Journal of Privacy and Confidentiality (2016) **7**, Number 2, 159–184

Featherweight PINQ

Hamid Ebadi* and David Sands†

Abstract.

Differentially private mechanisms enjoy a variety of composition properties. Leveraging these, McSherry introduced PINQ (SIGMOD 2009), a system empowering non-experts to construct new differentially private analyses. PINQ is an LINQ-like API which provides automatic privacy guarantees for all programs which use it to mediate sensitive data manipulation. In this work we introduce *featherweight PINQ*, a formal model capturing the essence of PINQ. We prove that any program interacting with featherweight PINQ's API is differentially private.

Keywords: Differential privacy, dynamic database, PINQ, Formalization

1 Introduction

Differential privacy (Dwork, 2006; 2008; 2011) shows that by adding the right amount of noise to statistical queries, one can get useful results, and at the same time provide a quantifiable notion of privacy. The definition of differential privacy for a query mechanism (a randomized algorithm) is made by comparing the results of a query on any database with or without any one individual: a query Q is ε-differentially private if the difference in probability of any query outcome on a data-set only changes by a factor of e^{ε} (approximately $1+\varepsilon$ for small ε) whenever an individual is added or removed.

Of the many papers on differential privacy, a mere handful (at the time of writing) describe implemented systems which provide more than just a static collection of differentially private operations. The first such system is the PINQ system of McSherry (McSherry, 2009). PINQ is designed to allow non-experts in differential privacy to build privacy-preserving data analyses. The system works by leveraging a fixed collection of differentially private data aggregation functions (counts, averages, etc.), and a collection of data manipulation operations, all embedded with a LINQ-like (Don Box, February 2007 (accessed November 18, 2015) interface from otherwise arbitrary C# code. PINQ mediates all accesses to sensitive data in order to keep track of the sensitivity of various computed objects, and to ensure that the intended privacy budget ε is not exceeded; a budget could be exceeded by answering too many queries with too high accuracy. In this way PINQ is intended to make sure that the analyst (programmer) does not inadvertently break differential privacy.

*Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden. mailto:hamide@chalmers.se

†Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden. mailto:dave@chalmers.se

Foundations of PINQ McSherry argues the correctness of PINQ by pointing out the foundations upon which PINQ rests. In essence these are:

- A predefined collection of aggregation operations (queries) on tables, each with a parameter specifying the required degree of differential privacy. Standard aggregation operations such as (noisy) count and average are implemented. The core assumption is that each aggregation operation Q with noise parameter ε, written here as Q_ε, is an ε-differentially private randomised function.
- Sequential composition principle: if two queries performed in sequence (e.g. with differential privacy ε_1 and ε_2 respectively) then the overall level of differential privacy is safely estimated by summing the privacy costs of the individual queries $(\varepsilon_1 + \varepsilon_2)$.[1]
- Parallel composition principle: if the data is partitioned into disjoint parts, and a different query is applied to each partition, then the overall level of differential privacy is safely estimated by taking the maximum of the costs of the individual queries.
- Stability composition: the stability of a database transformation T is defined to be c if whenever you add n extra elements to the argument of T, the result of T changes by no more than $n \times c$ elements. If you first transform a database by T, then query the result with an ε-private query, the privacy afforded by the composition of the two operations is safely approximated by $c \times \varepsilon$.

These foundations of PINQ provide an intuition about how and why PINQ works, but although a novel aim of PINQ was "providing formal end-to-end differential privacy guarantees under arbitrary use", the foundations are not strictly sufficient to build an end-to-end correctness argument since they fall short of describing a number of PINQ features of potential relevance to the question of its differential privacy:

1. Parallel queries partition data, but the data which is partitioned might not be the original input, but some intermediate table. The informal argument for taking the maximum of the privacy costs of the query on each partition relies on the respective queries applying to disjoint data points. But the data might not be disjoint when seen from the perspective of the original data set of individuals. Data derived from a participant might end up in more than one partition, so a correctness argument must model this possibility to show that it is safe.
2. Parallel queries are not parallel at all, but can be adaptive - the result of a query on one partition might depend on the result of a query on another. This means that the implementation is complicated by the bookkeeping necessary to track the "maximum" cost of the queries.

[1] McSherry informally justifies PINQs implementation of privacy bookkeeping by appealing to a *non-adaptive* composition theorem (McSherry, 2009)[Theorem 3] which assumes that the queries are chosen statically; fortunately for PINQ, and as is well-known, this result also holds for adaptive queries where the choice of second query might depend on the result of the first (Dwork et al., 2006; Roth, 2011).

3. The foundations suggest how to compute the privacy cost of composed operations from their privacy and stability properties. But in practice PINQ does not *measure* the amount of privacy lost by a PINQ program, it *enforces* a stated bound. Because of this, there are two kinds of results from a query: the normal noisy answer, or an exception. An exception is thrown if answering the query normally would break the global privacy budget. To prove differential privacy it is not enough that the query is differentially private in the normal case – it must also be shown to be private in the case when an exception is thrown, since this information is communicated to the program.

Although McSherry's original informal explanations of the correctness of PINQ fall short, we do not intend to imply that there is something fundamentally suspect about the *design* of PINQ. Instead we give a formal argument for the correctness of the core design of PINQ, through a judicious combination of *explicit modelling* for features such as (1) and (3) above, as well as *simplification* of features, such as parallel adaptive queries (2) which do not appear to be used in practice. Regarding the *implementation* of PINQ, work by Haeberlen et al. (2011) shows that there are clearly some flaws and other issues in the enforcement of certain abstractions, allowing direct access to the dataset, and leaking sensitive information by a variety of channels, including privacy budget exceptions.

Our aim in this paper is to clarify the design of PINQ by constructing a model of its core, simplified as much as possible to express the essence of the design, while abstracting away from detail of the *implementation* of the key abstractions, but at the same time providing enough detail to prove the correctness of the design. In this way we are able to make a clear separation between the implementation issues (which are attacked in Haeberlen et al. (2011)), and the core design which still includes details such as budget exceptions.

The approach of building a minimalist semantic model is common in the research area of programming languages, and our approach uses standard techniques from that area – in particular, *operational semantics* (see e.g. Winskel (1993)) expressed in the form of *(probabilistic) labelled transition systems*.

We call our model *Featherweight PINQ*, as a nod to a very successful minimal model for the Java type system, *Featherweight Java* (Igarashi et al., 2001). Unlike Featherweight Java, our approach does not introduce a minimal programming language containing PINQ-like primitives, but instead focuses on a minimal API that sits between the runtime system and the client program. Thus we model the client program completely abstractly as a deterministic labelled transition system which interacts with tables via the PINQ-like API but which is otherwise completely unconstrained. For this model we instantiate the definition of differential privacy, taking into account the interactive nature of the system, and prove that Featherweight PINQ provides differential privacy for any client program.

Our model makes some tradeoffs, mildly restricting the functionality of PINQ in exchange for a greatly simplified runtime system. To show that these restrictions do

not have a significant practical impact we investigate the available PINQ algorithms and show that they can all be rewritten to use our simplified API.

2 PINQ

In this section we provide a brief description of the PINQ system from the user perspective. PINQ is a .NET API which provides an interface similar to the Language Integrated Queries (LINQ) that is a language extension to .NET. Analyses that use PINQ are typically written in C#.

Listing 1.1 shows a code fragment for a sample analysis producing the average ages of adult males and adult females, respectively, and then separately computing the average of age for all individuals.

Listing 1.1: PINQ sample code

```
1  var agent = new PINQAgentBudget(budget);
2  var data = new PINQueryable<Recordstype>(rawdata.AsQueryable(), agent);
3  var adults = data.Where(x => x.age > 17);
4  var genders = new [] {0,1};
5  var parts = adults.Partition(genders, x=>x.gender);
6  foreach (var a in genders) {
7       result[a]= parts[a].NoisyAverage(budget/2, x=>x.age /100) ;
8  }
9  foreach (var a in genders) {
10    Console.WriteLine("Average age of {0} is {1}",
11       a==0 ? "Males " : "Females ",
12       result[a] * 100) ;
13    }
14 Console.WriteLine("Average age (all):"
15    + data.NoisyAverage(c, x=>x.age/100) * 100) ;
```

The first two lines of the program initialises a PINQueryable object with sample sensitive data (`rawdata`) structures and set the privacy limit (`budget`). A PINQueryable object is a wrapper to the database which enables PINQ to track the properties that are relevant for differential privacy. The supplied "agent" parameter expresses the amount of differential privacy that the system will enforce on this database.

The analysis starts by selecting (line 3) a subset of records of interest (those who are adults). Behind the scenes PINQ records the fact that the *stability* of `data` is unchanged: adding a single record to the `rawdata` does not change the size of the result of this transformation by more than a single record.

In line 5 a partitioning operation splits the data into two groups based on the gender field (0 for Male, 1 for Female). Partition is not a standard LINQ/SQL style operation, but is specific to PINQ. For each partition (i.e. for each gender), the code outputs a noisy average of the `age`. `NoisyAverage` is one of a collection of built-in differentially private primitive aggregation operations provided by PINQ. The amount of differential privacy for each query in the loop is `budget/2`. After executing the `foreach` loop there will be `budget/2` of the original budget remaining. The outcome of the last line depends

on the accuracy/privacy parameter `c`. If c is larger than `budget/2` the program will throw an exception (because answering the query with that degree of precision would break the budget).

3 Modelling Preliminaries

In this section we briefly introduce the modelling methods that we will use, which can be broadly described as *operational semantics* using *labelled transition systems*. We will introduce the basic terminology and mention how it will be used in this particular work. Readers familiar with these areas can safely skip the remainder of this section.

Operational semantics refers to the formal description of computation mechanisms (typically of programming languages). In the case of the present work we will be working with *small-step operational semantics* in which a computation is described by defining its constituent computation steps. Standard references include, for example, Winskel (1993).

Labelled Transition Systems The basic mathematical object used to describe the small-step operational semantics in this work is a *labelled transition system*.

A labelled transition system, formally, is a 4-tuple $\langle P, A, T, p_0 \rangle$, where

- S is the set of *system states* (i.e. the states of the system under consideration), and $s_0 \in S$ is a distinguished initial state, modelling the starting point for a computation.
- A is a set of *actions*, typically used to model the interactions between the system and its external environment – for example input and output events. The environment itself may or may not be modelled explicitly as a labelled transition system.
- T, the *transition relation*, is a subset of $S \times A \times S$. If $(s, a, s') \in T$ then this models the fact that if the system is in state s, if it interacts with the environment in the manner described by action a, then it will, after this interaction, be in state s'.

Common Notational Conventions It is typical to use some form of arrow-symbol to denote the transition relation T, for example $\rightarrow$. In this case we write $(s, a, s') \in \rightarrow$ in a more graphical syntax as $s \xrightarrow{a} s'$.

When defining a suitable set of actions to model interactions with an environment, a distinguished action τ is usually used to denote a *silent* or null action. A transition $s \xrightarrow{\tau} s'$ then models a system that evolves in one computation step from s to s' without any interaction with its environment.

Common Definition Conventions A specific labelled transition system can be defined in a number of ways. If the states are finite then the system can be written as a graph. In the case of systems which are not finite state (as the systems defined in this

article), a labelled transition system may be defined *hierarchically* in terms of other labelled transition systems, or *inductively* using recursive specifications. In both cases it is common for such definitions to be presented in the form of a collection of *deduction rules* of the form

$$\frac{\text{premise}_1 \cdots \text{premise}_n}{s \xrightarrow{a} s'}.$$

The premises are typically a list of zero or more predicates, possibly involving metavariables used in the description of s, a and s'. If the premises do not involve any transition relations they are also commonly written to the right of the rule literally as side conditions.

A rule of this form says "If the *premises* are all true, then $s \xrightarrow{a} s'$". In the case where the premises themselves make use of the transition relation itself then the definition is inductive.

Probabilistic Labelled Transition Systems Labelled transition systems can be extended to model discrete probabilistic systems by adding an additional component to the transition relation, representing the probability of a given transition. There are many variations on this idea depending on whether both probability and nondeterminism is modelled, and whether the probabilistic behaviour comes from the system being modelled, or from the environment. In the present paper we will use transitions of the form $s \xrightarrow{a}_p s'$ to denote the transition of a system from a state s by action a to a state s' where s' with probability p determined by the system.

Usage in this Paper In this paper we will be defining two kinds of labelled transition system. To model a *client program* we will use a labelled transition system where the states are the program states (e.g. the program code, the local variables etc.), and the actions model the PINQ API calls. However, we do not model any specific client program, but make our overall model parametric in the choice of client program (with some modest restrictions) as long as the labels correspond to our chosen model of PINQ API calls.

To model the complete system we define a probabilistic labelled transition system. In this system, the *states* are tuples consisting of the program state of the client program, together with an environment storing the privacy-sensitive data, and various privacy bookkeeping information. The *actions* of the complete system are the values returned by database queries as the computation proceeds. The transition relation is defined by deductive rules, with premises using the transition relation of the client program. The non zero/one probabilities arise solely from running one of a family of primitive probabilistic queries.

4 Idealised Program

In this section we describe the abstract model of the program and API to the PINQ operations. In the section thereafter we go on to model the PINQ internals, what we

call the *protected system*, before combining these components into a the overall model of Featherweight PINQ.

The first thing that we will abstract from is the host programming language. Here one could choose to model a simple programming language, but it is not necessary to be that concrete. Instead we model a program as an arbitrary deterministic system that maintains its own internal state, and issues commands to the PINQ internals. In this sense we idealise PINQ by assuming that the API cannot be bypassed. In fact the PINQ system does not successfully encapsulate all the protected parts of the system, and so some programs can violate differential privacy by bypassing the encapsulation (Haeberlen et al., 2011), or by using side effects in places where side-effects are not intended. By idealising the interface we make clear the intended implementation, but not the details of its realisation in any particular language. By treating programs abstractly we also simplify other features of PINQ including aspects of its architecture which promote certain forms of extensibility. Before describing the program model it is appropriate to say a few words about the *protected system* (described formally in the next section). The protected system contains all the datasets (tables) manipulated by the program. Since these are the privacy sensitive data, we only permit the program to access them via the API. The protected system tracks the stability of all the tables which it maintains, together with a global budget. Our program interacts with the protected system by the following operations:

Assignment Tables in the protected system are referred to via *table variables*. A program can issue an assignment command. The model allows the program to manipulate a table using transformation that assign a new value to table variables.

The general form of assignment is of the form $tv := F(tv_1, \ldots, tv_n)$, where F is taken from a set of *function identifiers* representing a family of transformations with bounded stability (i.e. for each argument position i there is a natural number c_i such that if the size of the ith argument changes by n elements, then the result will change by at most $c_i \cdot n$ elements). This stability requirement comes from PINQ and is discussed in more detail in the next section. Transformations include standard operations such as the `.Where(x => x.age > 17)` from the example in listing 1.1, and simple assignments $t_1 := t_2$ (taking F to be the identity function), as well as assignments of literal tables (the case when F has arity 0).

Query The only other operation of the PINQ API is the application of a primitive differentially private query. In the example above we saw a compound transformation and query operation `parts[a].NoisyAverage(budget/2, x=>x.age)`. It is sufficient to model just the query, since the transformation (`x=>x.age`) can be implemented via an intermediate assignment. Thus we assume a set of primitive queries **Query**, ranged over by Q, which take as argument a positive real (the ε parameter) and a table, and produce a discrete probability distribution over a domain of result values **Val**. We generalise the single query operation to a *parallel query*, with syntax $\mathsf{query}(tv, f, \vec{Q}, \varepsilon)$, where

1. tv is the table variable referring to the table that will be used for the analysis,

2. f is the partitioning function that maps each record to an index in $\mathsf{codomain}(f) = \{1, \ldots, k\}$ for some $k \in \mathbb{N}$,
3. $\vec{Q}$ is a vector of k queries from **Query**.

The execution of this operation (as described in the next section) involves computing the sequence of randomised values

$$Q_i(\varepsilon, \{r \in T \mid f(r) = i\}), i \in \mathsf{codomain}(f)$$

where T is the table bound to tv. This is the "parallel query" operation described informally in the description of PINQ (McSherry, 2009). We use a single ε for all queries because if we chose an ε_i for each query the privacy cost will be maximum of all the epsilons in any case, so we may as well enjoy the accuracy of the largest epsilon. However, we note that the implementation of PINQ is more general than this, since the queries on each partition may be performed in an adaptive way. Here we are making a trade-off in keeping our model simple at the expense of not proving differential privacy for quite as general a system.

Client Program Model The above abstraction of the PINQ API allows us to abstract away from all internal details of the programming language using the API. Following Ebadi et al. (2015), we model a program as an arbitrary labelled transition system with labels representing the API calls:

Definition 1 (**ProgAct** Labels)**.** *The set of program action labels* **ProgAct***, ranged over by a and b, are defined as the union of three syntactic forms:*

1. *the distinguished action τ, representing computational progress without interaction with the protected system,*
2. *$tvar := F(tv_1, \ldots, tv_n)$ where F is a function identifier, i.e. the formal name of a transformation operation of arity n,*
3. *$\mathsf{query}(tv, f, \vec{Q}, \varepsilon)?\vec{v}$, where f is a function from records to $\{1, \ldots, k\}$ for some $k > 0$, where $\vec{v}$ is a vector of values in $\mathbf{Val}^k$, and $\vec{Q}$ is a vector of k queries.*

Every label represents an interaction between a client program and the protected system. The labels represent the observable output of a system which are a sequence of those actions: internal (silent) steps (τ) modelling no interaction, and vectors of values $\vec{v}$ which are the results of some query being answered and returned to the program.

To define these transitions, we assume a client program modelled by a labelled transition system modelling the API to the protected system. For client programs, the label corresponding to a query call is of the form $\mathsf{query}(tv, f, \vec{Q}, \varepsilon)\ ?\ \vec{v}$, and models the pair of query and the returned result (as described before) as a single event. This allows us to model value passing with no need to introduce any specific syntax for programs. Note that the value returned by the query is known to the program, and the program can act on it accordingly. From the perspective of the program and the protected system together, this value will be considered an observable output of the whole system.

Definition 2 (Client Program)**.** *A* client program *is a labelled transition system* $\langle \mathbb{P}, \rightarrow, P_0 \rangle$*, with labels from* $ProgAct$*, where* $\mathbb{P}$ *is all possible program states,* P_0 *is the initial state of the program, and the transition relation* $\rightarrow \subseteq (\mathbb{P} \times \mathbf{ProgAct} \times \mathbb{P})$ *is deadlock-free, and satisfies the following determinacy property: for all states* P*, if* $P \xrightarrow{a} P'$ *and* $P \xrightarrow{b} P''$ *then*

1. *if* $a = b$ *then* $P' = P''$*,*
2. *if* a *is not a query then* $a = b$*,*
3. *if* $a = \mathsf{query}(tv, f, \vec{Q}, \varepsilon)\ ?\ \vec{v}$ *then* $b = \mathsf{query}(tv, f, \vec{Q}, \varepsilon)\ ?\ \vec{u}$ *for some* $\vec{u}$*, and for all actions* c *of the form* $\mathsf{query}(tv, f, \vec{Q}, \varepsilon)\ ?\ \vec{w}$ *there exists a state* P_c *such that* $P \xrightarrow{c} P_c$*.*

The conditions on client programs are mild. Deadlock (i.e. termination) freedom simplifies reasoning; a program that terminates in the conventional sense can be modelled by adding a transition $P \xrightarrow{\tau} P$ for all terminated states P. Query transitions model both the query sent and the result received. Since we are modelling message passing using just transition labels, the condition on queries states that the program must be able to accept any result from a given query. Modulo the results returned by a query, the conditions require the program to be deterministic. This is a technical simplification which (we believe) does not restrict the power of the attacker.

Remark: Implicit parameters We will prove that Featherweight PINQ provides differential privacy for any client program. To avoid excessive parametrisation of subsequent definitions, in what follows we will fix some arbitrary client program $\langle \mathbb{P}, \rightarrow, P_0 \rangle$ and some arbitrary initial budget ε and make definitions relative to these. As an example, the program provided in Listing 1.1 can be modelled as follows:

$$\begin{array}{l} adult = adultSelector(data) \\ [girls, boys] = \mathsf{query}(adult, genderPartitioner, [Average, Averag], \varepsilon). \end{array} \tag{1}$$

5 Featherweight PINQ

In this section we turn to the model of the internals of PINQ, and the overall semantics of the system. We begin by describing the components of the protected system, and then give the overall model of Featherweight PINQ by giving a probabilistic semantics (as a probabilistic labelled transition system) to the combination of a client program and a protected system.

5.1 The Protected System

Global Privacy Budget The first component of the protected system is the global privacy budget. This is a non-negative real number representing the remaining privacy budget. The idea is that if we begin with initial budget b then Featherweight PINQ will

enforce b-differential privacy. The global budget is decremented as queries are computed, and queries are denied if they would cause the budget to become negative. In PINQ the budget is associated with a given data source. In our model we assume that there is only one data source, and hence only one budget. Further, PINQ allows the budget to be divided up and passed down to subcomputations. This does not fundamentally change the expressiveness of PINQ since, as we show later, we are free to extend Featherweight PINQ with the ability to query the global budget directly. Thus any particular strategy for dividing the global budget between subcomputations can be easily programmed.

The Table Environment The other data component of the protected system is the table environment, which maps each table variable to the table it denotes, together with a record of the *scaling factor*, which is a measure of the *stability* of the table relative to the initial data set. We define this precisely below. Formally we define a table as power-set of records, $\mathcal{P}(\mathbf{Record})$, a protected table is a pair of a table with its scaling factor:

$$\mathbf{ProtectedTable} \stackrel{\text{def}}{=} \mathbf{Table} \times \mathbb{N}.$$

5.2 The Featherweight PINQ Transition System

Featherweight PINQ is defined by combining a client program with the protected system to form the states of a probabilistic transition system.

Definition 3 (Featherweight PINQ States)**.** *The* states *(otherwise known as* configurations*) of Featherweight PINQ, ranged over by* $\mathbb{C}$*,* $\mathbb{C}'$ *etc., are triples of the form* $\langle P, E, B\rangle$ *where* P *is a client program state,* $E \in \mathbf{TVar} \to \mathbf{ProtectedTable}$ *is the table environment, and* $B \in \mathbb{R}^+$ *is the global budget.*

There is a family of possible initial states, indexed by the distinguished input table, and the initial budget. We define these by assuming the existence of a distinguished table variable, *input*, which we initialise with the input table, while all other table variables are initialised with the empty table:

Definition 4 (Initial configuration)**.**

$$\mathbf{Init}(T, B) \stackrel{\text{def}}{=} \langle P_0, E_T, B\rangle \text{ where } E_T(tv) \triangleq \begin{cases} (T, 1) & \text{if } tv = input \\ (\{\}, 0) & \text{otherwise.} \end{cases}$$

The operational semantics of featherweight PINQ can now be given:

Definition 5 (Semantics)**.** *The operational semantics of configurations is given by a probabilistic labelled transition relation with transitions of the form* $\mathbb{C} \xrightarrow{a}_p \mathbb{C}'$ *where* $a \in \mathbf{Act} \stackrel{\text{def}}{=} \{\tau, \bot\} \cup \bigcup_{n\in\mathbb{N}} \mathbf{Val}^n$*, and (probability)* $p \in [0, 1]$*. The definition is given by cases in Figure 1.*

We note at this point that some of the primitives have not yet been defined (e.g. stability in the Assign rule), and that the rules of the system do not, *a priori*, define a probabilistic transition system. We will elaborate these points in what follows. We begin by explaining the rules in turn.

$$\text{Silent}\ \frac{P \xrightarrow{\tau} P'}{\langle P, E, B\rangle \xrightarrow{\tau}_1 \langle P', E, B\rangle}$$

$$\text{Assign}\ \frac{P \xrightarrow{tv:=F(tv_1,\ldots,tv_n)} P'}{\langle P, E, B\rangle \xrightarrow{\tau}_1 \langle P', E[tv \mapsto (T', s)], B\rangle}\ \text{where}\ \begin{cases} E(tv_i) = (T_i, s_i), i \in \{1,\ldots,n\} \\ \mathsf{stability}(F) = (c_1,\ldots,c_n) \\ s = \sum_{i=1}^{n} c_i \times s_i \\ T' = [\![F]\!](T_1,\ldots,T_n) \end{cases}$$

$$\text{Query}_\perp\ \frac{P \xrightarrow{\mathsf{query}(tv,f,\vec{Q},\varepsilon)\ ?\ \perp} P'}{\langle P, E, B\rangle \xrightarrow{\perp}_1 \langle P', E, B\rangle}\text{where}\ \begin{cases} E(tv) = (T, s) \\ \varepsilon \cdot s > B \end{cases}$$

$$\text{Query}\ \frac{P \xrightarrow{\mathsf{query}(tv,f,\vec{Q},\varepsilon)\ ?\ \vec{v}} P'}{\langle P, E, B\rangle \xrightarrow{\vec{v}}_p \langle P', E, B - t \cdot \varepsilon\rangle}\text{where}\ \begin{cases} E(tv) = (T, s), \quad \varepsilon \cdot s \leq B \\ \mathsf{codomain}(f) = \{1,\ldots,n\} \quad \vec{v} \in \mathbf{Val}^n \\ T_i = \{t \mid t \in T, f(t) = i\}, i \in \{1,\ldots,n\} \\ p = \prod_{i=1}^{n} \Pr[Q_i(\varepsilon, T_i) = v_i] \end{cases}$$

Figure 1: Operational semantics

Assign When a program issues an assignment command $tv := F(tv_1, \ldots, tv_n)$, the value of the stored table for tv is updated in the obvious way. We must also record the scaling factor of the table thus computed. The scaling factor is computed from the scaling factors of the tables for $tv_1, \ldots, tv_n$, and the *stability* of the transformation f. We assume a mapping $[\![\cdot]\!]$ from formal function identifiers F to the actual table transformation functions $[\![F]\!]$ of corresponding arity.

Definition 6. *A table transformation f of arity n has stability $(c_1, \ldots, c_n)$ if for all $i \in \{1, \ldots, n\}$, we have*

$$|f(T_1, \ldots, T_i, \ldots, T_n) \ominus f(T_1, \ldots, T_i', \ldots T_n)| \leq c_i \times |T_i \ominus T_i'|.$$

This is the n-ary generalisation of McSherry's definition (McSherry, 2009), and bounds the size change in a result in terms of the size change of its argument. This is made more explicit in the following:

Lemma 1. *If f has stability $(c_1, \ldots, c_n)$ then $|f(T_1, \ldots, T_n) \ominus f(T_1', \ldots, T_n')| \leq \sum_i^n (c_i \times |T_i \ominus T_i'|)$.*

Note that not all functions have a finite stability. An example of this is the database join operation (essentially the cartesian product); adding one new element to one argument will add k new elements to the result, where k is the size of the other argument. Thus there is no static bound on the number of elements that may be added. Thus PINQ (and hence Featherweight PINQ) supports only transformation operations which have a finite stability. The variant of the join operation, Join* deterministically produces bounded numbers of join elements. For the purpose of this paper we do not need to be specific about the transformations. We simply assume the existence of a function $\mathsf{stability}$ which soundly returns the stability of a function identifier, i.e., if $\mathsf{stability}(F) = (c_1, \ldots, c_n)$ then $[\![F]\!]$ has stability $(c_1, \ldots, c_n)$.

The transition rule for assignment in featherweight PINQ is thus

$$\frac{P \xrightarrow{tv:=F(tv_1,\ldots,tv_n)} P'}{\langle P, E, B\rangle \xrightarrow{\tau}_1 \langle P', E[tv \mapsto (T', s)], B\rangle} \text{ where } \begin{cases} E(tv_i) = (T_i, s_i), i \in \{1,\ldots,n\} \\ \mathsf{stability}(F) = (c_1,\ldots,c_n) \\ s = \sum_{i=1}^{n} c_i \times s_i \\ T' = [\![F]\!](T_1,\ldots,T_n) \end{cases}.$$

The label on the rule τ says that nothing (other than computational progress) is observable from the execution of this computation step. The subscript 1 is the probability with which this step occurs.

Transformation	Stability
Select(T, *maper*)	(1)
Where(T, *predicate*)	(1)
GroupBy(T_1, *keyselector*)	(2)
Join*(T_1,T_2, n, m, *keyselector*$_1$, *keyselector*$_2$)	(n,m)
Intersect(T_1,T_2)	(1,1)
Union(T_1,T_2)	(1,1)
Partition(T, *keyselector*, *keysList*)	(1)

Table 1: Transformation stability

Understanding the scaling factor Here we provide more intuition about the scaling factor calculations, and explain some differences between the PINQ implementation and the Featherweight PINQ model. As an example, suppose we have a computation of a series of tables A–G depicted in Figure. 2.

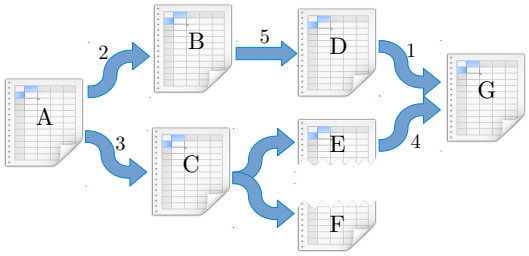

Figure 2: Transformations

	s	Calculation
A	1	Input table
B	2	$s(A) \times 2$
C	1	$s(A) \times 3$
D	10	$s(B) \times 5$
E	3	$s(C)$
F	3	$s(C)$
G	22	$s(D) \times 1 + s(E) \times 4$

Figure 3: Scaling factors (s)

The figure represents a PINQ computation involving three unary transformations (producing B, C, and D), one binary transformation producing G, and one partition operation (splitting C into E and F). We have labelled the transformation arcs with the stability constants of the respective transformations. What is the privacy cost of an ε differentially private query applied to, say, table D? Since D is the result of two transformations on the input data, the privacy cost is higher than just ε. The product of the sensitivities on the path from D backwards to the input A provide the *scaling factor* for ε. In this case the scaling factor for a query on D is 10. The remaining scaling factors are summarised in the table in Figure 3.

The scaling factor is the stability of that specific table; it bounds the maximum possible change of the table as a result of a change in the input dataset, assuming that it was produced using the same sequence of operations. The scaling factor is computed from the stabilities of transformations that produced it. The scaling factor for each protected table (except input table which has the scaling factor one) is computed compositionally using the scaling factors (s_i) of all the arguments and the sensitivities of corresponding transformation arguments (c_i) using the following formula: $\mathrm{s}_A = \sum_{i \in parent(A)} \mathrm{c}_i \times \mathrm{s}_i$. Figures 2 and 3 allow us to explain two key differences between PINQ and our model:

1. In PINQ, the tree structure depicted in the figure is represented explicitly, and scaling factors are calculated lazily: at the point where a query with accuracy ε is made on a table it is necessary to calculate its scaling factor s in order to determine the privacy cost $s.\varepsilon$. To do this the tree is traversed from the query at the leaf back to the root, calculating the scaling factor along the way. At the root the total privacy cost is then known and deducted from the budget (providing the budget is sufficient). In Featherweight PINQ the scaling factors of each table are computed eagerly, so the tree structure is not traversed.

2. In Featherweight PINQ we restrict the partition operation to the leaves of the tree, and combine it with the application of primitive queries to partitions.

The consequence of these two simplifications is that we do not need to represent the PINQ computation tree at all – all computations are made locally at the point at which a table is produced or queried.

Queries Parallel queries were described in detail in the previous section. When a program issues a query is it represented as a parallel query and a possible result – i.e. we model the query and the returned result as a single step. There are two cases to consider, according to whether the budget is sufficient or not. If the queried table T has scaling factor s then the cost of an ε query is $s \times \varepsilon$. If this is greater than the current global budget then the result is the exceptional value $\perp$. This value is the observable result of the query, and it occurs with probability 1. On the other hand, if the budget is sufficient, then the vector of query results $\vec{v}$ is returned with probability $p = \prod_i \Pr[Q_i(\varepsilon, T_i) = v_i]$ where T_i is the ith partition of T. Note that p is indeed a probability, since the component queries are independent.

6 Differential Privacy for Featherweight PINQ

In this section we prove that Featherweight PINQ is differentially private. We begin by recapping the goals of differential privacy, before showing how to specialise the definition to Featherweight PINQ. Doing this entails building a trace semantics for Featherweight PINQ. Differential privacy guarantees that a data query mechanism (abstractly, a randomized algorithm) behaves similarly on similar input databases. This "similarity" is a quantitative measure ε on the difference in the information obtained

from any data set with or without any individual. When this difference is small, the presence or absence of the individual in the data set is difficult to ascertain.

Definition 7. *Mechanism f provides ε-differential privacy if for any two datasets A and B that differ in one record ($\mid A \ominus B \mid = 1$), and for any two possible outcome $f(A)$ and $f(B)$, the following inequalities hold:* $\mathrm{e}^{-\varepsilon} \leq \frac{\Pr[(f(A) \in S)]}{\Pr[(f(B) \in S)]} \leq \mathrm{e}^{\varepsilon}$.

In this definition, S is subset of the range of outcomes for f ($S \subseteq \mathrm{Range}(f)$) and for similarity of outcomes we use the ratio between the probabilities of observing outcomes $\Pr[(f(\mathit{datasets}) \in S)]$ when the analyses are executed on any two similar datasets A and B. Finally for similarity of datasets hamming distance is used as a metric. In this work we assume that the primitive query mechanisms (and thus Featherweight PINQ) provide answers over a discrete probability distribution, so that it is sufficient to consider S to be a singleton set.

6.1 Trace semantics

The first step to instantiating the definition of differential privacy to Featherweight PINQ is to be able to view Featherweight PINQ as defining a probabilistic function. In fact each client program gives rise to a family of probabilistic functions, one for each length of computation that is observed. This is given by building a *trace semantics* on top of the transition system for Featherweight PINQ.

The semantics of Featherweight PINQ is a probabilistic labelled transition system of the simplest kind: for each configuration $\mathbb{C}$, the sum of all probabilities of all transitions of $\mathbb{C}$ is equal to 1. The system is also deterministic, in the sense that if $\mathbb{C} \xrightarrow{a}_{p_1} \mathbb{C}_1$ and $\mathbb{C} \xrightarrow{a}_{p_2} \mathbb{C}_2$ then $p_1 = p_2$ and $\mathbb{C}_1 = \mathbb{C}_2$. This makes it particularly easy to lift the probabilistic transition system from single actions to traces of actions:

Definition 8 (Trace semantics)**.** *Define the trace transitions $\Rightarrow \subseteq \mathbf{Config} \times \mathbf{Act}^* \times [0,1] \times \mathbf{Config}$ inductively as follows: (i) $\mathbb{C} \overset{[]}{\Rightarrow}_1 \mathbb{C}$ where $[] \in \mathbf{Act}^*$ is the empty trace, and (ii) if $\mathbb{C} \xrightarrow{a}_p \mathbb{C}'$ and $\mathbb{C}' \overset{t}{\Rightarrow}_q \mathbb{C}''$ then $\mathbb{C} \overset{at}{\Rightarrow}_{p.q} \mathbb{C}''$.*

Traces inherit determinacy from the single transitions:

Proposition 1 (Traces are Deterministic)**.** *If $\mathbb{C} \overset{t}{\Rightarrow}_{p_1} \mathbb{C}_1$ and $\mathbb{C} \overset{t}{\Rightarrow}_{p_2} \mathbb{C}_2$ then $p_1 = p_2$ and $\mathbb{C}_1 = \mathbb{C}_2$.*

This follows by an easy induction on the trace, using the fact that the single step transitions are similarly deterministic.

Lemma 2 (Traces are Probabilistic)**.** *Define*

$$\mu(\mathbb{C}, t) \overset{\text{def}}{=} \begin{cases} p & \text{if } \mathbb{C} \overset{t}{\Rightarrow}_p \mathbb{C}' \\ 0 & \text{otherwise.} \end{cases}$$

For all configurations $\mathbb{C}$*, and all* $n > 0$*,*

$$\sum_{t \in Act^n} \mu(\mathbb{C}, t) \ = 1$$

where Act^n *is the set of traces of length* n*.*

The proof is a simple induction on n, using the proposition above. The lemma says that whenever $\mathbb{C} \stackrel{t}{\Rightarrow}_p$, then p is the probability that you see trace t after having observed $\mathsf{size}(t)$ steps of the computation of $\mathbb{C}$. We will thus refer to the probability of a given trace to mean the probability of producing that trace from the given configuration among all traces of the same length. We denote this by writing $\Pr[\mathbb{C} \stackrel{t}{\Rightarrow}] = p$ when $\mathbb{C} \stackrel{t}{\Rightarrow}_p$.

Differential Privacy for Traces We are now in a position to specialise the definition of differential privacy for Featherweight PINQ. The probabilistic function is determined by the client program (which we have kept implicit but unconstrained), the initial budget ε, and the length of trace n that is observed for any combination. We define the function which maps a table T to trace t of length n with probability p precisely when $\Pr[\mathbf{Init}(T, \varepsilon) \stackrel{t}{\Rightarrow}] = p$.

The instantiation of the differential privacy condition to Featherweight PINQ is thus:

$$\forall t, T, T', \varepsilon. \text{ if } |T \ominus T'| = 1 \text{ then } \mathrm{e}^{-\varepsilon} \leq \frac{\Pr[\mathbf{Init}(T, \varepsilon) \stackrel{t}{\Rightarrow}]}{\Pr[\mathbf{Init}(T', \varepsilon) \stackrel{t}{\Rightarrow}_]} \leq \mathrm{e}^{\varepsilon}.$$

Towards a proof of this property we introduce some notation to reflect key invariants between the pairs of computations (for T and T' respectively).

Definition 9 (Similarity)**.** *We define similarity relations* $\sim$ *between tables, environments, and configurations as follows:*

- *For tables* T *and* T'*, and* $s \in \mathbb{N}$ *define* $T \sim_s T'$ *("*T *is* s*-similar to* T'*") if and only if* $|T \ominus T'| \leq s$*.*
- *For protected environments* E *and* E'*, define* $E \sim E'$ *if and only if for all* tv*, if* $E(tv) = (T, s)$ *and* $E'(tv) = (T', s')$ *then* $s = s'$ *and* $T \sim_s T'$*.*
- *For configurations, define* $\langle P, E, B\rangle \sim \langle P', E', B'\rangle$ *if and only if* $P = P'$*,* $E \sim E'$ *and* $B = B'$*.*

The configuration similarity relation captures the key invariant between the two computations in our proof of differential privacy. First we need to show that the invariant is established for the initial configurations:

Lemma 3. *If* $T \sim_1 T'$ *then* $\mathbf{Init}(T, B) \sim \mathbf{Init}(T', B)$*.*

This follows easily from the definition of the initial configuration. Now the main theorem shows that this is maintained throughout the computation:

Theorem 1. *If* $T \sim_1 T'$ *and* $\mathbf{Init}(T, B) \stackrel{t}{\Rightarrow}_p \mathbb{C} = \langle P, E, B - \varepsilon \rangle$, *then* $\mathbf{Init}(T', B) \stackrel{t}{\Rightarrow}_q \mathbb{C}' = \langle P, E, B - \varepsilon \rangle$ *where* $\mathbb{C} \sim \mathbb{C}'$ *and* $p \leq q . \exp(B - \varepsilon)$ *for some* $\varepsilon \leq B$.

Corollary 1 (B-differential privacy)**.** *If* $T \sim_1 T'$ *and* $\Pr[\mathbf{Init}(T, B) \stackrel{t}{\Rightarrow}] = p$ *then* $\Pr[\mathbf{Init}(T', B) \stackrel{t}{\Rightarrow}] = q$ *for some* q *such that* $p \leq q \cdot \exp(B)$.

Proof. Assume $\mathbf{Init}(T, B) \stackrel{t}{\Rightarrow}_p \mathbb{C}$. We proceed by induction on the length of the trace t, and by cases according to the last step of the trace.

Base case: $t = []$. In this case $p = q = 1$ and $\mathbb{C} = \mathbf{Init}(T, B)$ and $\mathbb{C}' = \mathbf{Init}(T', B)$. So $\varepsilon = \varepsilon' = 0$ and $\mathbb{C} \sim \mathbb{C}'$.
Inductive step: $t = t_1 a$. Suppose that $\mathbf{Init}(T, B) \stackrel{t_1}{\Rightarrow}_{p_1} \langle P_1, E_1, B_1 \rangle \stackrel{a}{\rightarrow}_{p_2} \langle P, E, B \rangle = \mathbb{C}$, and hence that $p = p_1 p_2$.

The induction hypothesis gives us q_1, P_1, E_1' and ε_1 such that

$$\mathbf{Init}(T', B) \stackrel{t_1}{\Rightarrow}_{q_1} \langle P_1, E_1', B_1 \rangle \tag{2}$$
$$E_1 \sim E_1' \tag{3}$$
$$p_1 \leq q_1 . \exp(B - \varepsilon_1) \tag{4}$$

by cases that is applied to the rule as the last transition ($\langle P_1, E_1, B_1 \rangle \stackrel{a}{\rightarrow}_{p_2} \langle P, E, B \rangle$) we have $p_2 = 1$ except for query execution and that $\langle P_1, E_1', B_1 \rangle \stackrel{a}{\rightarrow}_1 \mathbb{C}'$ for some $\mathbb{C}'$. In those cases it follows that $p \leq q \cdot \exp(B - \varepsilon)$ by taking $\varepsilon = \varepsilon_1$ and using (4).

Case 1: Silent. In this case $a = \tau$ and $P_1 \stackrel{\tau}{\rightarrow} P$.

$$\langle P_1, E_1, B_1 \rangle \stackrel{\tau}{\rightarrow}_1 \mathbb{C} = \langle P_1, E_1, B_1 \rangle$$
$$\langle P_1, E_1', B_1 \rangle \stackrel{\tau}{\rightarrow}_1 \mathbb{C}' = \langle P_1, E_1', B_1 \rangle.$$

It follows directly from (3) that $\mathbb{C} \sim \mathbb{C}'$.

Case 2: Assign. Here $P_1 \xrightarrow{t:=F(t_1,\ldots,t_n)} P$, and so we have

$$\mathbb{C} = \langle P, E_1[tv \mapsto (T, s)], B_1 \rangle$$
$$\mathbb{C}' = \langle P, E_1'[tv \mapsto (T', s)], B_1 \rangle$$

where for $i \in (1, \ldots, n)$

$$E_1(tv_i) = (T_i, s_i)$$
$$E_1'(tv_i) = (T_i', s_i')$$
$$\mathsf{stability}(F) = (c_1, \ldots, c_n)$$
$$s = \sum_{\sum_i^n} c_i \times s_i$$
$$T = [\![F]\!](T_1, \ldots, T_n)$$
$$T' = [\![F]\!](T_1', \ldots, T_n').$$

From (3) we have $E_1(tv_i) \sim E_1'(tv_i)$ which means $s_i = s_i'$ and $T_i \sim_{s_i} T_i'$. Using similarity definition and Lemma 1 we have $T \sim_s T'$ and hence we have $\mathbb{C} \sim \mathbb{C}'$.

Case 3: Query. The result of query execution depends on the remaining budget and the sensitivity of the table that the query is executed on. If privacy budget is insufficient an exception is thrown to inform the program about the shortage of budget, otherwise each query in the list of queries will be executed on its corresponding partition and the result of execution is returned as a list of values, $\vec{v}$.

Case 3.1: Query(run out of budget). Here we have a rule instance of the form:

$$\text{Query}_{\perp}\ \frac{P \xrightarrow{\mathsf{query}(tv,f,\vec{Q},\varepsilon)\ ?\ \perp} P'}{\langle P,E,B\rangle \xrightarrow{\perp}_1 \langle P',E,B\rangle}\text{where}\begin{cases}E(tv)=(T,s)\\ \varepsilon\cdot s > B\end{cases}.$$

In this case $\mathbb{C}\sim\mathbb{C}'$ and is similar to silent case.

Case 3.2: Query. Similarly we have a rule instance of the form:

$$\text{Query}\ \frac{P \xrightarrow{\mathsf{query}(tv,f,\vec{Q},\varepsilon)\ ?\ \vec{v}} P'}{\langle P,E,B\rangle \xrightarrow{\vec{v}}_p \langle P',E,B-t\cdot\varepsilon\rangle}\text{where}\begin{cases}E(tv)=(T,s), \quad \varepsilon\cdot s\le B\\ \mathsf{codomain}(f)=\{1,\ldots,n\} \quad \vec{v}\in\mathbf{Val}^n\\ T_i=\{t\mid t\in T, f(t)=i\}, i\in\{1,\ldots,n\}\\ p=\prod_{i=1}^{n}\Pr[Q_i(\varepsilon,T_i)=v_i]\end{cases}.$$

Hence we have a transition : $\langle P_1,E_1,B_1\rangle \xrightarrow{\vec{v}}_{p_2} \mathbb{C}=\langle P,E,B\rangle$ and the analogous transition : $\langle P_1,E_1',B_1\rangle \xrightarrow{\vec{v}}_{q_2} \mathbb{C}'=\langle P,E',B\rangle$. The needed value for theorem 1 is $\varepsilon=\varepsilon_1+(t\cdot\varepsilon_2)$.

For parallel queries on disjoint set we have the following equation:

$$\Pr[P_1 \xrightarrow{\mathsf{query}(tv,f,\vec{Q},\varepsilon)\ ?\ \vec{v}} P] = \prod_{i=1}^{n}\Pr[Q_i(s\cdot\varepsilon_2,T_i)=v_i].$$

Here we need to show that the following inequality is valid:

$$\prod_{i=1}^{n}\Pr[Q_i(s\cdot\varepsilon_2,T_i)=v_i] \le \prod_{i=1}^{n}\Pr[Q_i(s\cdot\varepsilon_2,T_i')=v_i'] \times \prod_{i=1}^{n}\exp(\varepsilon_2\times \mid T_i-T_i'\mid).$$

From $\sum_{i=1}^{n}(\mid T_i-T_i'\mid)=s$, we have $\prod_{i=1}^{n}\exp(\varepsilon_2\times\mid T_i-T_i'\mid)\le\exp(\varepsilon_2\times s)$ which we conclude:

$$\prod_{i=1}^{n}\Pr[Q_i(s\cdot\varepsilon_2,T_i)=v_i] \le \prod_{i=1}^{n}\Pr[Q_i(s\cdot\varepsilon_2,T_i')=v_i'] \times \exp(\varepsilon_2\cdot s).$$

These parallel queries provide $(s\cdot\varepsilon)$-differential privacy which means:

$$p_2 \le q_2\cdot\exp(\varepsilon_2\cdot s).$$

Multiplying two sides of the previous inequality with (4) we get:

$$p_1\cdot p_2 \le q_1\cdot q_2\cdot\exp(\varepsilon_1)\cdot\exp(\varepsilon_2\cdot s).$$

Knowing $B_1=B-\varepsilon_1$ result in choosing ε to be $\varepsilon=B-\varepsilon_1-(\varepsilon_2\cdot s)$. Finally it is easy to see $\mathbb{C}\sim\mathbb{C}'$ as the proper reduction in the global budget is the only change in the configuration. □

7 Practical Evaluation

We have developed a minimalistic model of PINQ and shown that the model is sufficiently precise to give a rigorous proof of differential privacy. A remaining concern, addressed in this section, is the extent to which the simplifications and tradeoffs made in the modelling of PINQ actually capture the true essence of PINQ. In particular, we simplified the concept of a parallel query to closely match the informal description of PINQ (McSherry, 2009), but not the actual implementation. The key difference between Featherweight PINQ and actual PINQ was described in Section 5 in connection with Figure 2, which depicts a partition operation which is not supported by Featherweight PINQ since it is not immediately followed by queries on the partitions. In fact the "parallel" queries in PINQ are not parallel at all, but are implemented by some sequential traversal of the partitions. Furthermore, the queries could, in principle, be adaptive (i.e., the result of a query on one partition can be used to influence the choice of query on other partitions). This feature is not easily supported by a small change to our model since it does not seem to be implementable using Featherweight PINQ's simple history-free use of explicit scaling factors.

In this section we report on the results of practical experimentation with our model, studying all the existing PINQ programs available in the distribution, plus further examples from McSherry and Mahajan (2010). Our approach was to implement the Featherweight PINQ API in PINQ, and see which PINQ examples can be reimplemented in a faithful way with this simpler API. Our observations based on this practical experiment can be summarised as follows:

- No existing PINQ programs take advantage of adaptiveness of parallel queries.
- All examples can be rewritten to use the simpler Featherweight PINQ API.

In the remainder of this section we describe the implemented Featherweight PINQ API, and summarise the difficulty of reimplementing the examples.

7.1 Implementing the Parallel Query Operator in PINQ

The essence of our simplified model is a parallel query operation. In our model the parallel query operation on a table requires (i) a mapping f from records to a set of indices $\{1, \ldots, n\}$, and (ii) a vector of n queries. The idea is that the ith query is applied to the table of all records r for which $f(r) = i$.

The signature we use in the C# implementation, function `Partition_Query`, is isomorphic to this, but more general for the sake of programming convenience, narrowing the gap to PINQ in a technically insignificant way. Instead of using a set of natural numbers $\{1, \ldots, n\}$, `Partition_Query` allows (as for the `Partition` operation of PINQ) an arbitrary set of keys K; instead of a vector of n queries, a dictionary mapping keys to queries is used. In concrete C# terms, queries are represented by a `QueryObj` class, `Partition_Query` accepts two parameters:

1. a dictionary of query objects `Dictionary<K, QueryObj<T>>` with generic key type `K`, and

2. a partitioning function `Func<T, K>` that maps each element to one of the provided keys[2].

Upon the execution, the partitioner function creates lists of elements that are mapped into the same key and executes the corresponding query (stored in `QueryObj`) on the elements of each list).

7.2 Evaluation

To examine the limitation of our proposed model we attempted to adapt projects that are implemented using PINQ in McSherry and Mahajan (2010) and McSherry (2009). The aim was to replace unrestricted use of PINQ's partition method with `Partition_Query` to see to what extent this was possible. The results are summarised in Table 2. Simple observation of the existing PINQ projects confirmed that none of them use adaptiveness.

The conclusion is that we did not find any examples that cannot be rewritten (with the same privacy cost) to use `Partition_Query`. Here we analyse the examples from the perspective of the relative difficulty of the translation, which we have summarised in right-hand column of the table.

The "Trivial" cases In majority of cases (marked "Trivial") the code takes the form of a partition followed by a simple for-loop over the keys of the partition, applying a static query to the table for each partition. Listing 1.2 illustrates the result of this process on the examples in Listing 1.1.

In translating this and similar examples, we simply remove the partition operation, replicate the structure of the for-loop, but instead of *applying* the query to the partition, we simply *build* the query object and add it to the key-query dictionary(line 8-11), and after the for-loop we call `Partition_Query` with the partition function and the query dictionary thus constructed (line 13).

A canonical example is a differentially private analysis K-means clustering algorithm described in McSherry (2009). For K-means algorithm the parts of the code using partition are listed in Listing 1.5 and its translation is provided in Listing 1.6.

Listing 1.2: FeatherweightPINQ adaption for the sample code

```
1 Dictionary<int, QueryObj<Recordstype>> kq =
2   new Dictionary<int, QueryObj<Recordstype>> ();
3 var agent = new PINQAgentBudget(budget);
4 var data = new PINQueryable<Recordstype>(rawdata.AsQueryable(), agent);
5 var adults = data.Where(x => x.age > 17);
6 var genders = new [] {0,1};
```

[2]Since PINQ is built on top of an embedded database query language LINQ, this is further packaged as a LINQ expression `Expression<Func<T, K>> keyFunc`

```
7
8  foreach (var a in genders) {
9     kq.Add(a, new QueryObj<Recordstype>
10       (queryType.Average, budget/2, x => x.age/100) );
11 }
12
13 var partsValue = adults.Partition_Query(kq, x=>x.gender);
14
15 foreach (var a in genders) {
16    Console.WriteLine("Average age of {0} is {1}",
17       a==0 ? "Males " : "Females ", partsValue[a] * 100) ;
18 }
19 Console.WriteLine("Average age (all):"
20    + data.NoisyAverage(c, x=>x.age/100) * 100) ;
```

Simple Nested Partition In some cases the algorithm partitions the dataset and recursively visits the partitions. In the simplest form the queries that are executed in base case (leaf queries) are executed on disjoint partitions, and each query is independent of the results of other leaf queries. To rewrite this in a way that is faithful to the amount of computation (the number of queries) and the use of the budget we flattened the recursive structure of the algorithm into a single partition. The example which exhibits this behaviour is Example 6 from McSherry (2009) convertible to one single partition transformation (labelled as "Flattening nested partition"). The original and Featherweight PINQ versions are given in Appendix A, listing 1.4.

Multi-Query Nested Partition The trickiest examples involve a more elaborate form of nested partitioning. In these cases not only are there nested partitions, but the queries are applied not only at the leaves (the smallest sub-partitions) but also to intermediate partitions. These examples must be refactored into multiple parallel queries. One such example (*CDF3*) is the case where one wants to calculate a cumulative frequency histogram. A cumulative frequency histogram *could* be computed by partitioning the data and computing the histogram (a parallel query, of cost ε) from which the cumulative histogram can be obtained by cumulative summation. However this approach results in a histogram in which the results are increasingly noisy from left to right. The present algorithm recursively computes the cumulative frequency with a more even error, and a cost $\varepsilon \log n$, where n is the histogram dimension. It is more difficult to convert because it must be refactored in two dimentions: the control-flow, to flatten the recursion, but also in the parallel queries, of which (in this example) there are $\log n$.

Transformation One example of a partition operation (*Stepping Stone*) appeared, at first inspection, to be impossible to convert. It makes a binary partition and then computes an intersection of the two partitions before applying a query to the result. Because the intersection is computed from different partitions, the query sensitivity is not magnified. One can nevertheless view this as an instance of the Featherweight PINQ API without conversion, by observing that the process of partitioning the data followed by a combination operation is itself just a (compound) transformation. Thus the example can be seen as a transformation followed by a simple query, and not a parallel query at all.

Project	Function	Difficulty
Trace Analysis	Discover	Multi-Query Nested Partition
	CDF3	Multi-Query Nested Partition
	FindSets	Trivial
Query Frequency	-	Simple Nested Partition
Anomaly Detection	Main	Trivial
Clustering TTLs	kMeansStep	Trivial
Cumulative Density	CDF2	Trivial
Machine Learning	kMeansStep	Trivial
Social Networking	Main	Trivial
Test Harness	Main	Trivial
Stepping Stones	Main	Transformation
Visualization	Histogram	Multi-Query Nested Partition
Worm Detection	Main	Trivial

Table 2: Result of converting projects to Featherweight-PINQ

8 Related Work

The approach described in this paper owes much to the model used in the formalisation developed in our recent work on *personalised differential privacy* (Ebadi et al., 2015). The idea to model the client program as an abstract labelled transition system comes from that work. That work also shows how dynamic inputs can be handled without major difficulties.

The closest other prior work is developed by Tschantz et al. (2011). Their work introduces a way to model interactive query mechanisms as a probabilistic automata, and develop bisimulation-based proof techniques for reasoning about the differential privacy of such systems. As a running example they consider a system "similar to PINQ", and use it to demonstrate their proof techniques. From our perspective their system is significantly different from PINQ in a number of ways: (i) it does not model the transformation of data at all, but only queries on unmodified input data, (ii) it models a system with a bounded amount of memory, and implements a mechanism which deletes data after it has been used for a fixed number of queries (neither of which relate to the implementation of PINQ). Regarding the proof techniques developed in Tschantz et al. (2011), as previously noted in Ebadi et al. (2015), a key difference between our formalisation and theirs is that they model a passive system which responds to external queries from the environment. In contrast, our model includes the adaptive adversary (the client program) as an explicit part of the configuration. In information-flow security (to which differential privacy is related) this difference in attacker models can be significant (Wittbold and Johnson, 1990). However it may be possible to prove that the passive model of Tschantz et al. (2011) is sound for the active model described here (c.f. a similar result for interactive noninterference (Clark and Hunt, 2009)).

Haeberlen et al. (2011) point out a number of flaws an covert channels in the PINQ system. This may seem at odds with our claims for the soundness of PINQ, but in fact all the flaws described are either covert timing channels (which we do not attempt

to model), flaws in PINQ's implementations of encapsulation, or failure to prevent unwanted side-effects, or combinations of these. Following this analysis, Haeberlen et al introduce a completely different approach to programming with differential privacy (an approach further developed and refined in Reed and Pierce (2010) and Gaboardi et al. (2013)) based on statically tracking sensitivity through sensitivity-types. This non-interactive approach is rigorously formalised and proven to provide differential privacy.

Barthe et al. (2013) introduce a relational Hoare-logic for reasoning formally about the differential privacy of algorithms. They include theorems relating to sequential and parallel composition of queries in the style of those stated by McSherry (2009). Unlike the present work, Barthe et al. (2013) does not rely on differentially private primitives, but is able to prove differential privacy from first principles.

9 Conclusion

We started by presenting some shortcomings(gaps) between the theory of differential privacy and the implementation of PINQ framework. To verify privacy assurance of analysis written in PINQ framework and to address the mentioned concerns, we introduced an idealised model for the implementation of PINQ. In the model, only PINQ's internal implementation has direct access to the sensitive data. An analysis written in this framework has indirect access to the protected system by calling some limited well defined/crafted interface APIs. In addition to the standard PINQ APIs, we extended the model with our own proposed APIs responsible to retrieve scaling factor and the budget from the protected environment. Furthermore we instantiated the definition of differential privacy to prove any analysis constructed in this setting and its communications with protected system would not violate the privacy guarantee promised by PINQ.

We believe that our model (and our general approach to modelling such systems) could be of benefit to formalise emerging variants on the PINQ framework, such as wPINQ (Proserpio et al., 2014), or Streaming PINQ (Waye, 2014).

Extensions to the PINQ API We mention one extension to PINQ that emerges from the details of the correctness proof. In PINQ, the budget and the actual privacy cost of executing an ε differentially private query on some intermediate table is not directly visible to the program:

> *"An analyst using PINQ is uncertain whether any request will be accepted or rejected, and must simply hope that the underlying PINQAgents accept all of their access requests."* (McSherry, 2009)(§3.6)

Recall that the key invariant that relates the two runs of the systems on neighbouring data sets (Definition 9) states that the budgets and the scaling factors in the respective environments are equal. This means that they contain *no* information about the sensitive data. This, in turn, means that we can freely permit the program to query them. This would allow the analyst to calculate the cost of queries and to make accuracy

decisions relative to the current privacy budget. Here we briefly outline this extension. We add two new actions to the set of program actions **ProgAct**, namely a query on the sensitivity of a table variable of the form $tv\ ?\ s$, where $s \in \mathbb{N}$, and a query on the global budget, $\mathsf{budget}\ ?\ v$ where $r \in \mathbb{R}^{\geq 0}$. The transition rules are given in Figure 4.

$$\text{Query sensitivity}\ \frac{P \xrightarrow{tv?s} P'}{\langle P, E, B\rangle \xrightarrow{\tau}_1 \langle P', E, B\rangle}\ \text{where } E(tv) = (T, s)$$

$$\text{Query budget}\ \frac{P \xrightarrow{\mathsf{budget}?B} P'}{\langle P, E, B\rangle \xrightarrow{\tau}_1 \langle P', E, B\rangle}$$

Figure 4: Budget and Scaling Factor

References

Barthe, G., Köpf, B., Olmedo, F., and Béguelin, S. Z. (2013). "Probabilistic Relational Reasoning for Differential Privacy." *ACM Trans. Program. Lang. Syst*, 35(3): 9.

Clark, D. and Hunt, S. (2009). "Noninterference for Deterministic Interactive Programs." In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, volume 5491 of *LNCS*.

Don Box, A. H. (February 2007 (accessed November 18, 2015)). "LINQ: .NET Language-Integrated Query." https://msdn.microsoft.com/en-us/library/bb308959.aspx.

Dwork, C. (2006). "Differential Privacy." In *ICALP (2)*, volume 4052 of *LNCS*, 1–12. Springer.

— (2008). "Differential privacy: A survey of results." In *Theory and Applications of Models of Computation*, 1–19. Springer.

— (2011). "A Firm Foundation for Private Data Analysis." *Commun. ACM*, 54(1).

Dwork, C., McSherry, F., Nissim, K., and Smith, A. (2006). "Calibrating noise to sensitivity in private data analysis." In *Theory of Cryptography*, 265–284. Springer.

Ebadi, H., Sands, D., and Schneider, G. (2015). "Differential Privacy: Now It's Getting Personal." In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15. ACM.

Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., and Pierce, B. C. (2013). "Linear Dependent Types for Differential Privacy." In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*.

Haeberlen, A., Pierce, B. C., and Narayan, A. (2011). "Differential Privacy Under Fire." In *USENIX Security Symposium*.

Igarashi, A., Pierce, B. C., and Wadler, P. (2001). "Featherweight Java: a minimal core calculus for Java and GJ." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3): 396–450.

McSherry, F. and Mahajan, R. (2010). "Differentially-private Network Trace Analysis." *SIGCOMM Comput. Commun. Rev.*, 40(4): 123–134.

McSherry, F. D. (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*, 19–30. ACM.

Proserpio, D., Goldberg, S., and McSherry, F. (2014). "Calibrating Data to Sensitivity in Private Data Analysis." *40th International Conference on Very Large Data Bases, VLDB'14*, 7(8): 637–648.

Reed, J. and Pierce, B. C. (2010). "Distance makes the types grow stronger: a calculus for differential privacy." *ACM Sigplan Notices*, 45(9): 157–168.

Roth, A. (2011). "The Algorithmic Foundations of Data Privacy, Lecture 4, Composition Theorems." Lecture Notes, University of Pennsylvania. Http://www.cis.upenn.edu/ aaroth/courses/slides/Lecture4.pdf.

Tschantz, M. C., Kaynar, D., and Datta, A. (2011). "Formal Verification of Differential Privacy for Interactive Systems (Extended Abstract)." *Electron. Notes Theor. Comput. Sci.*, 276: 61–79.

Waye, L. (2014). "Privacy Integrated Data Stream Queries." In *Proceedings of the 5th annual conference on Systems, programming, and applications: software for humanity*. ACM.

Winskel, G. (1993). *The formal semantics of programming languages: an introduction*. MIT press.

Wittbold, J. T. and Johnson, D. M. (1990). "Information Flow in Nondeterministic Systems." In *IEEE Symposium on Security and Privacy*, 144–161.

Appendix

Further example of translating a PINQ program to Featherweight PINQ's API.

Listing 1.3: Measuring many query frequencies in PINQ

```
// Original PINQ code
var parts = data.Select (line => line.Split (',')).Partition(keys, fields => fields[20]);
foreach (var query in keys)
{
   // use the searches for query, grouped by IP address
   var users = parts[query].GroupBy(fields => fields[0]);
   // further partition by the frequency of searches
```

```
8     var freqs = users.Partition(new int[] {1,2,3,4,5},
9     group => group.Count());
10    // output the counts to the screen, or anywhere else
11    Console.WriteLine(query + ":");
12    foreach (var count in new int[] {1,2,3,4,5})
13       Console.WriteLine(freqs[count].NoisyCount(100));
14 }
```

Listing 1.4: Measuring many query frequencies in Featherweight-PINQ

```
1  var groupedData = data.Select (line => line.Split (','))
2     .GroupBy (fields => new Tuple<string, string>
3        (fields [0], fields [20]));
4  foreach (var query in keys) {
5     foreach (var freq in Enumerable.Range (1, 5).AsQueryable ()) {
6        keyQuery.Add(new Tuple<string,int>(query,freq)
7           ,new QueryObj<IGrouping<Tuple<string,string>,string[]>>
8              (queryType.Count,epsilon, x=> 1 ));
9     }
10 }
11 var partValue = groupedData.Partition_Query (keyQuery
12    , x => new Tuple<string
13    , int> (x.Key.Item1, x.Count()));
14 foreach (var pv in partValue) {
15    Console.WriteLine ("Query "+ pv.Key.Item1
16          +  ",Freq "+ pv.Key.Item2
17          +  ":" + pv.Value);
18 }
```

Listing 1.5: k-Means Clustering in PINQ

```
1  public static void kMeansStep(PINQueryable<double[]> input, double[][] centers, double
       epsilon)
2  {
3     var parts = input.Partition(centers, x => NearestCenter(x, centers));
4     // update each of the centers
5     foreach (var center in centers)
6     {
7        var part = parts[center];
8        foreach (var index in Enumerable.Range(0, center.Length))
9           center[index] = part.NoisyAverage(epsilon, x => x[index]);
10    }
11 }
```

Listing 1.6: k-Means Clustering in Featherweight-PINQ

```
1  public static void kMeansStep(PINQueryable<double[]> input, double[][] centers, double
       epsilon)
2  {
3  foreach (var index in Enumerable.Range(0, centers[0].Length)) {
4     var keyQuery = new Dictionary<double[], QueryObj<double[]>> ();
5     foreach (var center in centers) {
6        var queryObject = new QueryObj<double[]> (queryType.Average, epsilon, x =>
             x[index]);
7        keyQuery.Add(center, queryObject);
```

```
8     }
9     int j = 0;
10 var x = input.Partition_Query (keyQuery, x => NearestCenter (x, centers))
11    foreach (var partValue in x) {
12       centers [j++][index] = partValue.Value;
13       }
14    }
15 }
```