Don't Be a Tattle-Tale: Preventing Leakages through Data Dependencies on Access Control Protected Data

Primal Pappachan
UC Irvine, USA
primal@uci.edu

Shufan Zhang
University of Waterloo, Canada
shufan.zhang@uwaterloo.ca

Xi He
University of Waterloo, Canada
xi.he@uwaterloo.ca

Sharad Mehrotra
UC Irvine, USA
sharad@ics.uci.edu

ABSTRACT

We study the problem of answering queries when (part of) the data may be sensitive and should not be leaked to the querier. Simply restricting the computation to non-sensitive part of the data may leak sensitive data through inference based on data dependencies. While inference control from data dependencies during query processing has been studied in the literature, existing solution either detect and deny queries causing leakage, or use a weak security model that only protects against exact reconstruction of the sensitive data. In this paper, we adopt a stronger security model based on *full deniability* that prevents any information about sensitive data to be inferred from query answers. We identify conditions under which full deniability can be achieved and develop an efficient algorithm that minimally hides non-sensitive cells during query processing to achieve full deniability. We experimentally show that our approach is practical and scales to increasing proportion of sensitive data, as well as, to increasing database size.

PVLDB Reference Format:
Primal Pappachan, Shufan Zhang, Xi He, and Sharad Mehrotra. Don't Be a Tattle-Tale: Preventing Leakages through Data Dependencies on Access Control Protected Data . PVLDB, 15(11): 2437 - 2449, 2022.
doi:10.14778/3551793.3551805

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zshufan/Tattle-Tale.

1 INTRODUCTION

Organizations today collect data about individuals that could be used to infer their habits, religious affiliations, and health status — properties that we typically consider as sensitive. New regulations, such as the European General Data Protection Regulation (GDPR) [33], the California Online Privacy Protection Act (CalOPPA) [34], and the Consumer Privacy Act (CCPA) [22], have made it mandatory for organizations to provide appropriate mechanisms to enable users control over their data, i.e., (how| why| for how long) their data is collected, stored, shared, or analyzed. *Fine Grained Access Control Policies (FGAC)* supported by databases is an integral technology component to implement such user control. FGAC policies enable data owners/administrators to specify which data (i.e., tables, columns, rows, and cells) can/cannot be accessed by which querier (individuals posing queries on the database) and is, hence, sensitive [15] for that querier. Traditionally, Database Management Systems (DBMS) implement FGAC by filtering away data that is sensitive for a querier and computing the query on only the non-sensitive part of the data. Such a strategy is implemented using either query rewriting [1, 26] or view-based mechanisms [30]. It is well recognized that restricting query computation to only non-sensitive data may not prevent the querier from inferring sensitive data based on semantics inherent in the data [4, 14]. For instance, the querier may exploit knowledge of data dependencies to infer values of sensitive data as illustrated in the example below.

Example 1. Consider an Employees table (Figure 1) and a FGAC policy by a user *Bobby* to hide his salary per hour (*SalPerHr*) from all the queries by other users. If the semantics of the data dictate that any two employees who are faculty should have the same *SalPerHr*, then hiding *SalPerHr* of *Bobby* would not prevent its leakage from a querier who has access to *Carrie*'s *SalPerHr.* □

In general, leakage may occur from direct/indirect inferences due to different type of data dependencies, such as conditional functional dependencies (CFD) [13], denial constraints [5], aggregation constraints [31], and/or *functional constraints* that exist when a dependent data values are derived/computed using other data values as shown below.

Example 2. Consider the Employee and Wage tables shown in Table 1. Let *Danny* specify FGAC policies to hide his *SalPerHour* in Employee Table and *Salary* in Wage Table. Suppose there exists a constraint that employees with role *Staff* cannot have a higher salary per hour than a faculty in the state of California. Using *Bobby*'s salary per hour that is leaked in Example 1, the new constraint about the staff salary, and the functional constraint between that *Salary* and the fields function of *WorkHrs* and *SalPerHr*, information about the salary and the salary per hour of *Danny* will be leaked even though they are sensitive. □

To gain insight into the extent to which leakage could occur due to data semantics, we conducted a simple experiment on a synthetic dataset [2, 5] that contains the address and tax information of individuals. The tax data set consists of 14 attributes and has associated with it 10 data dependencies, an example of which is

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551805

Eid	EName	Zip	State	Role	WorkHrs	SalPerHr
c_{1} 34	c_{2} Alice Land	c_{3} 45678	c_{4} AZ	c_{5} Student	c_{6} 20	c_{7} 40
c_{8} 56	c_{9} Bobby Hill	c_{10} 54231	c_{11} CA	c_{12} Faculty	c_{13} 40	c_{14} 200
c_{15} 78	c_{16} Carrie Sea	c_{17} 53567	c_{18} CA	c_{19} Faculty	c_{20} 40	c_{21} 200
c_{22} 12	c_{23} Danny Des	c_{24} 54231	c_{25} CA	c_{26} Staff	c_{27} 30	c_{28} 70

Eid	DeptName	Salary
c_{29} 34	c_{30} CS	c_{31} 800
c_{32} 56	c_{33} EE	c_{34} 8000
c_{35} 78	c_{36} CS	c_{37} 8000
c_{38} 12	c_{39} BIO	c_{40} 2100

Figure 1: Employee and Wages Table

a denial constraint "if two persons live in the same state, the one earning a lower salary has a lower tax rate". An adversary can use the above dependency to infer knowledge about the sensitive cells. Suppose the *salary* attribute of an individual is sensitive and therefore hidden. If the disclosed data contains information about another individual who lives in the same state and has a lower tax rate, an adversary can infer the upper bound of this individual's salary using the dependency. To demonstrate this leakage, we considered an attribute with large number of data dependencies defined on them (e.g., state) to be sensitive, and thus, replaced its values by *NULL*. We then used the state-of-the-art data cleaning software, Holoclean [29], as a real-world attacker to reconstruct the *NULL* values associated with the sensitive cells. Holoclean was able to reconstruct the actual values of the state 100% of the time highlighting the importance of preventing leakage through data dependencies on access control protected data.

Prior literature has studied the challenge of controlling inferences about sensitive data using data dependencies and called it the "inference control problem". [14]. Existing techniques used to protect against inferences can be categorized based on when the leakage prevention is applied [3]. In the first category, inference channels between sensitive and non-sensitive attributes are detected and controlled at the time of database design [8, 16]. A database designer uses methods in this category to detect and prevent inferences by upgrading classification of inferred attributes. However, they result in poor data availability if a significant number of attributes are marked as sensitive to prevent leakages. The second category of works include detection and control at the time of query answering. Works such as [3, 32] determine if answers to the query could result in inferences about sensitive data using data dependencies, and reject the query if such an inference is detected. Such query control approaches can lead to rejection of many queries when there are non-trivial number of sensitive cells and background knowledge. Another limitation of the prior work is the weak security model used in determining how to process queries. All prior work on inference control considers a query answer to leak sensitive data if the answer can be used to reconstruct the exact value of a sensitive object. Leakages that do not reveal the exact value but, perhaps, limit the values a sensitive object may take are not considered as leakage. For instance, in Example 2 above, since the constraints do not reveal *Danny*'s exact salary but only that it is below $200 per hour, prior works will not consider it to be a leakage even though querier/adversary could eliminate a significant number of possible domain values based on the data constraints. As we explain in detail in Section 8, the existing solutions to the inference control cannot be easily generalized to prevent such leakages.

In this paper, we study the problem of answering user queries under a new, much stronger model of security — viz., *full deniability*. Under full deniability, any new knowledge learned about the sensitive cell through data dependencies is considered as leakage. Thus, eliminating a domain value as a possible value a attribute / cell can take violates full-deniability. One can, of course, naively, achieve full deniability by hiding the entire database. Instead, our goal is to identify the minimal additional non-sensitive cells that must be hidden so as to acheive full deniability. In addition, we require the algorithm that identifies data to hide in order to achieve full deniability to be efficient and scalable to both large data sets and to large number of constraints.

We study our approach to ensuring full deniability during query processing under two classes of data dependencies [1]:

- *Denial Constraints (DCs)*: that are general forms of data dependencies expressed using universally quantified first order logic. They can express commonly used types of constraints such as functional dependencies (FD) and conditional functional dependencies (CFD) and are more expressive than both
- *Function-based Constraints (FCs)*: that establish relationships between input data and the output data it is derived from, using functions. Such constraints arise naturally when databases stored materialized aggregates or when data sensor data, collected over time (e.g., from sensors), is enriched (using appropriate machine learning tools) to higher level observations.

To achieve full deniability, we first develop a method for *Inference Detection*, that detects, for each sensitive cell, the non-sensitive cells that could result in violation of full deniability. The candidate cells identified by Inference Detection are passed to the second function, *Inference Protection* that minimally selects the non-sensitive data to hide to prevent leakages. Our technique is geared towards maximizing utility when preventing inferences for large number of sensitive cells and their dependencies. After hiding additional cells, Inference Detection is invoked repeatedly to detect any indirect leakages on the sensitive cells through the new set of hidden cells and their associated dependencies. These methods are invoked cyclically until no further leakages are detected either on the sensitive cells or any additional cells hidden by Inference Protection. Using these two

[1]Other data dependencies such as Join dependencies (JD) and Multivalued dependencies are not common in a clean, normalized database and therefore not interesting to our problem setting.

different methods, we are able to achieve the security, utility, and performance objectives of our solution.

The main contributions in our paper are:

- A security model, entitled *Full deniability* to protect against leakage of sensitive data due to data semantics in the form Denial Constraints and Function-based Constraints.
- Identification of conditions under which full deniability can be achieved and efficient algorithms for inference detection and protection to achieve full deniability while only minimally hiding additional non-sensitive data.
- A prototype middleware that works alongside DBMS to ensure full deniability given set of dependencies and policies.
- Experimental results on two different data sets show that our approach is efficient and only minimally hides non-sensitive cells while achieving full deniability.

Paper Organization. We introduce the notations used in the paper and describe access control policies and data dependencies in Section 2. In Section 3, we present the security model — full deniability — proposed in this work. In Section 4, we describe how the leakage of sensitive data occurs through dependencies and introduce function-based constraints. We present in Section 5, the algorithms for inference detection and protection along with optimizations to improve utility. In Section 7, we present results from an end-to-end evaluation of our approach with two different data sets and different baselines. In Section 8 we go over the related work and we conclude the work by summarizing our contributions, and possible future extensions in Section 9.

Extended version. Due to space limitations, we could not describe several details, which can be found in the extended version [27], including: proofs of theorems, details of extensions for improving utility, and scalability experiments on a much larger dataset.

2 PRELIMINARIES

Table 1 summarizes the common notations we use. Consider a database instance $\mathbb{D}$ consisting of a set of **relations** $\mathcal{R}$. Each relation $\mathbb{R} \in \mathcal{R} = \{A_1, A_2, \ldots, A_n\}$ where A_j is an attribute in the relation. Given an attribute A_j in a relation $\mathbb{R}$ we use $Dom(A_j)$ to denote the domain of the attribute and $|Dom(A_j)|$ to denote the number of unique values in the domain (i.e. the domain size)[2]. A relation contains a number of indexed **tuples**, t_i represents the i^{th} tuple in the relation $\mathbb{R}$, and $t_i[A_j]$ refers to the j^{th} attribute of this tuple.

We will use the **cell-based** representation of a relation to simplify notation when discussing the fine-grained access control policies and data dependencies. Figure 1 shows two tables, the *Employee* table with cells c_1 to c_{28} and the *Wages* table with cells c_{29} to c_{40}. Note that in the cell-based notation each table, row, column corresponds to a set of cells. For instance, the second tuple/row of *Wages* table is the set of cells $\{c_{32}, c_{33}, c_{34}\}$ and the column for attribute *Zip* in the *Employee* table is the set $\{c_3, c_{10}, c_{17}, c_{24}\}$. Each cell has an associated value. For instance, the value of cell c_{11} is "CA".

[2]We say the domain size in the context of an attribute with discrete domain values and for continuous attributes we discretize their domain values into a number of non-overlapping bins.

Table 1: Notation Cheatsheet

Notation	*Definition*
$\mathbb{D}$	A database instance
c	A cell in a database relation
$\mathbb{C}, \mathbb{C}^H$	Set of cells, hidden cells
δ, Δ	A schema level data dependency / set
$\tilde{\delta}, S_\Delta$	An instantiated data dependency / set
$Cells(\tilde{\delta})$	Cells involved in $\tilde{\delta}$
$Preds(\tilde{\delta})$	The set of predicates associated with a DC
$Preds(\tilde{\delta}, c)$	The set of predicates in $\tilde{\delta}$ that involves the cell c
$Preds(\tilde{\delta} \backslash c)$	The set of predicates in $\tilde{\delta}$ without the cell c
$\mathbb{V}(\mathbb{C})$	Set of value assignments for cells in $\mathbb{C}$
$\mathbb{I}(c \mid \mathbb{V}, \tilde{\delta})$	Inference function for the cell c

2.1 Access Control Policies

Data sharing is controlled using access control policies, or simply policies. We classify users U as data owners, who set the access control policies, and as queriers, who pose queries on the data. Ownership of data is specified at tuple level and a data owner of a tuple may specify policies marking one or more cells (c_i) in the tuple t as sensitive against queries by other users. When another user queries the database, the returned data has to be policy compliant (i.e., policies relevant to the user are applied to the query results). In this section, we present a simplified model of policy that is sufficient to describe the approach and refer the reader to the extended version [27] for the complete policy model.

A policy P is represented using the 3-tuple $\{\mathbb{R}, \sigma, \Phi\}$ where $\mathbb{R}$ is the relation to which P applies, σ is the set of selection conditions that select the set of tuples in $\mathbb{R}$ to which P applies, and finally Φ is the projection conditions that identifies the set of columns to which the policy is applied. The application of an policy is done by a function over the database that returns *NULL* for a cell if it is disallowed by the policy or the original cell value if it is allowed.

Definition 2.1 (Sensitive Cell). Given a policy P, we say that a cell c is sensitive to a user U if after applying P c is replaced with *NULL*. The set of cells sensitive to the user U is denoted by $\mathbb{C}^S_U$ or simply $\mathbb{C}^S$ when the user is clear from the context.

2.2 Data Dependencies

Semantics of data is expressed in the form of *data dependencies*, that restrict the set of possible values a cell can take based on the values of other cells in database. Several types of data dependencies have been studied in the literature such as foreign keys, functional dependencies (FDs), and conditional functional dependencies (CFDs), etc. We consider two types of dependencies as follows:

Denial Constraints (DC), is a first-order formula of the form $\forall\, t_i, t_j, \ldots \in \mathbb{D}, \delta : \neg(Pred_1 \wedge Pred_2 \wedge \ldots \wedge Pred_N)$ where $Pred_i$ is the ith predicate in the form of $t_x[A_j]\theta t_y[A_k]$ or $t_x[A_j]\theta const$ with $x, y \in \{i, j, \ldots\}$, $A_j, A_k \in R$, $const$ is a constant, and $\theta \in \{=, >, <, \neq, \geq, \leq\}$. DCs are quite general — they can model dependencies such as FDs & CFDs and are flexible enough to model much more complex relationships among cells. Data dependencies in the form of DCs has been used in recent prior literature for data cleaning [6, 17], query optimization [19], and secure databases [3, 10]. Moreover, systems, such as [5], have been designed to automatically discover DCs in a given database. This is the type of DCs considered throughout the paper.

Function-based Constraints (FCs) capture the relationships between derived data and its inputs. As described in Example 2, the *Salary* in the *Wages* table (see Table 1) is a attribute derived using *WorkHrs* and *SalPerHr* i.e., *Salary* := *fn(WorkHrs, SalPerHr)*. In general, given a function fn with $r_1, r_2, \dots, r_n$ as the input values and s_i as the derived or output value, the FC can be represented by $fn(r_1, r_2, \dots, r_n) = s_i$.

3 FULL DENIABILITY

In this section, we discuss the assumptions in our setting and present the concept of *view* of a database for the querier and formalize an *inference function* with respect to the view and data dependencies. We formally define our security model — *Full Deniability* — based on the inference function and use it to determine the leakage on sensitive cells.

3.1 Assumptions

We will assume that tuples (and cells in tuples) are independently distributed except for explicitly specified dependencies that are either learnt automatically or specified by the expert. The database instance is assumed to satisfy the data dependencies. The querier, who is the adversary in our setting, is assumed to know the dependencies and can use them to infer the sensitive data values. This assumption leads to a stronger adversary than the standard adversary considered by many algorithms for differential privacy or traditional privacy notions like k-anonymity or access controls, which assumes the adversary knows no tuple correlations (or tuples are independent). A querier is free to run multiple queries and can attempt to make inferences about sensitive data based on the results of those queries. Two queriers, however, do not collude (i.e., share answers to the queries). We note that if such collusions were to be allowed, it would void the purpose of having different access control policies for different users.

As queriers are service providers or third parties who are interested in obtaining user data to provide a service and therefore we assume that queriers and data owners do not overlap. We also assume that a querier cannot apriori determine if a cell is sensitive or not (i.e., they do not know the access control policies) [3]. To see why this is important, consider a FD defined on the *Employee* table (Table 1) *Zip*→*State*. Suppose $c_{11}(State = \text{“}CA\text{”})$ is sensitive based on the policy and in order to prevent inferences using the FD, let c_{24} be hidden. If the querier has knowledge that c_{24} is hidden due to our approach (and hence know that c_{11} was sensitive), they can deduce that c_{25} and c_{11} have the same value.

3.2 Querier View

For each querier, given the set of policies applicable to the querier, the algoritm first determines which cell is sensitive to them. Such cells are set to *NULL* in the view of the database shared with the querier. As noted in the introduction, if only the sensitive cells are set to *NULL* and all the non-sensitive cells retain their true values, the querier may infer information about the sensitive cells through the various dependencies defined on the database. It is necessary, therefore, to set some of the non-sensitive cells to *NULL* in order to prevent leakages due to dependencies. Henceforth, we will refer to the cells, both sensitive and non-sensitive, whose values will be replaced by *NULL* as *hidden* cells, denoted by $\mathbb{C}^H$. We now present the concept of a querier view on top of which queries are answered.

Definition 3.1 (Querier View). The set of value assignments for a set of cells $\mathbb{C}$ in a database instance $\mathbb{D}$ with respect to a querier is denoted by $\mathbb{V}(\mathbb{C})$ or simply $\mathbb{V}$ when the set of cells is clear from the context. The value assignment for a cell could be either the true value of this cell in $\mathbb{D}$ or *NULL* value (if it is hidden).

We also define a concept of the *base view* of database for a querier, denoted by $\mathbb{V}_0$. In $\mathbb{V}_0$, *all* the cells in $\mathbb{D}$ are set to be *NULL*. We consider the information leaked to the querier based on computing the query results over the base view $\mathbb{V}_0$ as the least amount of information revealed to the querier. For instance, the base view may provide querier with information about number of tuples in the relation, but, by itself it will not reveal any further information about the sensitive cells, despite what dependencies hold over the database. Our goal in developing the algorithm to prevent leakage would be to determine a view $\mathbb{V}$ for a querier that hides the minimal number of cells, and yet, leaks no further information than the base view. Next, we define an inference function that captures what the querier can infer about a sensitive cell in a view using dependencies.

3.3 Inference Function

Dependencies such as denial constraints are defined at schema level, such as the dependency δ on Table 1:

$$\delta : \forall t_i, t_j \in Emp\ \neg(t_i[State] = t_j[State] \land t_i[Role] = t_j[Role] \land t_i[SalPerHr] > t_j[SalPerHr]).$$

Given a database instance $\mathbb{D}$, the schema level dependencies can be *instantiated* using the tuples. If the *Employee* Table has 4 tuples, then there are $\binom{4}{2} = 6$ number of instantiated dependencies at cell level. For example, one of the instantiated dependencies for δ is

$$\tilde{\delta} : \neg((c_{11} = c_{18}) \land (c_{12} = c_{19}) \land (c_{14} > c_{21})) \quad (1)$$

where $\{c_{11}, c_{18}, c_{12}, c_{19}, c_{14}, c_{21}\}$ correspond to $t_2[State]$, $t_3[State]$, $t_2[Role]$, $t_3[Role]$, $t_2[SalPerHr]$, and $t_3[SalPerHr]$ in the *Employee* Table respectively. From now on, we use S_Δ to denote the full set of instantiated dependencies for the database instance $\mathbb{D}$ at cell level. We use $Preds(\tilde{\delta})$, $Preds(\tilde{\delta}, c)$, and $Preds(\tilde{\delta}\backslash c)$ to represent the set of predicates in the instantiated dependency $\tilde{\delta}$, the set of predicates in $\tilde{\delta}$ that involves the cell c, and the set of predicates in $\tilde{\delta}$ that do not involve the cell c respectively. We also use $Cells(\tilde{\delta})$ and $Cells(Pred)$ to represent the set of cells in an instantiated dependency and a predicate respectively. For each instantiated dependency $\tilde{\delta} \in S_\Delta$, when every cell $c_i \in Cells(\tilde{\delta})$ is assigned with a value $x_i \in Dom(c_i)$, denoted by $\tilde{\delta}(\dots, c_i = x_i, \dots)$, the expression associated with an instantiated dependency can be evaluated to either *True* or *False*. Note that since the database is assumed to satisfy all the dependencies, all of the instantiated dependencies must evaluate to *True* for any instance of the database.

[3]In the extended version of our paper, we have included a discussion on relaxing both these assumptions.

We use the notation $\mathbb{I}(c \mid \mathbb{V})$ to denote the set of values (inferred by the querier) that the cell c can take given the view $\mathbb{V}$ but without any knowledge of the set of dependencies. Likewise, $\mathbb{I}(\mathbb{C} \mid \mathbb{V})$ denote the cross product of the inferred value sets for cells in the cell set $\mathbb{C}$, i.e., $\mathbb{I}(\mathbb{C} \mid \mathbb{V}) = \times_{c\in\mathbb{C}}\mathbb{I}(c \mid \mathbb{V})$. Clearly, if in a view, a cell is assigned its original/true value (and not *NULL*) then $\mathbb{I}(c \mid \mathbb{V})$ consists of only its true value. We will further assume that:

Assumption 1. *Let $\mathbb{V}$ be a view and c be a cell with value NULL assigned to it in $\mathbb{V}$. $\mathbb{I}(c \mid \mathbb{V}) = Dom(c)$. That is, a querier without knowledge of dependencies, cannot infer any further information about the value of the cell beyond its domain.*

Knowledge of the dependencies can, however, lead the querier to make inferences about the value of the cell. The following example illustrates that the querier may be able to eliminate some domain values as possible assignments of $Dom(c)$.

Example 3. Let c_{14} in Table 1 be sensitive for a querier and let the view $\mathbb{V}$ be the same as the original table with c_{14} replaced with *NULL*. Furthermore, let $\tilde{\delta}$ (Eqn. (1)) (that refers to c_{14}) hold. If the querier is not aware of this dependency $\tilde{\delta}$, the inferred value set for c_{14} is the full domain, i.e., $\mathbb{I}(c_{14} \mid \mathbb{V}) = Dom(c_{14})$. However, knowledge of $\tilde{\delta}$ leads to the inference that $c_{14} \leq 200$ since the other two predicates ($c_{11} = c_{18}, c_{12} = c_{19}$) are *True*. □

Definition 3.2 (Inference Function). Given a view $\mathbb{V}$ and an instantiated dependency $\tilde{\delta}$ for a cell $c_i \in Cells(\tilde{\delta})$, the inferred set of values for c_i by $\tilde{\delta}$ is defined as

$$\mathbb{I}(c_i|\mathbb{V},\tilde{\delta}) := \{x_i \mid \exists(\ldots,x_{i-1},x_{i+1},\ldots) \in \mathbb{I}(Cells(\tilde{\delta})\backslash\{c_i\} \mid \mathbb{V}) \\ s.t.\ \tilde{\delta}(c_1 = x_1,\ldots,c_i = x_i,\ldots,c_n = x_n) = True\} \quad (2)$$

where n denotes the size of the cell set $|Cells(\tilde{\delta})|$ and $x_i \in Dom(c_i)$.

Given a view $\mathbb{V}$ and a set of instantiated dependencies $S_\Delta = \{\ldots,\tilde{\delta},\ldots\}$, the inferred value for a cell c is the intersection of the inferred values for c_i over all the dependencies, i.e.,

$$\mathbb{I}(c_i|\mathbb{V},S_\Delta) := \bigcap\nolimits_{\tilde{\delta}\in S_\Delta}\mathbb{I}(c_i|\mathbb{V},\tilde{\delta}) \quad (3)$$

3.4 Security Definition

We can now formally define the concept of full deniability of a view. Note that given a view $\mathbb{V}$ and a set of dependencies S_Δ, the following always holds: $\mathbb{I}(c|\mathbb{V},S_\Delta) \subseteq \mathbb{I}(c|\mathbb{V}_0,S_\Delta)$. We say that a $\mathbb{V}$ achieves full deniability if the two set are identical i.e., the query results does not enable the querier to infer anything further about the database than what the querier could infer from the $\mathbb{V}_0$ (which, as mentioned in Sec. 3.2, is the least amount of information leaked to the querier).

Definition 3.3 (Full Deniability). Given a set of sensitive cells $\mathbb{C}^S$ in a database instance $\mathbb{D}$ and a set of instantiated dependencies S_Δ, we say that a querier view $\mathbb{V}$ achieves full deniability if for all $c^* \in \mathbb{C}^S$,

$$\mathbb{I}(c^*|\mathbb{V},S_\Delta) = \mathbb{I}(c^*|\mathbb{V}_0,S_\Delta). \quad (4)$$

4 FULL DENIABILITY WITH DATA DEPENDENCIES

In this section, we first identify conditions under which denial constraints could result in leakage of sensitive cells (i.e., violation of full deniability) and further consider leakages due to function-based constraints (discussed in Section 2).

4.1 Leakage due to Denial Constraints

An instantiated denial constraint consists of multiple predicates in the form of $\tilde{\delta} = \neg(Pred_1 \wedge \ldots \wedge Pred_N)$ where each predicate is either $Pred_N = c\ \theta\ c'$ or $Pred_N = c\ \theta\ const$. A valid value assignment for cells in $\mathbb{C}(\tilde{\delta})$ has at least one of the predicates in $\tilde{\delta}$ evaluating to *False* so that the entire dependency instantiation $\tilde{\delta}$ evaluates to *True*. Based on this observation, we identify a sufficient condition to prevent a querier from learning about a sensitive cell $c^* \in \mathbb{C}^S$ in an instantiated DC $\tilde{\delta}_i$ with value assignments.

As shown in Example 3, for an instantiated DC $\tilde{\delta}$ with cell value assignments, when all the predicates except for the predicate containing the sensitive cell ($Pred(\tilde{\delta}\backslash c^*)$) evaluates to *True*, a querier can learn that the remaining predicate $Pred(\tilde{\delta},c^*)$ evaluates to *False* even though c^* is hidden. Thus, it becomes possible for the querier to learn about the value of a sensitive cell from the other non-sensitive cell values. We can prevent such an inference by hiding additional non-sensitive cells.

Example 4. Suppose, in Example 3, we hide the non-sensitive cell (e.g., c_{11}) in addition to c_{14} (i.e., replace it with *NULL*). Now, the querier will be uncertain of the truth value of $c_4 = c_{11}$, and as a result, cannot determine the truth value of the predicate $c_{14} > c_{21}$ containing the sensitive cell. Since the predicate, $c_{14} > c_{21}$ could either be true or false, the querier does not learn anything about the value of the sensitive cell c_{14}. □

We can formalize this intuition into a sufficient condition that identifies additional non-sensitive cells to hide which we refer to as the *Tattle-Tale Condition* (TTC) [4] in order to prevent leakage of sensitive cells, as follow:

Definition 4.1 (Tattle-Tale Condition). Given an instantiated DC $\tilde{\delta}$, a view $\mathbb{V}$, a cell $c \in Cells(\tilde{\delta})$, and $Preds(\tilde{\delta}\backslash c) \neq \phi$

$$TTC(\tilde{\delta},\mathbb{V},c) = \begin{cases} True, & \forall\ Pred_i \in Preds(\tilde{\delta}\backslash c), \\ & eval(Pred_i,\mathbb{V}) = True \\ False, & otherwise \end{cases} \quad (5)$$

where *eval(Pred, $\mathbb{V}$)* refers to the truth value of the predicate *Pred* in the view $\mathbb{V}$ using the standard 3-valued logic of SQL i.e., a predicate evaluates to true, false, or unknown (if one or both cells are set to *NULL*). The predicates only compare between the values of two cells or the value of a cell with a constant as defined in Sec 2.2.

Note that $TTC(\tilde{\delta},\mathbb{V},c)$ is *True* if and only if all the predicates except for the predicate (s) containing c ($Preds(\tilde{\delta},c)$) evaluate to *True* in which case, the querier can infer that the one of the predicates containing c must be false and, as a result, could exploit the knowledge of the predicate (s) to restrict the set of possible values that c could take. This leads us to a sufficient condition to achieve full

[4]Tattle-Tale refers to someone who reveals secret about others

deniability as captured in the following two theorems. In proving the theorems, we will assume that none of the predicates in the denial constraints are trivial That is, there always exist a domain value for which the predicate can be true or false. This also means that in the base view $\mathbb{V}_0$ (where all cells are hidden), for any cell $c_i \in cells(\tilde{\delta})$ and for any predicate $Pred \in Preds(\tilde{\delta}, c)$, there exists a possible assignment for c_i in $\mathbb{I}(c_i \mid \mathbb{V}_0, \tilde{\delta})$ such that $eval(Pred, \mathbb{V}_0)$ returns *False*.

Theorem 4.2. *Given an instantiated DC $\tilde{\delta}$, a view $\mathbb{V}$, and a sensitive cell $c^* \in Cells((\tilde{\delta}))$ whose value is hidden in this view. If the Tattle-Tale Condition $TTC(\tilde{\delta}, \mathbb{V}, c^*)$ evaluates to False, then the set of inferred values for c^* from $\mathbb{V}$ is the same as that from the base view $\mathbb{V}_0$ (where all the cells are hidden), i.e., $\mathbb{I}(c^*|\mathbb{V}, \tilde{\delta}) = \mathbb{I}(c^*|\mathbb{V}_0, \tilde{\delta})$.*

Corollary 4.3. *Given a set of instantiated DCs S_Δ, a view $\mathbb{V}$, and a sensitive cell c^* whose value is hidden in this view. If for each of the instantiations $\tilde{\delta}_i \in S_\Delta$, $TTC(\tilde{\delta}_i, \mathbb{V}, c^*)$ evaluates to False then the set of inferred values c^* from the $\mathbb{V}$ is same as that from the base view $\mathbb{V}_0$ i.e., $\mathbb{I}(c^* \mid \mathbb{V}, S_\Delta) = \mathbb{I}(c^* \mid \mathbb{V}_0, S_\Delta)$.*

4.2 Selecting Cells to Hide

As shown in Theorem 4.2, the Tattle-Tale condition evaluating to *False* is the sufficient condition of achieving full deniability requirement. $TTC(\tilde{\delta}, \mathbb{V}, c)$ evaluates to *False* when one of the following holds: (i) none of the predicates involve the sensitive cell i.e., $Preds(\tilde{\delta}, c^*) = \phi$ (trivial case); (ii) one of the other predicates in $Preds(\tilde{\delta} \backslash c^*)$ evaluates to *False* in $\mathbb{V}$; or (iii) one of the other predicates in $Preds(\tilde{\delta} \backslash c^*)$ involve a hidden cell in $\mathbb{V}$ and thus evaluates to *Unknown*.

We define *cuesets*[5] as the set of cells in an instantiated DC that can be hidden to falsify the Tattle-Tale condition.

Definition 4.4 (Cueset). Given an instantiated DC $\tilde{\delta}$, a cueset for a cell $c \in cells(\tilde{\delta})$ is defined as

$$cueset(c, \tilde{\delta}) = Cells(Preds(\tilde{\delta} \backslash c)). \tag{6}$$

If $\tilde{\delta}$ only contains a single predicate, we consider the remaining cell in the $cueset(c, \tilde{\delta}) = c_j$ given that $Pred(c) = c_i \theta c_j$.

Example 5. In the instantiated DC from Example 3, the cueset for c_{14} based on $\tilde{\delta}_4$ is given by $cueset(c_{14}, \tilde{\delta}_4) = \{c_4, c_{11}, c_5, c_{12}\}$. □

We could falsify the Tattle-Tale condition w.r.t. a given cell c and dependency $\tilde{\delta}$ by hiding any one of the cells in the cueset independent of their values in $\mathbb{V}$. The cuesets for a cell c is defined for a given dependency instantiation. We can further define cueset for c for given a set of instantiated DCs S_Δ by simply computing the $cueset(c, \tilde{\delta})$ for each instantiated dependency in the set $\tilde{\delta} \in S_\Delta$. In order to prevent leakage of c through $\tilde{\delta}$, we will hide one of the cells in the $cueset(c, \tilde{\delta})$ corresponding to each of dependency instantiations $\tilde{\delta} \in S_\Delta$.

This, alone, however, might not still falsify the tattle-tale condition to achieve full-deniability. Leakage can occur indirectly since the value of the cell, say c_j chosen from the $cueset(c^*, \tilde{\delta}_i)$ to hide (in order to protect leakage of a sensitive cell c^*) could, in turn, be inferred due to additional dependency instantiation, say $\tilde{\delta}_j$. If this dependency instantiation does contain c^* (as in that case c^* is already hidden and therefore it cannot be used to infer any information about c_j), such a leakage can, in turn, lead to leakage of c^* as shown in the following example.

Example 6. Consider we hide the cell c_{11} in the cueset shown in Example 5 to protect the sensitive cell c_{14}. Let $\tilde{\delta}_j$ be another instantiated dependency, that is $\neg((c_{10} = c_{24}) \land (c_{11} \neq c_{25}))$ (i.e. a FD indicating Zip determines State). The dependency $\tilde{\delta}_j$ itself does not lead to the leakage to the sensitive cell c^*. However, based on dependency $\tilde{\delta}_j$, the querier can first infer the value of the hidden cell c_{11}, which in turn, leads to leakage of the sensitive cell c^*. □

Achieving full deniability for the sensitive cells requires us to recursively select cells to hide from the cuesets of not just sensitive cells, but also, from the cuesets of all the hidden cells. This recursive hiding of cells terminates when the cueset of a newly hidden cell includes an already hidden cell. The following theorem states that after the recursive hiding of cells in cuesets has terminated, the querier view achieves full deniability.

Theorem 4.5 (Full Deniability for a Querier View). *Let S_Δ be the set of dependencies, $\mathbb{C}^S$ be the sensitive cells for the querier and $\mathbb{C}^S \subseteq \mathbb{C}^H$ be the set of hidden cells resulting in a $\mathbb{V}$ for the querier. $\mathbb{V}$ achieves full deniability if $\forall c_i \in \mathbb{C}^H$, $\forall \tilde{\delta} \in S_\Delta$, $\forall$ non-empty $cueset(c_i, \tilde{\delta}) \in cuesets(c_i, S_\Delta)$, there exists a $c_j \in \mathbb{C}^H$ such that $c_j \in cueset(c_i, \tilde{\delta})$.*

4.3 Leakage due to Function-based Constraints

In order to study the leakages due to Function-based Constraints (FC) we first define the property of invertibilty associated with functions.

Definition 4.6 (Invertibility). Given a function $fn(r_1, r_2, \ldots, r_n) = s_i$, we say that fn is invertible if it is possible to infer knowledge about the inputs $(r_1, r_2, \ldots, r_n)$ from its output s_i. Conversely, if s_i does not lead to any inferences about $(r_1, r_2, \ldots, r_n)$, we say that it is non-invertible

The *Salary* function, in Example 2, is invertible as given the Salary of an employee, a querier can determine the minimum value of *SalPerHr* for that employee given that the maximum number of work hours in a week is fixed. Complex user-defined functions (UDFs) (e.g., sentiment analysis code which outputs the sentiment of a person in a picture), oblivious functions, secret sharing, and many aggregation functions are, however, non-invertible. Instantiated FCs can be represented similar to denial constraints. For example, an instantiation of the dependency δ : *Salary := fn(WorkHrs, SalPerHr)* is: $\tilde{\delta} : \neg(c_6 = 20 \land c_8 = 40 \land c_{31} \neq 800)$ where c_7, c_8, c_{31} corresponds to Alice's *WorkHrs*, *SalPerHr* and *Salary* respectively.

For instantiated FCs, if the sensitive cell corresponds to an input to the function, and the function is not invertible, then leakage cannot occur due to such an FC. Thus, the $TTC(c^*, \tilde{\delta}, \mathbb{V})$ returns *False* when the function is non-invertible. For all other cases, the leakage can occur in the exact same way as in denial constraints. We thus, need to to ensure the Tattle-Tale Condition for all the instantiations of a FC evaluates False.

[5]These cells give a *cue* about the sensitive cell to the querier.

Algorithm 1: Full Algorithm

```
    Input: User $U$, Data dependencies $S_\Delta$, A view of the
           database $\mathbb{V}$
    Output: A secure view $\mathbb{V}_S$
1   $\mathbb{C}^S$ = SensitivityDetermination($U$, $\mathbb{V}$)
2   $\mathbb{C}^H = \mathbb{C}^S$, $\mathbb{V}_S = \mathbb{V}$
3   $cuesets$ = InferenceDetect($\mathbb{C}^H$, $S_\Delta$, $\mathbb{V}$)
4   while $cuesets \neq \phi$ do
5       for $cs \in cuesets$ do
6           if $cs.overlaps(\mathbb{C}^H)$ then
7               $cuesets$.remove($cs$)
8       end
9       $toHide$ = InferenceProtect ($cuesets$)
10      $\mathbb{C}^H$.addAll($toHide$)
11      $cuesets$ = InferenceDetect($toHide$, $S_\Delta$, $\mathbb{V}$)
12  end
13  for $c_i \in \mathbb{C}^H$ do
14      Replace $c_i.val$ in $\mathbb{V}_S$ with $NULL$
15  end
16  return $\mathbb{V}_S$
```

Cueset for Function-based Constraints. The cueset for a FC $\tilde{\delta}$ is determined depending on whether the derived value (s_i) or input value ($\{\ldots, r_j, \ldots\}$) is sensitive and the invertibility property of the function fn.

$$cueset(c, \tilde{\delta}) = \begin{cases} \{c_i\}\ \forall c_i \in \{\ldots, r_j, \ldots\}, & \text{if } c = s_i \\ \{s_i\}, & fn \text{ is invertible and if } c \in \{\ldots, r_j, \ldots\} \\ \phi, & fn \text{ is non-invertible and if } c \in \{\ldots, r_j, \ldots\} \end{cases}$$

As the instantiation for FC is in DC form and their Tattle-Tale Conditions and cueset determination are almost identical, in the following section we explain the algorithms for achieving full deniability with DCs as extending it to handle FCs requires only a minor change (disregard cuesets when one of the input cell(s) is sensitive and function is non-invertible).

5 ALGORITHM TO ACHIEVE FULL DENIABILITY

In this section, we present an algorithm to determine the set of cells to hide to achieve the full-deniability based on Theorem 4.5. Full-deniability can trivially be achieved by sharing the base view $\mathbb{V}_0$ where all cells values are replaced with *NULL*. Our goal is to ensure that we hide the minimal number of cells possible while achieving full deniability.

5.1 Full-Deniability Algorithm

As shown in Figure 2, our approach (Algorithm 1) takes as input a user U, a set of schema level dependencies S_Δ, and a view of the database $\mathbb{V}$ (initially set to the original database). The algorithm first determines the set of sensitive cells $\mathbb{C}^S$ (*Sensitivity Determination* function for U and $\mathbb{V}$). Sensitivity determination identifies the policies applicable to a querier using the subject conditions in policies and marks a set of cells as sensitive thus assigning them

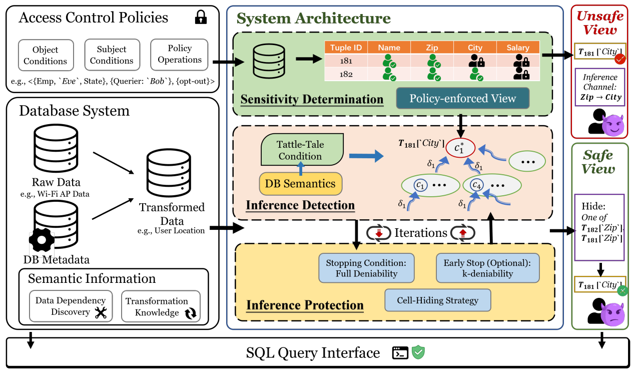

Figure 2: System Architecture

with *NULL* in the view. The set of sensitive cells are added into a set of hidden cells (*hidecells*) which will be finally hidden in the secure view ($\mathbb{V}_S$) that is shared with the user U. Next, the algorithm generates the cuesets for cells in *hidecells* using S_Δ and $\mathbb{V}$ (*Inference Detection*, Step 3). Given the cuesets, the algorithm chooses a set of cells to hide such that the selected cells covers each of the cuesets (*Inference Protection*). This process of cueset identification protection continues iteratively as new hidden cells get added. The algorithm terminates when for all of the cuesets there exists a cell that is already hidden. Finally, we replace the value of *hidecells* in $\mathbb{V}_S$ (initialized to $\mathbb{V}$) with *NULL* and returns this secure view to the user (Steps 13-16). The following theorem (proof included in the extended version) states that the algorithm successfully implements the recursive hiding of cells in $\mathbb{C}^H$ which is required for generating a querier view that achieves full deniability (as discussed in Theorem 4.5).

THEOREM 5.1. *When Algorithm 1 terminates,* $\forall c_i \in \mathbb{C}^H$, $\forall \tilde{\delta} \in S_\Delta$, *for all* $cueset(c_i, \tilde{\delta})$ *that is non-empty, there exists* $c_j \in cueset(c_i, \tilde{\delta})$ *such that* $c_j \in \mathbb{C}^H$ *(i.e., Algorithm 1 has recursively hidden at least 1 cell from all the non-empty cuesets of cells in* $\mathbb{C}^H$*).*

5.2 Inference Detection

Inference detection (Algorithm 2) takes as input the set of sensitive cells ($\mathbb{C}^S$), the set of schema-level dependencies (S_Δ), and a view of the database ($\mathbb{V}$) in which sensitive cells are hidden by replacing with and others are assigned the values corresponding to the instance. For each sensitive cell c^*, we consider the given set of dependencies S_Δ and instantiate each of the relevant dependencies δ using the database view $\mathbb{V}$ (Steps 5-7). The *DepInstantiation* function returns the corresponding instantiated dependency $\tilde{\delta}$. For each such dependency instantiation, if it is a dependency containing a single predicate i.e., $\tilde{\delta} = \neg(Pred)$ where $Pred = c^* \theta c_k$, we add the non-sensitive cell (c_k) to the cueset (Steps 9, 10). If the dependency contains more than a single predicate, we determine if there is leakage about the value of the sensitive cell by checking the Tattle-Tale Condition (TTC) for the sensitive cell c^* (Step 11)[6]. If $TTC(\tilde{\delta}, \mathbb{V}, c^*)$ evaluates to *False*, we can skip that dependency instantiation as

[6]While not shown in the algorithm for simplicity, when an input cell is sensitive in a FC instantiation, if the FC is non-invertible we ignore its cuesets as they are empty.

Algorithm 2: Inference Detection

```
Input: A set of sensitive cells $\mathbb{C}^S$, Schema-level data
       dependencies $S_\Delta$, A view of the database $\mathbb{V}$
Output: A set of cuesets cuesets
Function InferenceDetect($\mathbb{C}^S$, $S_\Delta$, $\mathbb{V}$):
    cuesets = {}
    for $c^* \in \mathbb{C}^S$ do
        $S_{S_\Delta}$ = {}            ▷ Set of instantiated dependencies.
        for $\delta \in \Delta$ do
            $S_{S_\Delta} = S_{S_\Delta} \cup$ DepInstantiation($\delta$, $c^*$, $\mathbb{V}$)
        end
        for $\tilde{\delta} \in S_\Delta$ do   ▷ For each instantiated dependency.
            if $|Preds(\tilde{\delta})| = 1$ then
                cueset = $\{c_k\}$          ▷ Note: $Pred(c^*) = c^* \theta c_k$
            else if $TTC(\tilde{\delta}, \mathbb{V}, c^*) = False$ then
                continue
            else
                cueset = $cells(Preds(\tilde{\delta} \backslash c))$
            end
            cuesets.add(cueset)
        end
    end
    return cuesets
```

there is no leakage possible on c^* due to it (Step 12). However, if $TTC(\tilde{\delta}, \mathbb{V}, c^*)$ evaluates to *True*, we get all the cells except for $Pred(c^*)$ (Step 14) [7]. After iterating through all the dependency instantiations for all the sensitive cells, we return *cuesets* (Step 19).

Note that in our inference detection algorithm, we did not choose the non-sensitive cell c' in $Pred(c^*) = c^* \theta c'$ as a candidate for hiding. We illustrate below using a counter-example why hiding c' might not be enough to prevent leakages.

Example 7. Consider a relation with 3 attributes A_1, A_2, A_3 and 3 dependencies among them ($\delta_1 : A_1 \rightarrow A_2, \delta_2 : A_2 \rightarrow A_3, \delta_3 : A_1 \rightarrow A_3$). Let there be two tuples in this relation $t_1 : 1(c_1), 2(c_2), 2(c_3)$ and $t_2 : 1(c_4), 2(c_5), 2(c_6)$. Suppose c_6 is sensitive. As leakage of the sensitive cell is possible through the dependency instantiation $\tilde{\delta}_2 : \neg((c_2 = c_5) \wedge (c_3 = c_6))$, c_5 is hidden. In the next iteration of the algorithm, to prevent leakages on the hidden cell c_5 through dependency instantiation $\tilde{\delta}_1 : \neg((c_1 = c_4) \wedge (c_2 = c_5))$, c_2 is also hidden. Note that c_2 is in the same predicate as c_5 in $\tilde{\delta}_1$. However, the querier can still infer the truth value of the predicate $c_2 = c_5$ as *True* based on the two non-hidden cells, c_1 and c_4, and the dependency instantiation $\tilde{\delta}_3 : \neg((c_1 = c_4) \wedge (c_2 = c_5))$. The querier also learns that $c_3 = c_6$ evaluates to *True* in $\tilde{\delta}_2$ which leads to them inferring that $c_6 = 2$ (same as c_3) and complete leakage. □

[7] If we wish to relax the assumption that queriers and data owners do not overlap stated in Section 3.1, we can do so here by only including the cells in the cueset that do not belong to the querier. We prove the correctness of this modification in the extended version [27].

Algorithm 3: Inference Protection

```
Input: Set of cuesets cuesets
Output: A set of cells selected to be hidden toHide
Function InferenceProtect(cuesets):
    toHide = {}                    ▷ Return list initialization.
    while cuesets ≠ $\phi$ do
        cuesetCells = Flatten(cuesets)
        $dict[c_i, freq_i]$ = CountFreq(GroupBy(cuesetCells))
        cellMaxFreq = GetMaxFreq($dict[c_i, freq_i]$)
        toHide.add(cellMaxFreq)          ▷ Greedy heuristic.
        for cs ∈ cuesets do
            if cs.overlaps(toHide) then
                cuesets.remove(cs)
        end
    end
    return toHide
```

To prevent any possible leakages on the sensitive cell c^* and its corresponding predicate $Pred(c^*)$, we only consider the solution space where a cell from a different predicate ($Preds(\tilde{\delta} \backslash c^*)$) is hidden.

5.3 Inference Protection

After identifying the cuesets for each sensitive cells based on their dependency instantiations, we now have to select a cell from each of them to hide to prevent leakages. The strategy for cell selection, described in Algorithm 3 utilizes Minimum Vertex Cover (MVC) [7] to minimally select the cells to hide from the list of cuesets. In this approach, each cueset is considered as a hyper-edge and cell selection strategy finds the minimal set of cells that covers all the cuesets. MVC is known to be NP-hard [11] and therefore we utilize a simple greedy heuristic based on the membership count of cells in various cuesets. Algorithm 3 takes as input the set of cuesets and returns the set of cells to be hidden to prevent leakages. First, we flatten all the cuesets into a list of cells and insert this list into a dictionary with the cell as key and their frequency count as the value (Steps 4-5). Next, we select the cell from the dictionary with the maximum frequency and add it to the set of cells to be hidden and remove any cuesets that contain this cell (steps 7-10). These steps are repeated until all the cuesets are covered i.e., at least one cell in it is hidden, and finally we return the set of cells to be hidden.

5.4 Convergence and Complexity Analysis

Algorithm 1 starts with s number of hidden cells. At each iteration, we consider that each hidden cell (including cells are hidden in previous iterations) is expanded to f number of cuesets on average by the Inference Detection algorithm (Algorithm 2). Among the cuesets, the average number of cells that are hidden, such that it satisfies full deniability, is given by $\frac{f}{m}$ where m is the coverage factor determined by minimum vertex cover (MVC). Then, at the end of ith iteration, the number of average hidden cells will be $s_i = s(\frac{f}{m})^i$, and the average number of cuesets will be $cs_i = sf(\frac{f}{m})^{i-1}$. As s_i is bounded by the total number of cells in the database, denoted by N, the number of iterations (T) to converge is bounded by $\log_{f/m}(N/s)$, when $f > m$ (which was verified in our experiments).

Given $|\Delta|$ which is the number of schema-level dependencies, we can estimate the time complexity with respect to I/O cost. At ith iteration of Algorithm 1, the I/O cost of (i) the dependency instantiation is $O(|\Delta|(N + s_i))$ (implemented using a "JOIN" query given sufficient, i.e. $\Theta(N)$, memory) and (ii) minimum vertex cover (MVC) with an I/O cost of $O(cs_i)$. Hence, the overall estimated I/O cost $\sum_{i=1}^{T} O(|\Delta|(N + s_i)) + O(cs_i)$ in which is equivalent to $O(N)$ given $T \leq \log_{f/m}(N/s)$ and thus is linear to the data size.

6 DISCUSSION

While full deniability studied in this paper offers a strong security model to protect sensitive data from inference attacks through data dependencies, it is not without limitations. In this section, we discuss these limitations and explore possible extensions to overcome them.

6.1 Weaker Security Model

The **first** potential limitation is from the perspective of utility as full deniability hides a number of non-sensitive cells to prevent leakages. It is possible to relax full deniability to a weaker security model which we call, *k-percentile deniability* in order to potentially hide fewer cells and thus improve utility.

Definition 6.1 (k-percentile Deniability). Given a set of sensitive cells $\mathbb{C}^S$ in a database instance $\mathbb{D}$ and a set of instantiated dependencies S_Δ, we say that a querier view $\mathbb{V}$ achieves k-percentile deniability if for all $c^* \in \mathbb{C}^S$,

$$|\mathbb{I}(c^*|\mathbb{V}, S_\Delta)| \geq (k \cdot |\mathbb{I}(c^*|\mathbb{V}_0, S_\Delta)|) \tag{7}$$

where $\frac{1}{|\mathbb{I}(c|\mathbb{V}_0, S_\Delta)|} \leq k \leq 1$.

Note that if $k = 1$, then k-percentile deniability is the same as full deniability, where the set of values inferred by the adversary from view $\mathbb{V}$ is the same as the set from the base view. With $k < 1$, it allows for a bounded amount of leakage. We also note that the security models used in prior works is subsumed by the notion of k-percentile deniability defined above. For instance, the model used in [3] ensures that the querier cannot reconstruct the exact value of the sensitive cell using data dependencies, which can be viewed as a special case of k-percentile deniability with the value of $k = \frac{2}{|\mathbb{I}(c|\mathbb{V}_0, S_\Delta)|}$, i.e., the number of values sensitive cell can take is more than 1.

We discuss k-percentile deniability in detail in the extended version of the paper. In Section 7, we show that the algorithm that achieves k-percentile deniability only marginally improves on full deniability even with low values of k (i.e., complete leakage). Therefore this approach is not useful in improving the utility in realistic settings. It is possible that in more complex domains with large number of sensitive cells, k-percentile deniability is more effective and this needs to be studied further.

6.2 Limitations of Security Model

The **second** potential limitation is from the perspective of security as our security model is based on the assumption that no correlations exist between attributes and tuples i.e., they are independently distributed other than what is explicitly stated through dependencies (that are either learnt automatically or specified by the expert). However, typically in databases, other correlations do exist which can be exploited to infer the values of the hidden cells. These correlations can be also learned by the database designer using dependency discovery tools or data analysis tools. If the correlations are very strong (e.g. hard constraints with no violations in the database), we call them out as constraints and consider them in our algorithms. For weak correlations, or soft constraints that only apply to a portion of the data, we do not consider them. Otherwise, everything in the database will become dependent, in which case our algorithm would be too conservative and hide more cells than necessary based on these soft constraints.

Hence, in our experiments (Section 7), we consider only the data dependencies (a set of hard constraints) that were defined on the database. To understand how this assumption plays out in inference attacks, we deployed a data cleaning adversary who tries to exploit all possible correlations in the data to reconstruct the values of the sensitive cells. The results (Section 7.4) show that by considering the major data dependencies, the adversary is only able to recover a small portion of the sensitive cells (14 out of 90 sensitive cells).

7 EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation results for our proposed approach to implement full-deniability. First, we explain our experimental setup including details about the datasets, dependencies, baselines used for comparison, evaluation metrics, and system setup. Second, we present the experimental results for each of the following evaluation goals: 1) comparing our approach against baselines in terms of utility, performance, and number of cuesets generated; 2) evaluating the impact of *dependency connectivity*, and 3) studying the extensions and limitations presented in Section 6. The dataset schemas, list of data dependencies, and results on a larger dataset are included in the extended version[27].

7.1 Evaluation Setup

Datasets. We perform our experiments on 2 different datasets. The first one is *Tax dataset* [2], a synthetic dataset with 10k tuples and 14 attributes, where 10 of them are discrete domain attributes and the rest are continuous domain attributes. Every tuple from the tax table specifies tax information of an individual with information such as name, state of residence, zip, salary earned, tax rate, tax exemptions etc. The second dataset is the *Hospital dataset* [5] which is a real-world dataset where all of the 15 attributes are discrete domain attributes.

Data Dependencies. For both datasets, we identify a large number of denial constraints by using a data profiling tool, Metanome [25]. Many of the output DCs identified by Metanome were soft constraints which are only valid for a small subset of the database instance. After manually analyzing and pruning these soft DCs, we selected 10 and 14 hard DCs for the Tax dataset and the Hospital dataset respectively. We also added a FC based on the continuous domain attribute named *"tax"* which is calculated as a function "tax = fn(salary, rate)".

Policies. (described in Section 2) control the sensitivity of a cell. The number of sensitive cells is equivalent to the number of policies

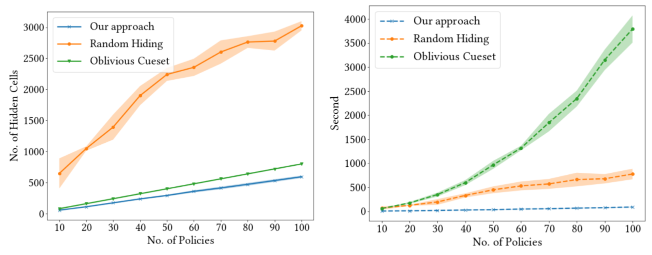

Figure 3: (a) Data utility (b) Performance. Experiments done on Tax dataset for *Our Approach*, *Random Hiding*, and *Oblivious Cueset*.

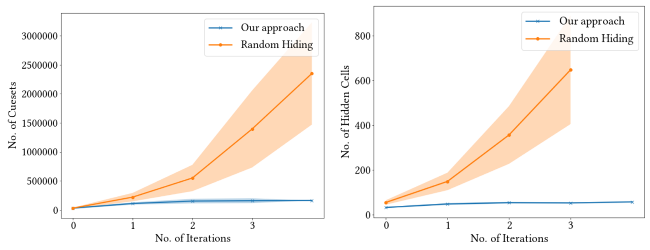

Figure 4: (a) Number of cuesets generated in each invocation of Inference Detection (b) Number of cells hidden in each invocation of Inference Protection. Experiments run with 10 sensitive cells on Tax dataset.

and it helps us in precisely controlling the number of sensitive cells in experiments using policies. While the experiments are performed for a single querier, the extension to multiple queriers is trivial.
Metrics. We compare our approach against the baseline methods using the following metrics: 1) *Utility*: measures the number of total cells hidden; 2) *Performance*: measures the run time in seconds.
System Setup. We implemented the system in Java 15 and build the system dependencies using Apache Maven. We ran the experiments on a machine with the following configuration: Intel(R) Xeon(R) CPU E5-4640 2.799 Ghz, CentOS 7.6, with RAM size 64GB. We chose as the underlying database management system MySQL 8.0.3 with InnoDB. For each testcase, we perform 4 runs and report the mean and standard deviation.
Baselines. In the following experiments we test our approach which implements Algorithm 1, denoted by *Our Approach* against baselines. To best of our knowledge, there exists no other systems which solves the same problem and therefore we have developed 2 different baseline strategies for comparison. In each baseline method, we replace one of the key modules in our system, determining cuesets and selecting cell to hide from the cueset, with a naïve strategy but without compromising full deniability of the generated querier view.

- *Baseline 1: Random selection strategy for hiding* (*Random Hiding*): which replaces the minimum vertex cover approach with an inference protection strategy that randomly selects cells from cuesets to hide.
- *Baseline 2: Oblivious cueset detection strategy* (*Oblivious Cueset*): which disregards Tattle-Tale Condition and uses an inference detection strategy that creates as many dependency instantiations as the number of tuples in the database for each dependency and generates cuesets for all of them.

7.2 Experiment 1: Baseline Comparison

We compare our approach against the aforementioned baselines and measure the utility as well as performance (see Figure 3(a)). We increase the number of policies from 10 to 100 (step=10) where each each sensitive cell participate in at least 5 dependencies. This ensures that there are sufficient inference channels through which information about sensitive cells could be leaked. The number of cells hidden by *Our Approach* increases linearly w.r.t the increase in number of policies/sensitive cells compared to *Random Hiding* (5.3×*Our Approach*) and *Oblivious Cueset* (1.4×*Our Approach*). *Random Hiding* performs the worst because it randomly hides cells without checking the membership count of a cell in cuesets (as with using *MVC* in Algorithm 3). The performance of *Oblivious Cueset* is better because it uses the same Inference protection strategy as *Our Approach*. However, it generates a larger number of cuesets as it doesn't check the Tattle-Tale Condition for the dependency instantiations (like in Algorithm 2)) and therefore has to hide more cells to ensure full deniability.

We also compare the performance (run time in seconds) against number of policies of these 3 approaches (see Figure3(b)). The run time of *Our Approach* is almost linear w.r.t the increase of the number of policies. On the other hand, *Oblivious Cueset* is exponential w.r.t number of policies, because it generates $O(|\Delta| \times n^2)$ cuesets where n denotes the number of tuples in $\mathbb{D}$ and it is expensive to run inference detection on such a large number of cuesets. In *Random Hiding*, we restrict the execution to the fifth invocation of the inference detection algorithm (Algorithm 2) i.e., if the execution doesn't complete by then, we force stop the execution. In order to study this further, we analyzed the total number of cuesets generated by *Random Hiding* vs. *Our Approach* (see Figure 4) in each invocation of Inference Detection. Due to the usage of MVC optimization in Inference Protection, *Our Approach* terminates after a few rounds where as with *Random Hiding* the number of cuesets generated in each invocation keeps increasing. We also note that *Our Approach* is more stable in different test cases and has a lower standard deviation on number of cuesets and hidden cells compared to *Random Hiding*.

7.3 Experiment 2: Dependency Connectivity

In the next set of experiments, we study the impact of dependency connectivity on the utility as well as performance. The relationship between dependencies and attributes can be represented as a *hypergraph* wherein the attributes are nodes and they are connected via data dependencies. We define the *dependency connectivity* of a node, i.e., an attribute, in this graph based on the summation of the degree (number of edges incident on the node) as well as the degrees of all the nodes in its closure. Using dependency connectivity, we categorize attributes on *Tax* dataset into three groups: low, medium, and high where attributes in high, low, and medium groups have the highest, lowest, and average dependency connectivity respectively. In Tax dataset, the high group contains 3 attributes (e.g. State), while the medium group has 3 attributes (e.g. Zip) and the low group includes 4 attributes (e.g. City).

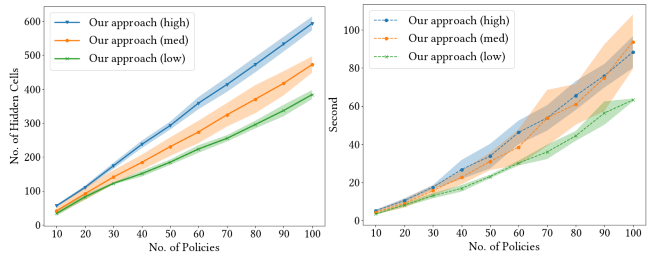

Figure 5: (a) Data utility (b) Performance. Experiments run with sensitive cells selected from (low, medium, high) dependency connectivity attributes in Tax dataset.

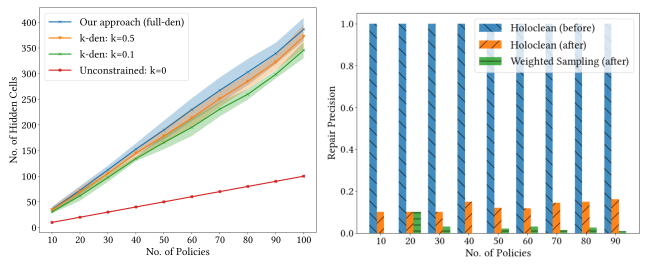

Figure 6: (a) Data utility on Tax dataset. Experiments done with full deniability and k-deniability (varying values of k). (b) Reconstruction precision of sensitive cells with two types of adversaries.

The results (see Figure 5) show that when sensitive cells are selected from attributes with higher dependency connectivity, *Our Approach* hides more cells than when selecting sensitive cells with lower dependency connectivity. This is because higher dependency connectivity leads to a larger number of dependency instantiations and therefore a larger number of cuesets from each of which at least one cell should be hidden.

7.4 Experiment 3: Extensions

In the final set of experiments, we test the extensions and limitations of *Our Approach* discussed in Section 6. **First**, we implemented *Our Approach* with a relaxed notion of security, k-percentile deniability, where k is a relative parameter based on the domain size of the sensitive cell. We analyze the utility of *Our Approach* when varying k and measure the utility. For the results shown in Figure 6(a), the sensitive cell is selected from "State" which is a discrete attribute with high dependency connectivity. Clearly, when $k = 0$, i.e., full leakage, *Our Approach* will only hide sensitive cells and when $k = 1$ i.e, Full deniability, *Our Approach* hides the maximum number of cells. When $k = 0.5$, i.e., the inferred set of values is half of that of the base view, *Our Approach* hides almost the same number of cells as $k = 1$ i.e., full deniability. When $k = 0.1$, i.e, the inferred set of values is $\frac{1}{10}$ of that of the base view, *Our Approach* hides $\approx 15\%$ less cells than the one that implements full deniability. On the *Hospital* dataset (results included in extended version), the utility improvement was marginal with k set to the smallest value possible (besides full leakage) i.e., $k = \frac{1}{|Dom(c^*)|}$. We note that *Our Approach* that implements full deniability is able to provide high utility with a stronger security model on both datasets compared to the one that implements k-percentile deniability.

Second, we study the effectiveness of *Our Approach* against inference attacks, i.e., to what extent can an adversary reconstruct the sensitive cells in a given querier view. We consider two types of adversaries. The first type of adversary uses *weighted sampling* where for each sensitive cell c^*, the adversary learns the distribution of values in $Dom(c^*)$ by looking at the values of other cells in the view. The querier, then tries to infer the sensitive cell value by sampling from this learned distribution. The second type of adversary utilizes a state-of-the-art data cleaning system, Holoclean [29], which compiles data dependencies, domain value frequency, and attribute co-occurrence and uses them into training a machine learning classifier. The adversary then leverages this classifier to determine values of sensitive cells by considering them as missing data in the database. The sensitive cell for this experiment is selected from "State" which is a discrete attribute with high dependency connectivity. We consider the 10 dependencies and drop the FC because Holoclean doesn't support it. We increase the number of policies from 10 to 90 and input the querier view (in which the values of hidden cells are replaced with *NULL*) to both adversaries. We measure the effectiveness by repair precision $= \frac{\#\text{correct repairs}}{\#\text{total repairs}}$ (where a *repair* is an adversary's guess of the value of a hidden cell) and therefore lower the *repair precision* of the adversary is, the more effective *Our Approach* is.

The results "Holoclean (before)" in Figure 6 (b) show that when only sensitive cells are hidden, an adversary such as Holoclean, is able to correctly infer the sensitive cells. When additional cells are hidden by *Our Approach*, indicated by "Holoclean (after)", the maximum precision of Holoclean is 0.15. On the other hand, the weighted sampling employed by the other type of adversary, indicated by "Weighted Sampling (after) ", could reconstruct between 3% and 10% of the sensitive cells. Note that Holoclean uses the learned data correlations (and attribute co-occurrence, domain value frequency) in addition to the explicitly stated data dependencies. However, it is only able to marginally improve upon weighted random sampling given the querier view generated by *Our Approach*.

8 RELATED WORK

The challenge of preventing leakage of sensitive data from query answers has been studied in prior work on inference control primarily in the context mandatory access control (MAC) wherein policies specify the security clearances for the users (subject) and the security classification/label for the data. Early work by Denning *et al.* [9] designed commutative filters to ensure answers returned by a query are equivalent to that which would be returned based on authorized view for the user. This work, however, did not consider data dependencies.

Preventing leakage through dependencies has been explored along different directions. One such direction is to control inferences by design time modifications by adding more MAC policies. Qian *et al.* [28] developed a tool to analyze potential leakage due to foreign keys in order to elevate the clearance level of data if such a leakage is detected. Delugachi *et al.* [8] generalized the work in [28] and developed an approach based on analyzing a conceptual graph to identify potential leakage from more general types of data associations (e.g., part-of, is-a). Later works such as [35], however, established that inference rules for detecting inferences at database design time is incomplete and hence is not a viable approach for preventing leakage from query answers. Design time approaches for disclosure control, however, have successfully been used in restricted settings such as identifying the maximal set of non-sensitive data to outsource such that it prevents inferences about sensitive data [10, 18, 23, 24].

Prior work has also explored query time approaches to prevent inferences from data dependencies. Thuraisingham [32] developed a *query control approach* in the context of MAC policies that uses an inference engine to determine if query answers can lead to leakage (in which case the query is rejected). While [32] assumed a prior existence of an inference detection engine, Brodsky *et al.* [3] developed a framework, entitled DiMon, based on chase algorithm for constraints expressed as Horn clauses. DiMon takes in current query results, user's query history, and Horn clause constraints to determine the additional data that maybe inferred by the subject. Similar to [32], if inferred data is beyond the security clearance of the subject then their system refuses the query. Such work (that identify if a query leaks/does not leak data) differs from ours since it cannot be used be used directly to identify a maximal secure answer that does not lead to any inferences — the problem we study in this paper. Also, the above work on query control is based on a much weaker security model compared to the full-deniability model we use. It only prevents adversary from reconstructing the exact value of a sensitive cell but cannot prevent them from learning new information about the sensitive cell.

Miklau & Suciu [21] also study the challenge of preventing information disclosure for a secret query given a set of views. Their security model is based on perfect secrecy as they characterize whether there exists any possible database instance for which information disclosure is possible through sharing of views. Our problem setting is different as we check for a given database instance whether it is possible to answer the query hiding as few cells as possible while ensuring full deniability. We could, of course, apply [21] to check if a query is unsafe and in such a case disallow the query. Such a usage of [21] will be extremely pessimistic as most queries will be rejected for a database with non-trivial number of dependencies.

Differential Privacy (DP) mechanisms promises to protect against an adversary with any prior knowledge [12]. In our problem setting of access control, the data is either hidden or shared depending upon whether it is sensitive which differs from querier to querier. This is called the *Truman model* of access control [30], in which the query is answered based on non-sensitive data. In such a model, the expectation of a querier is that the result doesn't include any randomized answers. However, DP based mechanisms involve randomized answers. Weaker notions of DP such as One-sided differential privacy (OSDP) [20] aim to prevent inferences on sensitive data by using a randomized mechanism when sharing non-sensitive data. An advantage of OSDP based approaches in the context of access control is that they offer security guarantees without the knowledge of (or the need to) explicitly specify data dependencies. However, such techniques offer only probabilisitic guarantees (and cannot implement security guarantees such as full deniability), and that too with the suppression of large amount of data. From the results on the expected percentage of released non-sensitive data as a function of privacy parameter ϵ presented in [20], OSDP suppresses approx. 91% non-sensitive data at $\epsilon = 0.1$ and approx. 37% at $\epsilon = 1$. In contrast, our techniques requires only a very small percentage of data to be suppressed by exploiting data dependencies explicity in inference control. OSDP may allow some non-sensitive data to be released based on a coin toss even when their values could lead to leakage of a sensitive cell as it doesn't take into account data dependencies. The current model of OSDP only supports hiding at the row level and is designed for scenarios where the whole tuple is sensitive or not. It is non-trivial to extend to suppress cells with fine-grained access control policies considered in our setting. Furthermore, most DP-based mechanisms (including OSDP) assume that no tuple correlations exist even through explicitly stated data dependencies.

9 CONCLUSIONS

We studied the inference attacks on access control protected data through data dependencies, DCs and FCs. We developed a new stronger security model called *full deniability* which prevents a querier from learning about sensitive cells through data dependencies. We presented conditions for determining leakage on sensitive cells and developed algorithms that uses these conditions to implement full deniability. The experiments show that we are able to achieve full deniability for a querier view without significant loss of utility for two different datasets. In future, we would like to extend the security model to not only consider hard constraints explicitly specified in the form of data dependencies but also soft constraints that exist as correlations between data items. The invertibility model in FCs could also be extended to model probabilistic relationship between input and output cells, instead of being deterministic as in the current model. Improving utility while implementing full deniability is also an open challenge. In k-percentile deniability, the improvement in utility as a factor of k needs to be studied further as factor of different properties of the dataset such as type of attributes, dependency connectivity, dependency instantiations, and possible number of cuesets for a given sensitive cell. Along with further exploration of k-percentile deniability considered in our paper, one could also consider releasing non-sensitive values (like in OSDP) randomly instead of hiding all. However, this requires addressing challenges of any inadvertent leakages through dependencies when sharing such randomized data.

ACKNOWLEDGEMENTS

This work was supported by NSF under Grants 2032525, 1952247, 2008993, and 2133391. This material was based on research sponsored by DARPA under Agreement Number FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation there on. The views and conclusions contained here in are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. We thank the reviewers for their detailed comments which helped to improve the paper during the revision process.

REFERENCES

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2002. Hippocratic Databases. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002.* Morgan Kaufmann, Hong Kong, China, 143–154. https://doi.org/10.1016/B978-155860869-6/50021-4

[2] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional Functional Dependencies for Data Cleaning. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007.* IEEE Computer Society, Istanbul, Turkey, 746–755. https://doi.org/10.1109/ICDE.2007.367920

[3] Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. 2000. Secure Databases: Constraints, Inference Channels, and Monitoring Disclosures. *IEEE Trans. Knowl. Data Eng.* 12, 6 (2000), 900–919. https://doi.org/10.1109/69.895801

[4] Jiayi Chen, Jianping He, Lin Cai, and Jianping Pan. 2020. Disclose More and Risk Less: Privacy Preserving Online Social Network Data Sharing. *IEEE Trans. Dependable Secur. Comput.* 17, 6 (2020), 1173–1187. https://doi.org/10.1109/TDSC.2018.2861403

[5] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *Proc. VLDB Endow.* 6, 13 (2013), 1498–1509. https://doi.org/10.14778/2536258.2536262

[6] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *29th IEEE International Conference on Data Engineering, ICDE 2013*. IEEE Computer Society, Brisbane, Australia, 458–469. https://doi.org/10.1109/ICDE.2013.6544847

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Cambridge, MA, USA.

[8] Harry S. Delugach and Thomas H. Hinke. 1996. Wizard: A Database Inference Analysis and Detection System. *IEEE Trans. Knowl. Data Eng.* 8, 1 (1996), 56–66. https://doi.org/10.1109/69.485629

[9] Dorothy E. Denning. 1985. Commutative Filters for Reducing Inference Threats in Multilevel Database Systems. In *1985 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 22-24, 1985.* IEEE Computer Society, Oakland, CA, USA, 134–146. https://doi.org/10.1109/SP.1985.10017

[10] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Giovanni Livraga, Stefano Paraboschi, and Pierangela Samarati. 2014. Fragmentation in Presence of Data Dependencies. *IEEE Trans. Dependable Secur. Comput.* 11, 6 (2014), 510–523. https://doi.org/10.1109/TDSC.2013.2295798

[11] Irit Dinur and Samuel Safra. 2005. On the hardness of approximating minimum vertex cover. *Annals of mathematics* 162, 1 (2005), 439–485. https://doi.org/10.4007/annals.2005.162.439

[12] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.* 9, 3-4 (2014), 211–407. https://doi.org/10.1561/0400000042

[13] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)* 33, 2 (2008), 1–48. https://doi.org/10.1145/1366102.1366103

[14] Csilla Farkas and Sushil Jajodia. 2002. The Inference Problem: A Survey. *SIGKDD Explor.* 4, 2 (2002), 6–11. https://doi.org/10.1145/772862.772864

[15] Elena Ferrari. 2010. Access control in data management systems. *Synthesis lectures on data management* 2, 1 (2010), 1–117. https://doi.org/10.1007/978-3-031-01836-7

[16] Thomas D. Garvey, Teresa F. Lunt, Xiaolei Qian, and Mark E. Stickel. 1993. Toward a Tool to Detect and Eliminate Inference Problems in the Design of Multilevel Databases. In *Results of the Sixth Working Conference of IFIP Working Group 11.3 on Database Security, VI: Status and Prospects: Status and Prospects* (Simon Fraser Univ., Vancouver, British Columbia, Canada). Elsevier Science Inc., USA, 149–167.

[17] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC Data-Cleaning Framework. *Proc. VLDB Endow.* 6, 9 (2013), 625–636. https://doi.org/10.14778/2536360.2536363

[18] Mehdi Haddad, Jovan Stevovic, Annamaria Chiasera, Yannis Velegrakis, and Mohand-Said Hacid. 2014. Access Control for Data Integration in Presence of Data Dependencies. In *Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014 (Lecture Notes in Computer Science)*, Vol. 8422. Springer, Bali, Indonesia, 203–217. https://doi.org/10.1007/978-3-319-05813-9_14

[19] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB J.* 31, 1 (2022), 1–22. https://doi.org/10.1007/s00778-021-00676-3

[20] Ios Kotsogiannis, Stelios Doudalis, Samuel Haney, Ashwin Machanavajjhala, and Sharad Mehrotra. 2020. One-sided Differential Privacy. In *36th IEEE International Conference on Data Engineering, ICDE 2020.* IEEE, Dallas, TX, USA, 493–504. https://doi.org/10.1109/ICDE48307.2020.00049

[21] Gerome Miklau and Dan Suciu. 2004. A Formal Analysis of Information Disclosure in Data Exchange. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, 2004.* ACM, Paris, France, 575–586. https://doi.org/10.1145/1007568.1007633

[22] State of California Department Justice Office of the Attorney General. 2020. California Consumer Privacy Act CCPA. https://oag.ca.gov/privacy/ccpa. [Online; accessed 01-Jul-2022].

[23] Kerim Yasin Oktay, Murat Kantarcioglu, and Sharad Mehrotra. 2017. Secure and Efficient Query Processing over Hybrid Clouds. In *33rd IEEE International Conference on Data Engineering, ICDE 2017.* IEEE Computer Society, San Diego, CA, USA, 733–744. https://doi.org/10.1109/ICDE.2017.125

[24] Kerim Yasin Oktay, Sharad Mehrotra, Vaibhav Khadilkar, and Murat Kantarcioglu. 2015. SEMROD: Secure and Efficient MapReduce Over HybriD Clouds. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* ACM, Melbourne, Victoria, Australia, 153–166. https://doi.org/10.1145/2723372.2723741

[25] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data Profiling with Metanome. *Proc. VLDB Endow.* 8, 12 (2015), 1860–1863. https://doi.org/10.14778/2824032.2824086

[26] Primal Pappachan, Roberto Yus, Sharad Mehrotra, and Johann-Christoph Freytag. 2020. Sieve: A Middleware Approach to Scalable Access Control for Database Management Systems. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2424–2437. https://doi.org/10.14778/3407790.3407835

[27] Primal Pappachan, Shufan Zhang, Xi He, and Sharad Mehrotra. 2022. Don't Be a Tattle-Tale: Preventing Leakages through Data Dependencies on Access Control Protected Data. https://doi.org/10.48550/ARXIV.2207.08757

[28] Xiaolei Qian, Mark E. Stickel, Peter D. Karp, Teresa F. Lunt, and Thomas D. Garvey. 1993. Detection and elimination of inference channels in multilevel relational database systems. In *1993 IEEE Computer Society Symposium on Research in Security and Privacy.* IEEE Computer Society, Oakland, CA, USA, 196–205. https://doi.org/10.1109/RISP.1993.287632

[29] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

[30] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. 2004. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France). Association for Computing Machinery, New York, NY, USA, 551–562. https://doi.org/10.1145/1007568.1007631

[31] Kenneth A. Ross, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1998. Foundations of aggregation constraints. *Theoretical Computer Science* 193, 1-2 (1998), 149–179. https://doi.org/10.1016/S0304-3975(97)00011-X

[32] Bhavani M. Thuraisingham. 1987. Security checking in relational database management systems augmented with inference engines. *Comput. Secur.* 6, 6 (1987), 479–492. https://doi.org/10.1016/0167-4048(87)90029-0

[33] Paul Voigt and Axel Von dem Bussche. 2017. The EU general data protection regulation (GDPR). *A Practical Guide, 1st Ed., Cham: Springer International Publishing* 10 (2017), 3152676.

[34] California Legislative Information website. 2020. California Online Privacy Protection Act (CalOPPA). https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=BPC&sectionNum=22575. [Online; accessed 01-Jul-2022].

[35] Raymond W. Yip and Karl N. Levitt. 1998. Data Level Inference Detection in Database Systems. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop.* IEEE Computer Society, Rockport, Massachusetts, USA, 179–189. https://doi.org/10.1109/CSFW.1998.683168