Privacy Preservation of Aggregates in Hidden Databases: Why and How?

Arjun Dasgupta
CSE Department
UT Arlington
arjundasgupta@uta.edu

Nan Zhang*
CS Department
George Wash. Univ.
nzhang10@gwu.edu

Gautam Das[ζ]
CSE Department
UT Arlington
gdas@uta.edu

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT
Many websites provide form-like interfaces which allow users to execute search queries on the underlying hidden databases. In this paper, we explain the importance of protecting sensitive aggregate information of hidden databases from being disclosed through individual tuples returned by the search queries. This stands in contrast to the traditional privacy problem where individual tuples must be protected while ensuring access to aggregating information. We propose techniques to thwart bots from sampling the hidden database to infer aggregate information. We present theoretical analysis and extensive experiments to illustrate the effectiveness of our approach.

Categories and Subject Descriptors
H.2.7 Database Administration

General Terms
Algorithms, Management, Measurement, Performance, Security

Keywords
Hidden Databases, Privacy Preservation

1. INTRODUCTION
Databases with public web-based search interfaces are available for many government, scientific, and health-care websites as well as for many commercial sites. Such sites are part of the much talked about hidden web and contain a lot of valuable information. These sites provide controlled access to their databases through their search interfaces.

The simplest and most widely prevalent kind of search interface over such databases allows users to specify selection conditions (point or range queries) on a number of attributes and the system returns the tuples that satisfy the selection conditions. Sometimes the returned results may be restricted to a few (e.g., top-*k*) tuples, sorted by one or more ranking functions predetermined by the hidden web site. To illustrate the scenario let us consider the following examples.

Example 1: *An auto dealer's web form lets a user choose from a set of attributes e.g., manufacturer, car type, mileage, age, price, engine type, etc. The top-k answers, sorted according to a ranking function such as price, are presented to the user, where k is a small constant such as 10.*

Example 2: *An airline company's flight search form lets a user search for a flight by specifying a set of attributes such as departure and destination cities, date, number of stops, carrier, and cabin preferences. The top-k flights, sorted according to a ranking function such as price, are presented.*

In this paper, our key observation is that there is a large class of such websites where individual result tuples from search queries are public information and raise no privacy concerns (e.g., availability of a specific model of car, schedule for a specific flight). However, such sites would like to *suppress inference of aggregate information*. In Example 1, while the auto dealer wishes to allow potential car buyers to search its database, it would not like to make public information that enables competitors to infer its inventory, e.g., that a certain popular car has been in short supply at the dealership in recent weeks. If the competitors were able to infer such aggregate information, then this would allow them to take advantage of the low inventory by a multitude of tactics (e.g., stock that vehicle, make appropriate adjustments to price).

Likewise in Example 2, an airline company would not wish to reveal information that enables terrorists to predict which flights, on what dates, are more likely to be relatively empty. In recent hijackings such as 9/11 and Russian aircraft bombing of 2004, terrorists' tactics are believed to be to hijack relatively empty flights because there would be less resistance from occupants. If terrorists are able to infer aggregate information such as Friday afternoon flights from Washington DC to Denver are emptier than others on average, they could leverage this information to plan their attacks.

Of course, extremely broad aggregate information is usually well known and publicly available – e.g., that family sedans are more common than sports cars, or that flights are usually empty on Christmas Day. It is the inferences of relatively *fine-grained aggregates* as illustrated by the examples above that need to be protected against, as the impact of such inference can range from being merely disadvantageous to the data publisher to posing serious security threats to society at large. Also, aggregates collected from human efforts (through domain knowledge) do not provide any form of statistical guarantees, which may be particularly relevant for *fine-grained aggregates.*

It is important to recognize that our scenario of **Privacy Preservation of Aggregates** is in sharp contrast to the traditional privacy scenarios, where the individual information needs to be protected against disclosure but aggregate information that does not lead to inference to an individual's information is considered acceptable disclosure. To our best knowledge, this form of data privacy is novel and has not been explored by researchers.

The goal of this paper is to propose techniques that can enable the owner of a hidden database to guarantee the privacy

* Partly supported by NSF grants 0852673, 0852674 and 0845644.
ζ Partly supported by NSF grants 0845644, 0812601 and grants from Microsoft Research and Nokia Research.

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

of aggregate information without compromising the utility of search queries submitted through the front-end search interfaces.

Because most interfaces only allow a limited number of search results to be returned, and do not allow aggregate queries (e.g., a query such as "SELECT COUNT(*) FROM *D* where <selection-condition>") to be directly executed, it may seem that aggregate information is already adequately protected from bots or malicious users, as they would have to execute an inordinate number of search queries to collect enough returned tuples to aggregate at their end. However, in our recent works [DDM07, DZD09], we have studied how an external application can leverage the public interface of a hidden database to draw high quality uniform random samples of the underlying data, which can then be used to *approximately infer* fine-grained aggregates. As the size of the samples collected increases, the estimation of aggregates becomes more robust. In particular, we proposed two samplers that can be used against a wide variety of existing hidden databases: HIDDEN-DB-SAMPLER [DDM07] and HYBRID-SAMPLER [DZD09]. These approaches can be very effective, as in many cases, approximate aggregates can provide the critical insights into the data. These samplers obtain samples from the restrictive interface provided by hidden databases in an efficient manner by optimizing the number of queries posed to the database. Thus, our main challenge is to develop techniques to thwart such sampling attacks.

Technical Challenge: *Given a hidden database, develop techniques that make it very difficult to obtain uniform random samples of the database via its search interface without necessitating human intervention.*

Note that if we were to accept a "human in the loop", then a seemingly simple solution is to embed into the search interface a human-readable machine-unrecognizable image called CAPTCHA [EDHS07], and to require each user to provide a correct CAPTCHA solution before executing every search query. This strategy is used in a number of real-world websites[1]. Nonetheless, a key limitation of this approach is that it eliminates the possibility of any automated access, including legitimate ones such as third-party web services, over the hidden database. Such limitation is becoming increasingly problematic with the growing popularity mash-up web applications.

Thus, we aspire to develop techniques that allow search queries issued by bona fide users, both human as well as third-party web applications, but at the same time make it very difficult for adversaries including automated bots by forcing them to execute an inordinate number of search queries before they can obtain a small random sample of the database.

While there has been significant recent work in the areas of privacy preserving data mining [AS00, CKV+03, ZZ07], data publishing [Swe02], OLAP [AST05], and information sharing [AES03], these techniques are inadequate for our purposes. Unlike our scenario, these techniques are appropriate where the privacy of the individual needs to be preserved - e.g., in a medical database, it is acceptable to reveal that the number of HIV patients is 30% more than cancer patients, but not acceptable to reveal that a particular patient has HIV. Tuplewise privacy preservation techniques such as *encryption* [AES03], *data perturbation* [AS00] and *generalization* methods [Swe02] cannot be used in our framework either as obfuscating individual data tuples is not an option since tuples need to be made visible to normal search users. The well-studied *online query auditing* [NMK+06], which answers or denies a query based on the user's query history, is also not applicable in our scenario as these websites provide public interfaces and cannot monitor individual users' history of past queries.

Our Approach: In this paper we present novel techniques for protecting aggregates over hidden databases. We propose a privacy-preserving algorithm called COUNTER-SAMPLER, which can be used to defend against *all possible* sampling attacks. The key idea used in COUNTER-SAMPLER is to insert into the hidden database a small number of carefully constructed *dummies tuples*, i.e., tuples that do not exist in the original hidden database but are composed of legitimate attribute values.

The reasons why dummy tuples can be effective at defending against sampling attacks are subtle - we defer a detailed discussion for later in the paper, but provide brief intuition here. Our approach builds on the observation that all existing sampling attacks retrieve uniform random sample from tuples returned by queries that select *at most k* tuples each (where *k* is a small constant such as 10 or 100), because otherwise the search on the hidden database will return only the top-*k* tuples sorted by ranking functions unknown to the sampler, and hence cannot be assumed to be random. In other words, we target a common characteristic of samplers to find "valid" search queries that neither "overflow" (i.e., do not have broad conditions that select more than *k* tuples) nor "underflow" (i.e., do not have narrow conditions that select no tuple). Thus, the key idea of COUNTER-SAMPLER is to carefully construct and insert dummy tuples into the database such that most valid (and some underflowing) queries are converted to overflowing queries, thus significantly decreasing the proportion of valid queries within the space of all possible search queries. As a result, any sampler which generates samples from valid queries has to execute a huge number of queries before it encounters enough valid queries be able to generate a uniform random sample.

Of course, the presence of dummy tuples presents an inconvenience to normal search users or applications, which have to now distinguish the real tuples from the dummy tuples in the returned results of any search query. How to distinguish dummy tuples from real tuples is discussed later in the paper. Nevertheless, to reduce such inconveniences, our objective is to *minimize* the number of inserted dummy tuples while providing the desired privacy guarantees. Our analytical as well as experimental results show that only a small number of dummy tuples are usually inserted by COUNTER-SAMPLER to provide adequate privacy guarantees.

Summary of Contributions:

- We define the novel problem of *Privacy Preservation of Aggregates in Hidden Databases*.
- We develop COUNTER-SAMPLER, a privacy-preserving algorithm that inserts dummy tuples to prevent the efficient sampling of hidden databases.
- We provide theoretical analysis on the privacy guarantee for sensitive aggregates provided by COUNTER-SAMPLER.
- We describe a comprehensive set of experiments that demonstrate the effectiveness of COUNTER-SAMPLER. Although it is universally effective against all possible sampling attacks, we demonstrate its effectiveness against the state-of-the-art sampling algorithms, HIDDEN-DB-SAMPLER [DDM07] and HYBRID-SAMPLER [DZD09].

The rest of this paper is organized as follows. We introduce preliminaries notions in Section 2, the basic ideas behind COUNTER-SAMPLER in Section 3, the actual algorithm and analysis in Section 4, case studies against existing attacks in

[1] e.g., *http://www.seatcounter.com/*

Section 5, experimental results in Section 5, related work in Section 6 and final remarks in Section 7.

2. PRELIMINARIES

2.1 Hidden Databases

In most of this paper, we restrict discussions to categorical data – we discuss extensions to numerical data in Section 4.5. Consider a hidden database table D with m tuples $t_1,\ldots,t_m$ on N attributes $a_1,\ldots,a_N$. The table is only accessible through a web-based interface where users can issue queries that specify values for a subset of the attributes, say $a_1, \ldots, a_n$. We refer to such queries as the *search queries*, which are of the form: Q_S: SELECT * FROM D WHERE $a_{c1} = v_{c1}$ AND ··· AND $a_{cu} = v_{cu}$, where $v_{c1},\ldots,v_{cu}$ are from the domains of $a_{c1},\ldots,a_{cu} \in \{a_1, \ldots, a_n\}$, respectively.

Let $Sel(Q_S)$ be the set of tuples in D that satisfy Q_S. As is common with most web interfaces, we assume that the query interface is restricted to only return up to k tuples, where $k << m$ is a pre-determined small constant. If the query is too broad (i.e., $|Sel(Q_S)| > k$), only the top-k tuples in $Sel(Q_S)$ will be selected according to a ranking function, and returned as the query result. The interface will also notify the user that there is an *overflow*. At the other extreme, if the query is too specific and returns no tuple, we say that an *underflow* occurs. If there is neither an overflow nor an underflow, we have a *valid* query result.

In this paper, we assume that the interface does not allow a user to "scroll through" the complete answer $Sel(Q_S)$ when an overflow occurs. We argue that this is a reasonable assumption because many real-world top-k interfaces only allow a limited number of "page turns". Google, for example, only allows 100 page turns (10 results per page) per query. This essentially makes Google a top-1000 interface in our model.

2.2 Privacy Requirements

Consider *aggregate queries* of the form: SELECT AGGR(*) FROM D WHERE $a_{c1}=v_{c1}$ AND···AND $a_{cu} = v_{cu}$, where AGGR(·) is an aggregate function such as COUNT, SUM, etc, and $v_{c1}, \ldots, v_{cu}$ are from the domains of $a_{c1}, \ldots, a_{cu}$, respectively. Let $Res(Q_A)$ be the result of such an aggregate query. As discussed Section 1, due to privacy concerns the owner of a hidden database may consider certain aggregate queries to be sensitive and would not willingly disclose their results.

To quantify privacy protection, we first define the notion of disclosure. Similar to the privacy models for individual data tuples [NMK+06], we can define the *exact* and *partial* disclosure of aggregates. Exact disclosure occurs when a user learns the exact answer to an aggregate query. Exact disclosure is a special case of partial disclosure; the latter occurs when there is a significant change between a user's prior and posterior confidence about the range of a sensitive query answer.

In this paper we consider the (broader) partial disclosure notion because approximate answers are often adequate for aggregate queries. Carrying the same spirit in the privacy-game notion for individual data tuples [KMN05], we define the following (ε, δ)-privacy game between a user and the hidden database owner for a sensitive aggregate query Q_A:

1. The owner chooses its defensive scheme.
2. The user issues search queries and analyzes their results to try and estimate $Res(Q_A)$.
3. The user wins if $\exists x$ such that the user has confidence $> \delta$ that $Res(Q_A) \in [x, x+\varepsilon]$. Otherwise, the user loses.

Based on the (ε, δ)-game notion, we define the privacy requirement for a hidden database as follows:

Definition 2.1: *We say that a defensive scheme achieves* (ε, δ, p)*-privacy guarantee for* Q_A *iff for any user,* $\Pr\{A$ *wins* (ε, δ)*-privacy game for* $Q_A\} \le p$.

The probability is taken over the (possible) randomness in both the defensive scheme and the attacking strategy. Note that if a defensive scheme achieves (ε, δ, p)-privacy guarantee, then no user can win a (ε_0, δ_0)-privacy game with probability of p_0 if $\varepsilon_0 \le \varepsilon$, $\delta_0 \ge \delta$, and $p_0 \ge p$. Thus, the greater ε or the smaller δ and p are, the more protection a (ε, δ, p)-privacy guarantee provides.

2.3 Attack: Cost and Strategies

A user cannot directly execute aggregate queries. Instead, it must issue search queries and infer sensitive aggregates from the returned results. There may be an access fee for search queries over a proprietary hidden database. Even for hidden databases with publicly available interfaces, the server usually limits the number of queries a user can issue before blocking the user's account or IP address (e.g., Google SOAP Search API enforces a limit of 1,000 queries per day [Google08]). Thus, we define the *attacking-cost limit* u_{max} as the maximum number of search queries that a user can issue. Such limits make it unrealistic for an attacker to completely crawl a large hidden database. However, as recent research [DDM07, DZD09] has shown, such databases are vulnerable to *sampling attacks*, which are based on the generation of uniform random samples (with replacement) from the database and the approximate estimation of sensitive aggregates from the samples.

2.4 Defense: Dummy Insertion

We study the strategy of inserting dummy tuples to defend against sampling attacks. To enable a *bona fide* search user to distinguish real tuples from dummies, we propose to accompany each returned tuple with a CAPTCHA flag indicating whether it is real or dummy. Unlike the CAPTCHA challenge discussed in Section 1, our scheme applies to a large class of third-party applications such as meta-search engines[2]. In particular, a meta-search engine queries several hidden databases based on a search condition specified by an end-user, and then returns the union of all returned tuples after (re-)sorting or filtering. In the process, it treats all tuples in the same manner regardless of real or dummy. The engine does not parse any CAPTCHA itself, and instead simply forwards the tuples with the CAPTCHA flags they receive from the hidden database to end-users.

An important note of caution is that the attribute values of a dummy tuple may allow an adversary to identify it as a dummy, essentially incapacitating the CAPTCHA flag. In particular, an adversary may identify a dummy tuple by checking whether its attribute values violate constraints obtained by the adversary through external knowledge (e.g., that the airline in Example 2 does not operate any flight out of Seattle, WA). A solution to this problem requires the proper modeling of external knowledge, which we will leave as an open problem. In this paper, we assume that there exists a dummy-generating oracle DUMMY-ORACLE(m_0, C) which generates dummy tuples that (1) satisfy the search condition C, and (2) cannot be identified as a dummy by the adversary. The oracle terminates when either no more tuples satisfying the above conditions can be generated, or m_0 dummy tuples have been generated, whichever occurs first.

2.5 Problem Definition

The objective of defense is to protect sensitive aggregates while maintaining the *utility* of search queries to normal users.

[2] e.g., *http://www.kayak.com/*

We measure the (loss of) utility by the number of inserted dummy tuples: more dummy tuples increases inconvenience for normal users and hence reduces utility[3]. The problem studied for the remainder of this paper is formally defined as follows:

Problem: *Given the attacking-cost limit and a set of sensitive aggregate queries, the objective of dummy insertion is to achieve* (ε, δ, p)*-guarantee for each aggregate query while minimizing the number of inserted dummy tuples.*

3. SAMPLING ATTACK AND DEFENSE

In this section, we develop the main intuitive ideas behind COUNTER-SAMPLER, our algorithm for defense against all possible sampling attacks. The actual algorithm and its analysis are presented later in Section 4. For simplicity, in this section we shall consider Boolean databases (extensions for categorical and other non-Boolean databases will be discussed later in Section 4). We assume the actual number of tuples m is much smaller than the space of possible tuples, which is at least 2^n. We shall frequently refer to the two running examples in Table 1:

Example A: A Boolean database with m unique tuples on n attributes. Only one tuple $t_1 = <0,\ldots, 0,1>$ satisfies $a_1=a_2=0$.

Example B: A Boolean database with $m = 2^l$ unique tuples on n attributes. All tuples satisfy $a_1 = \cdots = a_{n-l} = 0$.

Table 1. Examples

a_1	a_2	a_3	$\cdots$	a_{n-1}	a_n
0	0	0	$\cdots$	0	1
1	0	0	$\cdots$	1	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
0	1	1	$\cdots$	0	0

Example A

a_1	$\cdots$	a_{n-l}	$\cdots$	a_n
0	$\cdots$	0	$\cdots$	1
0	$\cdots$	0	$\cdots$	0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
0	$\cdots$	0	$\cdots$	1

Example B

3.1 A Common Characteristic of Sampling Algorithms

We begin by finding a common characteristic for all existing sampling attacks. Recall that a search query has three possible outcomes: overflow, valid, underflow. A common characteristic for all existing samplers [DDM07, DZD09] is: to obtain a uniform random sample tuple t, a sampler must have discovered at least one *valid* search query that contains t in its result. The reason for this property is that sample tuples are only retrieved from results of valid queries: An underflowing query of course does not return any tuple. And even though an overflowing query returns k tuples, these are the top-k tuples preferentially selected by a ranking function, and hence cannot be assumed to be random. We focus on defending against sampling algorithms with this property in the paper.

It is also worth noting a restriction on how valid queries can be discovered: they cannot be constructed using tuples returned by overflowing queries. Without this restriction, a trivial way to generate a valid query is to issue SELECT * FROM D, pick one of the k returned tuples, and construct an n-predicate query from it. Such valid queries are useless for sampling due to the same reason why overflowing queries cannot be used: The returned tuples are not randomly but preferentially selected by some ranking function.

[3] Admittedly, this is a simplified measure because a dummy tuple in a frequently issued search query might introduce more inconvenience than one that is rarely retrieved. We propose to study such query-distribution-based utility measures in the future.

3.2 Single-Sample Attack and Defense

We start with a simple version: how a sampler can obtain *one* uniform random sample tuple by finding one valid query, and how to defend against such single-sample attacks by forcing a sampler to issue a large number of queries. There are two important concepts related to the space of search queries used in this subsection: the *universal space* Ω and the *active space* Θ. The universal space Ω is the set of all possible search queries. The active space Θ will be defined during the discussion later, but intuitively, at any point during the search for a valid query, Θ is a subset of Ω containing only those queries that are candidates for issuing at a subsequent time. That is, after a sampler receives a query answer, it removes from Θ all queries which it determines should never be issued later.

3.2.1 How to Find a Single Valid Query

Consider the task of finding one valid query. A naïve sampler may randomly choose a query from Ω, submit it through the interface, and retry if the query returns not valid. However, this is extremely inefficient: Each tuple may be retrieved by (at most) ${}^nC_0 + {}^nC_1 + \cdots + {}^nC_n = 2^n$ queries, where ${}^nC_i$ is the number of i-predicate queries that retrieve the tuple[4]. On the other hand, the universal space Ω contains 3^n unique queries because each attribute a_i has three possibilities in the query specification (i.e., $a_i = 0$, $a_i = 1$, or not specified). Thus, the probability of picking a valid query is at most $m \cdot (2/3)^n$, which is extremely small when n is large. Thus, such a naïve sampler may need to issue a very large number of queries before it encounters a valid query.

Due to the low percentage of valid queries in Ω, in order to find a valid query without incurring a high query cost, any reasonable sampler should not blindly guess the next query to issue. Instead, it should adopt a "smarter" strategy to reduce the active space of candidate queries based on the previously received query answers and improve efficiency.

In what follows we investigate the characteristics of such smart samplers that attempt to reduce the active space of candidate queries. We first describe examples that illustrate the various ways in which Θ can be shrunk. We then follow with a more general quantification of the amount of shrinkage possible by receiving answers to any arbitrary search query.

Examples of Active Space Shrinkage: As a sampler issues queries and receives their answers, it is able to shrink the active space Θ. For example, let the first query issued be Q_1: SELECT * FROM D WHERE $a_1 = 1$. We discuss the three possible outcomes of Q_1: underflow, overflow, and valid, respectively:

Case 1: Q_1 underflows: In this case the size of Θ decreases to 3^{n-1} because

- Since Q_1 underflows, any query with predicate $a_1 = 1$ will underflow. Thus, all such queries can be removed from Θ.
- Any query with predicate $a_1=0$ can also be removed because it will remain the same after removing predicate $a_1 = 0$. That is, it can be reduced to another query in the remaining Θ.
- Thus, Θ contains only 3^{n-1} queries with no predicate of a_1.

Case 2: Q_1 overflows: Any *smart* sampler can first remove from Θ two queries, Q_1, and SELECT * FROM D, because they both will overflow. Furthermore, since Q_1 overflows, at least $k + 1$ queries with predicate $a_1 = 1$ are valid. Since the sampler only aims to find one valid query, it may choose to remove from Θ all queries except those with predicate $a_1 = 1$. After doing so, the size of Θ becomes 3^{n-1}.

[4] The reason is that once the i (out of n) attributes are chosen, their values in the search condition are determined by the tuple's values.

Case 3: Q_1 is valid: The *sampler* concludes the search for a valid query. The new size of Θ is shrunk to 1.

As we can see, no matter whether Q_1 underflows, overflows, or returns valid answers, it always shrinks Θ and thereby reduces the query cost for obtaining one valid query. The formal definition of Θ is as follows:

Definition 3.1 (Active Space): *The active space Θ is the minimum subset of Ω for which there exists a previously issued overflowing query Q such that every query Q′ ∈ Ω which contains all predicates of Q can be inferred from the previously received query answers and the answer to a query* in Θ.

Quantifying Active Space Shrinkage: We now generalize the above observations and derive the amount of shrinkage possible by any arbitrary query Q_c containing c Boolean predicates:

- An underflowing Q_c can remove from Θ as many as $(c + 1)\cdot 3^{n-c}$ queries consisting of two disjoint sets: (1) at most 3^{n-c} queries each of which includes all predicates of Q_c, as these queries always underflow, and (2) at most $c\cdot 3^{n-c}$ queries each of which includes $c - 1$ predicates of Q_c and the complement of the remaining predicate of Q_c, because such a query is equivalent to a corresponding query (that is retained in Θ) with the last predicate removed.
- An overflowing Q_c may lead to an active space only containing queries with all of Q_c's predicates, as it is guaranteed to contain at least one valid query. Thus, Θ may shrink to size as small as $\approx |\Theta|/3^c$. Note that when c is large, such reduction of Θ is usually much more significant than that with an underflowing query.
- A valid query Q_c concludes the search for the current valid query and shrinks Θ to size 1.

Key Observations: Based on the above discussion, we make two key observations on single-sample attack characteristics:

- Shrinking Θ significantly reduces sampling query cost. Thus any smart sampler should attempt to issue queries that maximize the shrinkage of Θ.
- In particular, valid queries as well as *long* overflowing queries contribute the most to shrinking Θ and thereby lead to the most reduction in the cost of sampling. Here "long" refers to queries having many predicates. Thus a smart sampler should attempt to issue these types of queries as often as possible.

3.2.2 Neighbor Insertion: Defense against Single-Sample Attacks

To defend against single-sample attacks, we introduce a strategy called **Neighbor Insertion** based on the attacking characteristics discussed above. Recall from Section 1 that we cannot change/remove existing tuples, as this will pose a severe adverse impact to normal search users. Instead, we propose to insert dummy tuples into the database in such a way that valid and/or underflowing queries get converted into overflowing ones, thus making the task of finding a valid query by a sampler difficult (note that with the insertion of dummy tuples, overflowing queries will continue to overflow).

Recall from the key observations above that an efficient single-sample attack relies on two types of queries: valid queries and long overflowing queries. These two types form the main threat to defense. Fortunately, long queries (both overflowing as well as valid) are usually difficult for a sampler to find even before dummy insertion: within the $(2^c \cdot {}^nC_c)$ c-predicate queries in Ω, the total number of valid and overflowing queries is at most $m\cdot {}^nC_c$; thus the probability of choosing one is no more than $m/2^c$, which is extremely small when c is large[5]. This inherent difficulty of finding long valid/overflowing queries leaves us with *short valid queries* as being the most dangerous threat to the defense. Thus, our objective of dummy tuples insertion is to convert short valid queries into short overflowing queries; the latter are orders of magnitude less effective in shrinking Θ (recall that a c-predicate overflowing query Q_c reduces Θ to approximately $|\Theta|/3^c$).

To convert short valid queries to short overflowing queries, our basic idea is to insert dummy tuples into the "neighboring zone" of real tuples (i.e., sharing the same values on a large number of attributes). In Example A, <0,…,0,0> is a neighbor of t_1 = <0, …,0,1>, and may be added as a dummy tuple to overflow an originally valid query SELECT * FROM D WHERE a_1=0 AND a_2=0 when k = 1. In our algorithm, we choose to add dummy tuples such that all valid queries with fewer than b predicates will overflow, where b is a parameter. We refer to such a method as *b-neighbor insertion*. The detailed algorithm and analysis will be provided in Section 4.

3.3 Multi-Sample Attack and Defense

Having discussed the simple single-sample cases, we now consider practical cases where a sampler must obtain *multiple* uniform random sample tuples. We will point out the key differences between the two cases, and present an additional strategy called *high-level packing* to defend against all possible multi-sample attacks.

3.3.1 How to Find Multiple Valid Queries

Recall from Section 3.1 that obtaining multiple sample tuples may require the sampler to find multiple valid queries. To do so, a sampler can always invoke a single-sample attack for multiple rounds by resetting Θ to Ω at the end of each round. For such samplers, our neighbor insertion technique discussed above would provide adequate defense. However, such samplers are hardly optimal - they are essentially memory-less and use no information from queries answered in the previous rounds. In what follows, we investigate the characteristics of "smart" samplers that can take advantage of historic queries in multi-sample attacks, and then describe additional defense techniques to counter such attacks.

Recall two important concepts: the universal space Ω and the active space Θ. For multi-sample cases, we further distinguish between two types of active spaces, the *essential space* Θ_E and the *focused space* Θ_F. They will be formally defined later in the discussion, but intuitively, Θ_E is a subset of Ω that excludes all queries the sampler knows will not be issued during the *entire* search process, while Θ_F is a subset of Θ_E that further excludes all queries the sampler knows should not be issued during the search for the *current* valid query. For single-sample attacks, $\Theta=\Theta_E=\Theta_F$. However, for multi-sample attack, Θ_E excludes queries based on historic information that a sampler is able to carry over to future rounds.

Examples Comparing with Single-Sample Attacks: Consider again Q_1: SELECT * FROM D WHERE a_1 = 1. The consequence of an underflowing Q_1 is similar to the single-sample case. All queries removed from Θ in that case can be simultaneously removed from Θ_E and Θ_F. However, there are two important differences in the aftermath of valid and overflowing queries:

[5] Esp. when the size of the database is much smaller than the space of all possible tuple values ($m << 2^n$). This is usually the case in practice.

First, a valid Q_1 shrinks Θ_E for the subsequent search of valid queries, but not to the size of 1 (as was the case with Θ for the single-sample case). In particular, the size of Θ_E will decrease to 3^{n-1} due to the following reasons:

- All queries with predicate $a_1 = 1$ can be removed from Θ_E because their answers can be inferred from the returned tuple of Q_1 (by matching them with all predicates).
- All queries Q without predicate of a_1 can also be removed from Θ_E because the answer to such a query is a union of (1) the tuple in Q_1 that match all predicates of Q, and (2) the answer to Q' which is the same as Q except having an additional predicate a_1=0. (i.e., Q can be reduced to Q').
- Thus, the essential space Θ_E contains only queries with predicate $a_1 = 0$. Similar to the discussion in Section 3.2.1 for overflowing queries, there are 3^{n-1} such queries.

Second, with an overflowing Q_1, the distinction between Θ_E and Θ_F becomes necessary. Recall that in a single-sample case, two types of queries are removed from Θ: The first type has two queries, Q_1 and SELECT * FROM D, which always overflow. These can be removed from both Θ_E and Θ_F in multi-sample attacks. The second type contains all queries without predicate $a_1 = 1$, because the sampler chooses to focus on queries with this predicate. These queries can be removed from Θ_F as they should never be issued during the search for the current valid query. However, they cannot be removed from Θ_E if the sampler needs to find more valid queries than the two guaranteed in Θ_F.

Formally, we have the following definition of Θ_E:

Definition 3.2 (Essential Space): *The essential space* Θ_E is the minimum subset of Ω *such that for every query* $Q \in \Omega$, *there exists* $Q' \in \Theta_E$ *such that the answer to* Q *can be inferred from the answer to* Q' *and the previously received query answers*.

That is, a query should be excluded from Θ_E iff (1) its answer can be inferred from the previously received query answers, or (2) it can be reduced to another query in the new Θ_E. Such a query never needs to be issued. Thus, the size of Θ_E decreases monotonically over time. Based on the definition of Θ_E, the formal definition of Θ_F is as follows:

Definition 3.3 (Focused Space): *The focused space* Θ_F *is the minimum subset of* Θ_E *for which there exists a previously issued overflowing query* Q *satisfying the next drawn sample tuple, such that* Θ_F *includes every query* $Q' \in \Theta_E$ *which contains all predicates of* Q.

The focused space Θ_F only contains queries in Θ_E that the sampler chooses to focus on for finding the current valid query. A query in $\Theta_E \backslash \Theta_F$ may be issued to find other valid queries in the future. In particular, to generate uniform random samples, Θ_F decreases monotonically during the process of finding each valid query, but must be reset to Θ_E to find the next valid query.

Quantifying Shrinkage of Θ_E and Θ_F: We *again* generalize the results to any c-predicate search query Q_c:

- An underflowing Q_c has similar consequence as the single-sample case: up to $(c+1)\cdot 3^{n-c}$ queries should be removed from both Θ_E and Θ_F.
- An overflowing Q_c removes from Θ_E as many as 2^c queries which are formed by a subset of the predicates of Q_c because they all overflow. In addition, Q_c leads to a focused space Θ_F consisting of queries with all of Q_c's predicates. The size of Θ_F can be as small as $|\Theta_E|/3^c$.
- A valid query Q_c concludes the search for the current valid query. Similar to the underflowing case, it also removes from Θ_E as many as $(c+1)\cdot 3^{n-c}$ queries.

Key Observations: Based on the above *discussions*, we make two key observations on the characteristics of multi-sample attacks:

- For multi-sample attacks, shrinking Θ_E contributes more to the efficiency of sampling than shrinking Θ_F, because the shrinkage of Θ_E accelerates the process of finding *all* remaining valid queries while the shrinkage of Θ_F only accelerates the search for the current valid query.
- In particular, *short* underflowing queries become a major threat to defense because each of them may remove from the "critical" Θ_E as many as $(c+1)\cdot 3^{n-c}$ queries, which is much more than the reduction of 2^c queries by an overflowing query. Thus, a smart multi-sample attacker should attempt to issue these types of queries as often as possible.

3.3.2 High-Level Packing: Defense against Multi-Sample Attacks

We introduce another dummy insertion defensive strategy called **High-Level Packing** based on the attacking characteristics discussed above. The objective here is to address the new threat from short underflowing queries. We do so by converting such queries into short overflowing queries which are much less effective for shrinking Θ_E. With such conversion, the reduction in the size of Θ_E may be lessened by orders of magnitude, from $(c+1)\cdot 3^{n-c}$ to 2^c (e.g., from 8.6×10^7 to 32 when n=20 and c=5). Although the converted short overflowing query also sets Θ_F to size $|\Theta_E|/3^c$, the impact of Θ_F only lasts during the search for the current valid query.

To convert short underflowing queries to overflowing ones, we need to "pack" such queries with dummy tuples. In Example B, when k = 1, <1,0,…,0> and <1,0,…,1> may be added as dummy tuples to overflow an originally underflowing query SELECT * FROM D WHERE $a_1 = 1$. More generally, in our algorithm we choose to add dummy tuples such that *all* underflowing queries with fewer than d predicates will overflow, where d is a parameter. We refer to such a method as *d-level packing*.

In our COUNTER-SAMPLER algorithm, high-level packing is used alongside neighbor insertion to defend against all multi-sample attacks. The detailed algorithm and analysis are provided in the next section.

4. COUNTER-SAMPLER

In this section, we present the detailed algorithm of COUNTER-SAMPLER and analyze its performance.

4.1 Algorithm COUNTER-SAMPLER

Algorithm COUNTER-SAMPLER consists of two steps: d-level packing (Lines 1-6) and b-neighbor insertion (Lines 7-15). The subroutine DUMMY-ORACLE was discussed in Section 2.4. Lines 1-6 ensure that no (d–1)-predicate (or shorter) query will underflow, while Lines 7-15 ensure that no (b–1)-predicate (or shorter) query will be valid. Thus, Algorithm COUNTER-SAMPLER achieves d-level packing and b-neighbor insertion.

Note that while COUNTER-SAMPLER does not require the hidden database D to be Boolean, for ease of discussion our following analysis will be restricted to Boolean databases. We will discuss categorical and numerical databases in Section 4.5. Also, for this moment, we assume b and d are parameters set by the hidden database owner. How they should be determined will be discussed in Section 4.2.

Algorithm COUNTER-SAMPLER(b, d, k)

// Start of d-level packing
1: **for each** set S of $d-1$ attributes
2: K_S = Cartesian product of domains of attributes in S;
3: D_S = SELECT S FROM D;
4: **for each** tuple t in SELECT * FROM $K_S - D_S$
5: $C = \wedge_{a_i \in S}(a_i = t[a_i])$;
6: $D = D \cup$ DUMMY-ORACLE($k+1$, C);
// Start of b-neighbor insertion
7: $D_{\text{DUMMY}} = \phi$;
8: **for each** set S of $b-1$ attributes
9: D_S = SELECT S FROM D;
10: N_S = SELECT S, COUNT(*) FROM $D \cup D_{\text{DUMMY}}$ GROUP BY S HAVING COUNT(*) > k;
11: **for each** tuple t in SELECT * FROM $D_S - N_S[S]$
12: $C = \wedge_{a_i \in S}(a_i = t[a_i])$;
13: c = SELECT COUNT(*) FROM D$\cup D_{\text{DUMMY}}$ WHERE C;
14: $D_{\text{DUMMY}} = D_{\text{DUMMY}} \cup$ DUMMY-ORACLE($k+1-c$, C);
15: $D = D \cup D_{\text{DUMMY}}$;

4.2 Privacy Guarantee

To provide a privacy guarantee against *any* sampler that aims to draw uniform random samples from the hidden database, we will first prove a lower bound on the expected number of search queries required for a sampler to find s uniform random sample tuples over a Boolean hidden database. Based on the bound, we will derive a (ε, δ, p)-privacy guarantee achieved by COUNTER-SAMPLER.

Before formally presenting the results, we would like to point out that although Algorithm COUNTER-SAMPLER is quite intuitive, the theoretical analysis is quite challenging and consequently the derived bounds are rather loose. We also point out that the results address the class of samplers that use finding *valid* queries as a part of their sampling strategy. Our goal of presenting these analytical results is not to promote the tightness of bounds, but to demonstrate the versatility of COUNTER-SAMPLER in defending against any arbitrary sampler. However, as our experiments in Section 6 show, the actual performance is much better in practice.

Lemma 4.1: *For a Boolean hidden database with m tuples, after COUNTER-SAMPLER has been executed with parameters b and d such that* $4s(b-d)(d+1) \le 3^d$ *and* $m \le 2^{d-1}$*, any sampler needs at least an expected number of* $4s(b-d)/3$ *search queries to find s uniform random sample tuples over the database.*

Due to space limitation, we omit the proof – it is based on the main ideas discussed in Section 3, and the full details are available at [DZDC08]. Interestingly, the bound is decreasing with d, which seems to suggest that d-level packing is hurting the defense. However, this is not true because, in order for the lemma to hold, d must be large enough to satisfy the two conditions in the lemma. The reason why the bound decreases with d is because a "smart" sampler (which may be aware of the COUNTER-SAMPLER algorithm and hence knows b and d) does not need to issue any queries with fewer than d-predicates.

Based on the lemma, the following Theorem 4.2 provides privacy guarantees for sensitive COUNT aggregate queries. Recall from Section 2 that the attacking-cost limit $u_{\max}$ is the maximum number of search queries that an adversary can issue.

Theorem 4.2. *For a Boolean hidden database with m tuples, when all samplers have an attacking-cost limit* $u_{\max}$*, for any COUNT query with answer in* $[x, y]$*, the hidden database owner achieves* $(\varepsilon, \delta, 50\%)$*-privacy guarantee if COUNTER-SAMPLER has been executed with parameters b and d which satisfy*

(a) $d \ge \log_2 m + 1$ and $3^{d-1}/(d+1) \ge u_{\max}$

(b) $b \ge d + (3\varepsilon^2 u_{\max} / (32\min(x(m-x), y(m-y))(erf^{-1}(\delta))^2))$

where $erf^{-1}(\cdot)$ *is the inverse error function.*

The details of the proof are available in [DZDC08], but in brief, Conditions (a) and (b) directly follow from Lemma 4.1 and a lower bound on the number of samples required to win a (ε, δ)-privacy game, which can be derived via standard sampling theory. The theorem can be easily extended to other types of aggregates (e.g., SUM) by making proper assumptions on the distribution of the measure attribute.

Theorem 4.2 provides guidelines on the parameter settings for b and d. In particular, Conditions (a) and (b) imply lower bounds on d and b, respectively. Note that the smaller b or d is, the fewer dummy tuples are required. Thus, to achieve a $(\varepsilon, \delta, 50\%)$-privacy guarantee, for a given attacking-cost limit $u_{\max}$, we should first set d to be the minimum value that satisfies Condition (a), and then compute b as the minimum value that satisfies Condition (b). Ideally, we can follow these settings to make Algorithm COUNTER-SAMPLER parameter-less. Nonetheless, as discussed earlier, this theorem only provides necessary but not sufficient conditions for $(\varepsilon, \delta, 50\%)$-privacy guarantee. In practice, other (tighter) bounds on the b and d suffice, as we will demonstrate in the experimental results.

An interesting observation is that Condition (b) depends on the range $[x, y]$ of COUNT query answers. In particular, for given values of $u_{\max}$, b, d, and δ, the value of ε is maximized when $x = y = m/2$. This shows that COUNTER-SAMPLER provides the strongest privacy guarantee when a COUNT query is neither too broad nor too narrow. This is consistent with our objective (Section 1) of protecting fine-grained aggregates.

It is important to note that the privacy guarantee holds for *all* COUNT queries against possible sampling algorithms. This renders inference-based attacks that try to infer aggregates by combining answers of two or more non-sensitive queries, useless.

4.3 Number of Inserted Dummy Tuples

The privacy guarantee derived above is independent of the interface parameter k. The number of inserted dummy tuples however, depends on k because, intuitively, for a larger k more dummy tuples are required for making underflowing or valid queries overflow. The number of inserted dummy tuples also depends on the original data distribution. For example, when the data is densely distributed into a few clusters, b-neighbor insertion requires much fewer dummy tuples than when all attributes are i.i.d. with uniform distribution. For d-level packing, the i.i.d. case requires much fewer dummy tuples than when the data is highly skewed (e.g., all attributes have probability of 99% to be 1). Because of this dependency, we will not attempt a theoretical analysis of the number of dummy tuples inserted, and instead present a thorough experimental evaluation in Section 6.

4.4 Efficiency and Implementation Issues

Given b and d, the time complexity of COUNTER-SAMPLER is $O({}^nC_{d-1} \cdot \max(2^d, m) + {}^nC_{b-1} \cdot m)$, where n is the number of (searchable) attributes and m is the number of tuples. The efficiency is usually not a concern because (1) COUNTER-SAMPLER only needs to be executed once as a pre-processing step for a static database, and (2) n is usually quite small for a hidden database with web interface. However, COUNTER-

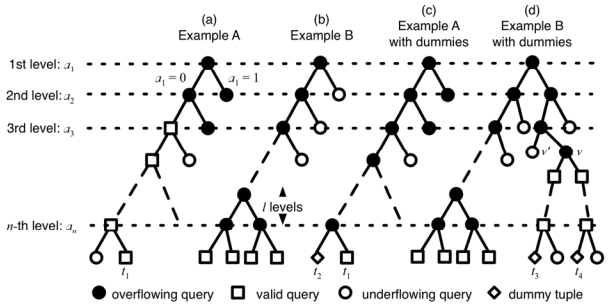

Figure 1. Query Trees for Examples A and B

SAMPLER can be slow when n is large. To address this problem, we present RANDOM-COUNTER-SAMPLER, a randomized version which replaces the enumeration of all (d–1)- or (b–1)-attribute sets by checking such attribute sets at random.

Algorithm RANDOM-COUNTER-SAMPLER(b, d, k)

1: **Randomly choose** a set S of d – 1 attributes
2: *Call Lines* 2 – 6 *of Algorithm COUNTER-SAMPLER*
3: Goto 1 until no dummy has been inserted for h iterations
4: $D_{\text{DUMMY}} = \phi$;
5: **Randomly choose** a set S of b – 1 attributes
6: *Call Lines* 9 – 14 *of Algorithm COUNTER-SAMPLER*
7: Goto 5 until no dummy has been inserted for h iterations
8: $D = D \cup D_{\text{DUMMY}}$;

Since the dummy tuples inserted for one attribute set may also overflow queries corresponding to other attribute sets, the number of dummy tuples required for the not-yet-chosen attribute sets decreases quickly. We terminate the process when no dummy tuple is inserted for h consecutive iterations. Thus, even when n is large, RANDOM-COUNTER-SAMPER is able to terminate quickly with a small probability of error.

4.5 Extensions for Categorical and Numerical Databases

Algorithm COUNTER-SAMPLER can be directly applied as-is to both Boolean and categorical databases. Nonetheless, for categorical data, the number of predicates of a query (e.g., b and d) is not an effective measure for the query's power in helping a sampler prune its search space. For example, consider an underflowing Q_1: SELECT * FROM D WHERE $a_1 = 1$. A binary a_1 can reduce Θ_E to about 2/3 of its original size, while an a_1 with 100 possible values can only reduce Θ_E to about 100/101 of its original size. Moreover, the number of all possible b-predicate queries is also determined by the size of the Cartesian product of all involved attributes' domains (i.e., multiplication of the domain sizes), rather than simply the number of predicates b. Thus, an extension of high-level packing and neighbor insertion to categorical databases is to replace d and b with two new parameters c_d and c_b, such that the Cartesian product size of underflowing and valid queries must be no less than c_d and c_b, respectively. Note that when the database is Boolean, we have $c_d = 2^d$ and $c_b = 2^b$. Our experiments presented in Section 6 adopt this strategy for categorical databases.

For numerical data, we can apply COUNTER-SAMPLER by appropriately discretizing the numerical data to resemble categorical data. However, different discretization techniques have different impact on the usability of the system. For example, a larger bucket size may reduce the precision of numerical values and affect search results usability. In contrast, a smaller bucket size may require a larger number of dummy tuples (e.g., to achieve the same c_d *and* c_b). How to choose an optimal discretization scheme is left as an open problem.

5. CASE STUDIES

The analysis in Section 4 showed that COUNTER-SAMPLER can effectively defend against *any sampler* over hidden databases. As examples, we now illustrate how COUNTER-SAMPLER defends against HIDDEN-DB-SAMPLER [DDM07] and HYBRID-SAMPLER [DZD09], two state-of-the-art sampling algorithms over hidden databases.

Defense against HIDDEN-DB-SAMPLER: We start our discussion by a brief overview of HIDDEN-DB-SAMPLER. Its basic idea is to repeatedly perform random "drill-down" searches over the query space. Consider Figure 1(a) and 1(b) which shows binary query trees for Examples A and B, respectively. Each tree is a complete binary tree with n+1 levels (though we only show parts of each tree due to space limitations). The i-th ($1 \le i \le n$) internal level represents attribute a_i, and the left (resp. right) edge downward from any i-th level node is labeled 0 (resp. 1), representing a predicate of a_i having the labeled value. Thus, the leaves (i.e., (n+1)-th level) represent all possible tuple values. Only a small proportion of the leaves will correspond to actual tuples; the vast majority will be empty.

The random drill-down approach essentially performs a random walk down this tree. The walk starts from the root node and takes a random path to level a_1 (by issuing a query with predicate corresponding to the path). If it returns overflow, then another step is taken to the next level (i.e., the query is appended by a random predicate involving a_2). This drill-down process shall either lead to a valid query, or return empty. If a valid query is reached, a tuple randomly chosen from the query result is returned as a sample after acceptance-rejection sampling.

HIDDEN-DB-SAMPLER has two variants, with fixed and random order of attributes, respectively. In the latter variant, the attributes order in the tree is randomly permuted before each random walk. For the ease of discussion, we consider a fixed order $[a_1,\ldots,a_n]$ (as Figure 1) for a Boolean database with $k = 1$.

Consider Example A. Before dummy insertion, an expected number of only 3 queries are needed to retrieve t_1:<0,…,0,1> (from SELECT * FROM D WHERE a_1=0 AND a_2=0). To see how b-neighbor insertion delays the sampling, consider $b = n$. A dummy tuple t_2:<0,…, 0,0> will be inserted to overflow a (b–1)-predicate query SELECT * FROM D WHERE a_1=0 AND ···AND a_{n-1}=0. Figure 1(c) depicts the tree after dummy insertion. As we can see, with the insertion of t_2, HIDDEN-DB-SAMPLER needs a minimum of n queries (traveling from the root to the leaf) to retrieve t_1 from a valid query. This is much more than the two queries needed before dummy insertion.

However, b-neighbor insertion alone is insufficient, as will be illustrated for Example B. In this case, for any $b \in [1, l]$, b-neighbor insertion will not insert any dummy tuple. As a result, the sampler remains efficient as it can still detect *underflows* on levels 1 to $n - l$ and prune a major portion (i.e., $1 - 1/2^{n-l}$) of the tree after issuing only $n - l$ underflowing queries.

Now consider the effect of d-level packing. Since there is no real tuple with $a_1 = 1$, two dummy tuples will be inserted with a_1=1 and equal values for any other $d - 2$ attributes, e.g., a_2 to a_{d-1}. Figure 1(d) depicts a tree after dummy insertion. As we can see, in the right half (i.e., a_1=1) of the tree, there is one overflowing node on every level between the 1st and the (d – 1)-th level. Consider such a node v at the x-th level. Since HIDDEN-DB-SAMPLER travels to left or right with equal probability, the probability for a sampler drawing s samples to reach v or its underflowing neighbor

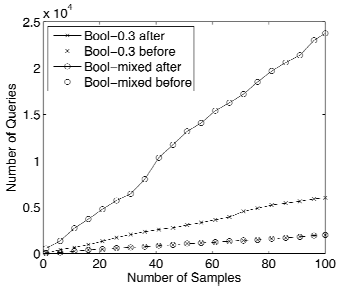

Figure 2: Number of queries before and after COUNTER-SAMPLER for Boolean

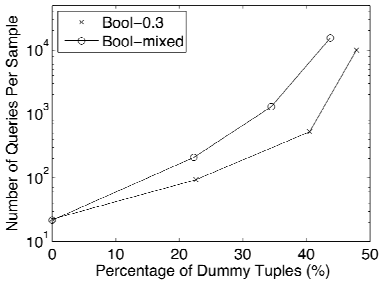

Figure 3: Delay of sampling vs. Percentage of dummy tuples

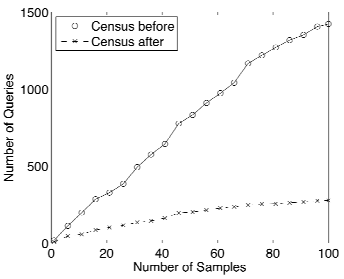

Figure 4: Number of queries before and after COUNTER-SAMPLER for Census

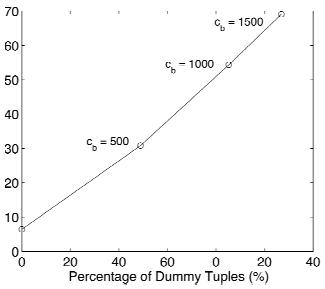

Figure 5: Delay of sampling vs. number of dummy tuples

v' is $p(v) = 1-(1-1/2^{x-1})^s$. Since $p(v)>1-1/e$ when $x \le \log_2 s + 1$, as long as $d \ge \log_2 s+1$, the two inserted dummy tuples increase the expected number of queries by at least $\Sigma_v(2p(v)) - 1 > 2(1 - 1/e)\log_2 s - 1$.

As we can see from the examples, by inserting a few dummy tuples, COUNTER-SAMPLER can substantially increase the query cost of HIDDEN-DB-SAMPLER.

Defense against HYBRID-SAMPLER: HYBRID-SAMPLER [DZD09] consists of two phases: pilot-sample collection and COUNT-assisted sampling. The vast majority of queries issued are in the phase of pilot-sample collection, where HIDDEN-DB-SAMPLER is called to collect a pre-determined number of pilot samples. As we illustrated above, COUNTER-SAMPLER can significantly increase the query cost of HIDDEN-DB-SAMPLER. Thus, it can also substantially delay sampling by HYBRID-SAMPLER. We will demonstrate such delay in the experimental results.

6. EXPERIMENTS AND RESULTS

In this section, we describe our experimental setup and present the experimental results. Note that the Section 4 provides theoretical privacy guarantees against all possible samplers. We now carry out empirical studies for two state-of-the-art sampling algorithms, HIDDEN-DB-SAMPLER and HYBRID-SAMPLER for Boolean and categorical hidden databases. We also draw conclusions on the individual impact of high-level packing and neighbor insertion on the delay of sampling attacks and the number of inserted dummy tuples.

6.1 Experimental Setup

1) Hardware and Platform: All our experiments were performed on a 1.99 Ghz Intel Xeon machine with 4 GB of RAM. The COUNTER-SAMPLER algorithm was implemented in MATLAB. We set $h = 40$ for the randomized version.

2) Data Sets: HIDDEN-DB-SAMPLER recommends different strategies for Boolean and categorical databases (with random order of attributes for the former and fixed order for the latter). Thus, we consider both Boolean and categorical datasets. Note that in order to apply COUNTER-SAMPLER, these datasets are offline ones to which we have full access.

Boolean Synthetic: We generated two Boolean datasets, each of which has 100,000 tuples and 30 attributes. The first dataset is generated as i.i.d. data with each attribute having probability of $p = 0.3$ to be 1. We refer to this dataset as the *Bool-0.3* dataset. The second dataset is generated in a way such that different attributes have different distribution. In particular, there are 30 independent attributes, 5 have probability of $p = 0.5$ to be 1, 10 have $p = 0.3$, the other 10 have $p = 0.1$. We refer to this dataset as the *Bool-mixed* dataset.

Categorical Census: The Census dataset consists of the 1990 US Census Adult data published on the UCI Data Mining archive [HB99]. After removing attributes with domain size greater than 100, the dataset had 12 attributes and 32,561 tuples. It is instructive to note that the domain size of the attributes of the underlying data is unbalanced in nature. The attribute with the highest domain size has 92 categories and the lowest-domain-size attributes are Boolean.

3) Sampling Algorithms: We tested two state-of-the-art sampling algorithms for hidden databases: HIDDEN-DB-SAMPLER [DDM07] and HYBRID-SAMPLER [DZD09].

HIDDEN-DB-SAMPLER has two variations, HIDDEN-DB-RANDOM and HIDDEN-DB-FIXED, which use random and fixed order of attributes, respectively. HIDDEN-DB-RANDOM can only be applied to Boolean data, while HIDDEN-DB-FIXED can also be applied to categorical data. HIDDEN-DB-RANDOM is parameter-less, while HIDDEN-DB-FIXED requires a parameter called scaling factor C for the acceptance/rejection module, in order to balance between efficiency and bias. Following the heuristic in [DDM07], we set $C = 1/2^l$ where l is the average length of random walks for collecting the samples.

HYBRID-SAMPLER has two parameters: s_1, the number of pilot samples collected for optimizing future sampling, and c_S, the count threshold for switching between HYBRID-SAMPLER and HIDDEN-DB-SAMPLER. Following the settings in [DZD09], we set $s_1 = 20$ and $c_S = 5$.

4) Performance Measures for COUNTER-SAMPLER: We evaluated two performance measures for COUNTER-SAMPLER. The first is *privacy* protection, i.e., the delay (or inefficiency) forced onto the sampling algorithms by COUNTER-SAMPLER. This is measured by the number of unique queries issued by HIDDEN-DB-SAMPLER and HYBRID-SAMPLER to obtain a certain number of samples. The second measure is the loss of *utility*, i.e., the overhead incurred to bona fide users. In particular, we used the number of dummy tuples inserted by COUNTER-SAMPLER, as discussed in Section 2.5.

6.2 Effectiveness of COUNTER-SAMPLER

HIDDEN-DB-RANDOM: Since HIDDEN-DB-RANDOM only supports Boolean data, we tested the effectiveness of COUNTER-SAMPLER on defending against HIDDEN-DB-RANDOM over the two Boolean synthetic datasets. We first applied COUNTER-SAMPLER with a fixed pair of parameters $b = 10$ (for b-neighbor insertion) and $d = 6$ (for d-level packing) when $k = 1$. In this case, COUNTER-SAMPLER inserts 29218

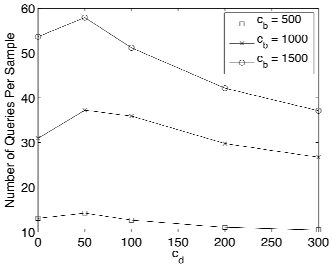

Figure 6: Delay of sampling vs. c_d for high-level packing

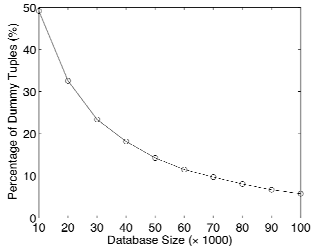

Figure 7: Percentage of Dummy Tuples vs. Database Size

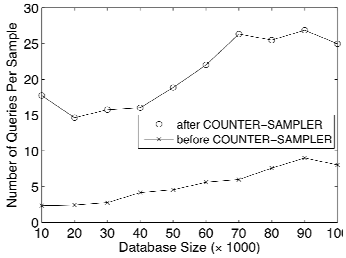

Figure 8: Efficiency of sampling vs. Database Size

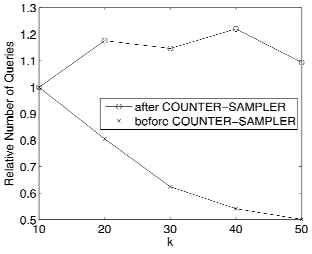

Figure 9: Efficiency of sampling vs. k

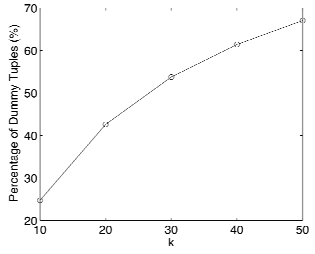

Figure 10: Percentage of Dummy Tuples vs. k

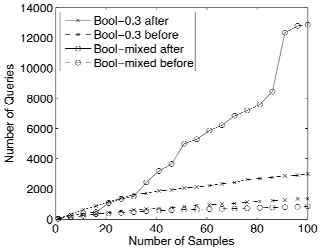

Figure 11: HYBRID-SAMPLER

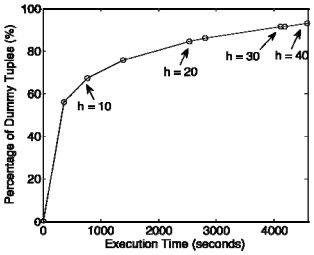

Figure 12: RANDOM-COUNTER-SAMPLER

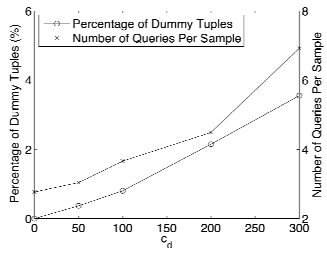

Figure 13: High-Level Packing Only

and 32717 dummy tuples for the Bool-0.3 and Bool-mixed datasets, respectively, leading to a proportion of 22.6% and 24.7% tuples of the final database being dummies. Figure 2 depicts the number of queries required by HIDDEN-DB-RANDOM to obtain 1 to 100 samples before and after COUNTER-SAMPLER is used. One can see from the figure that for both datasets, COUNTER-SAMPLER significantly increases the number of queries HIDDEN-DB-RANDOM has to issue. For example, to draw 100 samples from the Bool-mixed dataset, HIDDEN-DB-RANDOM requires only 1987 queries before COUNTER-SAMPLER is applied, but 23751 queries afterwards.

We also tested the tradeoff between the delay of HIDDEN-DB-RANDOM and the number of dummy tuples inserted by COUNTER-SAMPLER. To do so, we set $d = 5$ and vary b between 10, 12, and 15. Again, $k = 1$. Figure 3 depicts relationship between the percentage of dummy tuples in the final database and the average number of queries required by HIDDEN-DB-RANDOM to obtain a sample during the process of drawing 50 samples. One can see from the figure that the sampler can be delayed by orders of magnitude with a moderate number of dummy tuples. For example, HIDDEN-DB-SAMPLER requires over 700 times more queries when there are 43.7% dummy tuples in the dummy-inserted version of the Bool-mixed dataset.

HIDDEN-DB-FIXED: In [DDM07], HIDDEN-DB-FIXED is recommended over HIDDEN-DB-RANDOM for categorical data as it generally yields a smaller bias in the samples. Thus, we tested the effectiveness of COUNTER-SAMPLER in defending against HIDDEN-DB-FIXED over the Census dataset. We set $k = 10$ by default unless otherwise noted.

Similar to the experiments on Boolean synthetic datasets, we first applied COUNTER-SAMPLER with a fixed pair of parameters $c_d = 50$ (for high-level packing) and $c_b = 500$ (for neighbor insertion). In this case, COUNTER-SAMPLER inserts 10697 dummy tuples, leading to a proportion of 24.7% tuples of the final database being dummies. Figure 4 depicts the number of queries required by HIDDEN-DB-FIXED to obtain 1 to 100 samples before and after COUNTER-SAMPLER is used. One can see from the figure that COUNTER-SAMPLER significantly increases the number of queries HIDDEN-DB-FIXED needs to issue. For example, to obtain 30 samples, HIDDEN-DB-FIXED needs only 277 queries before COUNTER-SAMPLER is applied, but 1423 queries afterwards.

We again tested the relationship between the delay of HIDDEN-DB-FIXED and the number of inserted dummy tuples. To do so, we set $c_d = 100$ and varied c_b between 500, 1000, and 1500. Figure 5 depicts the results. As we can see, COUNTER-SAMPLER delays HIDDEN-DB-FIXED by more than ten-fold, from 6.41 to 69.07 queries per sample, when 53.4% tuples of the final database are dummies.

We also tested COUNTER-SAMPLER with varying c_d. Figure 6 shows the number of queries required by HIDDEN-DB-FIXED for each sample (while drawing 100 samples) when c_d varies from 0 to 300. One can observe from the figure a pattern that holds for all three cases of $c_b = 500$, 1000, and 1500: when c_d increases, the number of queries required by HIDDEN-DB-FIXED first increases, and then decreases. This verifies what is indicated by our theoretical results in Lemma 4.1 and Theorem 4.2: once the value of d (resp. c_d) used by high-level packing reaches a threshold, the further increase of d (resp. c_d) can only reduce, and not improve, the protection provided by COUNTER-SAMPLER.

We tested COUNTER-SAMPLER with varying database sizes. In particular, we constructed 10 databases with 10,000 to 100,000 tuples by sampling with replacement from the Census dataset. Then, we applied COUNTER-SAMPLER with $c_b = 500$ and $c_d = 50$ to all of them. Figure 7 shows the percentage of dummy tuples in the dummy-inserted databases. One can see that this percentage decreases rapidly when k increases. For example, the percentage is 49.11% for the 10,000-tuple database but only 5.77% for the 100,000-tuple database. Meanwhile, the privacy protection provided by COUNTER-SAMPLER remains effective, as demonstrated in Figure 8.

We also studied the impact of the interface parameter k on COUNTER-SAMPLER. In particular, we tested cases where k ranges from 10 to 50. To highlight the change of sampling efficiency with k, we used the case where $k = 10$ as the baseline

scenario, and calculated the *relative* number of queries required by HIDDEN-DB-FIXED for other values of k (for collecting the same number (100) of samples). Figure 9 shows the results before and after COUNTER-SAMPLER with c_b = 500 and c_d = 50 is applied. One can see from the figure that without COUNTER-SAMPLER, the number of queries required by HIDDEN-DB-FIXED decreases rapidly with increasing k. After COUNTER-SAMPLER is applied, however, the number of queries remains stable for all values of k. This is consistent with the fact that our privacy guarantees derived in Lemma 4.1 and Theorem 4.2 are independent of the value of k.

Figure 10 shows the change in the number of dummy tuples with k when c_b = 500 and c_d = 50. Naturally, with a larger k, more dummy tuples are needed to achieve the same values of c_b and c_d. Nonetheless, recall from Figure 7 that the percentage of dummy tuples decreases rapidly with the database size. Thus, for a real-world hidden database which has a very large number of tuples and also a large k, the percentage of dummy tuples inserted by COUNTER-SAMPLER should remain small.

HYBRID-SAMPLER: To demonstrate the universality of COUNTER-SAMPLER on defending against any samplers, we tested it against another sampling algorithm, HYBRID-SAMPLER [DZD09], over the two Boolean synthetic datasets. Similar to Figure 1, we set b = 10 and d = 6 when k = 1, leading to 29218 (22.6%) and 32717 (24.7%) dummy tuples for Bool-0.3 and Bool-mixed, respectively. Figure 11 depicts the number of queries required by HYBRID-SAMPLER to obtain 1 to 100 samples before and after COUNTER-SAMPLER is used. One can see from the figure that HYBRID-SAMPLER is also significantly delayed by COUNTER-SAMPLER. For example, to draw 100 samples from the Bool-mixed dataset, HIDDEN-DB-RANDOM requires only 851 queries before COUNTER-SAMPLER is applied, but 12878 queries afterwards.

PREPROCESSING EFFICIENCY: COUNTER-SAMPLER is essentially a preprocessing step; hence its runtime efficiency is usually not a concern. Nevertheless, we observed that it is quite efficient for real-world categorical hidden databases that usually have a smaller number of attributes. For example, for the Census dataset, the deterministic version of COUNTER-SAMPLER only requires 91.18 seconds to complete when c_b = 500 and c_d = 50. We also performed experiments on Boolean dataset with many attributes, which represents an extremely inefficient scenario for the deterministic version. We tested the execution time of COUNTER-SAMPLER as well as RANDOM-COUNTER-SAMPLER on the Bool-mixed dataset (which has 30 attributes) when b = 10 and d = 5. Unsurprisingly, while the deterministic version took days to complete, the randomized version was much more efficient. Figure 14 shows the relationship between the percentage of dummy tuples inserted and the execution time. One can see that when h = 40, with 4585 seconds, RANDOM-COUNTER-SAMPLER inserts more than 92.84% of all dummy tuples inserted by the deterministic version.

6.3 Individual Effects of Neighbor Insertion and High-level Packing

We also studied the individual effects of neighbor insertion and high-level packing. As an example, we considered the HIDDEN-DB-FIXED sampling algorithm and the Census dataset. First, we applied only high-level packing, and then executed HIDDEN-DB-FIXED to collect 100 samples. Figure 13 depicts the number of queries required by HIDDEN-DB-FIXED and the percentage of dummy tuples when c_d ranges from 0 to 300. One can observe that high-level packing alone only inserts a very small amount of dummy tuples, but is already quite effective against HIDDEN-DB-FIXED. For example, when only 3.55% tuples of the final database are dummies, high-level packing can delay HIDDEN-DB-FIXED by a factor of 2.50 (from 2.77 to 6.92 queries per sample).

Second, we applied only neighbor insertion to the Census dataset, and then executed HIDDEN-DB-FIXED to collect 100 samples. Figure 14 depicts the number of queries required by HIDDEN-DB-FIXED and the number of inserted dummy tuples when c_b ranges from 0 to 1500. One can see from the figure that neighbor insertion alone imposes significant delays to HIDDEN-DB-FIXED. For example, HIDDEN-DB-FIXED requires 13.64 times more queries (37.10 vs. 2.77 queries per sample) after applying COUNTER-SAMPLER with c_b = 1500. Meanwhile, 53.01% tuples in the final database are dummies.

To get a rough estimate on the effect of including the dummy tuples on usability, we sampled 1,000 queries with replacement from each level of the query tree on the Census dataset, and found very few queries returning only dummies: When k=10 and 24.7% (resp. 54.3%) tuples are dummies, only 1.1% (resp. 4%) queries return only dummy tuples.

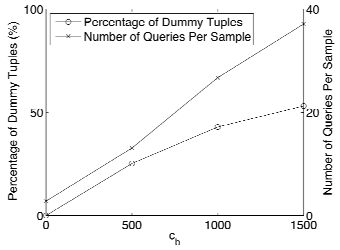

Figure 14: Neighbor Insertion Only

7. RELATED WORK

There has been recent work on crawling as well as sampling from hidden databases. However, here we restrict our discussion to prior works on sampling. In [DDM07, DZD09] the authors have developed techniques for sampling from structured hidden databases leading to the HIDDEN-DB-SAMPLER and HYBRID-SAMPLER algorithms respectively. A closely related area of sampling from a search engines index using a public interface has been addressed in [BB98] and more recently [BG06, BG07]. [CC01] and [IG02] use query based sampling methods to generate content summaries with relative and absolute frequencies while [HYJS04, HYJS06] uses two phase sampling method on text based interfaces. On a related front [CH02, BGM02] discuss top-k processing which considers sampling or distribution estimation over hidden sources. [CHW+08] discusses keyword search over a corpus of structured web databases in the form of HTML tables. [MKK+08] considers the surfacing of hidden web databases by efficiently navigating the space of possible search queries.

Much research on privacy/security issues in databases and data mining focused on the protection of individual tuples which is complementary to our proposed research. Traditional studies on access control and data sanitization are designed to limit the access to private data in relational databases [SCF+96, JSSS01]. Researchers have proposed various privacy-preserving (aggregate) query processing techniques, which can be classified as query auditing [NMK+06, KMN05] and value encryption/perturbation [AES03, AS00, Swe02, MKGV07, AST05]. The perturbation of (output) query answers has also been studied [DMNS06]. More closely related to the problem addressed in this paper is the existing work on protecting sensitive aggregation information [ABE+99, VEB+04, GV06]. Nonetheless, to our best knowledge, all existing work in this category focuses on the protection of sensitive association rules in frequent pattern mining.

8. FINAL REMARKS

In this paper, we have initiated an investigation of the protection of sensitive aggregates over hidden databases. We proposed Algorithm COUNTER-SAMPLER that inserts a number of carefully constructed dummies tuples into the hidden database to prevent the aggregates from being compromised by the sampling attacks. We derived the privacy guarantees achieved by COUNTER-SAMPLER against any sampler that aims to draw uniform random samples of the hidden database. We demonstrated the effectiveness of COUNTER-SAMPLER against the state-of-the-art sampling algorithms [DDM07, DZD09]. We performed a comprehensive set of experiments to illustrate the effectiveness of or algorithm.

Our investigation is preliminary and many extensions are possible. For example, we focused on the dummy tuple insertion paradigm in this paper. In the future work, we shall investigate other defensive paradigms, such as the integration of dummy insertion and query auditing, for protecting sensitive aggregates. We shall also investigate the techniques for the protection of sensitive aggregates against adversaries holding external knowledge about the underlying data distribution. Scenarios where hidden database interfaces return COUNT results [DZD09] also need to be explored. Another interesting future direction is the investigation of dynamic hidden databases and their impact on aggregates protection.

9. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their careful reading of the draft and thoughtful comments.

10. REFERENCES

[ABE+99] M. Atallah, E. Bertino, A. K. Elmagarmid, M. Ibrahim, V. S. Verykios, Disclose Limitation of Sensitive Rules. Knowledge and Data Exchange Workshop 1999.

[AES03] R. Agrawal, A. Evfimievski, and R. Srikant, Information Sharing Across Private Databases. SIGMOD 2003.

[AS00] R. Agrawal and R. Srikant, Privacy-Preserving Data Mining, SIGMOD 2000.

[AST05] R. Agrawal, R. Srikant, and D. Thomas, Privacy Preserving OLAP, SIGMOD 2005.

[BB98] K. Bharat and A. Broder. A Technique for Measuring the Relative Size and Overlap of Public Web Search Engines. WWW 1998.

[BG06] Z. Bar-Yossef and M. Gurevich. Random Sampling from a Search Engine's Index. WWW 2006.

[BG07] Z. Bar-Yossef and M. Gurevich: Efficient search engine measurements. WWW 2007.

[BGM02] N. Bruno, L. Gravano, A. Marian: Evaluating Top-k Queries over Web-Accessible Databases. ICDE 2002.

[CC01] J. P. Callan, M. E. Connell: Query-based sampling of text databases. ACM Trans. Inf. Syst. 19(2): 2001.

[CH02] K. C-C. Chang, S. Hwang: Minimal probing: supporting expensive predicates for top-k queries. SIGMOD 2002.

[CHW+08] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang, WebTables: Exploring the Power of Tables on the Web, VLDB 2008.

[CKV+03] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and M. Zhu, Tools for Privacy Preserving Distributed Data Mining, ACM SIGKDD Explorations, 4(28): 2003.

[DDM07] A. Dasgupta, G. Das, H. Mannila: A random walk approach to sampling hidden databases. SIGMOD 2007.

[DMNS06] C. Dwork, F. McSherry, K. Nissim, and A. Smith, Calibrating noise to sensitivity in private data analysis. Theory of Cryptography Conference 2006.

[DZD09] A. Dasgupta, N. Zhang, G. Das: Leveraging COUNT Information in Sampling Hidden Databases. ICDE 2009.

[DZDC08] A. Dasgupta, N. Zhang, G. Das, S. Chaudhuri, On Privacy Preservations of Aggregates in Hidden Databases, Technical Report TR-GWU-CS-09-001, George Washington University, 2009.

[EDHS07] J. Elson, J. R. Douceur, J. Howell, J. Saul: Asirra: a CAPTCHA that exploits interest-aligned manual image categorization, CCS 2007.

[Google08] http://code.google.com/apis/soapsearch/api_faq.html

[GV06] A. Gkoulalas-Divanis and V. S. Verykios, An Integer Programming Approach for Frequent Itemset Hiding. CIKM 2006

[HB99] S. Hettich and S. D. Bay, The UCI KDD Archive [http://kdd.ics.uci.edu]. Irvine, CA: University of California, Department of Information and Computer Science. 1999.

[HYJS04] Y. Hedley, M. Younas, A. E. James, M. Sanderson: A two-phase sampling technique for information extraction from hidden web databases. WIDM 2004.

[HYJS06] Y. Hedley, M. Younas, A. E. James, M. Sanderson: Sampling, information extraction and summarisation of Hidden Web databases. Data Knowl. Eng. 59(2): 2006.

[IG02] P. G. Ipeirotis, L. Gravano: Distributed Search over the Hidden Web: Hierarchical Database Sampling and Selection. VLDB 2002.

[JSSS01] S. Jajodia, P. Samarati, M. L. Sapino, V. S. Subrahmanian, Flexible support for multiple access control policies. TODS 26(2): 2001.

[KMN05] K. Kenthapadi, N. Mishra, and K. Nissim, Simulatable auditing. PODS 2005.

[MKGV07] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, *l*-Diversity: Privacy Beyond *k*-Anonymity. TKDD 1(1): 2007.

[MKK+08] J. Madhavan, D. Ko, Ł. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, Google's Deep-Web Crawl, VLDB 2008.

[NMK+06] S. U. Nabar, B. Marthi, K. Kenthapadi, N. Mishra, and R. Motwani, Towards robustness in query auditing. VLDB 2006.

[SCF+96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, Role-based access control models. IEEE Computer, 29(2): 1996.

[Swe02] L. Sweeney, *k*-anonymity: a model for protecting privacy. International Journal on Uncertainty, Fuzziness and Knowledge-based Systems, 10(5): 2002.

[VEB+04] V. S. Verykios, A. K. Elmagarmid, E. Bertino, Y. Saygin, and E. Dasseni, Association rule hiding, TKDE 16(4): 2004.

[ZZ07] N. Zhang and W. Zhao, Privacy-Preserving Data Mining Systems. IEEE Computer, 40(4): 2007.