12-2023

Preventing Inferences through Data Dependencies on Sensitive Data

Primal Pappachan
Portland State University

Shufan Zhang
University of Waterloo

Xi He
University of Waterloo

Sharad Mehrotra
University of California, Irvine

Citation Details

Pappachan, P., Zhang, S., He, X., & Mehrotra, S. (2023). Preventing Inferences through Data Dependencies on Sensitive Data. IEEE Transactions on Knowledge & Data Engineering, (01), 1-18.

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

Preventing Inferences through Data Dependencies on Sensitive Data

Primal Pappachan, *Member, IEEE,* Shufan Zhang, *Student Member, IEEE,* Xi He, *Member, IEEE,* and Sharad Mehrotra, *Fellow, IEEE*

Abstract—Simply restricting the computation to non-sensitive part of the data may lead to inferences on sensitive data through data dependencies. Inference control from data dependencies has been studied in the prior work. However, existing solutions either detect and deny queries which may lead to leakage – resulting in poor utility, or only protects against exact reconstruction of the sensitive data – resulting in poor security. In this paper, we present a novel security model called *full deniability*. Under this stronger security model, any information inferred about sensitive data from non-sensitive data is considered as a leakage. We describe algorithms for efficiently implementing full deniability on a given database instance with a set of data dependencies and sensitive cells. Using experiments on two different datasets, we demonstrate that our approach protects against realistic adversaries while hiding only minimal number of additional non-sensitive cells and scales well with database size and sensitive data.

Index Terms—Inference Control, Data Dependencies, Inference Protection, Security & Privacy, Access Control

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) [2], the California Online Privacy Protection Act (CalOPPA) [3], and the Consumer Privacy Act (CCPA) [4], 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 [5] 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 [6], [7] or view-based mechanisms [8]. 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 [9], [10]. 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 an 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 dictates 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 types of data dependencies, such as conditional functional dependencies (CFD) [11], denial constraints [12], aggregation constraints [13], and/or *functional constraints* that exist when 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 [12], [14] 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 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 a large number of data dependencies defined on them (e.g., state) to be sensitive, and thus, replaced

- *A preliminary version of this article was accepted and presented in VLDB 2022 [1].*
- *P. Pappachan is with Portland State University, Portland, OR 97201. E-mail: primal@pdx.edu.*
- *S. Zhang and X. He are with the University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1. E-mail: {shufan.zhang, xi.he}@uwaterloo.ca.*
- *S. Mehrotra is with University of California, Irvine, CA 92697 USA. E-mail: sharad@ics.uci.edu.*

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

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

Fig. 1. Employee and Wages Table

its values by *NULL*. We then used state-of-the-art data-cleaning software, Holoclean [15], 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". [9]. Existing techniques used to protect against inferences can be categorized based on when the leakage prevention is applied [16]. In the first category, inference channels between sensitive and non-sensitive attributes are detected and controlled at the time of database design [17], [18]. 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 work includes detection and control at the time of query answering. Works such as [16], [19] 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 the rejection of many queries when there is a 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 the querier/adversary could eliminate a significant number of possible domain values based on the data constraints. As we explain in detail in Section 9, 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 an 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 achieve 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 a 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 store 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 a 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 a 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 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 of 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 relaxed *k-percentile deniability* model, relaxations of security assumptions, and algorithms to achieve these relaxations.
- A prototype middleware (∼10K LOC) that works alongside DBMS to ensure full deniability given a set of dependencies and policies.
- Experimental results on two different data sets show that our approach is efficient and only minimally hides non-sensitive

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.

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 6, we extend the full deniability model to k-percentile deniability and in Section 7 we relax the security assumptions in our model. In Section 8, we present results from an end-to-end evaluation of our approach with two different data sets and different baselines. In Section 9 we go over the related work and we conclude the work by summarizing our contributions, and possible future extensions in Section 10.

Comparison to Conference Version. In this version, new contents include 1) novel algorithms for improving scalability and utility, i.e., a binning-then-merging algorithm to scale up inference protection and algorithms to achieve a weaker k-deniability security notion; 2) a detailed study of relaxing the assumptions w.r.t adversary presented in the preliminary version along with modified algorithms to achieve full deniability under new settings; 3) more ablation experiments for evaluating performance and utility under different settings; 4) expanded related work along with more details on the datasets and models used for experiments.

2 Preliminaries

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".

2.1 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). We assume queries have associated metadata that contains information about the querier [3].

Query model. The SELECT-FROM-WHERE query posed by a user U is denoted by Q. In our model, we consider that queries have associated metadata which consists of information about the querier and the context of the query. This way, we assume that for any given query Q, it contains the metadata such as the identity of the querier (i.e., $Q^{querier}$) as well as the purpose of the query (i.e., $Q^{purpose}$). For example, $Q^{querier}$="John" and $Q^{purpose}$="Analytics".

Policy model. A policy P is expressed as <*OC, SC, AC*>, where AC corresponds to the action, i.e., either deny or allow, SC corresponds to the subject condition i.e, the user to whom the policy applies (e.g., the identity of the querier, or the group for which the policy applies, in case queriers are organized into groups), and OC corresponds to a set of object conditions that identifies the cells on which the policy is to be enforced. Each object condition OC_i is represented using the following 3-tuple: $\{\mathbb{R}, \sigma, \Phi\}$ where $\mathbb{R}$ is the relation, σ and Φ are the selection and projection conditions respectively that together select the cells that are sensitive. The application of a 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. This is modelled after FGAC policy models used in previous works [7], [21]. We denote the set of cells identified by OC_i as $\mathbb{C}_{OC_i}$.

Definition 1 (Sensitive Cell)**.** *Given a policy $P =$<OC, SC, AC>, we say that a cell c is sensitive to a user U if $c \in \mathbb{C}_{OC_i}$ where $OC_i \in OC$, $U = SC.querier$, and $AC = deny$. 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 context is clear.*

Example 3. An example policy from scenario in Section is <{*Employee, EName = "Carrie Sea", SalPerHr*}, {*"John Doe"*, }, {*deny*}>. The policy specifies that the salary information (*SalPerHr*) of Employee Carrie (*EName = "Carrie Sea"*) in the *Employee* table should be denied (i.e., it is sensitive) to the *Querier = "John Doe"* . □

2.2 Data Dependencies

The 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 the 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 have been used in recent prior literature for data cleaning [22], [23], query optimization [24], and secure databases [16], [25]. Moreover, systems, such as [12], have

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.

3. In general, policies control access to data based not just on the identity of the querier, but also on purpose [20]. Thus, metadata associated with the query will also contain purpose in addition to the querier identity.

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

been designed to automatically discover DCs in a given database. This is the type of DCs considered throughout the paper. We used a data profiling tool, Metanome [26] to identify the complete set of denial constraints.

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 $r_1, r_2, \ldots, r_n$ as the input values and s_i as the derived or output value, the FC can be represented by $fn(r_1, r_2, \ldots, r_n) = s_i$.

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). To see why this is important, consider a FD defined on the *Employee* table (in Fig. 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 algorithm 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 2 (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:

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

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

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

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 $\tilde{\delta}(\ldots, c_i = x_i, \ldots)$, 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.

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

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

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 4. 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 (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*

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

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 4 (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 $\tilde{\delta} = \neg(Pred_1 \land \ldots \land Pred_N)$ 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 4, 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 5. Suppose, in Example 4, 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 $c_{11} = c_{18}$, 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 5 (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.

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 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*. The proof is inclduded in the appendix.

Theorem 1. *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*

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

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

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 1. *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., $\mathbb{I}(c^* \mid \mathbb{V}, S_\Delta) = \mathbb{I}(c^* \mid \mathbb{V}_0, S_\Delta)$.*

Proof. From Theorem 1, we know that for each $\tilde{\delta}_i \in S_\Delta$ when the TTC is False, we have $\mathbb{I}(c^*|\mathbb{V}, \tilde{\delta}_i) = \mathbb{I}(c^*|\mathbb{V}_0, \tilde{\delta}_i)$. As each individual set based on individual dependency instantiation are equal in both the released view and base view, the joint set of values in both views computed by the intersection of all the sets should also be equal i.e., $\bigcap_{\tilde{\delta}_i \in S_\Delta} \mathbb{I}(c^*|\mathbb{V}, \tilde{\delta}_i) = \bigcap_{\tilde{\delta}_i \in S_\Delta} \mathbb{I}(c^*|\mathbb{V}_0, \tilde{\delta}_i)$. According to Equation 3, this joint set is the final inferred set of values for c^* based on S_Δ in a given view and as they are equal we have $\mathbb{I}(c^* \mid \mathbb{V}, S_\Delta) = \mathbb{I}(c^* \mid \mathbb{V}_0, S_\Delta)$. □

If the dependency $\tilde{\delta}$ only contains a single predicate, the Tattle-Tale condition evaluates to True even in $\mathbb{V}_0$ when all the cells are hidden $TTC(\tilde{\delta}, \mathbb{V}_0, c_i)$ =*True* in the cases of *Pred*(c_i) and therefore it is not possible to prevent querier from learning about the truth value of the sensitive predicate.

4.2 Selecting Cells to Hide

As shown in Theorem 1, 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 6 (Cueset)**.** *Given an instantiated DC $\tilde{\delta}$, a cueset for a cell $c \in cells(\tilde{\delta})$ is defined as*

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

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 6. In the instantiated DC from Example 4, the cueset for c_{14} based on $\tilde{\delta}_4$ is $cueset(c_{14}, \tilde{\delta}_4) = \{c_4, c_{11}, c_5, c_{12}\}$. □

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.

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. The proof is included in the appendix.

Theorem 2 (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

To study the leakages due to function-based constraints (FCs), we define the property of invertibilty associated with functions.

Definition 7 (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: $\tilde{\delta} : \neg(c_6 = 20 \wedge c_7 = 40 \wedge c_{31} \neq 800)$ where c_6, c_7, 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.

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.

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

As the instantiation for FC is in DC form and their Tattle-Tale Conditions and cueset determination are almost identical, in

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

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

Algorithm 1: Full Algorithm

Input: User U, Data dependencies S_Δ, 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** $\mathbb{V}_S$

Algorithm 2: Inference Detection

Input: A set of sensitive cells $\mathbb{C}^S$, Schema-level data dependencies S_Δ, A view of the database $\mathbb{V}$
Output: A set of cuesets $cuesets$

1 **Function** `InferenceDetect`($\mathbb{C}^S$, S_Δ, $\mathbb{V}$)**:**
2   cuesets = { }
3   **for** $c^* \in \mathbb{C}^S$ **do**
4   S_{S_Δ} = { }   ▷ Set of instantiated dependencies.
5   **for** $\delta \in \Delta$ **do**
6   S_{S_Δ} = S_{S_Δ} $\cup$ **DepInstantiation**(δ, c^*, $\mathbb{V}$)
7   **end**
8   **for** $\tilde{\delta} \in S_\Delta$ **do**   ▷ For each instantiated dependency.
9   **if** $|Preds(\tilde{\delta})| = 1$ **then**
10   cueset = $\{c_k\}$ ▷ Note: $Pred(c^*) = c^*\theta c_k$
11   **else if** $TTC(\tilde{\delta}, \mathbb{V}, c^*) = False$ **then**
12   **continue**
13   **else**
14   cueset = $cells(Preds(\tilde{\delta} \backslash c))$
15   **end**
16   cuesets.add(cueset)
17   **end**
18   **end**
19   **return** *cuesets*

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).

Remark. We extend the invertibility notion to a more general model, i.e., (m,n)-*invertibility*, that can capture the partial leakage due to function-based constraints. The details for this notion and computing partial leakage according to (m,n)-invertibility can be found in supplementary materials.

5 Algorithm to Achieve Full Deniability

In this section, we present an algorithm to determine the set of cells to hide to achieve full-deniability based on Theorem 2. Full-deniability can trivially be achieved by sharing the base view $\mathbb{V}_0$ where all cell 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

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 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 cover 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 return this secure view to the user (Steps 13-16). The following theorem 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 2).

Theorem 3. *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* $\geq$ *1 cell from all the non-empty cuesets of cells in* $\mathbb{C}^H$*).*

Proof. By contradiction, we suppose there exists a cueset $cs \in cueset(c_i, \tilde{\delta})$ in which no cell is not hidden. This means the cueset cs has no overlap with the hidden cell set $\mathbb{C}^H$. Then by lines 6-7 in Algorithm 1, the cueset cs exists in the cueset list $cueset(c_i, \tilde{\delta})$, which indicates that the While loop will not terminate. This contradicts the pre-assumed condition. □

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

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

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 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 $\tilde{\delta}_3 : \neg((c_1 = c_4) \wedge (c_3 = c_6))$. 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. □

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.

Query-based method. For each dependency and each sensitive cell, inference detection instantiates the dependency to generate $|\mathbb{D}| - 1$ instantiations. The algorithm then iterates over each instantiation and checks the Tattle-tale condition and if satisfied generates a cueset. The inference detection algorithm will be time and space-intensive given a substantial number of dependencies and/or sensitive cells. To improve upon this, we propose a query-based technique for implementing inference detection.

Instead of generating one instantiation per sensitive cell and dependency, this method produces one query for all the sensitive cells. First, this method retrieves the tuples containing sensitive cells, sets the values of sensitive cells to *NULL* and stores them in a temporary table called ***temp***. Next, the Tattle-tale condition check is turned into a join query between this ***temp*** table and the original table.

The join condition in this query is based on the tuples being unique (*T1.tid* $\neq$ *T2.tid*). Furthermore, this query checks for each relevant attribute in the tuple whether it is sensitive i.e., it is set to *NULL* in the *temp* table (*T2.Zip is NULL*), or whether the corresponding predicate from the dependency evaluates to *True* (*T1.Zip=T2.Zip*). The *WHERE* condition in this query is only satisfied if all the predicates in a dependency instantiation except for the sensitive predicate evaluate to *True*. Thus, the result of this join query contains all instantiations for which the Tattle-tale condition evaluates to *True* from which the cuesets can be readily identified.

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

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 show algorithms to achieve so and prove the correctness of this modification in Section 7

Algorithm 3: Inference Protection (Minimum Vertex Cover)

Input: Set of cuesets *cuesets*
Output: A set of cells selected to be hidden *toHide*

```
1  Function InferenceProtect(cuesets):
2      toHide = {}                          ▷ Return list initialization.
3      while cuesets ≠ φ do
4          cuesetCells = Flatten(cuesets)
5          dict[c_i, freq_i] =
             CountFreq(GroupBy(cuesetCells))
6          cellMaxFreq = GetMaxFreq(dict[c_i, freq_i])
7          toHide.add(cellMaxFreq)          ▷ Greedy heuristic.
8          for cs ∈ cuesets do
9              if cs.overlaps(toHide) then
10                 cuesets.remove(cs)
11         end
12     end
13     return toHide
```

5.3 Inference Protection

After identifying the cuesets for each sensitive cell based on their dependency instantiations, we now have to select a cell from each of them to hide to prevent leakages. The first strategy for cell selection, described in Algorithm 7, randomly selects a cueset and a cell from it to hide (if no cells in it have been hidden already). We use this approach as our first baseline (*Random Hiding*) in Section 8. The second strategy for cell selection, described in Algorithm 3 utilizes Minimum Vertex Cover (MVC) [27] to minimally select the cells to hide from the list of cuesets. In this approach, each cueset is considered as a hyper-edge and the cell selection strategy finds the minimal set of cells that covers all the cuesets. MVC is known to be NP-hard [28] 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 the 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 that 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

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

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 $\mathcal{O}(|\Delta|(N+s_i))$ (where inference detection is implemented using the query-based method given sufficient, i.e. $\Theta(N)$, memory) and (ii) minimum vertex cover (MVC) with an I/O cost of $\mathcal{O}(cs_i)$. Hence, the overall estimated I/O cost $\sum_{i=1}^{T} \mathcal{O}(|\Delta|(N+s_i)) + \mathcal{O}(cs_i)$ in which is equivalent to $\mathcal{O}(N)$ given $T \leq \log_{f/m}(N/s)$ and thus is linear to the data size.

The cost of the dependency instantiation for the ith iteration depends on the I/O cost of the join query which is $\mathcal{O}(N+s_i)$ when given sufficient (i.e., $\Theta(N)$) memory. This query is executed $|\Delta|$ times. Hence, the cost for the dependency instantiation is $\mathcal{O}(|\Delta|(N+s_i))$.

Hence, the total estimated I/O cost for T iterations can be derived as follows given $T \leq \log_{f/m}(N/s)$.

$$
\begin{aligned}
& \sum_{i=1}^{T}(|\Delta|(N+s_i)) + c_i \\
= \quad & |\Delta|(N + s\sum_{i=1}^{T}(f/m)^i) + sf\sum_{i=1}^{T}(f/m)^{i-1} \\
\leq \quad & |\Delta|(N + s(f/m)^{T+1}) + sf(f/m)^T \\
= \quad & |\Delta|(N + s(N/s)(f/m)) + sf(N/s) \\
= \quad & N|\Delta|(1 + f/m) + fN
\end{aligned}
$$

We complement the complexity analysis with the required sufficient memory storage discussion. For (i) dependency instantiation, the join query between two tables of size N and s_i, we need memory size $\Omega(N+s_i) = \Omega(N)$ since $s_i \leq N$. In (ii) the algorithm of computing MVC, all cuesets are read into the memory, which requires the memory size $\Omega(c_i) = \Omega(N*m) = \Omega(N)$ for constant m. Thus we need $\Omega(N)$ memory to finish all operations in our system implementation, which is feasible in practice. We also note that this complexity analysis only holds with $\Theta(N)$ size of memory, in which case the cost of memory operations is much cheaper than the overhead of I/O operations. Given $\Omega(N^2)$ memory, which can be *impractical*, all the operations can then be finished within memory and the total computational cost is bounded by $\mathcal{O}(N^2)$, according to the following analysis.

If all operations are taken within memory, then the cost of dependency instantiation is bounded by $\mathcal{O}(Ns_i)$ and the computational cost of the MVC algorithm is bounded by $\mathcal{O}(c_i^2)$. Then we derive the following bound similarly.

$$
\begin{aligned}
& \sum_{i=1}^{T}(N|\Delta|s_i) + c_i^2 \\
= \quad & N|\Delta|s\sum_{i=1}^{T}(f/m)^i + s^2f^2\sum_{i=1}^{T}(f/m)^{2i-2} \\
\leq \quad & N|\Delta|s(f/m)^{T+1} + s^2f^2(f/m)^{2T} \\
= \quad & N|\Delta|s(N/s)(f/m) + s^2f^2(N/s)^2 \\
= \quad & N^2|\Delta|(f/m) + f^2N^2
\end{aligned}
$$

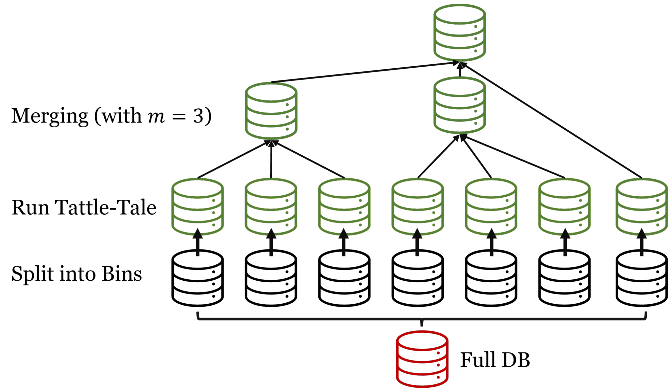

Fig. 2. An Illustration of the Binning-then-Merging Algorithm (with binning size $b = 7$ and merging size $m = 3$).

5.5 Wrapper for Scaling out Full-Deniability Algorithm

The complexity analysis above shows that, given sufficient memory, full deniability algorithm is linear to the size of the database. On larger databases, the memory requirement becomes unsustainable due to the substantial number of dependency instantiations and cuesets. We present a wrapper which partitions the database in order so that our algorithm is able to run with a smaller memory footprint. The high-level idea of the algorithm is illustrated in Figure 2.

Algorithm 4 partitions the full database into a number of *bins*, where b is the bin size parameter. It then calls the *Full Algorithm* (presented in Section 5.1 and denoted by `runMain()` in Algorithm 4) on each of these bins in order to generate a view per bin that satisfies full deniability. As the full algorithm is executed on smaller bins, the memory requirement is much lower than the entire database. Next, it merges m number of these bins, where m is the merge size parameter, and executes *Full Algorithm* on the merged bins. The wrapper iterates over the merged bins until there is only 1 bin left. It then executes *Full Algorithm* on this last bin which is full database and the final view that satisfies full deniability is returned. As each of the bins has achieved full deniability, the number of relevant dependency instantiations and cuesets will be much lower in the merged bin compared to running the full algorithm on the entire database. The output view generated by Algorithm 4 trivially satisfies full-deniability as the *Full Algorithm* is executed on each of the individual bins as well as the full database in the final step.

6 Weaker Security Model

Achieving full deniability on a view can lead to hiding a number of non-sensitive cells to prevent leakages. In this section we describe how 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.

6.1 k-Percentile Deniability

The weaker security notion of k-Percentile Deniability is defined as follows.

Definition 8 (k-percentile Deniability)**.** *Given a set of sensitive cells $\mathbb{C}^S$ in a database instance $\mathbb{D}$ and a set of instantiated*

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

Algorithm 4: Binning-then-Merging Wrapper Algorithm

Input: User U, Data dependencies S_Δ, A view of the database $\mathbb{V}$, Bin size b, Merge size m
Output: A secure view $\mathbb{V}_S$

Function `BinningThenMerging` $(U, S_\Delta, \mathbb{V}, b, m)$ **:**
  $\mathbb{V}_1, \ldots, \mathbb{V}_k \leftarrow$ **Binning**$(\mathbb{V}, b) \triangleright k := \frac{\|\mathbb{V}\|}{b}$, no. of bins.
  binQueue = $[\mathbb{V}_1, \ldots, \mathbb{V}_k]$
  mergeQueue = { }
  while $\|binQueue\| \neq 1$ *or* $mergeQueue \neq \emptyset$ **do**
    $\mathbb{V}_i \leftarrow$ binQueue.**pop**()
    mergeQueue.**push**(**runMain**(U, S_Δ, $\mathbb{V}_i$))
    if $\|mergeQueue\| \geq m$ *or* $\|binQueue\| = 0$ **then**
      $\mathbb{V}_j \leftarrow$ **Merge**(mergeQueue)
      binQueue.**push**(**runMain**(U, S_Δ, $\mathbb{V}_j$))
      mergeQueue.**clear**()
  end
  return *binQueue.**pop**()*

dependencies S_Δ, we say that a querier view $\mathbb{V}$ achieves k-percentile deniability if for all $c^* \in \mathbb{C}^S$,

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

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 as defined above. For instance, the model used in [16] 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.

6.2 Algorithms to Achieve k-Percentile Deniability.

In k-percentile deniability or simply *k-den*, we quantify the leakage on the sensitive cell in a given view $\mathbb{V}$ and the set of instantiated data dependencies S_Δ. The decision to hide additional cells is only made if the set of possible values inferred by the querier is larger than the given threshold (k). Stated differently, given the fan-out tree of the cuesets as selected in the full-deniability algorithm, we can *prune* some of the cuesets at the first fan-out level based on this threshold.

To show this, we need to first find a good representation of the set of inferred values for a cell. The set of inferred values for a c^* given by the $\mathbb{I}(c \mid \mathbb{V}, S_\Delta)$ (defined in Section 3.3) can be represented as follows

$$\mathbb{I}(c \mid \mathbb{V}, S_\Delta) = \begin{cases} minus\_set, & Dom(c) \text{ is discrete} \\ Dom(c) - [low, high] & Dom(c) \text{ is continuous} \end{cases}$$

When attribute for the cell c is discrete, the operator θ in *Pred*(c) is limited to either $\neq$ or $=$. Therefore, we represent the inferred set of values by a set, called minus_set, containing the values that cannot be assigned to the cell in the view $\mathbb{V}$. On the other hand, when the attribute for the cell c is continuous, the operator θ could be one of the following: $\{>, \geq, <, \leq\}$ and therefore we use a range, denoted by $(low, high)$ to represent the set of values. Computing the set of inferred values for a cell is relatively easy and due to space constraints, we deferred the details to supplementary materials.

This function computes the exact leakage on a sensitive cell with respect to various instantiated dependencies. We utilize this to implement k-den where for each sensitive cell after we detect the cuesets (Step 3 in Algorithm 1), we compare the leakage on a sensitive cell due to the instantiated dependencies (associated with the cuesets). The k parameter, specified as a fraction of the maximum domain size of a sensitive cell, provides a bound on the acceptable leakage on a sensitive cell. If the sensitive cell c^* has a discrete domain and $|c^*.minus\_set| \leq |Dom(c^*)| \times (1 - k)$ evaluates to *True*, we do not hide any cells from any of its cuesets. On the other hand, for a sensitive cell c^* with a continuous domain we check if $high - low \geq |Dom(c^*)| \times (k)$ evaluates to *True*. The difference between low and $high$ gives the actual domain size after taking into count leakages due to dependencies.

When the leakage is under the threshold, k-den approach can halt earlier than the full-den algorithm, pruning out a large number of cuesets and cells to hide. If the leakage is above the threshold, then we order the cuesets in the descending order of leakage and hide cells from them (using Inference Protection) until the leakage is below the threshold. We execute Maximum Vertex Cover (MVC) in Inference Protection on all the cuesets of the sensitive cell even if only a portion of them have hidden cells. We note that this k-pruning step is only executed in the first fan-out level as an early stop condition. This ensures that the final solution generated by k-den is strictly an improvement over the strict full deniability model.

Theorem 4. *The algorithm to achieve k-percentile deniability (i.e. algorithm 5) always performs as well as (or better than) the algorithm to achieve full-deniability (i.e. algorithm 1).*

Proof. We note that the KPrune algorithm implicitly simulates the full-deniability algorithm. It does not immediately prune the cuesets or the cells to hide from the fan-out tree generated by the full-deniability algorithm (since this can change the result of running the greedy minimum vertex cover). Instead, we collect those cuesets that can be pruned but actually prune out them after simulating the overall full-deniability algorithm. Therefore, the KPrune algorithm won't hide more cells than the algorithm to achieve full-deniability. □

In Section 8, we show through experiments 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.

7 Relaxing Security Assumptions

In this section, we explore relaxing an important assumption stated in Section 3.1 about the adversary, that the adversary cannot apriori determine whether a cell is sensitive or not. There may be scenarios where the adversary can accurately guess the relative sensitivity of the attributes in a database schema. For example, in

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

Algorithm 5: KPrune: An Early-stop Algorithm to Achieve *k-percentile* Deniability

```
Input: Last level hidden cells trueHide, Current level
       hidden cells to prune toHide, Current level level,
       Leakage parameter k
Output: An updated minimum set of hidden cells in this
        level that satisfy k-deniability trueHide
Function KPrune(trueHide, toHide, level, k):
    bestCuesets = {}          ▷ Cuesets cannot be pruned.
    for cell ∈ trueHide do
        cellCuesets = cell.getCuesets()
        cell.leakage = InferredValues(cell, cellCuesets)
        if isDeniable(cell, k) then
            continue
        for cs ∈ cellCuesets do
            if level > 1 then          ▷ Simulate full-den.
                bestCuesets.add(cs)
        end
        if level = 1 then              ▷ KPrune early-stop.
            cellCuesets.Sort(leakageToParent, 'desc')
            while not isDeniable(cell, k) do
                lcs = cellCuesets.head        ▷ Max leakage.
                bestCuesets.add(lcs)
                cellCuesets.remove(lcs)
                        ▷ Recalculate the leakage of the cell.
                cell.leakage = InferredValues(cell,
                  cellCuesets)
            end
    end
    for bestCS ∈ bestCuesets do
                ▷ Update trueHide based on the pruning.
        trueHide = trueHide ∪ (toHide ∩
          bestCS.cells)
    end
    return trueHide
Function isDeniable(cell, k):
    if |Dom(c*)| − |cell.leakage| ≥ k · |Dom(c*)| then
        return True                ▷ Based on k-deniability.
    return False
```

an employee table *Salary* is more likely to be sensitive than *Zip Code* and if both are hidden in a tuple the adversary can guess that one was due to policy and the other due to the algorithm. This situation can be handled by our algorithm with a slight modification under the assumption that any tuple in the database instance could contain a sensitive cell. This means that while the adversary knows that salary is more likely to be sensitive, they do not know salaries of exactly which employees are sensitive.

The key idea behind this modified algorithm is to hide the sensitive cell in a tuple where only the non-sensitive cell is hidden. From the previous example, we would also hide the *Salary* attribute of a tuple (even when it is not sensitive) if our algorithm chooses to hide *Zip Code*. Therefore the adversary cannot be certain whether all the hidden cells under *Salary* attribute were done so by policy or the algorithm. We slightly modify the original Inference Protection algorithm (Algorithm 3) and propose Algorithm 6 in order to handle this relaxed assumption.

First, the original Inference Detection algorithm (Algorithm 2) identifies the cuesets based on dependency instantiations as an input to the Inference Protection algorithm. Second, the original Inference Protection algorithm will select at least 1 cell from each cueset to hide. Third, the steps in the modified Inference Protection Algorithm (Steps 13-21) go through the set of hidden cells and for each of them check if they belong to a non-sensitive attribute. If it does, then add the cells under the sensitive attribute from the corresponding tuple to the set of hidden cells.

Algorithm 6: Modified Inference Protection

```
   Input: Map<sensitive cell c*: Set of cuesets cuesets >,
          A view of the database V
   Output: A set of tuples to hide toHide
1  Function InferenceProtection*(Map):
2      toHide = {}                 ▷ Return set initialization.
3      while Map.cuesets ≠ φ do
4          cuesetCells = Flatten(Map.cuesets)
5          dict[c_i, freq_i] =
             CountFreq(GroupBy(cuesetCells))
6          cellMaxFreq = GetMaxFreq(dict[c_i, freq_i])
7          toHide.add(cellMaxFreq)     ▷ Greedy heuristic.
8          for cs ∈ Map.cuesets do
9              if cs.overlaps(toHide) then
10                 Map.cuesets.remove(cs)
11         end
12     end
13     additionalHiddenCells = {}  ▷ Hiding additional cells.
14     for c_h ∈ toHide do
15         tid = c_h.getTupleID()      ▷ Hidden cell's tuple ID.
16         for c* ∈ Map.sensitiveCells do
17             sensitiveAttr = c*.attributeID ▷ Sensitive cell's
                 attribute.
18             additionalHiddenCells.add(V.get(tid,
                 sensitiveAttr))
19         end
20     end
21     toHide = toHide ∪ additionalHiddenCells
22     return toHide
```

We note that the assumption of equal likelihood of tuple containing sensitive cell can be further relaxed by adopting a probabilistic approach (motivated by OSDP [29]) in which certain non-sensitive cells are randomly hidden to prevent adversary from inferring if it was part of a sensitive cell's cueset. However, such an approach will be a non-trivial extension and is an interesting future direction to explore.

Remark. In supplementary materials, we discuss ideas on how to relax another assumption that an adversary can be a data owner.

8 Experimental Evaluation

In this section, we present the experimental evaluation results for our proposed approach to implementing 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 the number of cuesets generated; 2) evaluating the impact of *dependency connectivity*; 3) testing the scalability of our system; 4) validating k-percentile deniability presented in Section 6 and

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

the modified inference protection algorithm in Section 7; 5) evaluating the query-driven utility in a case study when query workloads are presented; and 6) testing effectiveness against real-world adversaries.

8.1 Evaluation Setup

Datasets. We perform our experiments on 2 different datasets. Some statistics of the datasets are summarized in the supplementary materials. The first one is *Tax dataset* [14], 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 the 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* [12] which is a 100K dataset where all of the 15 attributes are discrete domain attributes. We select a subset of this dataset (which includes the first 10K tuples of the dataset), called *Hospital10K*, for the experiments included in the paper. We then conduct a scalability experiment that makes use of the binning-then-merging wrapper on the original Hospital dataset, i.e. 100, to show the scalability of our system. It is also notable that both datasets have a large domain size, as shown in Table 2. The active domain size in the table refers to the domain of the attributes participating in the data dependencies that we consider in the experiments.

Data Dependencies. For both datasets, we identify a large number of denial constraints by using a data profiling tool, Metanome [26]. 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 an FC based on the continuous domain attribute named *"tax"* which is calculated as a function "$\text{tax} = fn(\text{salary}, \text{rate})$". Since the Hospital dataset does not have continuous domain attributes, we cannot create a function-based constraint on it and just use the 14 DCs for evaluation. If any of them were soft DCs, we updated/deleted the violating tuples to turn them into hard DCs. The data dependencies used for experiments can be found in supplementary materials.

Policies control the sensitivity of a cell. The number of sensitive cells is equivalent to the number of policies and it helps us in precisely controlling the number of sensitive cells in experiments using policies. We randomly sample each policy by first sampling a tuple ID among all the tuples and an attribute from a selected group of attributes without replacement, until obtaining a certain number of policies determined by a control parameter. For each experiment (with the same set of control parameters), we generate 4 different access control views with different policies to represent 4 users. We execute our algorithm independently over these 4 views and report the mean and standard deviation in the results.

Metrics. We compare our approach against the baselines using the following metrics: 1) *Utility*: measures the number of total cells hidden; 2) *Workload-driven utility, i.e., visibility percentage*: measures the percentage of visible cells in queries from a workload; 3) *Performance*: measures the run time in seconds.

Besides, we study the fan-out of the number of cuesets, the attack precision of real-world adversary, and the distribution of the hidden cells in access control and inference control views.

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 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.

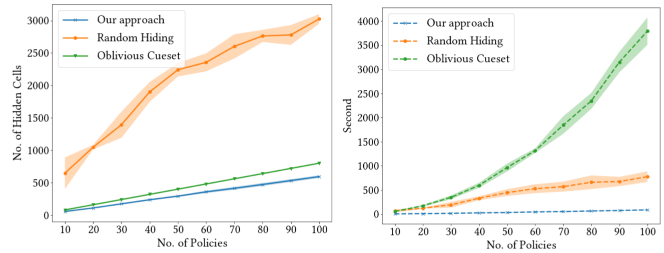

Fig. 3. (a) Data utility (b) Performance. Experiments done on Tax dataset for *Our Approach*, *Random Hiding*, and *Oblivious Cueset*.

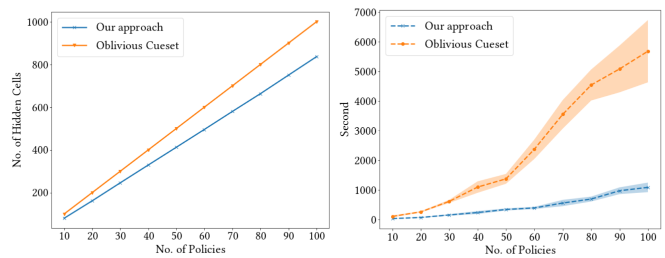

Fig. 4. (a) Data utility (b) Performance. Experiments on Hospital10K dataset for *Our Approach*, and *Oblivious Cueset*.

Reproducibility. We open-source our codebase (including ∼10K lines of code) on GitHub (https://github.com/zshufan/Tattle-Tale). This codebase includes the implementation of our system as well as scripts to set up databases, generate testcases, run end-to-end experiments, and plot the empirical results. For experiment reproducibility instructions please follow the guidelines in the Readme file in the GitHub repository.

Baselines. In the following experiments, we test our approach which implements Algorithm 1, denoted by *Our Approach* against baselines. To the best of our knowledge, there exist no other systems which solve 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 cells to hide from the cueset, with a naïve strategy but without compromising the 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.

8.2 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 sensitive cell participates in at least 5 dependencies.

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

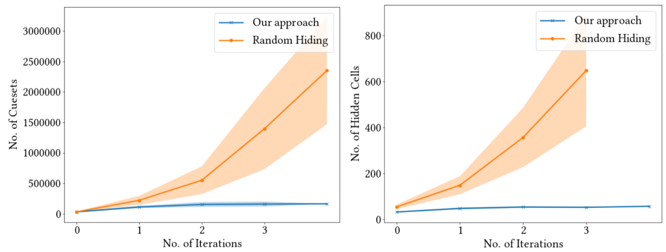

Fig. 5. (a) Number of cuesets generated in each invocation of Inference Detection (b) Number of cells hidden in each invocation of Inference Protection. Experiments run with 10 sensitive cells on Tax dataset.

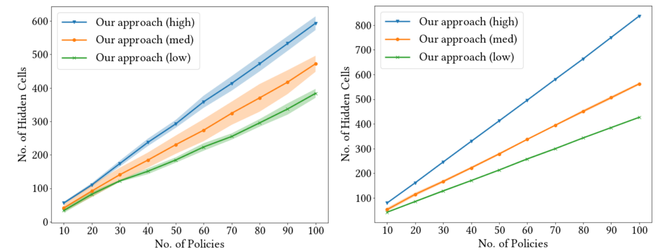

Fig. 6. Data utility experiments run with sensitive cells selected from (low, medium, high) dependency connectivity attributes in (a) Tax dataset (b) Hospital10K dataset.

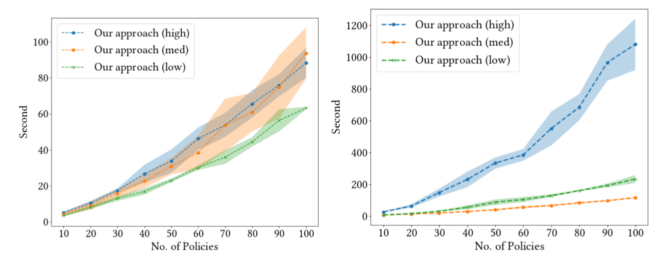

Fig. 7. Performance experiments run with sensitive cells selected from (low, medium, high) dependency connectivity attributes in (a) Tax dataset (b) Hospital10K dataset.

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 $\mathcal{O}(|\Delta| \times n^2)$ 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 5) 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*.

We show the supplementary evaluation results on the Hospital10K dataset. Figure 4 presents the end-to-end comparison between *Our Approach* and *Oblivious Cueset*, and supports our claim. In supplementary materials, we show experimental results with more sensitive cells (i.e., access control policies). Interestingly if the access control view is highly sensitive (e.g., 10% cells of the view are marked as NULL) and the sensitive cells are distributed over different columns, the sensitive cells can cancel out the channels leading to inference to each other. Therefore, in this case, our experimental results show that few additional cells are required to hide to achieve inference control.

8.3 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).

The results (see Figure 6) 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. The results are verified on both the Tax dataset and Hospital10K dataset (as shown in Figure 6(a) and Figure 6(b)). 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. Figure 7 demonstrates the evaluation among the dependency connectivity groups, on both datasets.

8.4 Experiment 3: Scalability Experiments

The results of the scalability experiments are shown in Figure 8. The y axis records the time consumption while the x axis denotes the size of the database (spanning from 10K tuples to 100K tuples). We consider two different settings for selecting sensitive cells, 1) randomly sample a fixed number of sensitive cells regardless of the database size, and 2) incrementally sample a fixed ratio of sensitive cells w.r.t the database size. The results

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

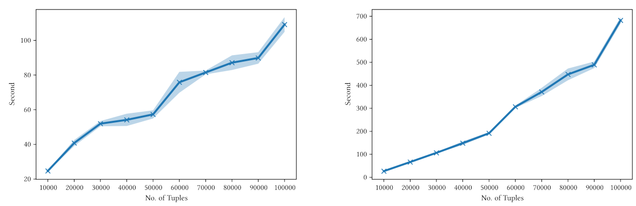

Fig. 8. (a) The results for randomly sampling a fixed number of sensitive policies (b) The results for incremental sampling a fixed ratio of sensitive policies. Evaluation was done using the Binning-then-Merging Wrapper Algorithm on the Hospital dataset.

of these two settings are presented in Figure 8(a) and Figure 8(b), resp. In both cases, we set the bin size as 10K tuples and the merging size as 5. In the first setting, the number of sensitive cells is set as 30 whereas, in the second setting, the ratio of sensitive cells to the total number of cells is 30 cells per 10K tuples. We note that the starting point of the plot (x = 10K tuples) corresponds to the experiments presented in Section 8 i.e., running our main algorithm on the dataset of size 10K (as there is only 1 bin). As shown in Figure 8, the time consumption scales near-linearly (depending on the data itself) to the size of the datasets.

8.5 Experiment 4: k-Percentile Deniability

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 9(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, the utility improvement was marginal with k set to the smallest value possible (besides full leakage) i.e., $k = \frac{1}{|Dom(c^*)|}$. *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. We measure the runtime performance of k-deniability for different k values and compare the results with full-deniability. As shown in Figure 9(b), algorithms to achieve k-deniability take longer time to complete than the full-deniability algorithms, because k-deniability algorithms reduce the fan-out of the cuesets in the first iteration, but more iterations are thus taken to converge. For different tested k values, the more we relax the k constraint, the less execution time the algorithm will take, because fewer cuesets, thus a smaller fan-out, are considered in calculating leakage.

8.6 Experiment 5: Modified Inference Protection

We implemented and tested the modified inference protection algorithm (Algorithm 6) on the Tax dataset and compare the results with our inference protection (w. MVC) to achieve full-deniability. As one can observe from Figure 10, the price of

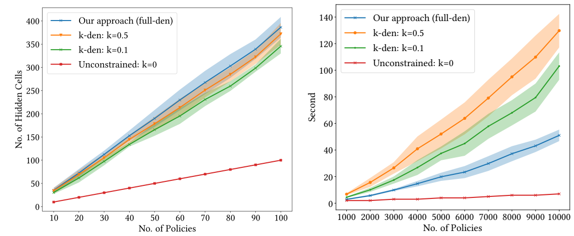

Fig. 9. (a) Data utility on Tax dataset. Experiments done with full deniability and k-deniability (varying values of k); (b) Performance on Tax dataset (varying values of k).

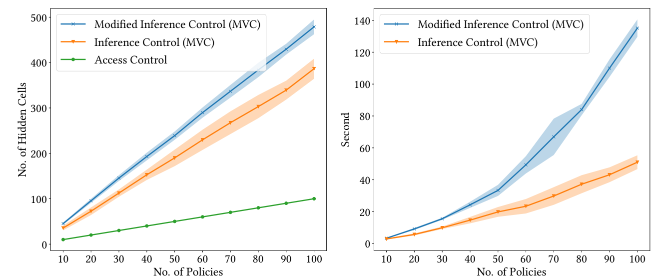

Fig. 10. Modified Inference Protection: (a) Data utility (b) Performance on Tax dataset for modified inference protection, inference protection, and access control.

relaxing the assumptions is compensating for utility and efficiency. Compared to our approach to achieving inference control, the modified inference control hides 1.3x cells and requires an average of 2-3x more time to converge (with a non-linear growth).

8.7 Experiment 6: Case Study over Query Workloads

We further study how inference control algorithms can affect the utilities of query workloads, especially when a large portion of the database view is marked as NULL by access control policies. We first investigate the distribution of the hidden cells (NULL's) across the views. We take the run with the access control view with 1,000 policies and execute our approach (w. MVC) to generate the inference control view and use these two views throughout this case study. Since this study involves a large number of policies that baseline methods take too much time to converge, we only compare the inference control view based on our approach with the access control view. We present in Figure 11 the heatmap, where a darker color represents more cells hidden in this column, and the density distributions of data that support the visualization. The distributions of NULL cells are similar in both views – most additional hidden cells in the inference control view are concentrated in the first 3 attributes that are directly correlated with the access control policies. Some but fewer additional cells from other columns are hidden as well in the inference control view, while none of the cells are hidden from the attributes not participating in dependencies.

Evaluating workload-driven utility metric. Next, we evaluate the utility of the database views over two types of query workloads: randomized range queries over one column and cross columns. In particular, for the first case, we randomly generate 1,000 set queries per column with randomly sampled range specifications (w. 300-1100 cells, varying). For the cross-column queries, we consider every possible pairwise combination of the attributes and similarly generate 1,000 queries for each combination. The

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

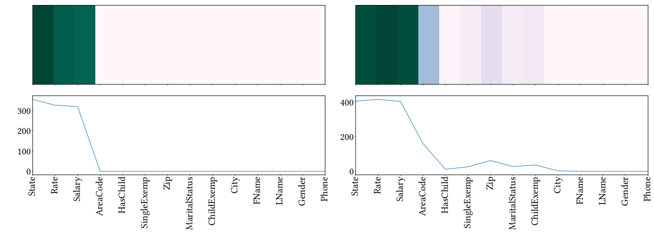

Fig. 11. Distribution of NULL's: (a) as policies in access control view; (b) as hidden cells in the inference control view.

range queries cover both attributes in each combination. As mentioned, we consider visibility (i.e., percentage of non-NULL cells in the query result) as the utility metric in this case study.

Figure 12 shows the empirical results. We take the workload that executes 1,000 queries on the "Rate" column to present the results in Figure 12(a) and (b) for access control and inference control views, resp. We use histograms to show the number of queries that has a certain percentage of visibility. As observed, most queries remain high visibility ($\sim$93-96% cells visible) in both views, indicating good utility for downstream analytics.

We then present results for cross-column queries in Figure 12(c) and (d) as heatmaps. Each block in the heatmap represents the average visibility percentage among 1,000 queries executed over this attribute combination. While the overall visibility is over 95% for both views, a darker color in the heatmap suggests more cells are visible from the query. The similarity between the two heatmaps indicates that inference control does not affect the query-driven utility much compared to the access control views.

8.8 Experiment 7: Case Study against Real-World Adversaries

A potential limitation of 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 is 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.

Therefore, 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 [15], 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 $= \dfrac{\#\text{correct repairs}}{\#\text{total repairs}}$ (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 13 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 only marginally improves upon weighted sampling given the view generated by *Our Approach*.

9 Related Work

The challenge of preventing leakage of sensitive data from query answers has been studied in many prior works on inference control [9]. Early work by Denning *et al.* [30] designed commutative filters to ensure answers returned by a query are equivalent to that which would be returned based on the authorized view for the user. This work, however, did not consider data dependencies. We categorize them based on when and how inference control is applied and what security model is used.

Design-time Prevention Methods which mark attributes that lead to inferences on sensitive data items as sensitive. Qian *et al.* [31] developed a tool to analyze potential leakage due to foreign keys in order to elevate the clearance level of data if such leakage is detected. Delugachi *et al.* [17] generalized the work in [31] 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 [32], however, established that inference rules for detecting inferences at database design time are incomplete and hence are not a viable approach for preventing leakage from query answers. Design time approaches for disclosure control 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 [25], [33], [34], [35], however, do not extend to our setting.

Query-time Prevention Methods that reject queries which lead to inferences on sensitive data items. Thuraisingham [19] developed a *query control approach* in the context of Mandatory Access Control (MAC) wherein policies specify the security clearances for the users (subject) and the security classification/label for the data. [19] presented an inference engine to determine if query answers can lead to leakage (in which case the query is rejected). While [19] assumed a prior existence of an inference detection engine, Brodsky *et al.* [16] developed a framework, DiMon, based on chase algorithm for constraints expressed as Horn clauses.

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

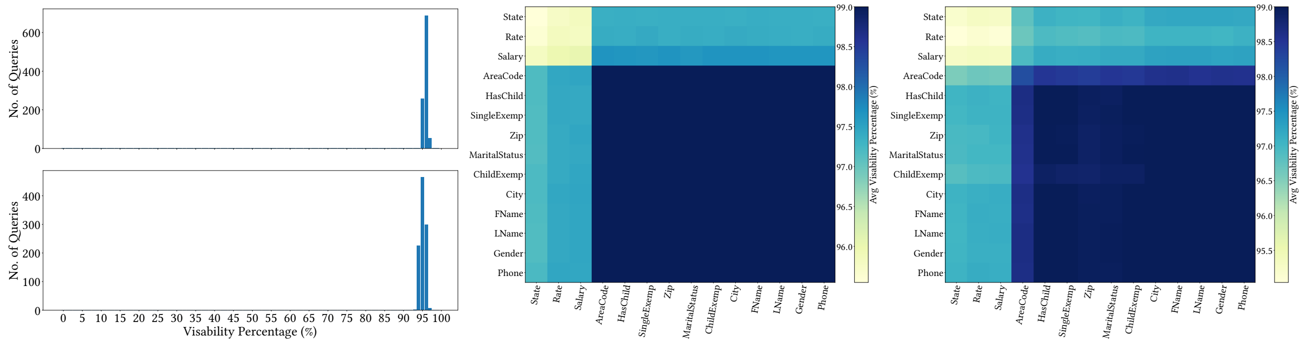

Fig. 12. Workload-driven utility: (a) Upper left: visibility percentage for queries in the workload over the access control view; (b) Bottom left: visibility percentage for queries in the workload over the inference control view; (c) Middle: average visibility percentage in cross-column workload over the access control view; (d) Right: average visibility percentage in cross-column workload over the inference control view.

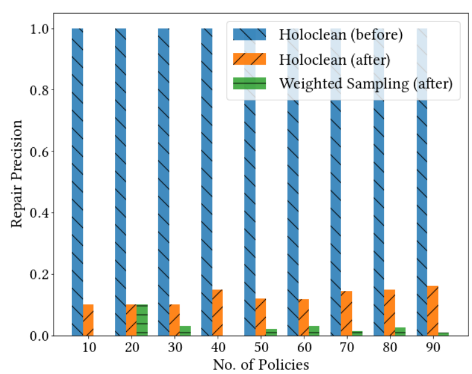

Fig. 13. Against real-world adversaries: Reconstruction precision of sensitive cells with two types of adversaries.

DiMon takes in current query results, the user's query history, and Horn clause constraints to determine the additional data that may be inferred by the subject. Similar to [19], if inferred data is beyond the security clearance of the subject then their system refuses the query. Such work (that identifies if a query leaks/does not leak data) differs from ours since it cannot 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 an adversary from reconstructing the exact value of a sensitive cell but cannot prevent them from learning new information about the sensitive cell.

Perfect Secrecy Models that characterizes inferences on any possible database instance as leakage. The most relevant of these works is from Miklau & Suciu [36] who study the challenge of preventing information disclosure for a secret query given a set 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. Applying their approach to our problem setting will be extremely pessimistic as most queries will be rejected on a database with a non-trivial number of dependencies.

Randomized Algorithms for Inference Prevention that suppress too many cells and does not look at dependencies as inference channels The most relevant of these are Differential Privacy (DP) mechanisms promise to protect against an adversary with any prior knowledge and thus have wide applications nowadays [37], [38], [39]. In our problem setting of access control, called the *Truman model* of access control [8], the data is either hidden or shared depending upon whether it is sensitive for a given querier. In such a model, the expectation of a querier is that the result doesn't include any randomized answers. Weaker notions of DP such as One-sided differential privacy (OSDP) [29] aims to prevent inferences on sensitive data by using a randomized mechanism when sharing non-sensitive data. However, such techniques offer only probabilistic guarantees (and cannot implement security guarantees such as full deniability), and therefore may allow some non-sensitive data to be released even when their values could lead to leakage of a sensitive cell. These techniques also lead to suppression of a large amount of data (suppresses approx. 91% non-sensitive data at $\epsilon = 0.1$ and approx. 37% at $\epsilon = 1$). 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.

Inference Control in Other Settings. Among these, [40] studies the problem of secure data outsourcing in the presence of functional dependencies. Access control policies are modelled using confidentiality constraints which define what combination of attributes should not appear together in a partition. They use a graph-based approach built upon on functional dependencies to detect possible inference channels. The goal is to then derive optimal partitioning so as to prevent inferences through these functional dependencies while efficiently answering queries on distributed partitions. Vimercati et al [25] also studied the problem of improper leakage due to data dependencies in data fragmentation. Similar to [40], they mark attributes as sensitive (using confidentiality constraints) and block the information flow from non-sensitive attributes to sensitive attributes through dependencies. In general, the works in this category look at sensitivity at the level of attributes and not at the level of cells through fine-grained access control policies, studied in our work. In our work, we enforce fine-grained access control policies and allow minimal hiding of additional cells to prevent inferences.

10 Conclusions and Future Work

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

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

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.

The Tattle-Tale problem from the paper can be extended and more discussions may be spawned by considering other access control research. We thereby envision future directions as follows.

- (Extending Constraint Modelling) 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 the probabilistic relationship between input and output cells, instead of being deterministic as in the current model. One potential approach is to use Markov Logic Network to encode the fuzzy constraint in our system.
- (Improving Utility) One may want to improve utility while implementing full deniability or by further exploring k-percentile deniability. To achieve so, one direction to go is to consider releasing non-sensitive values (like in OSDP) randomly instead of hiding all. However, this requires addressing the challenges of any inadvertent leakages through dependencies when sharing such randomized data.
- (Towards Other Use Cases in Access Control) While this work focuses on the Truman model of access control, future work can consider other settings, such as non-Truman models or a cryptographic modelling of access control [41], [42] or web applications [43]. Since data are stored in relational models and are thus often correlated via constraint, the Tattle-tale problem also exists in those directions. Future directions can consider similar problems in other use cases of access control.

Supplementary Material

Due to space constraints, we defer omitted proofs, algorithms, discussions, and some experimental details to the supplementary materials of this paper.

Acknowledgments

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] P. Pappachan, S. Zhang, X. He, and S. Mehrotra, "Don't be a tattle-tale: Preventing leakages through data dependencies on access control protected data," vol. 15, no. 11. VLDB Endowment, 2022, pp. 2437–2449. [Online]. Available: https://doi.org/10.14778/3551793.3551805

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

[3] C. L. I. website, "California online privacy protection act (CalOPPA)," https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=BPC&sectionNum=22575, 2020, [Online; accessed 01-Jul-2022].

[4] S. of California Department Justice Office of the Attorney General, "California consumer privacy act CCPA," https://oag.ca.gov/privacy/ccpa, 2020, [Online; accessed 01-Jul-2022].

[5] E. Ferrari, "Access control in data management systems," *Synthesis lectures on data management*, vol. 2, no. 1, pp. 1–117, 2010.

[6] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Hong Kong, China: Morgan Kaufmann, 2002, pp. 143–154. [Online]. Available: http://www.vldb.org/conf/2002/S05P02.pdf

[7] P. Pappachan, R. Yus, S. Mehrotra, and J.-C. Freytag, "Sieve: A middleware approach to scalable access control for database management systems," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2424–2437, jul 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407835

[8] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2004, p. 551–562. [Online]. Available: https://doi.org/10.1145/1007568.1007631

[9] C. Farkas and S. Jajodia, "The inference problem: A survey," *SIGKDD Explor.*, vol. 4, no. 2, pp. 6–11, 2002. [Online]. Available: https://doi.org/10.1145/772862.772864

[10] J. Chen, J. He, L. Cai, and J. Pan, "Disclose more and risk less: Privacy preserving online social network data sharing," *IEEE Trans. Dependable Secur. Comput.*, vol. 17, no. 6, pp. 1173–1187, 2020. [Online]. Available: https://doi.org/10.1109/TDSC.2018.2861403

[11] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for capturing data inconsistencies," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 2, pp. 1–48, 2008.

[12] X. Chu, I. F. Ilyas, and P. Papotti, "Discovering denial constraints," *Proc. VLDB Endow.*, vol. 6, no. 13, pp. 1498–1509, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p1498-papotti.pdf

[13] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan, "Foundations of aggregation constraints," *Theoretical Computer Science*, vol. 193, no. 1-2, pp. 149–179, 1998. [Online]. Available: https://doi.org/10.1016/S0304-3975(97)00011-X

[14] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for data cleaning," in *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007*. Istanbul, Turkey: IEEE Computer Society, 2007, pp. 746–755. [Online]. Available: https://doi.org/10.1109/ICDE.2007.367920

[15] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré, "Holoclean: Holistic data repairs with probabilistic inference," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1190–1201, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1190-rekatsinas.pdf

[16] A. Brodsky, C. Farkas, and S. Jajodia, "Secure databases: Constraints, inference channels, and monitoring disclosures," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 6, pp. 900–919, 2000. [Online]. Available: https://doi.org/10.1109/69.895801

[17] H. S. Delugach and T. H. Hinke, "Wizard: A database inference analysis and detection system," *IEEE Trans. Knowl. Data Eng.*, vol. 8, no. 1, pp. 56–66, 1996. [Online]. Available: https://doi.org/10.1109/69.485629

[18] T. D. Garvey, T. F. Lunt, X. Qian, and M. E. Stickel, "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 on Database Security, VI: Status and Prospects: Status and Prospects*. USA: Elsevier Science Inc., 1993, p. 149–167.

[19] B. M. Thuraisingham, "Security checking in relational database management systems augmented with inference engines," *Comput. Secur.*, vol. 6, no. 6, pp. 479–492, 1987. [Online]. Available: https://doi.org/10.1016/0167-4048(87)90029-0

[20] J.-W. Byun and N. Li, "Purpose based access control for privacy protection in relational database systems," *The VLDB Journal*, vol. 17, no. 4, p. 603–619, jul 2008. [Online]. Available: https://doi.org/10.1007/s00778-006-0023-0

[21] P. Colombo and E. Ferrari, "Efficient enforcement of action-aware purpose-based access control within relational database management systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 8, pp. 2134–2147, 2015.

[22] F. Geerts, G. Mecca, P. Papotti, and D. Santoro, "The LLUNATIC data-cleaning framework," *Proc. VLDB Endow.*, vol. 6, no. 9, pp.

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3336630

625–636, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p625-mecca.pdf

[23] X. Chu, I. F. Ilyas, and P. Papotti, "Holistic data cleaning: Putting violations into context," in *29th IEEE International Conference on Data Engineering, ICDE 2013*. Brisbane, Australia: IEEE Computer Society, 2013, pp. 458–469. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544847

[24] J. Kossmann, T. Papenbrock, and F. Naumann, "Data dependencies for query optimization: a survey," *VLDB J.*, vol. 31, no. 1, pp. 1–22, 2022. [Online]. Available: https://doi.org/10.1007/s00778-021-00676-3

[25] S. D. C. di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati, "Fragmentation in presence of data dependencies," *IEEE Trans. Dependable Secur. Comput.*, vol. 11, no. 6, pp. 510–523, 2014. [Online]. Available: https://doi.org/10.1109/TDSC.2013.2295798

[26] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann, "Data profiling with metanome," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1860–1863, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1860-papenbrock.pdf

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. Cambridge, MA, USA: The MIT Press, 2009.

[28] I. Dinur and S. Safra, "On the hardness of approximating minimum vertex cover," *Annals of mathematics*, vol. 162, no. 1, pp. 439–485, 2005.

[29] I. Kotsogiannis, S. Doudalis, S. Haney, A. Machanavajjhala, and S. Mehrotra, "One-sided differential privacy," in *36th IEEE International Conference on Data Engineering, ICDE 2020*. Dallas, TX, USA: IEEE, 2020, pp. 493–504. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00049

[30] D. E. Denning, "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*. Oakland, CA, USA: IEEE Computer Society, 1985, pp. 134–146. [Online]. Available: https://doi.org/10.1109/SP.1985.10017

[31] X. Qian, M. E. Stickel, P. D. Karp, T. F. Lunt, and T. D. Garvey, "Detection and elimination of inference channels in multilevel relational database systems," in *1993 IEEE Computer Society Symposium on Research in Security and Privacy*. Oakland, CA, USA: IEEE Computer Society, 1993, pp. 196–205. [Online]. Available: https://doi.org/10.1109/RISP.1993.287632

[32] R. W. Yip and K. N. Levitt, "Data level inference detection in database systems," in *Proceedings of the 11th IEEE Computer Security Foundations Workshop*. Rockport, Massachusetts, USA: IEEE Computer Society, 1998, pp. 179–189. [Online]. Available: https://doi.org/10.1109/CSFW.1998.683168

[33] M. Haddad, J. Stevovic, A. Chiasera, Y. Velegrakis, and M. Hacid, "Access control for data integration in presence of data dependencies," in *Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014*, ser. Lecture Notes in Computer Science, vol. 8422. Bali, Indonesia: Springer, 2014, pp. 203–217. [Online]. Available: https://doi.org/10.1007/978-3-319-05813-9_14

[34] K. Y. Oktay, S. Mehrotra, V. Khadilkar, and M. Kantarcioglu, "SEMROD: secure and efficient mapreduce over hybrid clouds," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne, Victoria, Australia: ACM, 2015, pp. 153–166. [Online]. Available: https://doi.org/10.1145/2723372.2723741

[35] K. Y. Oktay, M. Kantarcioglu, and S. Mehrotra, "Secure and efficient query processing over hybrid clouds," in *33rd IEEE International Conference on Data Engineering, ICDE 2017*. San Diego, CA, USA: IEEE Computer Society, 2017, pp. 733–744. [Online]. Available: https://doi.org/10.1109/ICDE.2017.125

[36] G. Miklau and D. Suciu, "A formal analysis of information disclosure in data exchange," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, 2004*. Paris, France: ACM, 2004, pp. 575–586. [Online]. Available: https://doi.org/10.1145/1007568.1007633

[37] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Found. Trends Theor. Comput. Sci.*, vol. 9, no. 3-4, pp. 211–407, 2014. [Online]. Available: https://doi.org/10.1561/0400000042

[38] L. Yu, S. Zhang, L. Zhou, Y. Meng, S. Du, and H. Zhu, "Thwarting longitudinal location exposure attacks in advertising ecosystem via edge computing," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, 2022, pp. 470–480. [Online]. Available: https://doi.org/10.1109/ICDCS54860.2022.00052

[39] S. Zhang and X. He, "DProvDB: Differentially private query processing with multi-analyst provenance," *Proc. ACM Manag. Data*, 2023.

[40] A. Jebali, S. Sassi, A. Jemai, and R. Chbeir, "Secure data outsourcing in presence of the inference problem: A graph-based approach," *Journal of Parallel and Distributed Computing*, vol. 160, pp. 1–15, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731521001842

[41] Y. Bao, W. Qiu, X. Cheng, and J. Sun, "Fine-grained data sharing with enhanced privacy protection and dynamic users group service for the iov," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–15, 2022.

[42] T. V. X. Phuong, G. Yang, and W. Susilo, "Hidden ciphertext policy attribute-based encryption under standard assumptions," *Trans. Info. For. Sec.*, vol. 11, no. 1, p. 35–45, jan 2016. [Online]. Available: https://doi.org/10.1109/TIFS.2015.2475723

[43] W. Zhang, E. Sheng, M. A. Chang, A. Panda, M. Sagiv, and S. Shenker, "Blockaid: Data access policy enforcement for web applications," *CoRR*, vol. abs/2205.06911, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2205.06911

[44] X. Xiao, Y. Tao, and N. Koudas, "Transparent anonymization: Thwarting adversaries who know the algorithm," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 2, pp. 1–48, 2010.

[45] N. Dalvi, G. Miklau, and D. Suciu, "Asymptotic conditional probabilities for conjunctive queries," in *International Conference on Database Theory*. Springer, 2005, pp. 289–305.

Primal Pappachan is an Assistant Professor in the Department of Computer Science at Portland State University. He received a Ph.D. in Computer Science from University of California, Irvine in 2021. Afterwards, he was a postdoctoral scholar in the College of Information Sciences and Technology at Pennsylvania State University. His research interests include data management and privacy, particularly designing and implementing data protection mechanisms.

Shufan Zhang received the M.Math degree from the University of Waterloo, Waterloo, ON, Canada, in 2022. He is currently working toward the Ph.D. degree in computer science at the University of Waterloo. His research interests include computer security and data privacy, on both theory and system aspects, as well as their intersections with database systems and machine learning.

Xi He is an Assistant Professor in the Cheriton School of Computer Science at the University of Waterloo, and Canada CIFAR AI Chair at the Vector Institute. Her research focuses on the areas of privacy and security for big data, including the development of usable and trustworthy tools for data exploration and machine learning with provable security and privacy guarantees. She has given tutorials on privacy at VLDB 2016, SIGMOD 2017, and SIGMOD 2021. She is a recipient of the Meta Privacy Enhancing Technologies Research Award in 2022 and Google Ph.D. Fellowship in Privacy and Security in 2017. Her book "Differential Privacy for Databases," coauthored by Joseph Near, was published in 2021. Xi graduated with a Ph.D. from the Department of Computer Science, Duke University, and a double degree in Applied Mathematics and Computer Science from the University of Singapore.

Sharad Mehrotra received the PhD degree in computer science from the University of Texas, Austin, Austin, Texas, in 1993. He is currently a professor with the Department of Computer Science, University of California, Irvine, Irvine, California. Previously, he was a professor with the University of Illinois at Urbana Champaign, Champaign, Illinois. He has received numerous awards and honors, including the 2011 SIGMOD Best Paper Award, 2007 DASFAA Best Paper Award, SIGMOD Test of Time Award, 2012, DASFAA ten year best paper awards for 2013 and 2014, 1998 CAREER Award from the US National Science Foundation (NSF), and ACM ICMR Best Paper Award for 2013. His primary research interests include the area of database management, distributed systems, secure databases, and Internet of Things.