Designing Private Data-Publishing Settings

Chul Lee Hyun
University of Toronto
Toronto, Canada
leehyun@cs.toronto.edu

Yaron Kanza
Technion
Haifa, Israel
kanza@cs.technion.ac.il

Renée J. Miller
University of Toronto
Toronto, Canada
miller@cs.toronto.edu

Zheng Zhang
University of Toronto
Toronto, Canada
zhzhang@cs.toronto.edu

ABSTRACT

When data are published or exchanged, we may want to ensure that certain information is kept private. We consider data-publishing settings (including views, or more generally, data-exchange settings) where the private information is specified by a secret query on the proprietary schema. For such settings, we formulate a privacy notion that ensures those accessing published information cannot learn, with certainty, any answer to the secret query, no matter what data is contained in the private database. We provide algorithms to test the privacy of such data-publishing settings. We consider settings based on a closed-world assumption (for example, settings defined by exact view definitions) as well as setting based on an open-world assumption (for example, settings defined using tuple-generating dependencies). We also propose a method for designing private settings. We experimentally validate the usability of our design solution on large settings.

1. INTRODUCTION

When databases are published or exchanged, we may want to ensure that some portion of the data is kept private and not released. In general, we may want to provide privacy guarantees that do not *unnecessarily* restrict the data that is published.

To define the published data, we may use a set of views, or more generally, the published data may be described by an independent schema and a set of constraints that define the relationship between the published schema and the proprietary schema. Note that the latter is a form of *data-exchange setting* [**?**], and has previously been used for describing published data in other studies of privacy [**?**, **?**]. The secret data is also defined declaratively using a *secret query*. That is, the data we wish not to disclose are the answers to a secret query. In our work, we will refer to a setting (defining the relationship between the published schema and the underlying proprietary schema) and a secret query (defining the data to be concealed) as a *data-publishing setting*. A data-publishing setting provides *privacy* (or is *private*) if no tuple in the answer to the secret query can be inferred with certainty from published data. We will make this notion precise in Section 2.

Our primary goal in this paper is to provide a *privacy test* that determines whether a data-publishing setting is private. When a given data-publishing setting is not private, it is often desirable to modify the given setting so that it becomes private. It is also desirable that the proposed modification will preserve, as much as possible, the original semantics of the setting. Our second goal is to provide a method that will, with a minimal number of modifications, convert a given setting that is not private into a private setting.

Many different definitions of privacy have been considered in the literature. These approaches can be broadly classified as *data-dependent privacy* [**?**, **?**, **?**, **?**] and *data-independent privacy* [**?**, **?**, **?**]. As pointed out in the latter work, data-dependent privacy may not be sufficient in dynamic environments (that is, in environments where the database may change over time).

EXAMPLE 1.1. *Consider a company database as shown in Figure 1 containing the relations Employee, Department and Participates. It may be important for a business to protect certain sensitive information. For example, the company might want to conceal the relationship between employees, the projects they work on, and their department manager. Such information may reveal private company strategies, for example, by potentially indicating that a highly valued employee is working on a certain project whose manager is the company CEO. However, it may be in the best interest of the business to reveal other information, such as who works for them, and what projects they have. In designing a data publishing setting, we want to have tests for determining whether the setting reveals any private information.*

For this example, consider the three relations presented in Figure 1, and the following secret query Q_s to specify the information that should be kept secret.

```
Qs  SELECT S.Name, S.ProjName, D.MgrName
    FROM Participates S, Department D
    WHERE S.DeptName = D.DeptName
```

The company, however, does want to disclose general information on its employees, departments, and projects. A system administrator decides to publish information about employees and all related projects. A project is related to an employee if there is someone else in the same department participating on the project.

```
CREATE VIEW V1 as
SELECT DISTINCT S1.Name, S1.DeptName,
       S2.ProjName
FROM   Participates S1, Participates S2,
WHERE  S1.DeptName = S2.DeptName and
       S1.Name <> S2.Name
```

Suppose that the company also publishes information on employees, their department, and their manager using the following view.

Figure 1: Proprietary instances and view instances.

```
CREATE VIEW V2 as
SELECT DISTINCT S.Name, S.DeptName, D.MgrName
FROM Participates S, Department D
WHERE S.DeptName = D.DeptName
```

Let us consider whether these views effectively conceal the information of the secret query Q_s. If we publish an instance of the views, say $\mathcal{I}$, users would learn some information about the proprietary instance. For example, a user, Alice, would learn some of the employee names, some project names, and some manager names in the proprietary instance. However, can Alice learn, for certain, whether the answer to Q_s over the proprietary instance contains a tuple with a specific employee name, a specific project name, and a specific manager name? To answer this question, consider that for a given instance of the views $\mathcal{I}$, there are many possible proprietary instances that could produce $\mathcal{I}$; we say that Alice can learn a secret tuple, for example a tuple such as ("John", "Privacy", "Amy"), if this tuple is in the answer to Q_s over every possible proprietary instance that produces the view instance $\mathcal{I}$. That is, if the tuple ("John", "Privacy", "Amy") is in the certain answers *of Q_s with respect to the published instance $\mathcal{I}$ [**?**].*

Suppose that the view instance $\mathcal{I}$ contains the relation I_1, an instance of the view V_1 with exactly two tuples ("John", "DB", "SQL") and ("Amy", "DB", "Privacy"). Instance $\mathcal{I}$ also contains the relation I_2, an instance of V_2 with two tuples ("John", "DB", "Amy") and ("Amy", "DB", "Amy"). These tuples are depicted in gray in Figure 1. Then, from the tuple ("John", "DB", "SQL") in I_1, Alice can infer that there exist, in the relation `Participates` *of the underlying proprietary database D, two tuples of the form ("John","DB",p) and (e,"DB", "SQL") for some e and p where $e \neq$ "John". She can also deduce that a tuple of the form Participates (e, "DB", p) must appear in I_1. If the instance I_1 consists of only these two tuples ("John", "DB", "SQL") and ("Amy", "DB", "Privacy"), then she can conclude that $e =$ "Amy" and $p =$ "Privacy". Hence, the tuple ("John", "DB", "Privacy") and the tuple ("Amy", "DB", "SQL") must appear in the* `Participates` *relation of any possible proprietary instance. From the tuple ("John", "DB", "Amy") in I_2, Alice can infer that there exists, in the relation* `Department` *of the proprietary instance D, a tuple of the form ("DB", "Amy"). Hence, she can conclude that the tuple ("John", "Privacy", "Amy") is an answer to Q_s over* **any** *proprietary instance that produces $\mathcal{I}$, and therefore is in $Q_s(D)$. Notice that the reasoning we used here was very specific to the instance $\mathcal{I}$. For (many) other instances of the views, Alice is not able to deduce secret answers. For example, consider a view instance $\mathcal{I}'$ comprises of the instances I'_1 and I'_2 that are depicted in Figure 1. Let D_1 and D_2 be the two proprietary databases that are shown in Figure 1 (for each relation it is written in parenthesis whether it belongs to D_1, to D_2 or to both.) The two proprietary databases D_1 and D_2 both produce $\mathcal{I}'$ whereas $Q_s(D_1) \cap Q_s(D_2) = \emptyset$. Hence, from $\mathcal{I}'$ Alice cannot conclude, with certainty, any secret answer.*

As shown in the previous example, a data-publishing setting may conceal secret data for one published instance while revealing it for another instance. Our goal is, however, to guarantee that the secret data is concealed for any possible instance.

A possible attempt to avoid disclosure of private information is to require that all the published data will satisfy k-anonymity [**?**]. In order to provide k-anonymity, data may be altered to ensure any answer is indistinguishable (in a precise sense) from k other answers. In many data-publishing applications, we may not want to mutate data values. Furthermore, k-anonymity was originally designed for static data and the issues of using it with dynamic data are just emerging [**?**].

Another approach to avoid disclosure of private information is to check each time the data is updated, using a data-dependent test, whether the data-publishing setting is private. However, this approach does not guarantee privacy [**?**]. Moreover, this solution is costly for an instance that has frequent updates, and when a setting is determined to not be private for an instance, we would have to change the setting to permit any data to be published. Hence, in this work, we focus on data-independent privacy.

There are several data-independent privacy notions that have been proposed. One notion is Multi-level Security Databases (MLSD) which was proposed by Brodsky *et al.* [**?**]. Their work only considers select-project type queries over a single table. Nash and Deutsch [**?**] have investigated the privacy problem in GLAV data-integration systems. In a data-integration system, user queries are answered by certain answers (*i.e.*, the published data is not materialized as in data exchange). A number of different privacy notions are proposed in their work, and a breach of privacy is defined as completely revealing a source or completely revealing the secret. Note that our notion of privacy is stricter than the privacy notion of Nash and Deutsch [**?**]. We consider as a breach of privacy the ability to infer at least one tuple from the secret whereas they consider it as the ability to infer the whole secret. This difference leads to an apparent disparity in complexity results. Nash and Deutsch [**?**] show that, in general, it is undecidable to test whether a GLAV data-integration system provides data-independent privacy. In contrast to their work, we will show that in data-publishing settings, testing data-independent privacy, is decidable and in some cases even polynomial.

A setting that is not private can be converted into a private one. In some cases privacy can be achieved by using an *open* setting rather than a *close setting*. In an open data-publishing setting, the published data is not required to contain *all* the data in the proprietary instance. Hence, if the proprietary instance would reveal a secret, we simply elect not to publish all the data. An open data-publishing setting is defined in terms of tuple-generating dependencies (tdgs), which are often referred to as *open* dependencies [**?**] defined from the published view or schema to the proprietary database. In a *closed* data-publishing setting, the views are completely revealed. It is defined in terms of exact or closed dependencies [**?**, **?**].

EXAMPLE 1.2. *In Example 1.1, we argued that the setting is not private because there are instances for which Alice could reveal tuples of the secret. We showed that for an instance I_1 that consists of only the two tuples ("John", "DB", "Privacy") and ("Amy", "DB", "Privacy"), and I_2 that only consists of the two tuples ("John", "DB", "Amy") and ("Amy", "DB", "Amy"). However, we also showed that adding tuples to I_1 and I_2 can generate the instances I'_1 and I'_2 for which Alice cannot infer any tuple of the secret with certainty.*

Now, suppose that the instances I_1 and I_2 are published in an open setting. Alice does not know what the company elected not to publish. The underline proprietary database can be D (the database that contains only the tuples depicted in gray in Figure 1), D_1, D_2 or any other database that contains the tuples of D. Hence, there are no certain answers to Q_s for the published instance $\{I_1, I_2\}$, and Alice cannot infer with certainty any tuple of the secret. Furthermore, if Alice does not have any information about the number of employees, projects or project managers in the company, she would not even be able to estimate the likelihood of some tuple to be in the secret.

Not every open data-publishing setting is private. As an example, consider adding Q_s as a view V_3 to the setting of Example 1.1 (*i.e.,* the case when Q_s is both the secret query and a view). Obviously, in such case, any answer to V_3 is a certain answer to Q_s. Hence, we need not only a privacy test for closed settings, but also a privacy test for open settings.

When a setting cannot be made private by publishing it as an open setting, or when there are reasons not to publish only part of the data, a user may use the approach of modifying the setting for making it private. For example, we can remove data from the published database by removing an attribute from a view. We can also hide information by removing a join condition from the view.

EXAMPLE 1.3. *A possible modification to the setting of Example 1.1 is to remove the attribute `MgrName` in the definition of the view V_2,* i.e., *V_2 will consist of only the attributes `Name` and `DeptName` from the `Participates` relation. Another possible change is to delete the join condition `S.DeptName = D.DeptName` in the definition of the view V_2. In both cases, after the modification Alice will not be able to infer from the view instances any certain answer to Q_s. Hence, the modified settings are private.*

Modifying the setting to provide privacy also changes the original view semantics. In general, it may be very hard for a human user to design a data-publishing setting that is private. Our examples illustrate the intricate reasoning needed to verify privacy. Hence, we propose a solution where the user can design a data-publishing setting based on her application requirements, use our tests to determine if it is private, and then, for settings that are not private, invoke our design algorithm to generate a modified setting that is private. Our algorithm tries to minimize the change to the semantics of the data publishing setting.

Example 1.2 and Example 1.3 illustrate the degree of privacy provided in our approach. In these examples, for every given employee e and every given manager m, Alice cannot learn from any published instance whether e works in some project managed by m. In k-anonymity, for comparison, it is required that there will be at least k possible managers for e, where, typically, k is a number around 5. In practical scenarios where the number of managers is greater than the k used in k-anonymity, our approach will provide, at least, the anonymity provided by k-anonymity, for every instance of the proprietary database.

Our main contributions are the following.

- We propose a new data-independent notion of privacy that provides finer-grained privacy guarantees than previous proposals. Specifically, we guarantee that a user will not be able to determine, with certainty, any answer to a secret query.
- We provide algorithms for testing whether a setting is private.
- We evaluate our algorithms over a large number of synthetically generated settings and show that they are efficient.
- We develop a design algorithm that can convert a setting that is not private into a private setting. Our approach is based on searching for edits that minimally change the original setting.
- We implement and evaluate our design algorithm, and show that it is indeed effective in finding modified (private) settings containing only small changes from the original (non-private) setting. To the best of our knowledge, this is the first work in automatic design of private data publishing settings.

The paper is organized as follows. In Section 2, we will formally present a new notion of data-independent privacy. In Section 3, we present algorithms for testing whether a data-publishing setting is private. We will describe the application of our approach and how to implement our approach in Section 4. We experimentally validate the efficiency and applicability of our proposed approach in Section 5. Related work is reviewed in Section 6. Finally, we present future work and conclude in Section 7.

2. FRAMEWORK

We now define data-publishing settings, and a new data-independent privacy notion.

Schemas and Instances Let $\mathbf{P}$ be a proprietary schema $\mathbf{P}$, composed of a set of relations $\{P_1, \ldots, P_n\}$, modelling the proprietary data. Let $\mathbf{V} = \{V_1, \ldots, V_m\}$ be a public schema modelling the structure of the data to be published where $\mathbf{V}$ and $\mathbf{P}$ are disjoint. Instances over the public schema $\mathbf{V}$ are called *view instances* for consistency with the privacy literature where data is typically published through views. We use I (possibly with a subscript) to represent instances of $\mathbf{V}$. Instances over $\mathbf{P}$, denoted by D, are called *proprietary instances* (or *possible instances*).

EXAMPLE 2.1. *In Example 1.1, the proprietary schema consists of the relations `Department` (in some examples we will refer to it as `Dept`), `Employee`, and `Participates` while the view schema consists of V_1 and V_2.*

Queries In our paper, we will consider select-project-join queries. To make the notation for our algorithms concise, we will represent queries using standard conjunctive query notation.

EXAMPLE 2.2. *Consider the secret query Q_s in Example 1.1* *which specifies that the relationship between employees, their department manager, and their projects should be concealed.*

$$Q_s(n,p,m) \quad \vdash \quad \exists\, d\ .\ \texttt{Participates}(n,d,p), \texttt{Dept}(d,m)$$

Data-Publishing Setting Let $\Gamma = (\mathbf{V}, \mathbf{P}, \Sigma_{vp})$ where $\mathbf{V}$ is a public schema, $\mathbf{P}$ is a proprietary schema and Σ_{vp} is a constraint over $\mathbf{P}$ and $\mathbf{V}$. Then, a *data-publishing setting* is (Γ, Q_s) where Q_s is a secret query over $\mathbf{P}$. In this work, we assume Q_s is a project-select-join query.

We define two different types of *data-publishing settings* distinguished by the type of constraint Σ_{vp} they permit.

In an *open data-publishing setting*, Σ_{vp} is a set of source-to-target (view-to-proprietary) tuple generating dependencies (st-tgds) [**?**]. Such assertions are the most common forms of schema mappings used in the literature [**?**, **?**, **?**]. Specifically, in an open setting Σ_{vp} contains a set of constraints of the following form.

$$\forall \bar{x}\ Q_V(\bar{x}) \rightarrow \exists y\ Q_P(\bar{x}, \bar{y})$$

where Q_V and Q_P are project-select-join queries over the view schema $\mathbf{V}$ and the proprietary scheme $\mathbf{P}$, respectively.

In an open setting, the constraints assert that data in the (published) view instance, must be justified data in the proprietary instance conforming to Σ_{vp}. Note that in such a setting, the view instance may contain a subset of the data satisfying Σ_{vp}.

Open settings do not include the most common form of data-publishing settings, those defined by views. If Σ_{vp} is a standard view definition, then the published view instance is fully defined. To model such settings, we use exact constraints [**?**, **?**]. In a *closed* data-publishing setting, for each tgd from the view to the proprietary schema of the following form:

$$\forall \bar{x}\ Q_V(\bar{x}) \rightarrow \exists \bar{y}\ Q_P(\bar{x}, \bar{y})$$

we have a corresponding tgd from the proprietary to the view schema of the form:

$$\forall \bar{w}\ Q_P(\bar{w}) \rightarrow \exists \bar{z}\ Q_V(\bar{w}, \bar{z})$$

Let Σ_{pv} be the set of all proprietary-to-view tgds that are added to make the setting closed.

EXAMPLE 2.3. *Consider the view V_2 of Example 1.1. In an open setting, V_2 is represented by the following tgd (universal quantifiers are suppressed):*

$$V_2(n,d,m) \rightarrow \exists\, p, b.\ \texttt{Participates(n,d,p),Dept(d,m)}$$

In a closed setting, Σ_{vp} includes the constraint above and the following proprietary-to-view dependency.

$$\texttt{Participates(n', d', p'), Dept(d', m')} \rightarrow V_2(n', d', m')$$

Certain Answers Let (Γ, Q_s) be a data-publishing setting, where $\Gamma = (\mathbf{V}, \mathbf{P}, \Sigma_{vp})$ and Q_s is a query over the proprietary schema $\mathbf{P}$. Let I be a view instance. Given an instance I of $\mathbf{V}$, we can define *solution* to Γ as a proprietary instance D such that all the dependencies of Γ are satisfied [**?**]. A solution represents a possible propriety instance according to the constraints in the setting, that could have produced the published view instance I. Given I, *universal solution* to Γ is defined in the usual way [**?**]. Intuitively, a universal solution is a representation of all the possible solutions to (Γ, I).

Given two instances D_1 and D_2 containing constants and labeled nulls, a *homomorphism* $h : D_1 \rightarrow D_2$ is a mapping from the constants and labeled nulls in D_1 to those in D_2 such that (1) $h(c) = c$ for every constant c, (2) for every tuple $\bar{t}$ of relation R in D_1, $h(\bar{t})$ is a tuple of relation R in D_2. For every solution, there is a homomorphism from the universal solution to it. Given an instance I of $\mathbf{V}$, the certain answers of Q_s, denoted by *certain*(Q_s, I), are the set of all tuples $\bar{t}$ of constants such that for any possible solution D, $\bar{t}$ is in $Q_s(D)$, *i.e., for all* D *such that* D *is a solution to* (Γ, I). *The certain answers of Q_s with respect to a setting* Γ, denoted by *certain*(Q_s, Γ), are the union of all certain answers of Q_s for all possible view instances I of the view schema $\mathbf{V}$. That is, *certain*$(Q_s, \Gamma) = \cup_{I \textit{ is an instance of } \mathbf{V}}(\textit{certain}(Q_s, I))$.

Given a published instance I, a possible solution D to Γ may contain labelled nulls. We assume that a proprietary instance contains constants only (hence, the published instance contains constants only). Let *certain*$_c(Q_s, I)$ denote the set of certain answers of Q_s given I when the possible solutions D contain no labelled nulls. Similarly, let *certain*$_c(Q_s, \Gamma)$ denote the set of certain answers of Q_s when the possible solutions D contain no labelled nulls. Since a labeled null represents an arbitrary constant, it is easy to see that *certain*$_c(Q_s, \Gamma) \subseteq$ *certain*(Q_s, Γ). Since labelled nulls can be considered as distinct unique constants, *certain*$(Q_s, \Gamma) \subseteq$ *certain*$_c(Q_s, \Gamma)$.

PROPOSITION 2.4. *Given a data-publishing setting* (Γ, Q_s)*, certain*$_c(Q_s, \Gamma) = \emptyset$ *if and only if certain*$(Q_s, \Gamma) = \emptyset$*.*

Privacy We now give the formal definition of privacy in a data-publishing setting.

DEFINITION 2.5. *A data-publishing setting* (Γ, Q_s) provides privacy *(or is private) if Q_s has no certain answer for any instance I of* $\mathbf{V}$*. That is, certain*$(Q_s, \Gamma) = \emptyset$*.*

Given an open setting that is private, the corresponding closed setting may not be private. The following proposition claims that privacy in closed settings is at least as strict as that in open settings.

PROPOSITION 2.6. *Given a data-publishing setting* (Γ, Q_s)*, if* (Γ, Q_s) *is private as a closed setting, then* (Γ, Q_s) *is private as an open setting as well.*

Complexity Since our notion of privacy is data-independent, we focus on the *query complexity* [**?**] when we present the complexity results of the algorithms presented in this paper. We refer to the number of relations in the schema as *the size of a schema*. We refer to the number of subgoals in the dependency (query) as *the size of a dependency (query)*. For a set of dependencies, Σ_{vp}, we refer to the sum of the dependencies' sizes as *the size of a set of dependencies*.

3. TESTING PRIVACY

In this section, we present a set of algorithms for testing privacy of data-publishing settings. We first discuss settings where both the secret query and the queries used in the Σ_{vp} dependencies are *project-join* queries (specifically, the queries do not contain constants) in Section 3.1. We next discuss settings where both the secret query and dependencies may also contain equality selections (Section 3.2). We finally discuss settings where either the secret query or views contain inequalities in Section 3.3

3.1 Project-Join Queries

We present algorithms for testing privacy for data-publishing settings where the queries and dependencies use only project-join queries.

3.1.1 Open Settings

As we briefly mentioned in Section 1, an open data-publishing setting is similar to a data-exchange setting [?] in the sense that its view and proprietary schemas can be viewed as source and target schemas in a data-exchange setting (respectively). Following Fagin et al. [?], given an instance of the view schema, we can use a finite chase to compute a universal solution to the given data-exchange setting. Fagin et al. [?] also showed that given a source instance I, the certain answers to a query Q over the target schema are exactly the answers to Q executed over the universal solution, which consist of merely constants (*i.e.,* those answers that do not contain any of the labeled nulls created by the chase).

Our notion of privacy in data-publishing settings has some differences from that of query answering in data exchange settings. In the context of data-exchange, it is sufficient to compute the certain answers for a single source (in this case, view) instance, whereas in the context of private data-publishing, we must check that there is no certain answer for any possible view instance. Since there may be an infinite number of possible view instances, it is practically impossible to separately test privacy with respect to each view instance. Therefore, our approach to testing privacy is to construct an instance I_c such that there is a certain answer to Q_s for some possible published view instance iff there is a certain answer to Q_s for I_c. Hence, we can restrict our privacy test to the checking of certain answers with respect to I_c. We further describe our intuition through the following example.

EXAMPLE 3.1. *Consider the secret query and data-publishing setting from Example 1.1 where we have simplified the setting by omitting the final inequality selection and defined the setting as an open setting.*

$$Q_s(n,p,m) \quad \vdash \quad \exists\, d\ .\ \texttt{Participates}(n,d,p), \texttt{Dept}(d,m)$$

$$\begin{aligned}\Sigma_{vp}: V_1(n,d,p) &\rightarrow \texttt{Participates(n,d,p')},\\ &\quad\ \texttt{Participates(n',d,p)}\\ V_2(n,d,m) &\rightarrow \texttt{Participates(n,d,p)},\\ &\quad\ \texttt{Dept(d,m)}\end{aligned}$$

Let us consider what properties view instances should satisfy so that a certain answer exists for Q_s (a necessary condition for our privacy test). One can see that some `Name and DeptName` *value in V_1 should be equal to a* `Name and DeptName` *value in Q_s. In fact, since we only have project-join type queries in our view definitions, we can set each attribute value of the views to be the same such that the join conditions in Q_s are properly satisfied. There will be a certain answer to Q_s for some possible view instances iff there is a certain answer to Q_s for a specially defined instance I_c. Note that this occurs when each constant in the view instance is continuously replaced by the same constant c. The constant c can be chosen arbitrarily. The constraint satisfaction in Γ does not depend on the choice of constant c.*

ALGORITHM 3.2. *Algorithm for testing privacy of an open data-publishing settings containing only project-join queries, in the dependencies and in the secret query.*

1. *Let c be some arbitrary constant. Create a view instance I_c as follows. For each relation V in the schema $\mathbf{V}$, the relation $I_c(V)$ contains a k-tuple $(c,\ldots,c)$ where k is the arity of V. There are no other tuples in I_c.*

2. *Compute a universal solution D_c to Γ for I_c. If D_c does not exist, then (Γ, Q_s) is private.*

3. *If $Q_s(D_c)$ contains the tuple $(c,\ldots,c)$ of all constants, then the setting is not private. Otherwise, (Γ, Q_s) is private.*

Next, we illustrate this algorithm on a concrete example.

EXAMPLE 3.3. *Consider the data-publishing setting in Example 3.1 (representing the setting of Example 1.1 without the inequality constraints.) The algorithm first generates an instance I_c where $I_c(V_1)$ consists of a single tuple (c,c,c) and $I_c(V_2)$ consists of a single tuple (c,c,c). A universal solution D_c is computed by applying the tgds that represent these (open) views. We refer to the i-th labeled null as $\perp_i$. The solution D_c is given as follows. The relation* `Participates` *contains tuples $(c,c,\perp_1)$, $(\perp_2,c,c)$ (from the dependency for V_1), and $(c,c,\perp_3)$ (from V_2). The relation* `Department` *contains a tuple (c,c). The evaluation of Q_s over D_c produces the tuples $(c,\perp_1,c)$, $(\perp_2,c,c)$ and $(c,\perp_3,c)$. Since there is no tuple in the result consisting of merely constants, there is no certain answer to Q_s for any view instance, hence, the given data-publishing setting is private.*

The correctness of Algorithm 3.2 is stated by the following theorem.

THEOREM 3.4. *Algorithm 3.2 tests correctly whether an open setting (Γ, Q_s) which uses only queries with projects and joins is private.*

We now discuss the complexity of Algorithm 3.2. This complexity is mostly influenced by the number of chase steps when computing the universal solution, and by the complexity of evaluating Q_s over the proprietary instance.

PROPOSITION 3.5. *Consider an open data-publishing setting (Γ, Q_s) which uses only project-join queries. Algorithm 3.2 has $O(|\Sigma_{vp}|^{|Q_s|})$ time complexity, where $|\Sigma_{vp}|$ and $|Q_s|$ are the sizes of Σ_{vp} and Q_s, respectively.*

3.1.2 Closed Case

Testing privacy in a closed setting requires additional work. A closed setting imposes additional constraints that will decrease the set of possible proprietary instances for some view instances. Before presenting our algorithm for closed settings, we first introduce some notation to simplify the discussion. Recall that in Section 2, we defined *implicit* propriety-to-view dependencies Σ_{pv} which are added to make a given open data-publishing setting closed. For a closed setting, let us make these additional dependencies explicit, that is $\Gamma = (\mathbf{V}, \mathbf{P}, \Sigma_{vp}, \Sigma_{pv})$. Furthermore, let $\Gamma^b = (\mathbf{P}, \mathbf{V}, \Sigma_{pv})$ be the *inverse setting* of Γ. Let I_D^b denote the instance created by chasing a proprietary instance D using Γ^b.

ALGORITHM 3.6. *Algorithm for privacy testing of closed data-publishing settings containing only project-join queries in the dependencies and secret query.*

1. *Apply Steps 1–2 of the Algorithm 3.2 and compute a solution D_c for Γ as in the open setting case. If a universal solution D_c does not exist, then (Γ, Q_s) is private.*

2. *Compute, over the instance D_c, a universal solution $I_{D_c}^b$ using the inverse setting $(\mathbf{P}, \mathbf{V}, \Sigma_{pv})$.*

3. *For each labeled null $\perp_i$ from D_c that appears in $I_{D_c}^b$, replace in D_c all the occurrences of $\perp_i$ by c.*

4. *If $Q_s(D_c)$ contains the tuple $(c, \dots, c)$ merely consisting of constants, then a breach of privacy occurs in (Γ, Q_s). Otherwise, (Γ, Q_s) is private.*

Next, we illustrate how Algorithm 3.6 works through an example.

EXAMPLE 3.7. *Consider the data-publishing setting in Example 3.1. When considering this setting as a close setting, in addition to Σ_{vp} we have the following two ts-tgds.*

```
Σ_pv :  Participates(n,d,p'),
            Participates(n',d,p) → V_1(n, d, p)
        Participates(n,d,p),
            Dept(d,m)→ V_2(n, d, m)
```

The run of Algorithm 3.6 over this setting is as follows. First, a universal solution D_c is computed by applying Σ_{vp} on the instance I_c that has a single tuple (c, c, c) in V_1 and a single tuple (c, c, c) in V_2. The relation `Participates` *contains tuples $(c, c, \perp_1)$, $(\perp_2, c, c)$ (generated by applying the st-dependency for V_1) and $(c, c, \perp_3)$ (generated by applying the st-dependency for V_2). The relation* `Department` *contains a tuple (c, c).*

Next, the algorithm applies the tgds backwards to generate $I^b_{D_c}$. In $I^b_{D_c}$, the relation V_1 contains tuples (c, c, c), $(c, c, \perp_1)$, $(c, c, \perp_3)$, $(\perp_2, c, c)$, $(\perp_2, c, \perp_1)$, and $(\perp_2, c, \perp_3)$; the relation V_2 contains tuples (c, c, c) and $(\perp_2, c, c)$. According to Step 3, the labeled nulls $\perp_1, \perp_2$, and $\perp_3$ that appear in $I^b_{D_c}$ should be replaced by c in D_c. As a result, in D_c there exists `Participates` *(c, c, c) and* `Department`*(c, c). Evaluating Q_s over D_c, after replacing the null values by c, produces a tuple (c, c, c) made of constants. The tuple (c, c, c) is a certain answer to Q_s over the instances I_c. Thus, Algorithm 3.6 completes the run indicating that the given setting is not private.*

THEOREM 3.8. *Algorithm 3.6 tests correctly if a closed setting (Γ, Q_s) containing only project-join queries is private.*

We now discuss the complexity of Algorithm 3.6. The complexity is dominated by the generation step for $I^b_{D_c}$. This is due to Step 2 where it applies tgds in a form $Q_{\mathbf{P}}(\bar{x}) \rightarrow Q_{\mathbf{V}}(\bar{x})$. Since the execution of this step requires the evaluation of conjunctive query $Q_{\mathbf{P}}$ over D_c, the complexity of Algorithm 3.6 is related to the complexity of conjunctive-query evaluation. Not that for open settings, we are evaluating queries over an instance of fixed size, but for closed settings, our evaluation is over an instance whose size depends on the size of the setting. This leads to a higher query complexity for privacy testing of closed settings.

PROPOSITION 3.9. *For any closed setting containing only project-join queries, the following two statements hold.*

- *For a fixed data exchange setting Γ, it is coNP-hard in the size of the given secret query Q_s to decide whether Γ is private for Q_s.*
- *For a fixed secret query Q_s, it is coNP-hard in the size of Σ_{vp} to decide whether a given data exchange setting Γ (with fixed $\mathbf{V}$, $\mathbf{P}$) is private for Q_s.*

Although the best complexity of our Algorithm 3.6 for closed settings without constants is not polynomial, we will experimentally show in Section 5 that our algorithm is relatively efficient in practice since it is independent of any data instance.

3.2 Select-Project-Join Queries

In this section, we present how to extend the previous algorithms for settings in which the secret query and dependencies may be select-project-join (SPJ) queries. We start our discussion with the open setting case.

3.2.1 Open Case

Settings containing selections may refer to specific constants (in the dependencies or secret query). We again want to construct a single instance, say I_C, such that there is a certain answer to Q_s given a possible view instance if and only if there is a certain answer to Q_s given I_C. However, we cannot have the same construction as that for settings without constants, since in settings with constants there might exist selection constraints that cannot be satisfied by an instance I_C formed using a single arbitrary constant c. So, we show how to construct a more general I_C that is sufficient and that uses the constants that appear in the setting (that is, in Q_s or Σ_{vp}).

ALGORITHM 3.10. *Algorithm for testing the privacy of open data-publishing settings with SPJ queries.*

1. *Let C denote the set of all constants in Γ and Q_s. We create a view instance I_C as follows. For each V in the view schema $\mathbf{V}$, $I_C(V) = \{(c_1, \dots, c_k) \mid c_i \in C, \text{for } 1 \le i \le k\}$, where k denotes the arity of V.*
2. *Apply Step 2 and Step 3 of Algorithm 3.2 using this I_C.*

THEOREM 3.11. *Algorithm 3.10 correctly tests whether an open setting (Γ, Q_s) with SPJ queries is private.*

It is easy to see that the size of I_C is exponential in the size of Γ, where the maximal arity of $\mathbf{V}$ is in the exponent and the size of C is in the base. In many practical cases, though, it is reasonable to assume that both the maximal arity of $\mathbf{V}$ and the size of C are bounded and relatively small. So, our algorithms has the following time complexity.

PROPOSITION 3.12. *The time-complexity of Algorithm 3.10 is $O((|\Sigma_{vp}| \cdot |C|^{|\mathbf{V}|})^{|Q_s|})$, where $|\mathbf{V}|$, $|\Sigma_{vp}|$, $|C|$ and $|Q_s|$ denote the sizes of $\mathbf{V}$, Σ_{vp}, C and Q_s, respectively.*

Note that this complexity is similar to the $O(|D|^{|Q|})$ time complexity of evaluating a conjunctive query Q over a database D.

3.2.2 Closed Case

Closed settings with constants require a more intricate approach than open settings. The privacy test should handle constants (as in Algorithm 3.10) and satisfaction of the target-to-source constraints (as in Algorithm 3.6). On one hand, we need to increase the view instance with tuples comprising different constants. On the other hand, inferring equality between a labeled null and a constant is due to a unique value in some attribute, which is more frequent in small instances than in large ones. Our solution to this is to construct an instance I_C similar to I_C that was constructed above for open settings, but apply the test to every subset of I_C. (We use the standard definition of subset for relational instances, that is $I \subseteq I'$, if $I(V) \subseteq I'(V)$ for every relation V in $\mathbf{V}$.)

EXAMPLE 3.13. *Suppose that a company wants to conceal participation of employees in projects. This is specified by the following query.*

```
Q_s  SELECT Name, Project
     FROM Participates
```

Now suppose that the company wants to publish the following two views. A view of all the names of employees living in San Diego

```
V_1  CREATE VIEW V1 AS
     SELECT DISTINCT Name
     FROM Employee
     WHERE City = ``San Diego''
```

and a view showing the relationships between cities and projects

```
V2  CREATE VIEW V2 AS
    SELECT DISTINCT ProjName, City
    FROM Employee E, Participates P
    WHERE E.Name = P.Name
```

Also, suppose that there are two views showing in which department John works and in which department Amy works. Now if we consider an instance I_1 of V_1 that contains the two tuples ("John") and ("Amy"), then even if Alice sees a tuple ("Privacy", "San Diego") in the instance I_2 of V_2, she cannot conclude, for certain, that John participates in the "Privacy" project. Yet, if the instance I_1 contains only the tuple ("John"), then Alice can infer that John is the only employee living in San Diego and, hence, ("John", "Privacy") is a certain answer of Q_s.

ALGORITHM 3.14. *Algorithm for privacy testing of closed data-publishing settings containing select-project-join queries in the dependencies and secret query.*

1. *Let $C = C_0 \cup C_1$, where C_0 is the set of all constants in Γ and Q_s, C_1 is a set of k fresh constants, and k is the number of relational atoms in the body of Q_s.*
2. *Create a view instance I_C from the constants of C, as in Step 1 of Algorithm 3.10.*
3. *If there exists $I \subseteq I_C$ such that* CertainAnswers(Γ, Q_s, I) *is not empty, then a breach of privacy occurs in (Γ, Q_s). Otherwise, (Γ, Q_s) is private.*

When constructing I_C, we use C that contains the constants of the setting and additional new constants. Intuitively, the new constants help decreasing repetitions of values in different tuples. The number of new constants in C_1 is k because every answer to Q_s can be produced from k tuples of the database; and k tuples can be generated by applying st-tgds on k tuples of the view.

Since the tgds in Σ_{pv} limit the possible solutions (I_C, D_C) to Γ, we cannot guarantee I_C is a solution for any D_C. We must make sure that every tuple t generated by applying some ts-tgd on D_C represents at least one tuple of I_C, that is, there must be a homomorphism from t to some tuple of I_C. For example, if a tuple $t = (c_1, \perp_1)$ is generated by applying a dependency $R(x, y) \rightarrow V(x, y)$ on D_C, then either $I_C(V)$ must include a tuple of the form (c_1, c_2) or (I_C, D_C) is not a solution to Γ. Furthermore, if (c_1, c_2) is the only tuple in V having c_1 in the first position, then we can infer $\perp_1 = c_2$.

PROCEDURE 3.15 (CertainAnswers(Γ, Q_s, I)). *A procedure for computing the certain answers of Q_s, for an instance I and a closed settings Γ containing select-project-join queries.*

1. *Compute a universal solution D for Γ and I.*
2. *Compute over D a universal solution I_D^b, using the inverse setting $(\mathbf{P}, \mathbf{V}, \Sigma_{pv})$.*
3. *For each tuple t in I_D^b, let H_t be the set of tuples t' in I such that there is a homomorphism $h_{t,t'}$ from t to t'. If H_t is empty, then return an empty set (since there are no certain answers to Q_s for the instance I).*
4. *For each tuple t in I_D^b and each labeled null $\perp_i$ in t, check the following. If there exists a constant c such that for each t' in H_t there is a homomorphism from t_c to t', where t_c is the result of replacing $\perp_i$ by c in t, then replace all the occurrences of $\perp_i$ in D by c.*
5. *If in Step 4 some labeled null was replaced by a constant, return to Step 2.*
6. *Return all tuples of $Q_s(D)$ consisting of merely constants.*

THEOREM 3.16. *Algorithm 3.14 tests correctly whether a close setting (Γ, Q_s) with SPJ queries is private.*

3.3 Further Extensions

In this section, we consider a few extensions to data-publishing settings.

3.3.1 Inequalities

First we consider setting that include SPJ queries with inequalities. We illustrate an approach for extending Algorithm 3.14 in which we include in I_C new constants (not equal to any constants that are added to I_C in Step 1 of Algorithm 3.14.) The number of such constants is determined by the setting (and hence is bounded by the size of Γ).

To illustrate our approach, suppose that the secret query Q_s has a subgoal of the form $x \neq y$, where x and y are two variables (the case of inequalities of the form $x \neq c$ where x is a variable and c is a constant is similar, hence, we will not discuss it specifically.) In our algorithms, we evaluate Q_s over a universal solution and we check if the result consists of a tuple with no labeled nulls. Hence, we need to determine when the subgoal $x \neq y$ is satisfied during the evaluation of Q_s over a universal solution.

It is easy to see that there are three cases in which $x \neq y$ is satisfied during the evaluation of Q_s. First, when x and y are mapped to two constants c and c' such that $c \neq c'$. Second, when x and y are mapped to two labeled nulls $\perp_1$ and $\perp_2$, and it is possible to infer that $\perp_1 \neq \perp_2$. Third, when one variable is mapped to some labeled null $\perp$, the other variable is mapped to some constant c and it is possible to infer that $\perp \neq c$. Inference that $\perp_1 \neq \perp_2$ can be done when during the chase we apply a dependency that has a subgoal $z \neq w$ in the target query, $\perp_1$ is mapped to z and $\perp_2$ is mapped to w. Inferring that $\perp \neq c$ is similar.

For satisfying a subgoal $x \neq y$ by mapping x and y to two different constants, we need to use at least two constants in our test. In general, we do not want our test to fail due to not using enough constants. Thus, we add fresh constants to C before applying the privacy test. How many new constants do we need to add? Note that any inequality can be satisfied by mapping one of the variables to a new constant. Also note that each time a dependency is being applied in a chase step, it is being done independently of inequalities in other dependencies. Therefore, it is sufficient to add m new constants to C, where m is the maximal number of inequalities in a single dependency of Γ, or in Q_s.

The following example illustrates our approach.

EXAMPLE 3.17. *Suppose that a company wants to conceal the names of employees that work in two different departments. The secret data is defined by the following query.*

```
Qs  SELECT P.Name
    FROM Participates P, Participates Q
    WHERE P.Name = Q.Name and
          P.DeptName<>Q.DeptName
```

Now, suppose that in an open setting we have the view

```
V1  CREATE VIEW V1 AS
    SELECT DISTINCT Name, DeptName
    FROM Participates
```

In this case, Σ_{vp} will include the dependency

Σ_{vp} :$V_1(n, d) \rightarrow \mathtt{Participates}(n, d, p)$

Using a single constant c in the instance of V_1 will not allow satisfying the constraint `P.DeptName<>Q.DeptName` *in the query. However, by using two constants c_1 and c_2 such that $c_1 \neq c_2$ we can create an instance of V_1 that consists of the two tuples (c_1, c_1) and (c_1, c_2). A chase over this instance produces a universal solution with the tuples $(c_1, c_1, \perp_1)$ and $(c_1, c_2, \perp_2)$ in* `Participates`*. Now, evaluating Q_s over this solution provides the certain answer (c_1), hence, the setting is deemed not private.*

As another example, suppose we have the same secret query Q_s and a view V_2 that reveals pairs of employees from two different departments

```
V2  CREATE VIEW V2 AS
    SELECT DISTINCT P.Name, Q.Name
    FROM Participates P, Participates Q
    WHERE P.DeptName<>Q.DeptName
```

In this case, when we apply a chase over an instance of V_2 that has a single tuple (n, n), we receive a universal solution that contains two tuples $(n, \perp_d, \perp_p)$ and $(n, \perp_{d'}, \perp_{p'})$ with the piece of information that $\perp_d \neq \perp_{d'}$. The inequality $\perp_d \neq \perp_{d'}$ can be used to satisfy the inequality constraint in Q_s, thus, the evaluation of Q_s over this solution produces the certain answer (n), showing that the setting is not private .

3.3.2 Constraints on the Proprietary Schema

In general, the proprietary database is a full schema, potentially with constraints Σ_p representing key constraints and foreign-key constraints. Our algorithms for open settings make use of the chase to compute a universal solutions for I_C. In the presence of constraints on the proprietary schema, we can still use the chase to compute a solutions D_c for $(\Gamma, \Sigma_{vp}, \Sigma_p)$, given I_C, providing that the dependencies Σ_p are weakly-acyclic [**?**]. Furthermore, this computation is still efficient. We claim that our privacy test for open settings is still a sufficient test for a setting to be private, but it may no longer be necessary. In other words, if the test determines a setting is private, then the setting is private. But the test may falsely determine a setting to not be private even when it is private. This may happen because our test is for *all I*, not just all I that could be produced by some proprietary instance $D \models \Sigma_p$.

EXAMPLE 3.18. *Consider the following simple setting where the proprietary schema has a single relation $R(x, y)$ with key $X \rightarrow Y$. Assume there is a single dependency $V(x, y) \rightarrow R(x, y), R(c, c)$ and the secret query is $Q_s(y) \vdash R(c, y), y \neq c$, where c is some constant. If we ignore the key constraint $X \rightarrow Y$, then our algorithm will use an instance of V containing a tuple $R(c, c')$ for some $c' \neq c$ and will assert that (c') is a certain answer. However, chasing the instance by the dependency on V produces two tuples $R(c, c')$ and $R(c, c)$ that violate the key constraint. Having the key constraint, for each tuple of R either $x \neq c$ or $y = c$, so the body of Q_s cannot be satisfied and the setting is private.*

Extending our algorithms, for open settings, to deal with constraints Σ_p on the proprietary schema requires adding a step of chasing the generated proprietary instance by Σ_p—after the chase by Σ_{vp} and before evaluating Q_s.

4. DESIGNING PRIVATE SETTINGS

In this section, we introduce some basic edits to help convert a non-private data-publishing setting into a private setting. In general, there are two approaches to modify a setting for making it private. The first approach that may only be applied to closed settings is to publish only part of the data and refer to these settings as open. Example 1.2 illustrated that. The second approach is to modify the view definition. This has been shown in Example 1.3. We consider a setting that does not provide privacy due to a set of tuples created by chasing a view instance I, where this set of tuples satisfies the secret query Q_s, as having a **Type II** form of privacy violation. Notice that such a setting is not private whether it is closed or open. Another form of privacy violation arises due explicitly to the setting being closed. A setting may not be private if it is closed and the dependencies Σ_{pv} lead us to replace nulls in chased solutions with constants using a homomorphism application (in a form of solution-aware-chase required in closed settings). We say this latter type of setting has a **Type I** privacy violation. Such settings could be made private if we make them open (and if they have no Type II violations), but we consider less alternative ways of editing the setting. We introduce two types of edits to modify settings that exhibit these two types of privacy violations respectively.

4.1 Type I Edits

During our privacy testing for closed settings Γ, for each published instance I, we can construct a directed graph where a node represents a tuple created or modified during the testing and where an edge from node $\bar{t}_1$ to node $\bar{t}_2$ with a set of labels $\{L_1, L_2, \ldots\}$ means that for each of its labels L_j, (1) if L_j is of the form $S_i : d_{i_1}, d_{i_2}, \ldots$, then $\bar{t}_1$ is required to generate $\bar{t}_2$ in the chase step S_i by the set of dependencies $\{d_{i_1}, d_{i_2}, \ldots\}$; (2) if L_j is of the form $S_i : d_{i_1} : \perp_{j_1} = c_{k_1}, d_{i_2} : \perp_{j_2} = c_{k_2}, \ldots$, then in the chase step S_i, $\bar{t}_1$ has its labeled nulls $\perp_{j_1}, \perp_{j_2}, \ldots$ replaced by constants $c_{k_1}, c_{k_2}, \ldots$ because of some homomorphism application (see Section 3.2 for details) related to the dependencies $d_{i_1}, d_{i_2}, \ldots$, respectively; moreover, these labeled nulls appear in $\bar{t}_2$ and need to replaced by their corresponding constants as well. If a labeled null appears in $\bar{t}_1$ only and it is replaced by a constant using a homomorphism application, then we use a loop on $\bar{t}_1$ to represent it, *i.e.*, $\bar{t}_2 = \bar{t}_1$. We call such a graph $G_{\Gamma,I}$ the *tuple-generation* graph for Γ given a published instance I.[1]

Suppose a given closed setting Γ is not private. Then, $Q_s(D_c)$ contains a constant tuple $\bar{t}$, where D_c is the universal solution created in our Algorithm 3.6. Note that in D_c, some tuples may have at least one of their labeled nulls replaced by a constant by some homomorphism and if all of these homomorphisms were not done, then $Q_s(D_c)$ does not contain $\bar{t}$ anymore. Next, we are going to show how to find these homomorphism applications using the tuple-generation graph $G_{\Gamma,I}$. Let $\mathcal{T}$ be the set of tuples in the body of Q_s during the evaluation of Q_s that creates $\bar{t}$. We can trace the creation of the tuples in $\mathcal{T}$ on $G_{\Gamma,I}$. Denote $\mathcal{T}_c$ the set of tuples generated during the creation. Note that $\mathcal{T}_c$ contains $\mathcal{T}$ and may contain many other tuples. We can identify the subset $\mathcal{T}_b$ of tuples in $\mathcal{T}_c$, where each tuple in $\mathcal{T}_b$ is generated by chasing the target-to-source dependencies Σ_{pv} and has one of its labeled nulls replaced by a constant because of a homomorphism application with some tuple $\bar{t}'$, where $\bar{t}'$ is the valuation of the variables $\bar{x} \cap \bar{y}$ of some source-to-target dependency $d : Q_{\mathbf{V}}(\bar{x}) \rightarrow Q_{\mathbf{P}}(\bar{y})$ in Σ_{vp} over the published instance I in our algorithms. We call $\mathcal{T}_b$ the set of *inverse chasing* tuples w.r.t. $\bar{t}$. Note that $\mathcal{T}_b$ is contained in $I^b_{D_c}$ in our Algorithm 3.6. We say a subset $\mathcal{T}_e$ of the tuples in $\mathcal{T}_b$ is *critical* w.r.t. $\bar{t}$ if the following happens: (1) if every element in $\mathcal{T}_e$ fails at least one of its corresponding homomorphism in the creation, then $\bar{t}$ is not an answer to Q_s. (2) if a critical set of tuples can be identified from $\mathcal{T}_b$ by undoing the homomorphism application, continuing the chase, and evaluating Q_s on the new solution created by the chase to see if $\bar{t}$ is an answer. Note that such homomorphism applications for $\mathcal{T}_e$ are called the *critical* for Γ w.r.t. $\bar{t}$. The *reduced critical set*

[1]Note that this graph is closely related to the *route* notion of [**?**].

of tuples w.r.t. $\bar{t}$ is a critical set of tuples w.r.t. $\bar{t}$ which will become non-critical if any tuple in the set is deleted. Their corresponding set of homomorphisms is called the *reduced critical set of homomorphisms* w.r.t. $\bar{t}$. Hence, for each constant answer $\bar{t}$ to Q_s, we need to make sure that for the reduced critical sets of tuples $\mathcal{T}_e$ in $\mathcal{T}_b$ w.r.t. $\bar{t}$, the published instances of the modified setting contain enough diversified tuples to avoid homomorphism application. We further describe our intuition through an example.

EXAMPLE 4.1. *Recall that Example 1.2 show that many non-private closed settings become private when the setting is modified to be an open one. For these closed settings, each constant tuple $\bar{t}$ of $Q_s(D_c)$ has at least one non-empty critical set of tuples. When the setting is considered as open, no homomorphism application will happen; thus, the critical sets of tuples do not exist; hence, $Q_s(D_c)$ contains no constant tuples and the setting becomes private. Furthermore, these settings do not leak information for most instances and only reveal secure information for a few instances that are either too small or contain data with little variance in attribute values.*

To see whether the setting in Example 1.1 requires modifications to make it private, we first test the privacy of the setting. Without loss of generality, we test the privacy of the setting using the following instance $I(V_1) = \{(c,c,c),(m,c,c)\}$ and $I(V_2) = \{(c,c,c),(m,c,c)\}$ (recall that we add a second constant $m \neq c$ due to the inequality in the definition of V_1). Using V_1 we create in D_c tuples $(c,c,\perp_1)$, $(\perp_2,c,c)$, $(m,c,\perp_3)$, $(\perp_4,c,c)$ in relation `Participates`*. Using V_2 we create tuples $(c,c,\perp_5)$ and $(m,c,\perp_6)$ in* `Participates`*, along with (c,c) in relation* `Department`*. The next step is to apply the tgds backwards for generating $I^b_{D_c}$. In $I^b_{D_c}$, relation V_1 contains tuples (c,c,c), $(c,c,\perp_3)$, $(\perp_3,c,c)$, $(\perp_4,c,\perp_1)$, $(\perp_4,c,\perp_3)$, (m,c,c), $(m,c,\perp_1)$, $(m,c,\perp_1)$, $(\perp_2,c,c)$, $(\perp_2,c,\perp_3)$, and $(\perp_2,c,\perp_1)$; relation V_2 contains tuples (c,c,c), (m,c,c), $(\perp_2,c,c)$, $(\perp_4,c,c)$. We next replace the label nulls by constants as $\perp_4 = c$, $\perp_2 = m$, $\perp_1 = c$ and $\perp_3 = c$. After the replacement, in D_c there exist* `Participates(c,c,c)` *and* `Participates(m,c,c)`*. Hence, (c,c,c) and (m,c,c) are the certain answers to Q_s given I_C. Now, we first consider the certain answer (c,c,c). We trace back and find that the evaluation of the query body contains a tuple* `Participates`*(c,c,c), which can be created by three types of critical homomorphisms. One type is on the possible values of* `V1.Name` *in the tuple $V_1(\perp_2,c,c)$ for some labeled null $\perp_2$, the second type is on the possible values of* `V1.ProjName` *in the tuple $V_1(c,c,\perp_1)$ for some labeled null $\perp_1$, and the third type is on the possible values of* `V1.Name` *and* `V1.ProjName` *in the tuple $V_1(\perp_4,c,\perp_3)$ for some labeled nulls $\perp_4$ and $\perp_3$. Moreover, the labeled nulls $\perp_2$ and $\perp_4$ must satisfy the inequality condition* `S1.Name <> S2.Name`*, since* `S1.Name` *is evaluated to be $\perp_1$ or $\perp_3$ in the two kinds of homomorphisms. Therefore, the published instance of V_1 should contain at least one more distinct value of* `V1.Name` *and one more distinct value of* `V1.ProjName`*. In particular, the value of* `V1.Name` *is different from both values c and m in I_C. Hence, for each tuple in the published instance of V_1, there should be two tuples with unique distinct values of* `V1.Name` *and one of the two has one distinct value of* `V1.ProjName`*. Similarly, we analyze the other certain answer (m,c,c) and reach the same conclusion. Our intuition motivates the following algorithm.*

ALGORITHM 4.2. *An algorithm to modify a non-private closed setting (Γ, Q_s) so that critical sets of homomorphism applications will not occur.*

- *Test the privacy of (Γ, Q_s) and construct the tuple-generation graph $G_{\Gamma,I}$ for Γ w.r.t. each tested published instance I.*
- *If (Γ, Q_s) is not private, for each of the certain answers $\{\bar{t}\}$ to Q_s generated in Step 1, do the following.*
  - *Let $\mathcal{T}$ be the set of tuples in the body of Q_s during the evaluation of Q_s that creates $\bar{t}$. Let I be the published instance during the test s.t. $\bar{t} \in certain(Q_s, I)$.*
  - *Trace the creation of the tuples in $\mathcal{T}$ in $G_{\Gamma,I}$. Denote $\mathcal{T}_c$ the set of tuples generated during the creation. Identify the $\mathcal{T}_b$ of backward chasing tuples in $\mathcal{T}_c$.*
  - *Identify the critical subsets $\{\mathcal{T}_e\}$ of tuples within $\mathcal{T}_b$ and record their corresponding critical homomorphisms in the form of $S_i : d_{l_1} : n_{i_1} = c_{k_1} : d_{l_2} : n_{i_2} = c_{k_2} : \cdots, S_j : d_{l'_1} : n_{i'_1} = c_{k'_1} : d_{l'_2} : n_{i'_2} = c_{k'_2} : \cdots, \ldots$ $(i < j)$,* i.e., *with the same semantic meaning as edge labels.*
  - *Find the reduced critical subset $\mathcal{T}_e$ of tuples w.r.t. $\bar{t}$, for each critical homomorphisms for $\mathcal{T}_e$, and for each expression in the form of $d_{l_1} : n_{i_1} = c_{k_1}$, do the following.*
    - *Suppose d_{l_1} is represented as $Q_{\mathbf{V}}(\bar{x}) \rightarrow Q_{\mathbf{P}}(\bar{y})$.*
    - *Add to the side of d_{l_1} that is defined over $\mathbf{P}$ (*i.e.*, the right-hand-side) a duplicated $Q_{\mathbf{P}}(\bar{y}')$ with distinct new variables $\bar{y}'$.*
    - *For the variables in $\bar{y}$ that are evaluated to n_{i_1} in the chase step S_i (note that these variables also appear in $\bar{x}$), add inequalities between them and their corresponding variables in $\bar{y}'$. Furthermore, if any of these variables y_i in $\bar{y}$ appears in an inequality term in $Q_{\mathbf{P}}(\bar{y})$, say $y_i \neq y_0$, add a copy of the term where y_i is replaced by y'_i,* i.e., *the corresponding variable of y_i in $\bar{y}'$.*
    - *For other variables in both $\bar{x}$ and $\bar{y}$, add equalities between each of them and its corresponding variable in $\bar{y}'$.*
    - *Update $Q_{\mathbf{P}}(\bar{y}) \rightarrow Q_{\mathbf{V}}(\bar{x})$ accordingly.*
- *Rewrite and simplify the modified dependencies.*

Let us revisit Example 1.1 and run the above algorithm to get a private setting.

EXAMPLE 4.3. *For the sake of clean and clear explanation, we only consider one kind of homomorphism for the tuple $V_1(\perp_3,c,\perp_4)$ since it covers the other two homomorphism. We first duplicate the view body with distinct variables and add it to V_1. Then, since* `V1.Name` *is evaluated to be a labeled null, which is replaced by a constant later, we add an inequality term so that the newly added variable* `S3.Name` *for the attribute* `Name` *of the relation* `Participates` *is different from* `S1.Name`*. Similarly, we add one inequality term so that the newly added variable* `S3.ProjName` *for the attribute* `ProjName` *of the relation* `Participates` *is different from* `S1. ProjName`*. Moreover, since* `S1.Name <> S2.Name` *exists in V_1, we need to add* `S3.Name <> S2.Name`*. Finally, we add an equality term so that the newly added variable* `S3.DeptName` *for the attribute* `DeptName` *of the relation* `Participates` *is the same as* `S1.DeptName` *as this is the variable that is evaluated to c in the homomorphism. The following is the modified view of V_1.*

```
CREATE VIEW V1 as
SELECT DISTINCT S1.Name, S1.DeptName
     S2.ProjName
FROM Participates S1, Participates S2,
     Participates S3
WHERE S1.DeptName = S2.DeptName and
      S1.DeptName = S3.DeptName and
      S1.Name <> S2.Name and
      S1.Name <> S3.Name and
```

```
S2.Name <> S3.Name and
S1.ProjName <> S3.ProjName
```

Note that, in our testing algorithm, all constant tuples in $Q_s(D_c)$ have a non-empty critical set of tuples; hence, the modified setting prevents the creation of these critical set of tuples and the modified setting becomes private.

THEOREM 4.4. *Given a non-private closed setting Γ, Algorithm 4.2 generates a setting Γ' s.t. for any proprietary instance D, the published instance I w.r.t. Γ contains the published instance I_c' w.r.t. Γ'. Moreover, let D_c' be a universal solution for Γ w.r.t. I_c', there is no critical set of tuples for any tuple in $Q_s(D_c')$.*

4.2 Type II Edits

Note that Type I edits apply when a data-publishing setting is not private only when it is considered as a closed setting. When the data-publishing setting is not private whether it is considered as an open setting or a closed setting, there exists a constant tuple $\bar{t}$ of $Q_s(D_c)$ s.t. the set of critical tuples in D_c w.r.t. $\bar{t}$ is empty. In other words, the non-private closed setting is still not private when the setting is modified to be an open one. In this case, we introduce two basic edits to turn a non-private data-publishing setting private: (1) *the addition of projections*, denoted as P^+, is a random deletion of a free variable in a dependency; (2) *the deletion of equality terms*, denoted as E^-, is a random replacement of a repeated variable by a distinct new variable in a dependency. While the intuition for P^+ is clear, the intuition behind E^- is illustrated as follows.

EXAMPLE 4.5. *Consider a data-publishing setting Γ where $Q_s(x) \vdash \exists\, y, z\ P(x,z), P(z,y)$ where Σ_{vp} is $V(x,y) \to P(x,z), P(z,y)$. Clearly, Γ is not private. However, if we delete the equality term in V (*i.e.*, by replacing one of the two appearances of z by z'), then the modified Γ is private. Note that the equality term in Q_s can not be satisfied on the universal solution produced by our privacy test.*

Between the two edits, we prefer E^- since the resulting view instance will contain the original view instance; however, the connection between information as specified by the deleted equality terms is hidden. Next, we will propose a *Type II* heuristic to apply these two basic edits and gradually find an optimal solution (*i.e.*, a private setting). In Section 5, we will show that our Type II heuristic is feasible in practice as the number of required edits to achieve privacy is considerably low.

Before providing the details of our Type II heuristic, we first introduce some notations. Given a dependency $\lambda \in \Sigma_{vp}$, let $E(\lambda) = \{E_0^-(\lambda), \ldots, E_m^-(\lambda)\}$ and $P(\lambda) = \{P_0^+(\lambda), \ldots, P_k^+(\lambda)\}$ be the set of all permitted "deletions of equality terms" and "additions of projections" over λ, respectively. Let $M(\lambda) = \{s | s \in 2^{E(\lambda)} \cup 2^{P(\lambda)}\}$ be the set of possible combinations of edits over the given dependency λ. Let $iE^-(\lambda) = \{s \in 2^{E(\lambda)} |\, |s| = i\}$ denote the set of all possible subsets of $E(\lambda)$, which contains i E^-s. Let $iP^+(\lambda) = \{s \in M(\lambda) |\, |s| = i\}$ denote the set of all possible subsets of $P(\lambda)$, which contains i P^+s. When it is clear in the context, we simply denote $iE^-(\lambda)$ and $iP^+(\lambda)$ as iE^- and iP^+, respectively. Since we strictly prefer E^- to P^+ as previously mentioned, we define a partial order $<_M$ over $M(\lambda)$ as follows. Without loss of generality, suppose that given $A(\lambda)$, $B(\lambda)$ in $M(\lambda)$, if $A(\lambda) \in 2^{E(\lambda)}$ and $B(\lambda) \in 2^{P(\lambda)}$, then $A(\lambda) <_M B(\lambda)$. Furthermore, suppose both of $A(\lambda)$ and $B(\lambda)$ are either in $2^{P(\lambda)}$ or $2^{E(\lambda)}$, then we say $A(\lambda) <_M B(\lambda)$ if $|A(\lambda)| < |B(\lambda)|$ and $A(\lambda) \geq_M B(\lambda)$ otherwise. (This implies that our Type II heuristic is going to first apply all possible subsets of $E(\lambda)$ over λ before applying any P^+.) Unless specified, we assume that for any subset of $M(\lambda)$, there is always a partial order $<_M$. Therefore, when a Type II heuristic wants to decide which subset of edits to apply next, it will pick the smallest one according to $<_M$.

Our Type II heuristic is based on an exhaustive solution search over the space of all possible combinations of edits, *i.e.*, $M(\lambda)$. We start our search with the initial data-publishing setting and the search is halted when it has found a private data-publishing setting. Therefore, in the worst case it requires $2^{|E(\lambda)|} + 2^{|P(\lambda)|}$ operations. While its cost looks expensive, we show later, by our experiments, that the performance of our Type II heuristic is acceptable in practice. formal description of our Type II heuristic as follows. Finally, we would like to point out that our design tool will tell users where in the dependencies, the edits should be applied. Sometimes, multiple choices will be provided so that the users could pick the one that best suites their preferred semantics.

Summary We summarize our approach to modify a non-private setting to become a private one. We first test the setting using both the algorithms for open and closed settings, respectively. If the setting is not private under the algorithm for closed settings and is private under the algorithm for open settings, then we use Algorithm 4.2 to edit it. Otherwise, we use our Type II heuristics to edit the setting.

5. EXPERIMENTS

In this section, we present the results of our experiments. We first provide details about our experimental setup: we implemented our basic algorithms (both open and closed cases) from Section 3 and the Type II heuristic from Section 4 in PERL. We chose Perl DBI as the interface to the underlying database management system and MySQL was the choice for our DB system. All the experiments were run on a Linux-based PC desktop with a 2.13 GHZ processor and 224 MB RAM.

5.1 Run-Time Test

The purpose of this experiment is to test the run-time of our algorithms as the complexity of a given data-publishing setting grows with a fixed secret query. For the experiment, our query generator produces views in the form of

$$V(x_0, y_1, \ldots, y_m, x_{k+1}) \to P(x_0, x_1), \ldots, P(x_k, x_{k+1})$$

where each y_i is randomly chosen from $\{x_1, \ldots, x_k\}$. A sequence of data-publishing settings are constructed as follows: the secret query is fixed as $Q_s(x,y) = \{P(x,y)\}$ and a new Σ_{vp} is created from the previous Σ_{vp} by adding a view from our query generator with k and m incremented by one each time. The process starts with $k = m = 1$ and stops when $k = m = 60$. The number 60 was chosen as the upper-bound on k and m since the maximum number of joins allowed in MySQL is 61. Our algorithms are tested over this sequence of open and closed data-publishing settings. In Figure 2, we summarize the results of our experiments: our algorithms achieve a run-time that is linear in the size of the open settings and a run-time that is exponential in the size of the closed settings. This is predicted by the complexity results in Section 3. Furthermore, note that the graph clearly indicates the efficiency of our privacy tests as it takes less than a second to perform each test even over a setting consisting of 60 complex views.

Figure 2: Run-Time of our privacy tests for open/closed settings

5.2 Type II Test

In this set of experiments, we evaluate the performance of our Type II heuristic (from Section 3) in two different types of settings:

(1) a set of chain views with subgoals over multiple binary relations, (2) a set of chain views with subgoals over a single binary relation. Note that in (1), the views are generated in a similar fashion to those in our run-time test except that the relations are also randomly generated now. Denote s the number of generated relations. We generated 50000 views with $s = 10$, $k = 9$, and $m = 9$, and 50000 secret queries with $s = 10$, $k = 4$, and $m = 4$. Type II heuristic is tested over each pair of the 50000 views and secret queries. Out of them, around 1% are not private, on which our Type II heuristic is performed. We plot the results of our experiments in Figure 3, where the X-axis represents the type of edits (either P^+ or E^-). Below each operator (inside parenthesis), we indicate the number of Type II heuristic steps required to apply the corresponding operator so that the modified setting become private. The Y-axis represents the frequency at which privacy is achieved after the corresponding operator is applied. Furthermore, in Table 1, we report the average running time of our Type II heuristic before finding a private setting.

Figure 3: Type II Heuristic experiment on chain views over multiple relations

Search Stopped when it had applied the following edits	Average Run-Time (secs)
1E-	0.253
2E-	0.393
3E-	0.649
4E-	0.952
1P+	2.986
2P+	3.064
3P+	2.905

Table 1: Average run-time of Type II Heuristic operations on chain views over multiple binary relations

Figure 4: Type II Heuristic experiments on chain views over one single binary relation

For the second type of settings, 2000 views were generated with $k = 9$ and $m = 4$, and 2000 secret queries with $k = 9$ and $m = 2$. Type II heuristic is tested over each pair of the 2000 views and secret queries. We report, in Figure 4, the results of our experiments and, in Table 2, the average running of our Type II heuristic before finding a private setting.

In all the experiments, high appearances of E^-s of small sizes are observed. Intuitively, there are important joins in the chain that cause the leak of secret information. The performance is appealing since it does not take long for the Type II heuristic to identify such important joins. For the second type of settings, high appearances of P^+s are also observed. This is because when the views and secret queries are defined over a single relation, in some cases it might not be enough to perform join relaxations (*i.e.*, deletions). Instead, our Type II heuristic has to rely on projections to obscure the secret information. Finally, our experiments show that the rum-time cost of our Type II heuristic is acceptable in practice.

6. RELATED WORK

In data publishing, one popular way of measuring privacy disclosure is by using the notion of k-anonymity [?]. Yao *et al.* [?] consider a data-publishing setting consisting of a single relational table (which has an identifier attribute and a private attribute), and a set of select-project views to publish data. In their setting, k-anonymity is violated when the value of the identifier attribute in a

Stopped when it has applied	Average Run-Time (secs)
1E-	0.377
2E-	0.487
1P+	4.095

Table 2: Average run-time of Type II Heuristic operations on chain views over one single relation

tuple of the table can be determined to be among less than k possibilities based on the materialized (published) views together with the schema information of the table. Their privacy notion is data-dependent and can be applied to a limited class of data-publishing settings: (1) the secret query is a projection query (no selections or joins) and it returns a binary relation; (2) the proprietary instance only contains one single table; (3) all published views are defined by select-project queries.

There have been some other probability-based privacy notions. The privacy notion proposed by Miklau and Suciu [?] considers a set of exact views to reveal private information if the probability distribution of the answers to the secret query is changed for some view materializations [?, ?, ?]. Their results show that this is too strict of a notion as most any setting will reveal some information about the distribution of answers. In information-theory terminology, their privacy requires *perfect secrecy* [?], which is often too restrictive for practical purposes. Dalvi *et al.* [?] propose five types of privacy for boolean queries differentiated by the asymptotic value of the probability of information disclosure as the domain size grows to infinity. Although this approach could be applied to non-boolean queries, it has to consider all possible answers to the query, which could be of exponential size in the domain and the query. The problem of testing relative privacy [?, ?] is to check whether adding new views in addition to the existing ones would change the probability distribution of the answers to a secret query. In addition to the privacy guarantee we discussed in the introduction, Nash and Deutsch [?] also define two relative privacy guarantees.

7. CONCLUSION

We presented a privacy notion for data-publishing settings where the secrets are defined by a conjunctive query and the published data is described by a set of constraints over the published schema and the proprietary schema. We proposed algorithms to verify that no certain answers can be inferred from the published data for different types of settings. Our tests are data-independent, meaning that a private setting will not reveal secret data for any possible database instance. Our tests guarantee that secret information indeed cannot be inferred with certainty from any published data. We also considered the problem helping a user to design private data-publishing settings. Given a setting that is not private, we provide constructive ways of changing the setting minimally to produce a private setting. Our experiments show that both our algorithms and design heuristic run efficiently in practice.

In this paper, we considered a privacy guarantee that ensures a user cannot deduce with certainty any answer to a secret query. A complementary notion of privacy to the one we have discussed is to require that a user not learn anything about *possible answers* to the secret query Q_s. Let t be in the possible answer to Q_s, that is, there exists some database D such that $t \in Q_s(D)$. Then, given I, a user should not be able to determine (again with certainty) that t is not an answer to Q_s over the current (secret) proprietary database. Under this notion of privacy, the published data should not decrease the set of possible answers to Q_s. We are currently investigating how to test this form of privacy for data-publishing settings.