Project Description: CAREER: A Policy-Agnostic Programming Framework for Statistical Privacy

1 Introduction

The digitization of data promises to transform our understanding of, among other areas, health and medicine. Unfortunately, security and privacy concerns present a growing obstacle to this vision. To realize the full potential of data-driven research, we need to disentangle the problem of protecting sensitive information from the goals of the data analysis.

Consider patient records in a health database and a query to count the number of patients with a positive HIV diagnosis. Whether this count leaks "too much" information depends on how easily someone may deduce a given person is HIV-positive. Conservatively, only those authorized to view individual diagnoses could be allowed to see the count. But the count may be large enough that it obscures individual patient identities, making it safe to release. And if the count were part of an algorithm for machine-learning diagnosis predictions, it may be even more the case that the results are safe to release.

From this example, one can see that determining the safety of data release may depend on both the inputs and the computations performed using them. Just as analyzing sensitive data requires reasoning about the privacy implications, reasoning about privacy also involves considering the data analysis algorithms. Needing to reason about core functionality in tandem with privacy creates a programming bottleneck. To enable data analysts without Computer Science backgrounds to produce privacy-preserving analyses, we need *privacy-aware programming models* and *automated tools*.

Our work on *policy-agnostic* programming addresses the issue of programmer burden by factoring out security and privacy concerns from core program functionality, allowing computations over sensitive data to be *agnostic* to their security and privacy requirements. In prior work [4, 35, 54, 55], we developed a policy-agnostic solution for *information flow security*. Without our solution, programmers must implement information flow policies as conditional access checks across the program, reasoning about both how to resolve policies for eventual viewers and how the program is computing with sensitive values. The policy-agnostic programming model allows programmers to specify each information flow policy once and rely on the compiler and language runtime to enforce the policies. Specifications also describe what should happen when checks fail: policies additionally specify alternate default values.

Information flow policies, while useful, can be too conservative. With an information flow policy, an unauthorized viewer Eve may not see the result of *any* computation involving a patient Alice's diagnosis, including the results from a machine learning algorithm that uses Alice's information to predict diagnoses. In many cases, however, the results of such learning algorithms are more likely to help advance knowledge than they are likely to hurt individuals. While there exist *declassification mechanisms* [42] for information flow and techniques such as k-anonymity [48, 49] and ℓ-diversity [30] for de-identifying sensitive data, these solutions can be difficult to use by those untrained in data privacy. Statistical privacy techniques such as *differential privacy* [15, 16, 17] are more promising, as they provide formal mathematical frameworks for protecting individual values while allowing the release of aggregate results.

The main idea in this proposal is the development of a policy-agnostic programming framework for differential privacy. Prior work on language-based techniques for differential privacy has focused only on preventing leaks, rejecting programs either statically [21, 37] or dynamically [32, 40]. The programmer remains responsible for constructing programs that meet privacy requirements, a process that involves reasoning about the degree to which individual computations preserve privacy. In this five-year program, we plan to develop a policy-agnostic solution for differential privacy. The goal is to be able to support implementations of sophisticated machine learning algorithms that are agnostic to the privacy concerns and allow the programmer to rely on the compiler and runtime to make decisions that satisfy privacy requirements.

The central hypothesis is that we can automate the exploration of the trade-off space between accuracy and privacy. With differential privacy, there are two ways of making a computation more private: (1) by adding noise to data values, or (2) by performing a similar computation that leaks less information by introducing more error. The existing work on analyzing the accuracy/privacy trade-offs [24, 25, 28] involves manually exploration of the error introduced by specific techniques. To generalize this approach, we propose a model where the programmer can *explicitly* trade off accuracy and privacy through *exposing algorithmic choice* to the compiler and runtime. To support this model, we plan to develop dynamic and static reasoning tools for *automatically* exploring these trade-offs.

We propose a new language policy-agnostic differential privacy, called JOSTLE, with both a dynamic execution strategy and compilation support.[1] The proposed work involves the following tasks:

1. **A dynamic semantics for policy-agnostic differential privacy.** We will develop a programming model that allows the programmer to specify privacy and accuracy constraints, as well expose algorithmic choice about alternate similar executions that the program may perform. In this first task, we develop a dynamic execution model that makes decisions to satisfy these specifications, so that data analyses may be implemented agnostic to privacy concerns. We will develop the programming model in collaboration with researchers and engineers working with medical records.

2. **A decidable type system for probabilistic relational verification.** In our prior work [35], we demonstrated how to use decidable refinement type-checking for automated program repair supporting policy-agnostic information flow. Towards supporting static reasoning for policy-agnostic statistical privacy, we plan to develop a *decidable probabilistic relational type system*, first by extending *liquid types* [38, 50] to be relational [11, 23] and then by adding a probabilistic component [5, 6].

3. **A compilation framework for policy-agnostic differential privacy.** As a fully dynamic exploration of privacy and accuracy trade-offs can become computationally expensive, we plan to complement our dynamic execution solution with compiler support for algorithmic choice. We plan to use our decidable probabilistic relational type inference to allow the compiler to statically navigate a significant subset of the accuracy/privacy trade-offs expressible in JOSTLE.

While the scope of this five-year proposal involves focusing on supporting analyses over health and medical data, we plan to develop generalizable techniques. We are collaborating with researchers analyzing genomic data and with health engineering teams to develop realistic case studies.

Evaluation plan. A policy-agnostic solution for statistical privacy would allow the medical researcher to rely on the query infrastructure to implement privacy concerns. Success means that we have a programming framework that (1) is expressive enough to support a representative set of programs, (2) provides strong mathematical guarantees, (3) reasons about a significant portion of these programs statically and automatically, (4) demonstrates reasonable performance, both in terms of compilation times and runtime overheads. We plan to evaluate JOSTLE using example programs written for the PINQ [32] and Airavat [39] languages, as well as machine learning benchmarks from our collaboration with the University of Pittsburgh Medical Center (UPMC). (We have attached a letter of collaboration from our liaison, Carnegie Mellon University's Center for Machine Learning and Health.)

Integration of research and education. Previously, only those with the requisite background in statistics and program analysis could reason about whether and why programs are differentially private. By automating privacy analysis for arbitrary programs, JOSTLE makes it possible for non-experts to build up intuitions

[1] The tool is named such for two reasons. First, making computations differentially private often involves "jostling" values by adding noise. Second, our approach "jostles" the entire computation by replacing some sub-computations.

about what makes programs differentially private. This will increase the accessibility of statistical privacy, both in the classroom and for practitioners. Towards developing a tutorial based on this, we plan to incorporate JOSTLE into the junior-level course that the PI co-designed and co-taught. We also plan to release both the code and materials publicly.

Basis of confidence. The proposed work builds on the PI's prior work on policy-agnostic information flow [4, 35, 54, 55] and the PI's experience working on F* [47], a dependently typed language for verifying security properties. Towards automated verification, the PI co-developed Verve [53], an operating system automatically verified for type safety. (The Verve paper won Best Paper Award at PLDI 2010.) While interning at Facebook the PI built Ask Reeves [26], a verifier for access control policies, and became familiar with many security and privacy issues that arise in practice. The PI has ongoing collaborations with experts in probabilistic and relational reasoning, as well as genomics researchers and engineering teams working with health records. In spring 2017 the PI co-designed and co-taught a junior-level formal security course [19] that included a unit on statistical privacy.

2 Goal: Policy-Agnostic Differential Privacy

In this section, we provide background on differential privacy, discuss the programming bottlenecks in constructing differentially private computations, and present our vision for policy-agnostic differential privacy.

2.1 A Primer on Differential Privacy

The formulation of ϵ-differential privacy [15, 16, 17] talks about the ability to distinguish between data set D and a data set D' that differs only by a small amount. At a high level, a given computation is differentially private if the results of the computation make it unlikely to differentiate whether the computation happened using D or D'. Formally, let $\mathcal{X}$ denote a data domain. We call data sets D and D' *neighbors* (written $D \sim D'$) if we can derive D' from D by replacing a single data point with some other element of $\mathcal{X}$. We define differential privacy as follows:

Definition 1 (Differential Privacy [17])**.** *Given a fixed $\epsilon \geq 0$, a randomized algorithm $A : \mathcal{X}^* \to \mathcal{O}$ is ϵ-differentially private if for every pair of neighboring data sets $D \sim D' \in \mathcal{X}^*$ and for every event $S \subseteq \mathcal{O}$:*

$$Pr[A(D) \in S] \leq e^{\epsilon} Pr[A(D') \in S].$$

The scaling factor ϵ exists to give some slack to the kinds of computations that are allowed. The probabilities in this expression come from the randomness of the output of A. Note that this definition is a property of the function A and not on the data A uses to compute output.

2.1.1 Adding noise

To get the property that viewers infer individual data points with low probability, the function A needs to produce random output. Since not all data analysis algorithms are sufficiently random, many methods add noise to make functions differentially private. A standard way of adding noise is to use the *Laplace mechanism*. The Laplace Distribution centered at 0 with scale b has probability density $\text{Lap}(z|b) = \frac{1}{2b} e^{-\frac{|z|}{b}}$. We say $X \sim \text{Lap}(b)$ when X has Laplace distribution with scale b. If $f : \mathcal{X}^* \to \mathbb{R}^d$ is an arbitrary d-dimensional function, then we define the ℓ_1-*sensitivity* of f as $\Delta_1(f) = \max_{D \sim D'} ||f(D) - f(D')||_1$. The Laplace mechanism with parameter ϵ adds noise drawn independently from $\text{Lap}(\frac{\Delta_1(f)}{\epsilon})$ to each coordinate of $f(x)$. We have the useful property that the Laplace mechanism is differentially private [17].

```
double NoisyCount(double epsilon):
{
    if (myagent.Alert(epsilon))
        return mysource.Count() + Laplace(1.0/epsilon)
    else
        throw new Exception("Access is denied")
}
```

Figure 1: Implementation of the NoisyCount function in the PINQ interface [32].

2.1.2 Properties of differentially private computations

Differential privacy has a couple of useful computational properties:

Theorem 1 (Post Processing [17])**.** *Let $A : \mathcal{X}^* $ be an ϵ-differentially private algorithm and let $f : \mathcal{O} \to \mathcal{O}'$ be any function. Then the algorithm $f \cdot A : \mathcal{X}^* \to \mathcal{O}'$ is also ϵ-differentially private.*

Theorem 2 (Composition [17])**.** *Let $A_1 : \mathcal{X}^* \to \mathcal{O}, A_2 : \mathcal{X}^* \to \mathcal{O}$ be algorithms that are ϵ_1- and ϵ_2-differentially private, respectively. The algorithm $A : \mathcal{X}^* \to \mathcal{O} \times \mathcal{O}$, defined as $A(x) = (A_1(x), A_2(x))$, is $(\epsilon_1 + \epsilon_2)$-differentially private.*

Post-processing allows us to safely use the results of differentially private algorithms, as it implies that any algorithm based on the output of a differentially private algorithm is also differentially private. Composition allows us to combine differentially private algorithms safely. With this formulation of privacy, we can think of ϵ as the *privacy budget* and individual computations as depleting the privacy budget.

2.2 Differential Privacy and Programmer Burden

Prior work on language-based techniques aims to make it possible for non-experts to ensure that programs are differentially private, but they leave the programmer responsible for the reasoning required to construct these programs. Dynamic language-based solutions [32, 40] expose restricted functionality that has a well-understood interaction with the privacy budget. Static solutions [21, 37] expose differential privacy constructs via types and use program analyses that support reasoning about statistical leakage. The issue is, however, that these approaches serve only to *prevent* leaks, rejecting programs that do not satisfy the differential privacy requirements. The programmer remains responsible for choosing algorithms so that computations do not exceed the privacy budget.

More concretely, consider the query *"How many patients at this hospital are over the age of 40?"*. Performing this query typically involves filtering the patients that satisfy the over-40 constraint and then counting the results. Doing this in PINQ [32] would involve filtering through the custom interface and then using the NoisyCount function (implementation shown in Figure 1) for differentially private counting, rather than the regular count function. When the programmer calls NoisyCount with a given budget, the function will add noise to the actual count if the computation does not exceed the budget ϵ, or raise an error otherwise. PINQ ensures that all computations are differentially private by providing interfaces only to differentially private functions and by performing the appropriate bookkeeping on how much budget remains. If the programmer does not implement the desired data analysis to be appropriately private, however, the query returns "Access is denied" and the system revokes access to the database.

Similarly, static approaches will reject programs that are not differentially private but make the programmer responsible for writing such programs. In Fuzz [37], a language with a differential privacy type system, the programmer can implement our query as follows:

$$\lambda d :!_\epsilon \text{dB}.add\_noise\ (size\ (filter\ over\_40\ d))\ :\ !_\epsilon \text{dB} \multimap \text{M}\ \mathcal{R},$$

Here, add_noise adds Laplace noise and $size$, $filter$, and $over\_40$ are available functions over the data values. In Fuzz, the typing judgment for expression e is $\vdash e : !_\epsilon \text{dB} \multimap \text{M}\ \mathcal{R}$, where the type M $\mathcal{R}$ represents discrete probability distributions over output $\mathcal{R}$, $\multimap$ represents the space of 1-sensitive functions from $!_\epsilon$dB to M $\mathcal{R}$, and the type $!_\epsilon$dB represents databases whose metric is that of dB multiplied by ϵ. The type system rules out differential privacy violations, but the programmer is responsible for writing programs that type-check.

Even in this small computation, the programmer is responsible for combining functionality—and, in the static case, adding noise—so that computing the final result does not exceed the privacy budget. Most data analyses are more complex than this one and so impose even more burden on the programmer. For instance, when using machine learning the programmer becomes responsible for selecting not only which methods to use, but also input and iteration parameters that may affect the differential privacy of the computation. With these existing approaches, the programmer remains responsible for selecting algorithms and parameters that ensure the overall computations are within budget. Doing any data analysis requires reasoning about how sub-computations deplete the privacy budget.

2.3 The Vision of Policy-Agnostic Differential Privacy

The goal of policy-agnostic differential privacy is to allow the programmer to be able to implement a computation independently of privacy considerations and rely on the runtime and compiler to ensure that executions adhere to privacy and accuracy requirements. We want to allow programmers to write the following code to answer our query, where **budget** is a keyword specifying the global budget:

```
budget E
def getPatientsOverForty(d):
  size (filter over_40 d)
```

It turns out we can't *quite* write this, as the runtime and compiler need to know the space of possible computations it may explore to satisfy the privacy budget, as well as desired accuracy. Accuracy specification is important because adding introducing more error is an easy way to make computations more private. Instead, a program in our new language JOSTLE will look like this:

```
budget E
def getPatientsOverForty(d)@accuracy(a):
  size similar{noisy_size} (filter over_40 d)
```

The runtime and compiler become responsible for choosing whether to use `noisy_size` instead of `size`, based on privacy and accuracy requirements. Reasoning about accuracy and privacy may seem straightforward for this function, but as implementations become complex it is increasingly helpful to automate this reasoning. JOSTLE automates this reasoning, propagating information about privacy and accuracy to make decisions about whether to replace computations with ones marked as **similar**. Here we specify the accuracy parameter a explicitly, but the runtime and runtime are responsible for propagating these specifications when they are not explicitly specified. Note that to support non-expert data analysts, we can abstract over **similar**, building notions of similarity into standard libraries.

While we plan to support this programming model fully dynamically at first, we propose the development of a decidable type system for probabilistic relational reasoning so that we can reason about these trade-offs at compile time and perform static analysis to provide compiler support for constructing differentially private programs when possible. The result of our research will be a policy-agnostic framework for differential privacy that we hope to generalize for other formulations of statistical privacy.

```
def showPaper(self, pid):
    u = self.getCurrentUser()
    authors = self.getAuthors(pid)
    phase = self.getPhase()
    status = self.getStatus(pid) \
        if (phase == Phase.NOTIFIED) else Status.NODECISION
    session = self.getSession(pid) if (status == Status.ACCEPTED) else ""
    print ('\n'.join[u, authors, status, session, print])
```

Figure 2: A version of the EDAS functionality, with policy checks highlighted. The policy-agnostic version may leave off these checks.

3 Prior Work: Policy-Agnostic Programming for Information Flow

Our prior work on policy-agnostic programming [4, 35, 54, 55] shows how to factor information flow policies out from core functionality. While statistical privacy is different than information flow, this proposal builds on the following techniques from our prior work: (1) the design of a programming model where the compiler and runtime are responsible for finding executions to satisfy high-level constraints, (2) the design of a language where the programmer specifies alternate functionality for when policy checks fail, (3) a dynamic execution strategy based on multi-execution, and (4) a compilation strategy based on type-based program analysis. In this section, we explain information flow bugs and then present the programming model [55], the dynamic semantics based on simulating multiple simultaneous executions [4, 54], and the static type-driven repair solution [35].

3.1 Information Leaks and Programmer Bottleneck

Consider the following documented bug [1] in the EDAS conference management system. Conference management systems for academic conferences typically support a range of confidentiality policies to help preserve anonymity in the reviewing process, as well as confidentiality about discussion internal to the program committee. In many conference management systems, chairs may also tentatively mark papers as "accepted" during the discussion phase before the committee makes final decisions. The convention is that this tag is internal to the program committee and should not be visible to authors. The bug, however, is that the EDAS system shows an additional field, the corresponding conference session if a paper has been marked as "accepted," regardless of whether authors are permitted to know the acceptance statuses of their papers. This is an information flow bug: the system allows an author to infer whether a paper has been accepted based on the displayed value of the session.

It is unsurprising that information leaks are so prevalent: using most existing programming paradigms, the programmer must implement information flow policies as conditional access checks across the program and any missing check can cause a leak. To prevent the EDAS leak, the programmer must write something like the code in Figure 2. And there are two main ways in which this code can become even more complicated: (1) each of the other fields of a paper could have their own policies, with some policies depending on each other and (2) these values could be used in more convoluted computations. As a result, programmers must worry about information flow policies throughout the code when implementing core functionality. Even using static and dynamic techniques for ensuring the absence of information flow leaks [3, 9, 13, 14, 27, 29, 36, 39, 41, 46, 47, 56], programmers must correctly perform this reasoning to avoid compile-time errors, runtime exceptions, or silent failures.

```
class Paper(JModel):
    status = CharField(max_length=256)
    session = CharField(max_length=256)

    # Policy for the session field.
    @staticmethod
    def j_restrict_status(paper, viewer):
    return (viewer.phase == Phase.NOTIFIED)

    @staticmethod
    def j_get_public_status(paper):
        return Status.NODECISION
```

Figure 3: Data schema specification with policies.

3.2 A Policy-Agnostic Programming Model and Dynamic Semantics

Policy-agnostic information flow mitigates programmer burden by allowing programs to associate policies directly with sensitive data values, rather than implementing them as checks and filters across the program. Developing a policy-agnostic programming model for information flow is challenging because the programming model requires (1) an execution strategy that can change control flow based on policy checks and (2) specifications for what should happen when checks fail. We have implemented our solution in the Jeeves programming language [4, 55] and extended the programming model to database-backed applications in the Jacqueline web framework [54]. In Jeeves, programmers implement the EDAS functionality simply by writing the un-highlighted code in Figure 2.

Our solution is to express sensitive values as *faceted values* $\langle \ell \ ? \ V_H \ : \ V_L \rangle$, where V_H is the value visible to high-confidentiality (privileged) viewers, V_L is the value visible to low-confidentiality viewers, and $\ell \in \{\mathsf{high}, \mathsf{low}\}$ is a label. A key innovation in our solution is that the *language runtime and compiler*, rather than the programmer, decides the value of the guard ℓ based on a higher-level policy language. Each sensitive value may have a different label ℓ_i and the programmer specifies policies for the ℓ_i in terms of the program state. Policies are functions that take the viewer as an argument and return Boolean values that describe when a label is allowed to be `high`.

To illustrate how policy declarations work in practice, in Figure 3 we show example Jacqueline web framework code for the `Paper` data schema. In Jacqueline, programmers can specify policies associated directly with *data schemas* describing the layout of data across the application and database. The first part of the declaration looks just like a regular data schema definition: we define a data class corresponding to `Paper` models and we define two fields, `status` and `session`. The `j_restrict_status` method defines the policies, taking the current `Paper` row and the viewer as arguments, and the `j_get_public_status` method defines the alternate default value, taking the current row as the argument. The Jacqueline framework attaches policies to data values when they leave the database and evaluates policies to determine how to display results to a given viewer.

Our dynamic semantics for *faceted execution* [4] describes how to ensure all values computed using sensitive values become associated with the appropriate policies. Faceted execution simulates simultaneous multiple executions on different facets, for instance evaluating the expression $\langle j \ ? \ 0 \ : \ 1 \rangle + \langle k \ ? \ 2 \ : \ 3 \rangle$ yields the faceted value $\langle j \ ? \ \langle k \ ? \ 2 \ : \ 3 \rangle \ : \ \langle k \ ? \ 3 \ : \ 4 \rangle \rangle$. The dynamic semantics also describes keeping track of labels while evaluating conditional statements so as to prevent implicit flows. We prove a modified noninterference theorem that takes into account that policies and viewers may both depend on computations involving sensitive values. We extend the semantics and guarantees to database-backed applications.

3.3 Static Type-Driven Program Repair for Policy-Agnostic Information Flow

We have also developed a type-driven program repair solution [35] that rewrites programs to adhere to the policy-agnostic semantics. We observe that we can decompose the problem of sound program repair into program verification and program synthesis problems, using the results of failed program verification to produce small localized program synthesis problems. The key to automation in our technique is our use of *decidable refinement types*, based on *liquid types* [38, 50], to reason about information flow security. The type system allows for decidable type-checking, precise error localization, and decidable type inference. We implemented our technique in a Liquid Haskell-based system called LIFTY.

In LIFTY, the programmer implements each policy once, as a type associated with the sensitive data value it protects. The rest of the program may be policy-agnostic, as the compiler will infer the intermediate types. The programmer introduces policies using special type constructor $\langle T \rangle^p$ ("T tagged with policy p"), denoting that values of type T may only be seen by a user u in a store s provided that $p(u, s)$ holds. As in Jeeves and Jacqueline, the programmer also specifies an alternate default value:

$$\mathsf{status} :: \mathsf{PaperId} \to \mathsf{Ref}\ \langle \mathsf{Status} \rangle^{\lambda s.\lambda u.s[\mathsf{phase}] = \mathsf{Notified}}$$
$$\mathsf{default\ status} = \mathsf{NoDecision}$$

LIFTY combines refinement type reconstruction, along with a monadic encoding of information flow to propagate policies through all computations depending on sensitive values, to repair programs.

The key insight with LIFTY is that we can use the result of a failed type-checking attempt to determine which sensitive data accesses we need to guard with a conditional access check, as well as the appropriate checks. The LIFTY compiler works as follows. First, the type checker propagates viewer information back from output statements, parameterized by the viewer, to ensure that all policies are satisfied. If type-checking fails, the compiler identifies the smallest subterm that should be wrapped in a check, as well as the corresponding type that failed to check. LIFTY relies on liquid type inference to assign the strongest possible predicate to every sensitive subterm in the program. The compiler then uses type-based synthesis [34] to generate checks corresponding to the predicates. The compiler ensures that the checks appropriately implement the desired policies and that they do not violate policies on other implicated sensitive values.

4 Task: A Dynamic Semantics for Policy-Agnostic Differential Privacy

Towards developing a policy-agnostic programming framework for differential privacy, we will begin by (1) developing a programming model for navigating these trade-offs and (2) developing a dynamic execution solution for this programming model. We will formalize our dynamic semantics and prove properties about executions. The main challenges in this direction involve understanding the trade-off space for well-studied algorithms and determining ways to generalize this to arbitrary programs.

4.1 A Programming Model for Trading Off Accuracy and Privacy

We begin by developing a programming model where the language runtime can take responsibility for navigating privacy/accuracy trade-offs in programs. In our new language JOSTLE the programmer specifies (1) an overall budget ϵ for an entire set of computations, (2) accuracy specifications for functions of interest, and (3) alternate equivalent computations that the runtime can choose from for satisfying privacy and accuracy requirements. We introduce three new operators: **budget** for specifying the global budget, **@accuracy** for specifying accuracy constraints associated with computations, and **similar** $\{\dots\}$ for specifying a set of computations that may be used interchangeably for a given purpose. Recall our use of size **similar**{ noisy_size } from the example. The programmer may annotate functions with **@accuracy** if

the function, or its sub-computations, may be replaced with a less accurate version. The **similar** keyword specifies similar functions that may be used instead of the provided function. It is up to the runtime and compiler to make decisions about selecting computations of sufficient accuracy to meet the privacy budget.

4.2 Using Faceted Execution to Explore Trade-offs

While it is more straightforward for the programmer to reason at a per-function level about equivalent computations, it is far more complicated to reason over a larger computation that is iterating over these function calls and combining the results from many different sub-computations. To automate the reasoning about privacy/accuracy trade-offs, we will develop a variant of faceted execution that (1) creates faceted values based on **similar** declarations, (2) dynamically tracks accuracy and privacy budget use for each facet, and (3) chooses a final result that satisfies privacy and accuracy constraints, if one exists. While the PINQ interface functions are only able to raise an error, the JOSTLE runtime may select alternate executions that use less privacy budget and introduce more error. Faceted execution allows the JOSTLE runtime to make these decisions with full information about the entire computation.

Before we develop techniques for analyzing accuracy and privacy trade-offs of arbitrary algorithms, we will use results from recent work on exploring these trade-offs for well-studied algorithms [24, 25, 28]. Ligett *et al.* [28] have developed an automated mechanism for searching this trade-off for a set of Empirical Risk Minimization algorithms, including ridge regression and output perturbation. They define a notion of *privacy loss*, provide privacy loss functions over data, and propose an algorithm for searching over possible values of ϵ to find that smallest ϵ with satisfactory *excess risk*. We plan to use the results of this work to manage the privacy/risk trade-off for computations that involve multiple different sub-algorithms. We will use faceted execution to track the results we can get for different levels of privacy and accuracy and use these facets to select intermediate results that satisfy the global privacy budget and accuracy constraints.

We plan to generalize this technique to analyze privacy and accuracy properties of arbitrary algorithms. We plan to build on Hay *et al.*'s DPComp [25] and DPBench [24] frameworks for principled evaluation of the privacy vs. utility trade-offs of different algorithms over given data sets. The evaluation principles they propose include evaluating under different ϵ values, and evaluating algorithms for expected error and bias in the result. We plan to incorporate these principles in developing a framework that can analyze arbitrary algorithms for privacy/accuracy trade-offs. To do this, we will also experiment with incorporating sampling [8, 44] into our dynamic analysis. We may need to restrict the functionality available to arbitrary implementations to make the analysis possible.

5 Task: A Decidable Type System for Probabilistic Relational Reasoning

In the previous task, we developed a programming model for JOSTLE that exposes algorithmic choice and dynamically explores privacy/accuracy trade-offs. The fully dynamic approach is computationally expensive: it is expensive to run programs to determine their level of privacy, to explore all possible algorithms at runtime, and to perform dynamic sampling to determine accuracy. For these reasons, we want to *statically* and *automatically* (1) reason about the privacy and accuracy of computations and (2) make algorithmic choices based on privacy and accuracy constraints.

The main challenge in developing such a reasoning framework is the automation. Existing approaches for static verification of differential privacy [5, 21, 37] require significant annotation on the part of the programmer and thus are not suitable to support policy-agnostic programming. Our experience developing LIFTY shows us that decidable type inference for refinement types makes it possible to propagate top-level types to and from subterms for verification and also for allowing the program to provide each specification

once. We cannot use the existing solution directly, as reasoning about privacy requires *probabilistic* reasoning about random functions. Differential privacy is also a *relational* property between pairs of executions.

As the next step, then, we plan to develop a *decidable probabilistic relational type system* for automated reasoning about privacy and accuracy. Recall our function from before:

```
budget E
def getPatientsOverForty (d)@accuracy(a):
  size similar{noisy_size} (filter over_40 d)
```

We want the type checker to be able to (1) analyze the privacy and accuracy of `size (filter over_40 d)` and `noisy_size (filter over_40 d)`, (2) propagate privacy information through the program, for instance to functions that call `getPatientsOverForty`, and (3) propagate accuracy requirements throughout the program, combining accuracies appropriately. Probabilistic relational reasoning will allow the type checker to reason about the privacy loss of `size` and `noisy_size`. Type-checking will occur with respect to the budget `E`. Note that deciding between calling `size` and `noisy_size` is part of the next task, and not this one.

5.1 Background: Dependent Types for Verification

Towards developing usable verification tools, several authors have proposed the use of *dependent types* [31], types that depend on program terms, for enhancing the expressiveness of type systems. The type `int` is an integer type that does not depend on program terms, while the type $\mathsf{i} :: \{\nu : \mathsf{int} \mid 0 < \nu\}$ conveys that integer `i` is a member of the set of positive integers. Because these types depend on terms, we can use *type reconstruction*, the process of inferring intermediate types from top-level type declarations, to verify program properties. *Refinement types* [7, 18, 20, 52] are a form of dependent type with particular practical promise because they allow the encoding of invariants in a combination of types and predicates from a restricted SMT-decidable logic. This restriction makes it safe to support arbitrary recursion while allowing expressive and useful type-based specifications.

Dependent types have proven useful for verifying security property of programs. Work on Fine [12, 46] and F* [47] demonstrate how to use dependent types for verifying stateful authorization policies. Work rF* [6], which extends F* with probabilistic relational reasoning, demonstrates how to use probabilistic relational dependent types to verify cryptographic properties. Work on Fuzz [37] and DFuzz [21] use *linear types*, dependent types that track resource usage, combined with a probability monad, to verify ϵ-differential privacy. None of the reasoning involved with this work is decidable.

5.2 Decidable Relational Verification

Towards developing a decidable relational type system, we plan to build on *liquid types* [38, 50] to develop a relational refinement type system, taking inspiration from the type system of rF*. Liquid types are a form of refinement types that are sufficiently restricted to be able to perform decidable type-checking and type inference, but expressive enough to prove a variety of safety properties. Liquid type checking and inference are decidable for three reasons: (1) the type system has a conservative but decidable notion of subtyping, (2) an expression has a liquid type derivation only if it has a valid type derivation in the host language ML, and (3) the space of possible types is bounded, in particular for expressions such as λ-abstractions and recursive functions.

Our main task will be to develop analogous restrictions in the relational setting. Relational refinements [6] are refinements that talk about properties over *pairs of executions*. A relational refinement has *left* and *right* values for every program value in scope. The type $\mathsf{x} : \mathsf{int}\{|\mathsf{L}\ x = \mathsf{R}\ x|\}$ means that for any pair of executions over the same program, the value `x` produces the same value. These relations allow us to prove

partial equivalences for properties such as information flow and, when combined with probabilistic reasoning, differential privacy properties. The main challenges in extending liquid types for relational reasoning involves finding restrictions that yield decidable reasoning and proving the soundness of the type system. Since we intend for static reasoning to complement a dynamic solution, we do not need our type system to be able to support the full expressiveness of our language.

5.3 Decidable Probabilistic Relational Verification

In order to verify statistical privacy properties, we need to add a decidable probabilistic reasoning layer to our system. This will be more straightforward, as we can always propagate probabilities through type-checking and inference at a coarse granularity. Our goal, however, is to be able to reason about the accuracy and privacy of a set of standard differential privacy benchmarks, inspired by the example programs for the PINQ language [32]. Reasoning about the benchmarks will drive the design of the probabilistic component of our type system. Note that the problem of determining accuracy, both dynamically and statically, is less well-understood than the problem of determining privacy. It is an open question how much we can use static analysis for accuracy.

6 Task: A Compilation Framework for Policy-Agnostic Privacy

At this point, we will have a programming model for trading off accuracy and privacy, and a decidable probabilistic relational type system for reasoning about these problems. We can now combine these results to develop a compiler for JOSTLE that makes algorithmic choices based on privacy and accuracy requirements. In this task, we develop the algorithm for deciding which computations to replace with ones marked as **similar**, and which of those computations to select. We plan to integrate this with our existing dynamic analysis framework and perform tests on case study examples to determine when static analysis, versus dynamic analysis, is preferable.

6.1 Making Algorithmic Choices Based on Static Analysis

To determine which computations to replace with similar ones based on the budget value and accuracy requirements, we plan to take an approach inspired by our prior work on LIFTY [35], in which we used the results of failed verification attempts of information flow security to determine how to repair the program. LIFTY relied on *precise error localization* in the type-checking algorithm, finding the smallest subterm around which to insert a check. To adapt this solution to JOSTLE programs, we plan to extend our probabilistic type-checking with a notion of precise error localization. When the type-checker determines that a program does not meet the privacy budget, it looks for the smallest subterm to change, and tries to replace computations within that term. The type-checker can do something analogous for accuracy violations. Because of how we have constructed our framework, we will be searching finite spaces using decidable algorithms, so our compiler will be fully automated and will not need interactivity from the programmer.

6.2 Integration with Dynamic Analysis

Decidability is a strong restriction, and so we anticipate that our static type-based analysis will not be able to handle the full expressiveness of our language. Thus, we plan for JOSTLE to combine static and dynamic techniques towards supporting policy-agnostic privacy. An important part of this will involve formally specifying the interface between static and dynamic analysis, and proving the soundness of this interaction. When we want to use static versus dynamic analysis will also depend on the code we want to write. Towards addressing practical considerations, we plan to empirically evaluate, based on benchmarks involving

representative use cases, the static and dynamic solutions. We will use these results during compilation to decide, at a fine granularity, whether to make decisions statically or use dynamic execution.

7 Education and Curriculum Development

Our proposed framework addresses the problem of making statistical privacy accessible by making it possible for non-experts to develop intuitions about differential privacy.

7.1 An Online Platform for Experimenting with Differential Privacy

The definition of differential privacy is highly technical and involves an understanding of probability theory. Understanding differential privacy with respect to programs is even more complex, as it involves reasoning about how the composition of different computations depletes a privacy budget. Because all existing automated tools for differential privacy tell the user only whether programs are differentially private, but not why, previously programmers needed background in reasoning about both probabilities and programs to understand the privacy of their programs.

An alternate approach to teaching the mathematical foundations of differential privacy is to convey *intuition* about what makes some algorithms more private than others, and how privacy budgets work. An important ingredient in building intuition about policies involves allowing the user to modify policies and the code to see how this affects both privacy and choice of algorithms. With prior solutions for differential privacy, this kind of experimentation required domain knowledge, and thus was not accessible to non-experts. By automating privacy analysis for arbitrary programs, JOSTLE makes it possible for non-experts to build up intuitions about differential privacy.

Towards democratizing intuition about statistical privacy, we will develop an online platform based on JOSTLE, along with a set of examples, that allow people to experiment with different algorithms, privacy budgets, and noise functions. To do this, we plan to address both scaling and usability issues in JOSTLE, and potentially carve out a subset of JOSTLE that we optimize for educational purposes. Such a system will increase the accessibility of statistical privacy, both in the classroom and for practitioners.

7.2 Curriculum Development

To learn about how to teach with JOSTLE, we will experiment with using it in the PI's undergraduate formal security course. Spring 2017, the PI co-designed and co-taught 15-316, a junior-level formal security course, with Matthew Fredrikson, another Assistant Professor at Carnegie Mellon University. Topics we covered include reference monitor automata, authentication logics, information flow, language-based techniques for information flow, differential privacy, and language-based techniques for differential privacy. While teaching the course, we observed the effectiveness of constructing examples, working with the students to modify the program and policies by hand, and working through the changes in output. Thus, we anticipate an automated tool will be helpful for students. In addition to the online framework, we plan to develop (1) a set of pedagogical examples and (2) a set of proposed changes to the programs and policies designed to illustrate security concepts. We will involve the students to provide feedback.

Since our tools are automated, we can easily scale them for online education. We are plan to eventually develop a massive open online course (MOOC) on privacy, aimed at working programmers and people who work with sensitive data and have a basic background in programming. Such a course is increasingly relevant as more sensitive data becomes digital.

8 Related Work

The proposed work builds on prior work on statistical privacy, verification, and languages that expose choice to the language runtime and compiler.

8.0.1 Statistical Privacy and Relaxed Information Flow

Information flow security is often too restrictive for data analytics. There is a relevant body of work on *declassification* [42], about ways of downgrading classified information, but these techniques still involve substantial programmer burden for deciding when downgrading should occur. Work on *quantitative information flow* [45] characterizes not just whether information is leaked, but some notion of how much information is leaked. With quantitative information, however, there remain many questions about how it should be used.

We chose to focus on *differential privacy* [15, 16, 17] because statistical privacy provides a good framework for reasoning about data release in the analytics computations we are interested in. Differential privacy, in particular, has a maturing body of algorithmic work and language-based around it. Our work builds on existing work on accuracy vs. privacy trade-offs [22, 25, 28, 33], dynamic programming solutions for ensuring the differential privacy of programs [32, 40], and systems for mechanically verifying the differential privacy of computations [5, 6, 21, 37]. Our approach differs in that it is the first we know of that addresses the issue of programmer burden.

8.0.2 Verification Approaches

Our approach builds on a tradition of verification using dependent types [31], types that depend on program terms. Specifically, we work with refinement types [7, 12, 18, 20, 46, 47, 52], types that are refined by logical predicates. Our proposed approach combines advances in liquid types [38, 50, 51], decidable refinement types, with work on relational verification [11, 23] and probabilistic relational verification [5, 6]. Our approach relies very much on our prior result [35] that decidable type reconstruction can yield a sound program repair algorithm. At the foundation of this work is sound program synthesis using liquid types [34]. Our solution uses probabilistic relational verification, as opposed to linear logic for representing resources, as Fuzz [37] and DFuzz [21] do.

8.0.3 Language Design

We take as a starting point work our prior work on policy-agnostic information flow, both the dynamic faceted execution semantics [4, 54, 55] and the static type-driven repair framework [35]. Rather than allowing the runtime and compiler to choose which value to use, however, JOSTLE allows the runtime and compiler to choose which *algorithms* to use. This make JOSTLE more similar to languages that expose algorithmic choice: PetaBricks [2], a language and compiler for algorithmic choice based that can be autotuned for different architectures, EnerJ [43], a language with approximate types for safe low-power computation, and ENT [10], a language that supports energy-adaptive programming with mixed static and dynamic checking. We also take inspiration from the Uncertain$\langle$T$\rangle$ language [8] and the MAYHAP tool [44] in specifying and propagating accuracies and in the dynamic sampling approaches we plan to experiment with. JOSTLE's goal of characterizing and exploring the trade-off between accuracy and privacy makes it so that we have different goals than these other tools.

9 Research and Evaluation Plan

The research described will occur concurrently with the following projects: (1) an information flow web browser, in collaboration with Limin Jia, Lujo Bauer, and Matt Fredrikson, an NSF Medium grant funded by the SaTC program, (2) automatic inference of policies from checks written in the code, and NSF CRII grant funded by the SaTC program, and (3) an effort to apply the faceted policy-agnostic programming semantics, funded by Carnegie Mellon University's Center for Machine Learning and Health, in collaboration with the University of Pittsburgh Medical Center. These three projects all explore different applications of policy-agnostic programming and will lead to use cases in which statistical privacy is useful. We will be able to build on our experience for case studies and the systems we develop for infrastructure for this project.

The budget for the current proposal covers the equivalent of one part-time academic-year graduate student, though the work may be split among multiple students. We have not included funding for undergraduate research assistants, but we plan to engage undergraduates in for-credit research on related projects. Towards developing our JOSTLE framework for policy-agnostic privacy, we plan first to develop a programming model for privacy/accuracy trade-offs that is dynamically enforced, then to develop a decidable probabilistic relational type system, and then to augment our dynamic enforcement with a compilation strategy capable of performing static analysis of privacy and accuracy.

We present the following work plan on the granularity of academic years.

Year 1. We plan to develop dynamic enforcement strategy for JOSTLE. As a starting point, we will take the example from Ligett *et al.* [28], and then we will investigate generalizations based on the work of Hay *et al.* [24, 25]. We will seek feedback from our collaborators at UPMC throughout this process.

Year 2. We will develop a system for probabilistic relational reasoning, first by extending liquid types for relational reasoning, and then by adding a probabilistic component. We will also iterate over the design of JOSTLE on an expanded set of benchmark examples, and experiment with dynamic sampling techniques for analyzing algorithms.

Year 3. We will begin developing a compilation strategy that makes fine-grained algorithmic choices based on static analysis using our decidable probabilistic relational type-checking. Using our type system to do this may involve modifying the type-checking algorithm. We will formally describe our system and prove mathematical guarantees about results.

Year 4. We will formalize the interaction between our dynamic and static solutions and prove that they interact soundly. We will also perform an empirical investigation of when it is more practically advantageous to use the static solution, versus waiting to make decisions at runtime.

Year 5. We will address scaling and usability issues, and work on developing educational modules based on our system. We will also continue to extend the expressiveness of our static analysis.

10 Broader Impacts

The proposed research direction makes it possible for people with a diversity of expertise to engage with sensitive data sets. In particular, our focus on data analysis and machine learning programs for health data will allow researchers to extract and share more insights without worrying about violating patient privacy.

Impact on society. The success of this research will make it easier to enforce privacy policies in programs that use sensitive data. The solution will not only make it possible to extract and share more insights without worrying about violating patient privacy, but it also has the potential to change the way regulation handles privacy. Currently, a significant obstacle towards protecting patient privacy in software is the difficulty of automating the deep code analysis that is required. This analysis is particularly challenging for statistical privacy properties, as it involves an understanding of statistics as well as an understanding of the program.

Factoring out statistical privacy policies from the program will not only make it easier for programmers to understand the privacy implications of the code but also for auditors and regulators.

The PI is also involved in promoting diversity in STEM, in improving science communication, and bridging gaps between academia and industry. Towards promoting diversity in STEM, the PI started Graduate Women at MIT, mentors students interested in starting student groups and efforts around diversity, and writes and speaks about gender and other issues in Computer Science. Towards improving science communication, the PI co-directed NeuWrite Boston, a working group of scientists and science writers, for two years, and has written general-audience pieces for the *ACM SIGCAS Newsletter*, *ACM Queue*, and the *MIT Technology Review* about technology and related issues. Towards bridging gaps between academia and industry, the PI co-founded Cybersecurity Factory, an accelerator for turning research security ideas into companies, in 2015, and regularly attends and speaks at practitioner conferences, including a keynote at PrivacySecurityRisk and invited talks at PhillyETE and Curry On.

Impact on education. By making it easier to write programs that preserve statistical privacy guarantees, we hope to encourage more people to write analyses of sensitive data that could provide useful, life-saving insights in health and medicine. We also hope that making it easier to program with statistical privacy policies will improve general understanding of statistical privacy.

Towards research mentoring, the proposed research program will support two graduate students over five years. We also plan to involve undergraduate students in this program. During the 2016-2017 school year, three undergraduate researchers in our group are doing senior theses on related topics.

Towards increasing understanding about privacy, we will make the resulting tools freely available online and incorporate results into courses taught by the PI. The PI will also incorporate the results into public lectures about software, security, and privacy. The PI has already been doing this by speaking in forums such as Papers We Love. Most recently, the PI developed an ACM Learning Webinar about policy-agnostic programming that also provided an overview of state-of-the-art information flow control techniques.

Impact on research. The success of this research will contribute (1) an understanding of the accuracy vs. privacy trade-offs in a set of widely used algorithms, (2) a programming model for policy-agnostic differential privacy, (3) a dynamic semantics for this programming model, (4) a decidable probabilistic relational type system, and (5) a compilation framework for policy-agnostic privacy based on algorithmic choice. By developing an accessible framework for reasoning about statistical privacy, this work has the potential to encourage more research on making it easier to program with statistical privacy. Encouraging human-computer interaction research on this topic will be particularly useful for satisfying regulatory concerns for encouraging more consumer-facing application developers to preserve privacy when releasing aggregate information. Finally, the decidable framework for probabilistic relational reasoning will provide a platform for experimenting with programming tools for other formulations of statistical privacy.

11 Prior Support

The most relevant grant is CNS-1657530, "CRII: SaTC: Repairing Code from Inferred Specifications of Information Flow Security," 2017-2019, $174,794, PI. **Intellectual Merit:** We are developing a framework for inferring information flow policies from access checks in the code to repair other code. This proposal has no overlap with this ongoing project, as the project focuses on inferring information flow policies. **Broader Impacts:** Success of this project can change the way we secure legacy code.

References

[1] Shreya Agrawal and Borzoo Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *CSF*, 2016.

[2] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542481. URL http://doi.acm.org/10.1145/1542476.1542481.

[3] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Symposium on Security and Privacy*, SP, 2012.

[4] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. *PLAS 2013*.

[5] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 97–110, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103670. URL http://doi.acm.org/10.1145/2103656.2103670.

[6] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. Probabilistic relational verification for cryptographic implementations. In *POPL*, 2014.

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, February 2011. ISSN 0164-0925. doi: 10.1145/1890028.1890031. URL http://doi.acm.org/10.1145/1890028.1890031.

[8] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain: A first-order type for uncertain data. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 51–66, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541958. URL http://doi.acm.org/10.1145/2541940.2541958.

[9] N. Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *European Symposium on Programming, ESOP*, volume 3924 of *LNCS*. Springer Verlag, 2006.

[10] Anthony Canino and Yu David Liu. Proactive and adaptive energy-aware programming with mixed typechecking. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 217–232, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062356. URL http://doi.acm.org/10.1145/3062341.3062356.

[11] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 316–329, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009858. URL http://doi.acm.org/10.1145/3009837.3009858.

[12] Juan Chen, Ravi Chugh, and Nikhil Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 412–423, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806643. URL http://doi.acm.org/10.1145/1806596.1806643.

[13] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Conference on Programming Language Design and Implementation*, PLDI, 2009.

[14] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.

[15] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35907-9, 978-3-540-35907-4. doi: 10.1007/11787006_1. URL http://dx.doi.org/10.1007/11787006_1.

[16] Cynthia Dwork. Differential privacy: A survey of results. In *Proceedings of the 5th International Conference on Theory and Applications of Models of Computation*, TAMC'08, pages 1–19, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-79227-9, 978-3-540-79227-7. URL http://dl.acm.org/citation.cfm?id=1791834.1791836.

[17] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, pages 265–284, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-32731-2, 978-3-540-32731-8. doi: 10.1007/11681878_14. URL http://dx.doi.org/10.1007/11681878_14.

[18] Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111059. URL http://doi.acm.org/10.1145/1111037.1111059.

[19] Matthew Fredrikson and Jean Yang. 15-316: Software foundations of security and privacy, 2017. URL https://cmu-15-316.github.io/.

[20] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113468. URL http://doi.acm.org/10.1145/113445.113468.

[21] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*, January 2013.

[22] Marco Gaboardi, James Honaker, Gary King, Kobbi Nissim, Jonathan Ullman, and Salil P. Vadhan. PSI (Ψ): a private data sharing interface. *CoRR*, abs/1609.04340, 2016. URL http://arxiv.org/abs/1609.04340.

[23] Niklas Grimm, Kenji Maillard, Cédric Fournet, Catalin Hritcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. arXiv:1703.00055, July 2017. URL https://arxiv.org/abs/1703.00055.

[24] Michael Hay, Ashwin Machanavajjhala, Gerome Miklau, Yan Chen, and Dan Zhang. Principled evaluation of differentially private algorithms using dpbench. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 139–154, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2882931. URL http://doi.acm.org/10.1145/2882903.2882931.

[25] Michael Hay, Ashwin Machanavajjhala, Gerome Miklau, Yan Chen, Dan Zhang, and George Bissias. Exploring privacy-accuracy tradeoffs using dpcomp. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2101–2104, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2899387. URL http://doi.acm.org/10.1145/2882903.2899387.

[26] S.C. Heise, J. Yang, D. Reeves, and Y. Jia. Privacy verification tool, September 18 2014. URL http://www.google.com/patents/US20140282837. US Patent App. 13/842,185.

[27] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[28] Katrina Ligett, Seth Neel, Aaron Roth, Bo Waggoner, and Z. Steven Wu. Accuracy first: Selecting a differential privacy level for accuracy-constrained ERM. *CoRR*, abs/1705.10829, 2017. URL http://arxiv.org/abs/1705.10829.

[29] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. SOSP. ACM, 2009.

[30] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007. ISSN 1556-4681. doi: 10.1145/1217299.1217302. URL http://doi.acm.org/10.1145/1217299.1217302.

[31] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc. ISBN 0-13-561465-1. URL http://dl.acm.org/citation.cfm?id=3721.3731.

[32] Frank D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 19–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559850. URL http://doi.acm.org/10.1145/1559845.1559850.

[33] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. Gupt: Privacy preserving data analysis made easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 349–360, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213876. URL http://doi.acm.org/10.1145/2213836.2213876.

[34] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.

[35] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. Type-driven repair for information flow security. *CoRR*, abs/1607.03445, 2016. URL http://arxiv.org/abs/1607.03445.

[36] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1), January 2003.

[37] Jason Reed and Benjamin C. Pierce. 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, pages 157–168, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863568. URL http://doi.acm.org/10.1145/1863543.1863568.

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

[39] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. PLDI 2009.

[40] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855711.1855731.

[41] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[42] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, October 2009. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=1662658.1662659.

[43] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993518. URL http://doi.acm.org/10.1145/1993498.1993518.

[44] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 112–122, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594294. URL http://doi.acm.org/10.1145/2594291.2594294.

[45] Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, pages 288–302, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00595-4. doi: 10.1007/978-3-642-00596-1_21. URL http://dx.doi.org/10.1007/978-3-642-00596-1_21.

[46] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in Fine. In *ETAPS*, 2010.

[47] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. ICFP, 2011.

[48] Latanya Sweeney. K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, October 2002. ISSN 0218-4885. doi: 10.1142/S0218488502001648. URL http://dx.doi.org/10.1142/S0218488502001648.

[49] Latanya Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):571–588, October 2002. ISSN 0218-4885. doi: 10.1142/S021848850200165X. URL http://dx.doi.org/10.1142/S021848850200165X.

[50] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 209–228, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37035-9. doi: 10.1007/978-3-642-37036-6_13. URL http://dx.doi.org/10.1007/978-3-642-37036-6_13.

[51] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: experience with refinement types in the real world. In *Haskell*, 2014.

[52] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277732. URL http://doi.acm.org/10.1145/277650.277732.

[53] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 99–110, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806610. URL http://doi.acm.org/10.1145/1806596.1806610.

[54] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. PLDI '16.

[55] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. POPL '12, 2012.

[56] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. *ACM Symposium on Operating Systems Principles*, 2009.