to help determine the next piece she will submit. The output of the whole program is the sequence of published results returned by evaluation of successive pieces, plus anything else that the analyst cares to print.

This dual-mode design requires that the programmer code in two distinct styles, but the language is built to make this easy. To minimize cognitive overhead and maximize opportunities for sharing code (e.g., of library functions) between the two modes, both modes use exactly the same syntax, which is shown in Figure 4. The difference is that functions in the data mode must pass the stringent data-mode typechecker, while typing of adaptive-mode functions is looser. Specifically, the data-mode typechecker uses elements of linear typing and makes explicit use of the s annotations in the function type form $\tau \multimap [s]\ \tau$, while the adaptive mode typechecker uses a System-F-like typechecker that ignores the annotations entirely.

After introducing the basics of a running example for the section, we will dive into specifics of the data and adaptive modes and then conclude with a discussion about implementation details.

4.1 Running Example: Gradient Descent

We use a differentially private gradient-descent algorithm as a running example to demonstrate various aspects of Adaptive Fuzz. This is not merely a toy example designed simply to show off language features; indeed, it is a bit more complex than that and will be the focus of our first case study in §5. We begin with a brief primer on gradient descent.

Gradient descent is a simple and widely used optimization method that works to train many machine learning models—particularly d-dimensional linear regression with the standard mean-squared-error objective. Given a set of n records, $x_i \in \mathbb{R}^d$, each with a real-valued *label* y_i, the goal is to find a parameter vector $\theta \in \mathbb{R}^d$ that minimizes the difference between the labels and their *estimates*, where the estimate of a label y_i is the inner product $\langle x_i, \theta \rangle$ of θ and the corresponding record. That is, the goal is to minimize the following *loss function*:

$$L(\theta, (x, y)) \quad = \quad \frac{1}{n} \cdot \sum_{i=1}^{n} (\langle x_i, \theta \rangle - y_i)^2$$

The algorithm starts with an initial parameter vector such as $(0, \ldots, 0)$, and it iteratively produces subsequent θ vectors until some termination criterion is reached. For simplicity, we assume here that the process continues until the privacy budget is exhausted, but of course, in practice, the analyst may well choose to stop earlier if she finds the accuracy gains per iteration are no longer worth the cost.

Each iteration proceeds as follows. First, the algorithm computes the difference between the actual label of each example and its estimate. This quantity is called the *residual*:

$$resid(\theta, i, (x, y)) \quad = \quad \langle x_i, \theta \rangle - y_i \quad = \quad \sum_{\ell=1}^{d} \theta_\ell \cdot x_{i,\ell} - y_i$$

The residual is weighted according to the value of each feature and summed over all records to find the *gradient* vector of the loss function $\nabla L(\theta, (x, y))$; the jth coordinate of the gradient is:

$$\nabla L(\theta, (x, y))_j \quad = \quad \frac{2}{n} \cdot \sum_{i=1}^{n} x_{i,j} \cdot resid(\theta, i, (x, y))$$

This is then multiplied by the learning rate η (a parameter of the algorithm that acts as a scaling factor) to yield a new parameter vector, which becomes the θ for the next round of the algorithm.

A simple way to make this algorithm private is to add noise to the gradient vector after each round—i.e., to each individual component of $\nabla L(\theta, (x, y))$. This noise must be scaled based on the magnitude of the values in x and y (i.e., the sensitive values in the database)—the bigger the values, the more noise must be added. However, if we allow this noise to get arbitrarily large, then we cannot make any claims about how *accurate* the algorithm is. Therefore, we must bound the values in the database; here, we will assume that they are all between 0 and 1. We can then leverage the elegant accuracy analysis of Bassily et al. [2014].

4.2 Data Mode

The verifier for Adaptive Fuzz needs to be able to prove that a given code fragment denotes a differentially private piece. Fortunately, this is precisely what the Fuzz type system [Reed and Pierce 2010] was designed to do. This is why the syntax of Adaptive Fuzz (shown in Figure 4) is modeled after Fuzz itself:[6] so that we can use the Fuzz typechecking algorithm as our verifier, essentially unchanged. This section describes informally how the Fuzz typechecker works, aiming for just enough detail that the reader can understand the empirical results and additional examples in the following section. Reed and Pierce [2010] provide a full treatment.

The Fuzz type system proves differential privacy by tracking *function sensitivity*, a measure of how strongly the output of a function can vary as its input varies. (Recall from §2 that d_τ is the metric over the type τ.)

Definition 4.1 (Function sensitivity, Dwork et al. 2006b). *A function $f : \tau_1 \longrightarrow \tau_2$ is said to be c-sensitive if and only if $d_{\tau_2}(f(x), f(y)) \leq c \cdot d_{\tau_1}(x, y)$ for all $x, y \in \tau_1$.*

For instance, $f(x) = 2x$ is 2-sensitive because if $|x - x'| \leq 1$, then $|f(x) - f(x')| \leq 2$; similarly, $g(x) = x + 3$ is 1-sensitive, while $h(x) = x^2$ is not c-sensitive for any finite c. The Adaptive Fuzz type $\tau_1 \multimap[c]\ \tau_2$ classifies functions with c as a measure of their sensitivity; here, f would have the type $\mathbb{R} \multimap[2]\ \mathbb{R}$, while g would have type $\mathbb{R} \multimap[1]\ \mathbb{R}$ and h would have type $\mathbb{R} \multimap[\infty]\ \mathbb{R}$, meaning that it may use its argument an arbitrary number of times. Furthermore, because any function that is c-sensitive is clearly c'-sensitive for any $c' > c$, our use of the ∞ annotation serves as a nice pun: all functions that have the type $\tau_1 \multimap[c]\ \tau_2$ also have the type $\tau_1 \multimap[\infty]\ \tau_2$ just as all finite values of c are less than infinity. To extend sensitivity to functions over non-numeric types, we equip every type with a *distance metric*. The typechecking process uses these metrics, along with techniques drawn from *linear type systems* [Wadler 1990], to calculate sensitivities. The form of the Fuzz typing judgment is

$$(x_1 :_{s_1} \tau_1, \ldots, x_n :_{s_n} \tau_n) \vdash e : \tau.$$

This says that the open expression e is s_k-sensitive in the variable x_k for all $1 \leq k \leq n$, or, in linear typing terms, that e can use x_k at most s_k times. If $s_k = \infty$, then e can use x_k arbitrarily.

[6] We do make two small technical changes to the formulation of Reed and Pierce [2010]. First, rather than having a unique $!_s$ operator for modifying the sensitivity of a variable, we merge this annotation into function definitions. Specifically, the type $\tau_1 \multimap[s]\ \tau_2$ can be thought of as desugaring to $!_s\tau_1 \multimap \tau_2$ when $s \in \mathbb{R}_{\geq 0}$ and to $\tau_1 \to \tau_2$ when $s = \infty$, and the expression **fun** $(x : [s]\ \tau)\ \{e\}$ to $\lambda x'.$ **let** $!x = x'$ **in** e when $s \in \mathbb{R}_{\geq 0}$ and to $\lambda x. e$ when $s = \infty$. Second, in Fuzz's original semantics, probability distributions are represented as lists of pairs of (non-lifted) values with their probabilities. Evaluating an expression that results in a distribution requires evaluating every possible outcome of the expression, which means that, if the expression has a non-zero probability of diverging, then its evaluation will simply diverge. Here, we instead use the conventions of §2, where probability distributions are defined as functions from real numbers on the unit interval to (lifted) output values. Because these functions are only evaluated on one sampled real number, they diverge precisely with the probability of divergence and otherwise return a value. The proofs of Reed and Pierce [2010] go through unchanged. Lastly, we omit &-pairs from this discussion, though they are handled by our implementation.

$$\frac{\Gamma \vdash e_1 : \bigcirc\tau \qquad \Gamma', x :_\infty \tau \vdash e_2 : \bigcirc\tau'}{\Gamma + \Gamma' \vdash \textbf{sample}\ x = e_1;\ e_2 : \bigcirc\tau'} \qquad \frac{\Gamma \vdash e : \tau}{\infty\Gamma \vdash \textbf{return}\ e : \bigcirc\tau}$$

Fig. 5. The typing rules for sample and return.

Function sensitivity by itself is not enough to prove differential privacy; in addition, the Fuzz type system makes sure that the output of a piece is properly randomized—i.e., that each piece's result type is $\bigcirc\tau$, where $\bigcirc$ is the probability monad described in §2. A key insight in Fuzz is exactly how sensitivity is tracked through the monad: the typing rules for this are shown in Figure 5. The typing rule for the bind operator, written **sample** $x = e_1;\ e_2$, says that if e_1 is c_1-sensitive in some argument and e_2 is c_2-sensitive in it, then no matter how e_2 uses x, the whole expression will still only be $(c_1 + c_2)$-sensitive in the argument. (This is described by the sum of the environments Γ and Γ', which combines the environments while adding together the sensitivities of matching variables.) Basically, this means that sampled values can be used arbitrarily—that is, that they are added to the context with an ∞ annotation when e_2 is typechecked. The typing rule for the **return** operator simply defines **return** to be a function with no finite sensitivity, reflecting the fact that the operation of injecting a value into the probability monad as a "point distribution" (assigning probability 1 to that value) is not finitely sensitive. This means that the argument of **return** cannot depend on any finite-sensitivity variables, which is expressed in the rule by scaling every sensitivity in the environment Γ by ∞.

To get useful (finite-sensitivity) randomized computations, Adaptive Fuzz provides three primitives implementing basic differential privacy mechanisms: the Laplace mechanism for adding random noise, the exponential mechanism for privatizing non-real-valued queries, and the report-noisy-max primitive for privately determining which of a group of counting queries has the highest value [Dwork and Roth 2014].

Together, sensitivity checking and the probability monad give rise to the type system's key theorem:

Theorem 4.2 (Typing Guarantees Differential Privacy, Reed and Pierce 2010). *A closed piece e of type $\tau_1 \multimap[\varepsilon]\ \bigcirc\tau_2$ denotes an $(\varepsilon, 0)$-differentially private function from τ_1 to τ_2.*

To make a formal connection between this soundness theorem and the framework proposed in §3, we define the *Fuzz Verifier* to be a program that accepts a proposed piece coming from the analyst, comprising an abstract syntax tree and a claimed privacy bound ε, uses the Fuzz typechecking algorithm to check that the code has the type $D \multimap[\varepsilon]\ \bigcirc\tau$, and, if so, produces a VerAcc value containing the function produced by compiling the code (and otherwise returns VerRej). Theorem 4.2 establishes that this verifier is indeed *valid* in the sense of §3.

One subtlety worth noting is that the data-mode type system only deals in $(\varepsilon, 0)$-differential privacy: all pieces have $\delta = 0$, and the typechecker uses only the simple composition theorem internally (it is essentially the typing rule for the **bind** operator of the probability monad). Non-zero δ values arise only when many pieces are composed in the adaptive mode, where we apply an advanced privacy filter in filtered (advanced adaptive) composition to calculate the privacy cost of a whole computation.

Example. Figure 6 shows a data-mode piece that is ready to be inspected by the Fuzz Verifier. It implements the portion of gradient descent that updates a single parameter of the estimate vector θ.

The first line of Figure 6 defines a function *updateParameter* of five arguments. The first four arguments are simple: ε is the amount of privacy budget the piece will use, p is the index of the parameter that this piece will update, θ is the current estimate vector, and n is the size of the

```
function updateParameter (ε : ℝ) (p : ℤ) (θ : ℝ list) (n : ℤ) (db : [ε] (𝕀 × 𝕀 list) bag) : ○ℝ {
  v = calcGrad n θ p db;
  sample v_sample = add_noise (ε/2) v;
  return (2/n · v_sample)
}
function calcGrad (n : ℤ) (θ : ℝ list) (p : ℤ) (db : [2] (𝕀 × 𝕀 list) bag) : ℝ {
  rowOp = fun (yrow : (𝕀 × 𝕀 list)) : 𝕀 {
              let (y, row) = yrow;
              x = listindex [𝕀] p row;
              clip (((calcResid θ row y) · x + 1)/2)
          };
  v = bagsum (bagmap [𝕀 × 𝕀 list → 𝕀] rowOp db);
  v + v − n
}
function calcResid (θ : ℝ list) (row : 𝕀 list) (y : 𝕀) : ℝ {
  (clip (listsum (listzip [ℝ → ℝ → ℝ] mult θ row))) − y
}
```

Fig. 6. Data-mode code defining the piece that is used for each parameter-update step of gradient descent

database.[7] The fifth argument is the sensitive database, with type $(\mathbb{I} \times (\mathbb{I}\ list))\ bag$, where *bag* is the built-in type of multisets. Here, the elements of the bag are pairs of a label and a list of features. $\mathbb{I}$ is a special "clipped real" type that represents numbers in the unit interval $[0, 1]$. Following Theorem 4.2, the output type of every piece must be in the probability monad; here it is a distribution over reals.

The next line of *updateParameter* is a call to *calcGrad*. The *calcGrad* function is 2-sensitive in its database argument (as can be seen from the type annotation on its *db* parameter). This is important for the next line, which injects v into the probability monad using *add_noise*, the primitive implementing the Laplace mechanism [Dwork and Roth 2014]. The first argument to *add_noise* is the amplitude of the noise that will be added to the second argument. To make the distribution ε-sensitive in *db*, this means adding noise with amplitude $\varepsilon/2$. Because this is in a **sample** statement, this distribution is sampled, and the result is bound to v_{sample}. The final line scales v_{sample} by $2/n$ and injects it back into the probability monad. Because **return** is not finitely sensitive in its argument, this means that the whole expression requires that v_{sample} have an ∞ annotation in the context, but because **sample** does this (that is, it allows a sampled value to be used arbitrarily without changing the sensitivity of the distribution it's sampling), the whole expression is still only ε-sensitive in *db* as required.

Next we have *calcGrad*, which calculates the gradient at a single parameter. It begins by defining a function *rowOp*, which takes a row of the database and returns the residual at that row (using the function *calcResid*) times the value of the parameter over which we are calculating the gradient (extracted using a simple *listindex* function). It is critical that *rowOp* returns a clipped value (type $\mathbb{I}$), so it adjusts the resulting value to be between 0 and 1 with some simple arithmetic. (The function *rowOp* has no explicit sensitivity declaration, so Adaptive Fuzz assumes that it is infinite.) With *rowOp* defined, *calcGrad* then maps it over the database and sums the resulting bag. Recall from

[7] Although there are formulations of differential privacy in which the metric on bags permits bag size to be public information, the one we use considers it to be private. For our purposes here, we assume that the size of the database has been previously released, perhaps as the result of a differentially private query (obviously, using up some of the privacy budget).

§ 2 that the distance between two multisets is the number of differing elements, which means that mapping (which does not change the shape of the multiset) is 1-sensitive regardless of the sensitivity of the mapped function, and because the map results in a bag of clipped values, summing is 1-sensitive too (while if the values were arbitrary reals, summing would be infinitely sensitive). Finally, the last line of *calcGrad* doubles the result and subtracts the size of the database to "undo" the simple arithmetic done in *rowOp* and recover the true gradient value; it is this last operation—adding together two 1-sensitive values—that makes the function 2-sensitive in *db* in total.

The last function, *calcResid*, is simpler than the first two because it has no sensitive arguments. It simply calculates the dot product of θ and the given parameter list by pairwise multiplying the lists and summing the results and then subtracts the label value. By clipping the dot product, this function guarantees that its result is between -1 and 1.

4.3 Adaptive Mode

In terms of syntax, the adaptive mode and the data mode are the same, a choice we made consciously to allow code to be used in either mode. That said, while functions running in the data mode must be carefully constructed so that the Fuzz verifier can verify them, code in the adaptive mode is much less restrictive. Instead of using the Fuzz typechecker, it uses a simple System-F-like typechecker that uses the standard rules for a monad and ignores the annotations on arrows (treating $\multimap[c]$ simply as $\rightarrow$).[8] This typechecker is used primarily as a basic sanity check and is not essential to the Adaptive Fuzz design: the adaptive mode could just as well be untyped.

The major additional feature of the adaptive mode over the data mode is the primitive form **runfuzz** $[\tau]\ \varepsilon\ e$, which allows the analyst to enter the data mode and propose a piece.[9] Its parameters are a result type τ (which must not contain any arrows), a privacy bound ε, and an expression e representing a piece to be run. To evaluate this expression, we first run a *partial evaluation* pass over e to make it easier to verify. This partial-evaluation step is a crucial detail, without which the Fuzz verifier would almost never be able to verify pieces built in Adaptive Fuzz. The Fuzz typechecker requires all sensitivity annotations to be constants, but Adaptive Fuzz allows the annotations to be arbitrary expressions to enable more adaptive programs. Thus, partial evaluation is used to reduce the sensitivity annotations within e to constants, allowing Adaptive Fuzz programs to be written with arbitrary expressions in sensitivity annotations. For instance, the function *updateParameter* from Figure 6 will not be accepted by the Fuzz typechecker until it is provided with its first argument, ε, and then partially evaluated so that the ε in the annotation for *db* is reduced to a constant. Indeed, it is this partial evaluation that allows us to write code that looks like it has dependent types (e.g., *updateParameter*) without having to actually support dependent types. Partial evaluation also monomorphizes polymorphic functions (since the Fuzz type system is also monomorphic) and unrolls loops involving sensitive data, allowing more sensitivity annotations to be reduced to constants.

Finally, the Fuzz verifier is used to check that the partially evaluated piece has type $D \multimap[\varepsilon] \bigcirc\tau$ for some τ. If not, **runfuzz** returns **inl** (), signaling failure. If so, the piece is executed, yielding a distribution over results of type τ; a concrete value v is sampled from this distribution, and **inr** v is returned as the result of **runfuzz**.

Operational Semantics. The operational semantics for Adaptive Fuzz is written in continuation-passing style to mimic the analyst definition from §3. The types *Exp* and *Val* are type synonyms for the type of abstract syntax trees and are used as the instantiation of the type X as well as the type of

[8] Additionally, it collapses both tensor product pairs and &-pairs into the same type.

[9] Of course, because it is part of the syntax, **runfuzz** could technically be used in the data mode, but the Fuzz verifier will always reject it, so it will never run there.

```
evaluate : Exp → A
evaluate exp = eval (exp, λx.AnDone)

eval : (Exp × (Val → A)) → A
eval (exp, κ) = match exp with
  | x → ⊥
  | c → κ c
  | op e1 . . . ek →
      evalList (toList e1 . . . ek, λl. κ (runPrim op l))
  | fun (x :[s] τ) {e} → κ exp
  | e1 e2 → eval (e1, λv1. eval (e2, λv2.
      match v1 with
      | fun (x :[s] τ) {e_body} → eval ([v2/x]e_body, κ)
      | otherwise → ⊥))
  | x = e; e_body → eval (e, λv. eval ([v/x]e_body, κ))
  | inl e → eval (e, λv. κ (inl v))
  | inr e → eval (e, λv. κ (inr v))
  | case e of {inl(x) ⇒ e_l | inr(x) ⇒ e_r} →
      eval (e, λv.
        match v with
        | inl v_l → eval ([v_l/x]e_l, κ)
        | inr v_r → eval ([v_r/x]e_r, κ)
        | otherwise → ⊥)
  | (e1, e2) → eval (e1, λv1. eval (e2, λv2. κ (v1, v2)))
  | let (x1, x2) = e; e_body → eval (e, λv.
      match v with
      | (v1, v2) → eval ([v1/x1][v2/x2]e_body, κ)
      | otherwise → ⊥)
  | fold_τ e → eval (e, λv. κ (fold_τ v))
  | unfold e → eval (e, λv.
      match v with
      | fold_τ v → κ v
      | otherwise → ⊥)
  | sample x = e; e_body → κ exp
  | return e → κ exp
  | runfuzz e_s e → eval (e_s, λs.
      AnNext ((peval e, s, 0), λr.
        match r with
        | None → κ (inl ())
        | Some x → κ (inr x)))

evalList : (Exp list × (Val list → a)) → a
evalList (lst, κ) = match lst with
  | nil → κ nil
  | (e :: es) → eval (e, λv. evalList (es, λvs. κ (v :: vs)))
```

Fig. 7. Adaptive Fuzz operational semantics

code used inside pieces; this makes it easy to generate pieces, and the Fuzz Verifier defined earlier, which works on ASTs, meets the specification exactly. The type D is defined on a case-by-case basis by the curator (this would have to be arranged with the analyst ahead of time). The other types defined in §3 are all type synonyms. We examine a few cases in detail.

To evaluate **let** $(x_1, x_2) \ = e;\ e_{body}$, we first evaluate e. In the continuation, if the result of evaluating e is a pair, then the elements of the pair are substituted for the variables x_1 and x_2 in e_{body}, which is then evaluated (with the original continuation). Otherwise, evaluation fails (which we model as diverging).

Because analysts are deterministic, they do not perform any computation within the probability monad. Therefore, **return** and **sample** expressions are treated as values; they can be bound with **let** and substituted elsewhere, but they are only truly evaluated by the curator when they are part of a piece.

To evaluate **runfuzz** e_s e, the semantics says to first evaluate e_s, which should yield the expected sensitivity of e. Then, we *return* an AnNext value to the framework. This value packages up the partial evaluation of the expression to be evaluated (i.e., *peval* e), the ε bound s, the constant 0 for δ, and a continuation that takes a result from the framework, converts None into an **inl**-tagged value and Some : X into an **inr**-tagged value, and passes it to the current continuation. Because the result of the query (i.e., the type X in §3) is represented as an abstract syntax tree just like the type *Exp* (Figure 4), it can be immediately provided to κ as an **inr**-tagged value.

```
function gradient (ε : ℝ) (η : ℝ) (θ : ℝ list) (n : ℤ)  : ℝ list {
  function inner (p : ℤ) : ℝ list {
    if p ≥ length [ℝ] θ then {
      nil [ℝ]
    } else {
      case runfuzz ε (updateParameter ε p θ n) of {
        inl (s) ⇒ nil [ℝ]
      | inr (v) ⇒ cons [ℝ] (η · v) (inner (p + 1))
  }}}
  θ′ = listzip [ℝ] [ℝ] [ℝ] minus θ (inner 0);
  if length [ℝ] θ′ < length [ℝ] θ then { θ } else { gradient ε η θ′ }
}
```

Fig. 8. Adaptive mode code that implements the outer loop of gradient descent by repeatedly calling *updateParameter* (from Figure 6)

Example. Continuing with the gradient descent example, Figure 8 shows the full gradient descent algorithm written in Adaptive Fuzz: on each iteration of *gradient*, it creates and runs a sequence of pieces (using *updateParameter* from Figure 6) for each parameter in θ.

The first line declares the *gradient* function. It accepts four parameters—ε is the budget it will use for each piece, η is the scaling parameter of the algorithm, θ is the initial parameter list, and n is the size of the database—and produces an estimate vector as a result.

Internally, the function defines another recursive function, *inner*. The goal of *inner* is to run one *updateParameter* piece for each parameter in θ. It is called initially with $p = 0$, and it iterates p until it reaches the length of θ. On each iteration, it runs the *updateParameter* piece, and assuming it completes successfully (i.e., the curator had enough budget to run it), the sampled result v is scaled by η and *cons*'d onto the recursive call to *inner*. When p is greater than the number of parameters, *inner* just returns an empty list (*nil*).

The *gradient* function continues by simply calling *inner* and then pointwise subtracting the resulting list from θ. If the budget is ever exhausted, *inner* will stop early, and θ' will be shorter than θ. At this point, *gradient* will stop as well, but otherwise, it will recur.

4.4 Implementation

Our Adaptive Fuzz prototype comprises about 4000 lines of OCaml plus a few hundred lines of library code written in Adaptive Fuzz. The adaptive mode is interpreted; the data mode, where most computation happens, is compiled. At calls to **runfuzz**, the data mode piece is partially evaluated as described above, typechecked using the Fuzz typechecking algorithm, transliterated to OCaml (assuming it typechecks), compiled using the OCaml compiler, and executed against the database (in a separate process) to produce a sampled result, which is internalized as an adaptive mode value.

The runtime system uses both simple and advanced filters to check privacy bounds. It first tries the simple filter; if this comes out over budget, it instead tries the advanced filter. Because the advanced filter has large constant factors, this two-part check can be beneficial for programs involving only a small number of high-ε pieces.

4.5 Shifting the Data-Zone / Adaptive-Zone Border

The gradient descent running example shows one implementation of private gradient descent, but another can be easily defined if we shift the border between the data mode and the adaptive mode.

In Figure 8, each iteration of our implementation performs k ε-pieces (where the database has k parameters); if we move the inner loop of *gradient* into the data mode, each iteration will instead perform a single piece that updates all of θ's parameters at once.

In general, the advantage of putting more computation in the data mode is that it gives the analyst a stronger up-front guarantee of privacy usage. In essence, putting more code in the data mode effectively limits the amount of adaptivity in the algorithm, which means at each invocation of **runfuzz**, there is more static knowledge about the piece. For instance, in our implementation of gradient descent, it is possible that the budget will run out in the middle of computing the gradient, and the portion of the budget used up to that point would be essentially wasted. However, if it were written so that the gradient were computed in one piece, then that piece would be accepted or rejected altogether without wasting any of the budget.

On the other hand, putting the computation in the adaptive mode like we did increases adaptivity, which allows the advanced filter to be more effective. It is very likely that k ε-pieces will adaptively compose to use up less than $k\varepsilon$ worth of the privacy budget, and, because the Fuzz Verifier uses simple composition, $k\varepsilon$ is exactly how much the one big piece would use. Furthermore, by the nature of the adaptive filter, the savings will compound with every subsequent iteration of the algorithm. We cannot say exactly how much the budget is reduced—it depends on what pieces run before and after—but because we can always fall back to the simple filter (Theorem 2.10), we are guaranteed that this adaptive method will never be worse.

5 CASE STUDIES

We next report results from two case studies in which we implemented widely used statistical algorithms in Adaptive Fuzz and ran them on a real data set. Our main goals are (1) to show that Adaptive Fuzz is powerful enough to express these algorithms and (2) to estimate how many more queries an analyst is able to run with Adaptive Fuzz, compared to a language with only simple composition. In these examples, Adaptive Fuzz typically sees a 5× to 10× increase in the number of pieces it can run, with that number rising in some cases to above 40. Additionally, we explore the accuracy that these algorithms achieve for different values of ε.

We implemented two machine-learning algorithms that predict a person's income based on a database of census data generated from the `USCensus1990raw` data set [UCI KDD Archive 2016], which contains person records from the 1990 census conducted by the U.S. Census Bureau. The two algorithms support only numerical features, but the raw data set contains many categorical features as well, so we applied a transformation to convert categorical features to numerical ones. (For example, military service, which was initially coded as a discrete set of options such as "Now on active duty", "Active duty in the past", and "No service", was converted into 5 separate values that answer the binary question of whether each of those statements is true or false, which can easily be represented as a numerical feature ranging from 0 for false to 1 for true.) We took as the *label*—the feature for which the algorithms try to produce a predictor—the feature that indicates whether the person's total income was above $50,000. In total, our database contains one million records, each with 145 features in addition to the label. A full description of how we generated our database can be found in the extended version of this paper.

In the graphs in this section, each line represents a single run of the given algorithm, rather than an average of many runs. This highlights the randomness inherent in differential privacy and provides a realistic view of the kinds of results an analyst may see, but it also introduces some artifacts in the graphs: higher-noise runs sometimes "get lucky" and perform better than lower-noise ones, or vice versa.

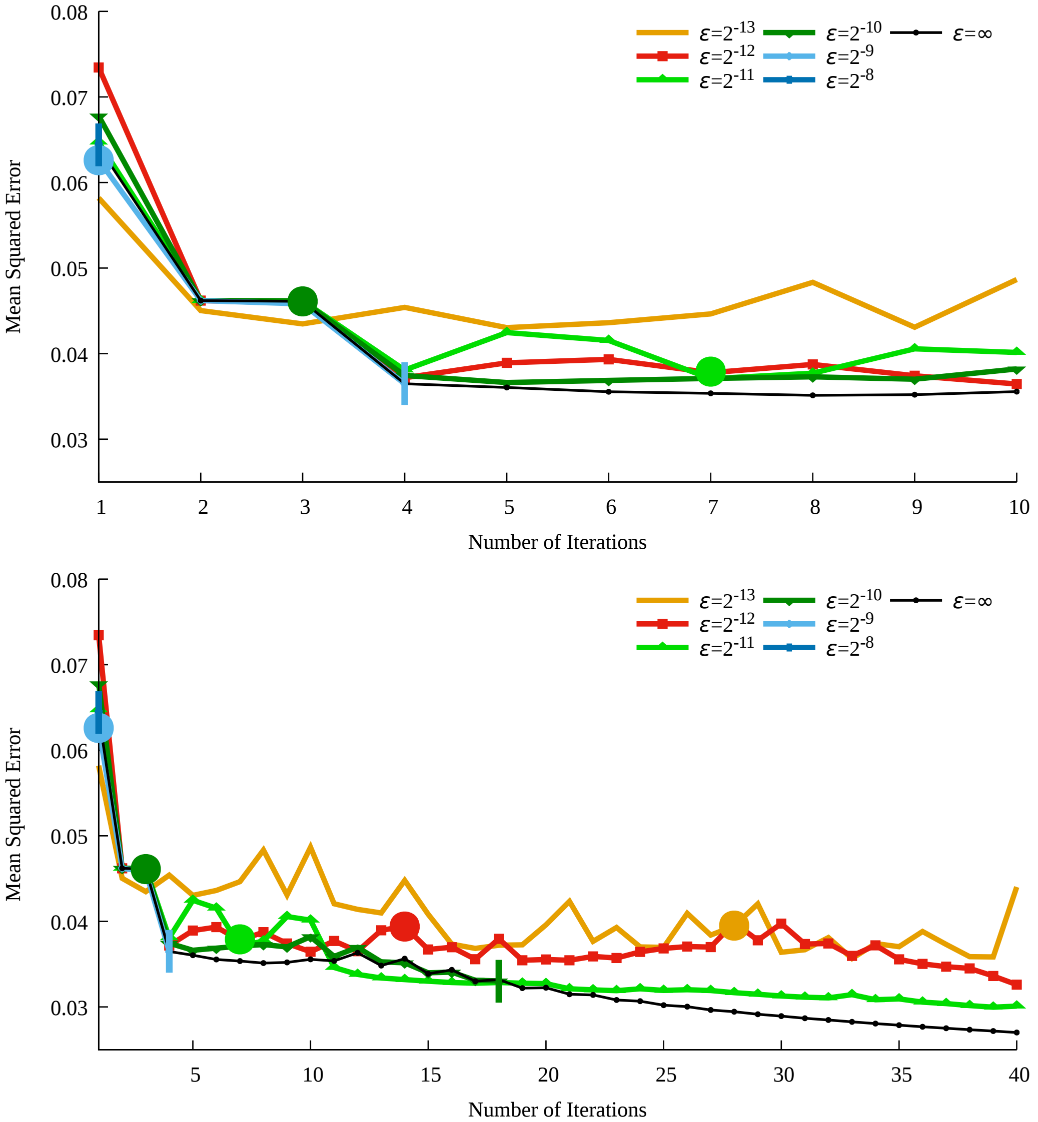

Fig. 9. The mean squared error after each iteration of gradient descent. The graph above shows just the first quarter of the iterations (1-10), and the graph below shows all of them (1-40). The lines extend until the filter cuts off computation (also marked by a vertical dash), and the larger dots indicate where it would stop using only simple composition (there is no large dot for $\varepsilon = 2^{-8}$ because, using simple composition, it was unable to perform even one iteration, and the vertical dash on the y-axis indicates that it performed exactly one iteration before being cut off by the filter). The line $\varepsilon = \infty$ indicates no noise, or how the algorithm would perform without differential privacy (there is no large dot because we are not tracking privacy budget in this case). The first point indicates MSE after one iteration, and the differences (as well as their convergence after iteration 2) are largely due to an artifact of starting at a vector of all zeros interacting with clipping done in calculating the residual.

5.1 Gradient Descent

Experimental Results. We ran this algorithm, implemented as in Figures 6 and 8, on our database with a global budget of $(2^{-1}, 2^{-30})$ and ε values ranging from 2^{-8} to 2^{-13} (and $\varepsilon = \infty$, i.e., no noise) and a fixed value of $\eta = 1/2$. (This value of η was chosen experimentally as one that works well for the database—much lower values reduce the performance of the algorithm and much higher values cause θ to diverge.) To show how accuracy is affected by ε, we plotted the mean squared error per iteration as a function of the chosen value of ε in Figure 9, and we let the algorithm run until the budget was exhausted or a maximum of forty iterations. Going further than 40 would have told us how well the algorithm continues to perform, but since only the runs with $\varepsilon \geq 2^{-11}$ run for more than 40 iterations, and these runs are mostly dominated by noise past this point, they are not worth showing. (One technical subtlety is that the analyst does not have direct access to the "true mean squared errors" shown in this graph. They could be calculated in a differentially private way, but then they would necessarily be only estimates, and the calculation would itself use some of the privacy budget.)

As the graph shows, the advanced filter allows much more analysis to be done. For instance, when $\varepsilon = 2^{-10}$, the advanced filter allows six times as many rounds (18 instead of 3), yielding a 30% improvement in accuracy. In fact, as ε shrinks, the gap between simple and advanced grows: when $\varepsilon = 2^{-11}$, the advanced filter allows 10 times as many rounds (72 instead of 7), at $\varepsilon = 2^{-12}$, this grows to 20 times (290 instead of 14), and at $\varepsilon = 2^{-13}$, 41 times (1163 instead of 28). Of course, there are diminishing returns in accuracy much earlier than that, and, with ε too small ($\varepsilon \geq 2^{-12}$), the noise overwhelms the signal. Looking at this graph, a thrifty analyst might choose $\varepsilon = 2^{-11}$. By using the advanced filter instead of simple composition, she could run 15 to 20 rounds and still have budget remaining for other tasks.

Recall that, if the curator had been using a non-parameter-adaptive composition theorem like standard advanced composition, then the analyst would have been forced to set the desired number of iterations in advance. Using a privacy filter, she can instead choose to stop the computation whenever the observed error has fallen to a satisfactory level.

5.2 Stagewise Regression

Stagewise regression is a commonly used variable-selection procedure for regression problems; a detailed description and analysis can be found in Tibshirani [2015]. Its goal is to select the variables that are most highly correlated to the label and thus will be most useful as predictors.

The algorithm initializes a weight vector over all features in the dataset (typically $(0, \ldots, 0)$) and then proceeds in rounds. Each round, it begins by computing the residual with the weight vector and finding the feature i that is most highly correlated with it (i.e., the coordinate of the gradient with the highest magnitude). It then increments the weight on feature i by a small step size η, in the opposite direction of its correlation with the residual. It can be run until an analyst-defined stopping condition is met, and at completion, the set of features with non-zero weight is the set of features selected by the algorithm.

Differentially private stagewise regression can be achieved by adding noise to the correlations between the features and the residual before determining which is the highest. This can be done so that each round uses a constant amount of privacy budget regardless of how many features there are.

Implementation. The implementation of stagewise regression is shown in Figure 10, where it is split into an adaptive-mode function (*stagewise*) and a data-mode function (*selectParam*). The algorithm begins in the adaptive mode with *stagewise*, where the first action is to run the piece produced by *selectParam*. The result is a pair of the selected parameter index and the value of the

```
function stagewise (ε : ℝ) (η : ℝ) (threshold : ℝ) (θ : ℝ list) (n : ℤ) : ℝ list {
  case runfuzz (2ε) (selectParam ε threshold θ n) of {
    inl (s)  ⇒ θ
  | inr (iv) ⇒ let (i, v) = iv;
               if (i < 0) then { θ } else {
                 θ′ = (listUpdate [ℝ] i ( + (negate (sign v) · η)) θ);
                 stagewise ε η threshold θ′
}}}

function selectParam (ε : ℝ) (threshold : ℝ) (θ : ℝ list) (n : ℤ)
                     (db : [2ε] (𝕀 × 𝕀 list) bag) : ◯(ℤ × ℝ) {
  sample i = reportNoisyMax [ℤ] ε 2 (fun (t : ℤ) { fun (b : [2] (𝕀 × 𝕀 list) bag) : ℝ {
                                       if t < 0 then { threshold }
                                                else { calcGrad n θ t b }}})
                                    (listtobag [ℤ] (countfrom (−1) (length [ℝ] θ)))
                                    db;
  if i < 0 then { return (i, 0) } else {
    sample val = add_noise (ε/2) (calcGrad θ i b);
    return (i, val)
}}
```

Fig. 10. The implementation of stagewise regression, presented as adaptive- and data-mode functions

gradient at that parameter, which is used to update the value of that parameter in θ by η in the right direction. Note that *sign* returns the sign of a value (either 1 or -1), and *listUpdate* takes an index i, an update function, and a list, and applies the function to the ith element of the list. If *selectParam* ever returns a parameter index less than zero, then *stagewise* halts, and otherwise, it recurs with the updated weight vector θ'.

The bulk of the computation occurs in *selectParam*, which relies critically on the report-noisy-max mechanism [Dwork and Roth 2014]. This takes a set of values and a quality function and selects the maximum-quality value from the set after a bit of ε-sensitive noise is added to each of the quality scores. Adaptive Fuzz implements this algorithm via the primitive

$$\begin{aligned} \mathit{reportNoisyMax} \;:\; & \textbf{forall}\, T.\, (\varepsilon : \mathbb{R}) \to (k : \mathbb{R}) \to (\mathit{qual} : T \to D \multimap[k]\, \mathbb{R}) \\ & \to (\mathit{ts} : T\ \mathit{bag}) \to (\mathit{db} : D) \multimap[\varepsilon]\, \bigcirc T. \end{aligned}$$

The first two arguments provide the sensitivity values for the future ones. The *qual* argument is the quality function, which produces a score for a value given the value and the database. The *ts* argument is the set of values, and the final argument is the database itself. The result is a distribution over the values in *ts*.

The idea behind *selectParam* is to use the features of the database as the set of values for report-noisy-max, and those features' gradient parameters as their quality, along with a dummy value whose score acts as a minimum threshold. Most of the function is just the call to *reportNoisyMax*. The first two arguments of this call declare that it will be ε differentially private and that its quality function is 2-sensitive. The quality function is the next argument; it calculates the feature's gradient parameter using the *calcGrad* function from Figure 6 (or returns the threshold when it is passed

-1, the dummy value). The set of features is next, and it is produced by using a simple function *countfrom* to produce a list from -1 to the length of θ and then converting it to a bag. The final argument is the database.

In the event that sampling *reportNoisyMax* produces an actual feature (not -1), the gradient at that feature is calculated, and because *calcGrad* is 2-sensitive, $\varepsilon/2$-sensitive Laplace noise is added to it. This is returned along with the index of the parameter. If sampling *reportNoisyMax* produces -1, then no feature was found to have a correlation higher than the threshold, and -1 is returned as the index (with an unused dummy value 0). In this case, *stagewise* does not recur, and the algorithm terminates.

Experimental Results. Stagewise regression is useful as a variable selection procedure before running gradient descent (since the privacy cost of gradient descent grows with the number of variables it is run on). Thus, we ran stagewise regression on our database with a fixed value of $\eta = 0.03$ and a threshold of 5% (maxing out at 50 iterations if the threshold is never reached) and followed this by gradient descent until the budget was exhausted (or maxing out at 200 rounds). We used ε values ranging from 2^{-8} to 2^{-13} (as well as once with $\varepsilon = \infty$, i.e., no noise), but note that stagewise regression runs with 2ε per iteration.

When $\varepsilon = \infty$, 9 features were selected after 38 rounds. For the other ε values, these numbers varied, but in general, the number of features selected grew as ε shrank. The selected features were not always the same, but certain ones were more common than others, e.g., sex, marital status, education, etc.

The mean squared error from the rounds of gradient descent after stagewise regression are shown in Figure 11. Using fewer variables has a slight negative impact on the accuracy, but it means that each round of the algorithm uses quite a bit less of the budget, so that many more rounds can be run. For instance, when $\varepsilon = 2^{-10}$, stagewise regression completes with 23 selected features, which means that gradient descent will require 23 ε-pieces per round instead of the 145 that would be required without stagewise regression. Thus, rather than stopping at 18 rounds as vanilla gradient descent was forced to, it can run for 109 rounds, leading to a 20% overall improvement in accuracy with the same privacy budget.

6 LIMITATIONS

Adaptive Fuzz uses OCaml's built-in random-number generator to sample distributions. However, these are pseudo-random numbers rather than truly random ones, which may lead to weaknesses in the differential privacy claim. This is not a fundamental flaw but rather an artifact of the prototype implementation; the `Random` library could easily be replaced with, e.g., a hardware-based, cryptographically secure random-number generator.

The runtime is also limited by physical processing capabilities since it needs to run on a real machine, and the analyst and curator are able to detect how long that processing takes, which opens the door to side channel and timing attacks. That is, the analyst could create a piece designed to take a long time (or even diverge) in certain conditions and then make strong assumptions when she does not receive results from the framework immediately. Haeberlen et al. [2011] addressed these types of attacks, but they are beyond the scope of this work.

Lastly, although the semantics for Adaptive Fuzz allows any type of value to be returned by a call to **runfuzz**, our implementation does not support returning functions. This would be tricky to implement, since Fuzz pieces are compiled and executed in a separate process, but even if it were easy to implement, it would add little benefit: for such a function to be approved as an output by the Fuzz typechecker, its dependence on the database would have to be so limited that it would be

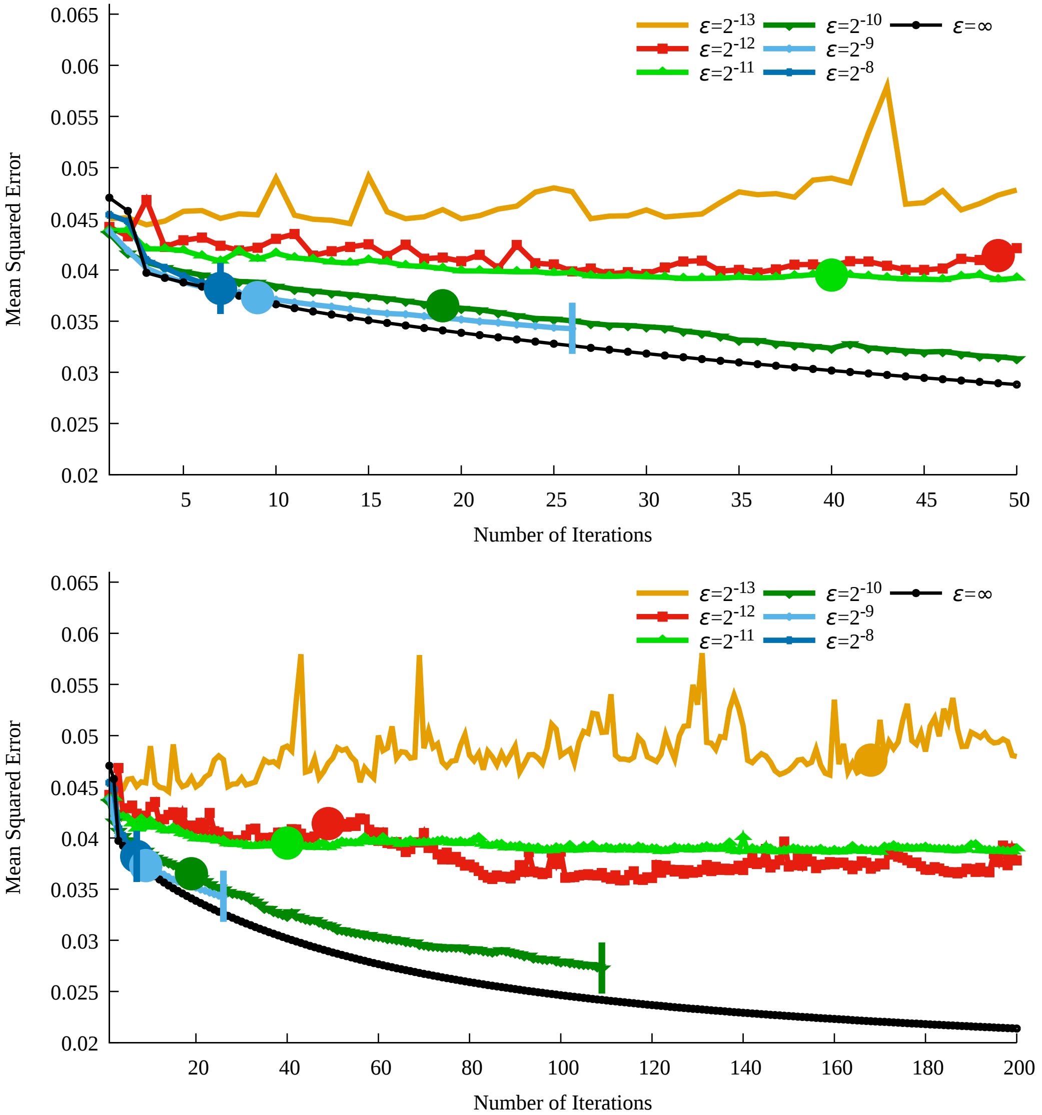

Fig. 11. The mean squared error of gradient descent performed after using stagewise regression for feature selection. The graph above shows just the first quarter of the iterations (1-50), and the graph below shows all of them (1-200). Again, the lines extend until the filter cuts off computation (also marked with a vertical dash), and the larger dots indicate where the algorithm would stop using simple composition. The line $\varepsilon = \infty$ indicates no noise, or how the algorithm would perform without differential privacy.

no more useful than just returning a constant and using it to build a function later, in the adaptive mode.

7 RELATED WORK

Differential privacy. Adaptive Fuzz provides differential privacy [Dwork et al. 2006b], which stands among the strongest privacy guarantees that have yet been proposed. A variety of other

privacy guarantees have been introduced over the years, including randomization [Agrawal and Srikant 2000], k-anonymity [Sweeney 2002], and l-diversity [Machanavajjhala et al. 2006]. These models are less restrictive than differential privacy, but they have been shown to be vulnerable to certain attacks [Evfimievski et al. 2002; Ganta et al. 2008; Kifer 2009; Li et al. 2007] that can cause private data to be disclosed. In contrast, differential privacy's provable guarantees hold even against worst case attackers with access to arbitrary auxiliary information.

Adaptive Fuzz's semantics shares a resemblance to the interactive structure proposed by Dwork et al. [2010]. Our semantics is slightly more general in that it can handle arbitrary adaptive composition instead of k-fold adaptive composition, and more realistic in that it handles computational issues such as non-termination, but that of Dwork et al. [2010] is more general in that it allows the analyst to choose different databases at different rounds.

Differential privacy implementations can be vulnerable to attacks, mostly having to do with the fact that computers use floating point numbers rather than true reals [Andrysco et al. 2015; Mironov 2012]. We consider these attacks important but orthogonal to our current work. A real deployed system based on Adaptive Fuzz would need to deal with them, using the methods suggested in these papers.

Privacy-preserving analysis tools. A number of existing tools [Barthe et al. 2015, 2012; Erlingsson et al. 2014; Gaboardi et al. 2013, aper; Haeberlen et al. 2011; McSherry 2009; Mohan et al. 2012; Narayan and Haeberlen 2012; Roy et al. 2010] can be used to work with sensitive data while preserving differential privacy. The main innovation in Adaptive Fuzz relative to these tools is support for advanced parameter-adaptive composition using filters. To the best of our knowledge, no current tool uses privacy filters, and existing tools support only basic sequential composition, with the exception of Psi [Gaboardi et al. aper], which uses an approximation of the exact composition theorem. Psi's goals are rather different from ours: it provides a user-friendly framework to run a collection of standard statistical analyses while managing privacy budgets, while Adaptive Fuzz is a general purpose programming language for private algorithms. One might imagine combining the two systems, allowing Psi power-users to write new analyses with Adaptive Fuzz.

Adaptive Fuzz is hardly the first differential privacy language to support adaptivity, especially considering that any system that only uses simple composition to string together pieces can be adaptive for free. Indeed, PINQ-like systems [Ebadi and Sands 2015; McSherry 2009] are based on a similar formalization to ours, including the clear distinction between analyst and curator, the use of streams to computationally sequence the probabilistic functions, and the release of intermediate results as observable outputs.

The semantics in §3 and the two-mode design of Adaptive Fuzz can be used to describe many languages that support differential privacy. For instance, PINQ [McSherry 2009] could be said to have a data mode consisting of just its primitives, each with a proof of their privacy, in which case its adaptive mode is all of C#. On the other end of the spectrum, Fuzz [Haeberlen et al. 2011] is a language entirely in the data mode, foregoing an adaptive mode altogether. Our semantics unifies this design space; with Adaptive Fuzz, we allow the programmer more freedom as to when and where she would like to switch between modes, which allows her to write complex pieces as well as adaptive behaviors between them.

Adaptive Fuzz builds on prior work using a combination of static analysis and linear typing [Reed and Pierce 2010] to certify differential privacy. In contrast to approaches that rely on run-time tracking, such as PINQ [McSherry 2009], this approach immediately rejects pieces that cannot be safely executed with the remaining privacy budget. The Fuzz type system is expressive enough to encode real examples; however, despite later extensions to richer type systems (such as the dependent types in DFuzz [Gaboardi et al. 2013] and the relational refinement types in HOARe^2 [Barthe et al.

2015]), it has proved challenging, with purely static analysis, to capture the end-to-end privacy cost of adaptive algorithms such as gradient descent after feature selection. Adaptive Fuzz's hybrid model of "piecewise static verification" is an attempt to get the best of both worlds.

Adaptive Fuzz uses the standard Fuzz typechecker as its verifier, but the system would also work with other, more complex verifiers. Notably, the natural extensions to Fuzz, such as DFuzz [Gaboardi et al. 2013] and HOARe^2 [Barthe et al. 2015], can be easily adapted into verifiers. However, due to how partial evaluation during **runfuzz** enables the use of arbitrary expressions at the type level and can even unroll loops, Adaptive Fuzz's typing is essentially as expressive as the more sophisticated dependent type analysis of DFuzz without requiring the complexity.

CertiPriv [Barthe et al. 2012] is a Coq-based framework that can be used to prove differential privacy from first principles. This approach is expressive enough to handle almost any piece, in principle, but it places a substantial burden on the analyst, who must have both time and expertise to write an end-to-end proof of privacy in Coq.

An alternative to verifying the privacy cost of individual pieces is simply to enforce an upper bound, e.g., by clipping outliers, as in Airavat [Roy et al. 2010], or using sample-and-aggregate techniques, as in GUPT [Mohan et al. 2012]. Unlike Adaptive Fuzz, this approach works for black-box computations written in any language, but it requires precise estimates of sensitivity: low estimates can waste precious privacy budget, while high estimates can lead to distorted results.

8 FUTURE WORK

We are working to expand our experimental evaluations to other statistical and machine learning algorithms that empirical scientists use in adaptive work flows, including other regression, classification, and hypothesis testing methods.

Adaptive Fuzz provides three differentially private primitives: the Laplace mechanism, the report-noisy-max mechanism, and the exponential mechanism (present in the language, although not used in this paper). We plan to expand its capabilities with primitives such as sparse vector [Dwork and Roth 2014] and Propose-Test-Release [Dwork and Lei 2009], which will allow the language to express a richer set of differentially private algorithms. Longer term, we hope to support user-defined plugins with accompanying machine-checked proofs.

Our work describes the most common form of differential privacy—that of a single curator with a single database and an analyst that queries this—but other forms exist as well. Another popular form is where the curator has multiple databases and the analyst can choose which to query from at every step. There is also a distributed form where different curators each hold a portion of a sensitive database. We would like to extend our formulation to permit more of these models.

As discussed in §4, Adaptive Fuzz does a bit of partial evaluation on the code given to a **runfuzz** expression before passing it to the verifier. This use of partial evaluation to enable typechecking appears to be novel, and we are currently investigating its theoretical properties (e.g., completeness of the partial evaluator / typechecker with respect to a declarative specification).

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback and suggestions. This work was supported in part by the National Science Foundation under grants CNS-1065060 and CNS-1513694, and a grant from the Sloan Foundation.

REFERENCES

Rakesh Agrawal and Ramakrishnan Srikant. 2000. Privacy-preserving Data Mining. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 439–450. https://doi.org/10.1145/342009.335438

Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 623–639.

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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 55–68.

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 (POPL '12)*, Vol. 47. ACM, ACM, New York, NY, USA, 97–110. https://doi.org/10.1145/2103656.2103670

Raef Bassily, Adam Smith, and Abhradeep Thakurta. 2014. 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*. IEEE Computer Society, 464–473.

Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006a. Our data, ourselves: Privacy via distributed noise generation. In *Advances in Cryptology-EUROCRYPT 2006*. Springer, 486–503.

Cynthia Dwork and Jing Lei. 2009. Differential privacy and robust statistics. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. ACM, 371–380.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006b. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography*. Springer, 265–284.

Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3-4 (2014), 211–407.

Cynthia Dwork, Guy N Rothblum, and Salil Vadhan. 2010. Boosting and differential privacy. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*. IEEE, 51–60.

Hamid Ebadi and David Sands. 2015. Featherweight PINQ. *CoRR* abs/1505.02642 (2015). http://arxiv.org/abs/1505.02642

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 (CCS '14)*. ACM, New York, NY, USA, 1054–1067. https://doi.org/10.1145/2660267.2660348

Alexandre Evfimievski, Ramakrishnan Srikant, Rakesh Agrawal, and Johannes Gehrke. 2002. Privacy Preserving Mining of Association Rules. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02)*. ACM, New York, NY, USA, 217–228. https://doi.org/10.1145/775047.775080

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 357–370.

Marco Gaboardi, James Honaker, Gary King, Kobbi Nissim, Jonathan Ullman, and Salil Vadhan. Working Paper. PSI (Îĺ): a Private data Sharing Interface. In *Theory and Practice of Differential Privacy*. New York, NY.

Srivatsava Ranjit Ganta, Shiva Prasad Kasiviswanathan, and Adam Smith. 2008. Composition Attacks and Auxiliary Information in Data Privacy. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08)*. ACM, New York, NY, USA, 265–273. https://doi.org/10.1145/1401890.1401926

Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, Berkeley, CA, USA, 33–33. http://dl.acm.org/citation.cfm?id=2028067.2028100

J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. 2014. Differential Privacy: An Economic Method for Choosing Epsilon. In *2014 IEEE 27th Computer Security Foundations Symposium*. 398–410. https://doi.org/10.1109/CSF.2014.35

Aaron Johnson and Vitaly Shmatikov. 2013. Privacy-preserving Data Exploration in Genome-wide Association Studies. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13)*. ACM, New York, NY, USA, 1079–1087. https://doi.org/10.1145/2487575.2487687

Peter Kairouz, Sewoong Oh, and Pramod Viswanath. 2015. The Composition Theorem for Differential Privacy. In *Proceedings of The 32nd International Conference on Machine Learning*. 1376–1385.

Shiva P Kasiviswanathan and Adam Smith. 2014. On the'Semantics' of Differential Privacy: A Bayesian Formulation. *Journal of Privacy and Confidentiality* 6, 1 (2014), 1.

Daniel Kifer. 2009. Attacks on Privacy and deFinetti's Theorem. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/1559845.1559861

Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. 2007. t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. In *2007 IEEE 23rd International Conference on Data Engineering*. 106–115. https://doi.org/10.1109/ICDE.2007.367856

Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. 2006. ℓ-Diversity: Privacy Beyond κ-Anonymity. In *Proceedings of the 22Nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, Washington, DC, USA, 24–. https://doi.org/10.1109/ICDE.2006.1

Frank McSherry and Ilya Mironov. 2009. Differentially Private Recommender Systems: Building Privacy into the Net. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '09)*. ACM, New York, NY, USA, 627–636. https://doi.org/10.1145/1557019.1557090

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

Elinor Mills. 2006. AOL sued over Web search data release. (Sept. 2006). cnet, http://www.cnet.com/news/aol-sued-over-web-search-data-release/.

Ilya Mironov. 2012. On Significance of the Least Significant Bits for Differential Privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 650–661.

Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. 2012. GUPT: Privacy Preserving Data Analysis Made Easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 349–360.

Jack Murtagh and Salil Vadhan. 2016. The Complexity of Computing the Optimal Composition of Differential Privacy. In *Theory of Cryptography*. Springer, 157–175.

Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially Private Join Queries over Distributed Databases.. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 149–162. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/narayan

Arvind Narayanan and Vitaly Shmatikov. 2008. Robust De-anonymization of Large Sparse Datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. 111–125. https://doi.org/10.1109/SP.2008.33

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 (ICFP '10)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/1863543.1863568

Ryan M. Rogers, Aaron Roth, Jonathan Ullman, and Salil P. Vadhan. 2016. Privacy Odometers and Filters: Pay-as-you-Go Composition. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 1921–1929.

Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and Privacy for MapReduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, Vol. 10. USENIX Association, Berkeley, CA, USA, 297–312. http://dl.acm.org/citation.cfm?id=1855711.1855731

Ryan Singel. 2009. Netflix Spilled Your "Brokeback Mountain" Secret, Lawsuit Claims. (Dec. 2009). Wired, http://www.wired.com/2009/12/netflix-privacy-lawsuit/.

Latanya Sweeney. 2002. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 5 (Oct. 2002), 557–570.

Ryan J. Tibshirani. 2015. A General Framework for Fast Stagewise Algorithms. *The Journal of Machine Learning Research* 16, 1 (Jan. 2015), 2543–2588. http://dl.acm.org/citation.cfm?id=2789272.2912080

UCI KDD Archive. 2016. 1990 US Census Data Set. (2016). Available from https://kdd.ics.uci.edu/databases/census1990/USCensus1990raw.html.

Philip Wadler. 1990. Linear Types Can Change the World. In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*. 546–566.