Privacy Implications of Database Ranking

Md Farhadur Rahman[‡], Weimo Liu[†], Saravanan Thirumuruganathan[‡], Nan Zhang[†], Gautam Das[‡]
The George Washington University[†], University of Texas at Arlington[‡]

ABSTRACT

In recent years, there has been much research in the adoption of Ranked Retrieval model (in addition to the Boolean retrieval model) in structured databases, especially those in a client-server environment (e.g., web databases). With this model, a search query returns top-k tuples according to not just exact matches of selection conditions, but a suitable ranking function. While much research has gone into the design of ranking functions and the efficient processing of top-k queries, this paper studies a novel problem on the *privacy implications* of database ranking.

The motivation is a novel yet serious privacy leakage we found on real-world web databases which is caused by the ranking function design. Many such databases feature private attributes - e.g., a social network allows users to specify certain attributes as only visible to him/herself, but not to others. While these websites generally respect the privacy settings by not directly displaying private attribute values in search query answers, many of them nevertheless take into account such private attributes in the ranking function design. The conventional belief might be that tuple ranks alone are not enough to reveal the private attribute values. Our investigation, however, shows that this is not the case in reality.

To address the problem, we introduce a taxonomy of the problem space with two dimensions, (1) the type of query interface and (2) the capability of adversaries. For each subspace, we develop a novel technique which either guarantees the successful inference of private attributes, or does so for a significant portion of real-world tuples. We demonstrate the effectiveness and efficiency of our techniques through theoretical analysis, extensive experiments over real-world datasets, as well as successful online attacks over websites with tens to hundreds of millions of users - e.g., Amazon Goodreads and Renren.com.

1. INTRODUCTION

1.1 Motivation

While traditional structured databases generally support the Boolean Retrieval model (i.e., return all tuples that exactly match the search query selection condition), in recent years there has been much research into exploring the applicability of an alternate Ranked Retrieval model (e.g., a kNN interface that returns top-k tuples according to a suitable ranking function). The ranked retrieval model has become an important component of many databases, especially in a client-server environment (e.g., web databases, where a client specifies and sends queries via a web interface to a backend database). Prior research has primarily focused on the effective design of ranking functions and the efficient processing of top-k queries for a given ranking function (e.g., [5, 7, 19]).

However, in this paper we investigate a novel problem on the *privacy implications* of database ranking, which has not been studied before. We show how privacy leakage (through the top-k interface) can be caused by a seemingly innocent design of the ranking function in such ranked retrieval models.

To understand how the privacy leakage occurs, note that many databases in a client-server environment feature both public and *private* attributes. For example, social networking websites often allow users to specify privacy settings that hide certain attributes from the public's view, e.g., profile demographics such as race, gender, income; location; past posts, etc. These websites honor the privacy settings by omitting the private attributes from being displayed in the returned query answers. Thus, the results include a ranked list of k tuples, but with only the public attributes displayed, and the private attributes hidden.

The problem here, however, is that many websites indeed include these private attributes as *input* to the ranking function. The purpose of doing so is, understandably, to make ranking more effective - e.g., the friend-search feature in a social network would preferably return users that have similar demographics or behavior patterns (e.g., posting with similar frequencies) as the user who executes the search, as common-sense indicates that they are more likely to be interested in each other. From the privacy perspective, this design might look harmless as well - after all, while a ranking function might take as input a large number of attributes, its output is merely the (relative) rank of a tuple among returned results - not even the actual ranking score! Naturally, the traditional belief here is that it is impossible to infer private attribute values from just the ranking of a returned tuple.

In our investigation of real-world client-server databases (including popular web databases), we found this traditional belief to be *wrong*. Specifically, in this paper, we develop a novel technique which, by asking a carefully constructed sequence of top-k queries and observing the corresponding change of tuple ranks in the query answers, may successfully infer the value of private attributes.

Before introducing our technical results, we would like to first illustrate the real-world impact of this privacy leakage by briefly demonstrating a very simple attack one can deploy using this technique on Renren.com, the equivalent of Facebook in China which

Proceedings of the VLDB Endowment, Vol. 8, No. 10
Copyright 2015 VLDB Endowment 2150-8097/15/06.

has hundreds of millions of users. We chose this website as an example not only because of its large user base, but because it supports extensive privacy settings - allowing a user to specify as private any subset of profile attributes such as hometown, work affiliation, university attended, etc. It also respects these privacy settings in the display of search results - e.g., if a user specifies hometown as private and "only visible to friends", then the user's hometown information will be hidden from all search and/or recommendation results unless the query is issued by a friend of the user.

Nevertheless, we also found that when ranking users in search or recommendation results, the ranking function used by Renren.com takes into account *all* attributes of a user's profile, regardless of whether a user has specified it to be private and/or who is issuing the query. For example, Figure 1a shows the screenshot[1] of the ranked list of tuples (i.e., users) returned for a friend-search query issued by a user LIONEL with hometown = Beijing, China and no other profile attribute specified. The query is formed using the only public attribute of our victim user TARGET (with a red target icon in the screenshot), name = Jia Ming. Since TARGET sets his hometown to be a private attribute "only visible to friends" and LIONEL is not a friend of TARGET, the hometown of TARGET is hidden from display in the query answer. Figure 1b shows the answer to the exact same query after LIONEL changes his hometown to Shanghai, China. The rank of TARGET now moves up from No. 3 to No. 1 in the new answer - and indeed ranks even higher than a few other users with the same name from Shanghai and studying in Fudan university (in Shanghai). The change of rank indicates a strong likelihood of TARGET having hometown = Shanghai (even though it does not form a proof). In this paper, we shall show how one can indeed prove that TARGET must come from Shanghai using just a few other query answers.

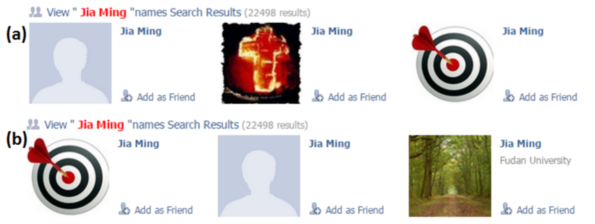

Figure 1: Demonstration of an attack over Renren

1.2 Novel Problem: Rank-Based Inference

The above motivating example led us to identify an important and novel problem of *ranked-based inference of private attributes*. From a conceptual standpoint, this problem is interesting as, to the best of our knowledge, privacy compromise from *tuple ranks* has not been studied before. From a practical standpoint, this problem is important as many client-server databases, especially web databases that attract large amounts of user contributions, commonly offer top-k query interfaces yet contain sensitive data (e.g., profiles, demographics) that users would like to keep private.

We formalize the problem as follows. Consider a database D with n tuples and $m + m'$ attributes, m of which $A_1, \ldots, A_m$ are public while the other m', $B_1, \ldots, B_m$, are private. The database allows top-k queries where k is a small number ($k \ll n$). To specify a query q, one assigns a predicate on *each* of the $m + m'$ attributes. The predicate can be point (i.e., $A_i = v$) or range (e.g., $B_i \in \{v_1, v_2\}$ or $*$, i.e., the entire domain).

[1]Note that, since Renren.com does not have an English version, this screenshot is taken in Chrome with the automated webpage translation feature of Google Translations enabled.

Given a top-k query q, the database computes a predetermined ranking function $s(t|q)$ for each tuple t in the database, and returns the k tuples with the smallest $s(t|q)$. Of course, only the m public attributes are displayed on the return interface - not the private attributes or the ranking score. In most websites, the ranking function is a closely guarded secret - so we assume the adversary has no knowledge of the ranking function other than two very basic properties, *monotonicity* and *additivity*, which we shall define in §2 and demonstrate that they hold for almost all reasonable ranking functions used in the real-world.

The objective of an adversary is to compromise the privacy of a pre-determined victim tuple v. Of course, the adversary can readily acquire the public attributes of v. Nonetheless, it does not know the ranking function being used (and of course no knowledge whatsoever of the tuples' ranking scores). Thus, the technical challenge for the adversary is to unveil the private attribute values, e.g., $v[B_1]$, by issuing a small number of queries through the web interface and observing only the public attribute values of the returned tuples and the order in which they are returned.

To the best of our knowledge, the above problem of inferencing sensitive data from the *ranking* of a tuple is very novel. While top-k querying has been extensively studied by the database community [7, 15, 19], much of the efforts were focused on (1) developing techniques to answer such queries efficiently [19, 20], and (2) designing proper distance/ranking functions for various applications [6, 22, 24]. There have been prior work on data privacy in the general area of *query inferencing* [8, 13, 16], but most focus was on learning individual values from aggregates such as SUM, MIN, MAX, etc. We discuss related work in more detail in §8.

1.3 Overview of Technical Results

As one of our important contributions, we introduce a comprehensive taxonomy of the problem space according to two dimensions: (1) the type of query interfaces widely used in practice and (2) the capability of adversaries. Then, for each subspace of the problem, we develop a novel technique which either guarantees the successful inference of private attributes, or (when such an inference is provably infeasible in the worst-case scenario) accomplishes the attack for a significant portion of real-world tuples.

Consider the first dimension. We distinguish between interfaces which only support "point queries" (i.e., a single value must be specified for each attribute in the query), and those that also support "IN queries" (i.e., where a subset/range of values can be specified for an attribute). For the second dimension, we distinguish between two types of adversaries: (1) those who are "query-only" (Q-only adversaries) - i.e., they are *passive* adversaries who only issue queries and observe query answers, but never tamper with (e.g., insert fake tuples into) the database; and (2) adversaries who "query-and-insert" (Q&I-adversaries), i.e., they only issue queries but also insert fake tuples into the database (e.g., by registering for fictitious user accounts on a social media website).

We have carefully investigated the four problem subspaces arising out of this taxonomy, and developed four novel attacks: Q&I-Point, Q-Point, Q&I-IN, and Q-IN. The fundamental ideas behind these attacks include two critical reductions: One reduces the problem of compromising a private attribute to finding so-called *differential queries* (defined in §4.1) which exclude all but one values in the domain. The second further reduces the problem to just finding a query which returns the victim tuple - nevertheless, this reduction holds only for Q&I-adversaries.

The differences on the applicability of these reductions lead to fundamentally different feasibilities of the attack, as illustrated in Table 1. Specifically, we find that while Q&I adversaries are always

Table 1: Feasibility, Worst- and Practical Query Cost

	Q&I-Point	Q-Point	Q&I-IN	Q-IN
Feasibility	Yes	Maybe	Yes	Maybe
Worst-case	$\prod_{i=1}^{m'} \lvert V_i^{\mathrm{B}} \rvert$	N/A	$\prod_{i=1}^{m'} \lvert V_i^{\mathrm{B}} \rvert$	N/A
In Practice	High	Highest	Lowest	Low

Note: $|V_i^{\mathrm{B}}|$ is the domain size for private attribute B_i.

able to accomplish the attack, there are cases where Q-only ones will fail. In terms of query cost, while the worst-case cost for even Q&I adversaries can be exponential, the query cost in practice is very reasonable - and can be significantly reduced when IN queries are available, even though IN has no impact on the (theoretical) worst-case query cost.

In summary, we make the following contributions in this paper:

- We have identified a novel and important problem of rank-based inferencing over web databases.
- We introduce a comprehensive taxonomy of the problem space, and identify four important subspaces based on varying database interface limitations and adversarial capabilities.
- For each problem subspace, we developed nontrivial adversaries, and carried out a rigorous theoretical analysis of their performance. Our results show that in almost all cases, the adversaries can launch efficient and successful attacks.
- We performed extensive experiments over real-world datasets and online experiments.

2. PRELIMINARIES

2.1 Model of Web Databases

As discussed in the introduction, many web databases store both public and private attributes of a user. Consider an n-tuple (i.e., n-user) database D with a total of $m + m'$ attributes, including m public ones $A_1, \ldots, A_m$ and m' private ones $B_1, \ldots, B_{m'}$. Let V_i^{A} and V_j^{B} be the attribute domain (i.e., set of all attribute values) for A_i and B_j, respectively. For the purpose of this paper, we consider V_i^{A} and V_j^{B} to be discrete and publicly known, and leave studies of numeric/infinite/unknown domains to future research.

We use $t[A_i]$ (resp. $t[B_j]$) to denote the value of a tuple $t \in D$ on attributes A_i (resp. B_j). For the purpose of this paper, we assume there is no duplicate tuple in the database (before an adversary makes any modification to the database) - i.e., every *bona fide* tuple has a unique value combination for the $m + m'$ attributes. While we assume that D does not change during the course of an attack, we include discussions in §4.2.1 to address the scenario where this assumption is violated.

Recall from the introduction that the database allows top-k queries where k is a small number such as 10 or 50. Given a *supported query* q defined below, the database computes the *ranking function* $s(t|q)$ for each tuple $t \in D$, and selects/returns k tuples in the *ascending* order of $s(t|q)$ (i.e., only the k tuples with minimum $s(t|q)$ will be returned some of which might not exactly match the query predicates). Of course, only the public attribute values, i.e., $t[A_1], \ldots, t[A_m]$, will be returned for each of the k tuples. Of course, since we allow duplicates on public attribute values - i.e., multiple tuples might share the same value combination on $A_1, \ldots, A_m$ - there must be a way to distinguish different returned tuples with the same public-attribute value-combination. For this purpose, we assume each tuple to be returned alongside a unique identifier (e.g., user ID) - and the adversary knows the unique identifier of the victim tuple as prior knowledge. It is important to note that the ranking score $s(t|q)$ is *not* returned - in addition, the design of $s(\cdot|\cdot)$ itself is a secret kept by the database owner.

Supported Queries: For the purpose of this paper, we consider ranking functions/queries that take into account all public and private attribute information. In other words, the web database supports queries which specify values/conditions on some or all of the $m + m'$ (public and private) attributes. In this paper, we consider two types of predicates that can be specified on an attribute: *point* and *IN*. Let the predicate specified in a query q for attribute A_i (resp. B_i) be $q[A_i]$ (resp. $q[B_i]$). A point predicate assigns a single value in the domain, i.e., $q[A_i] \in V_i^{\mathrm{A}}$, while an IN predicate assigns a subset of values, i.e., $q[A_i] \subseteq V_i^{\mathrm{A}}$. Consider a dating website as an example. While gender is often specified as a point predicate (i.e., male or female), interests and age can be considered IN ones (i.e., find users who most closely match the interest set {reading, travel, cycling, cooking} or age range $[25, 30]$). A special example of IN predicate is $q[A_i] = V_i^{\mathrm{A}}$ - i.e., $q[A_i] = *$ - indicating "do-not-care" on an attribute.

Practical Constraints: Most, if not all, web databases enforce practical constraints on how one might interact with the web interface. The two most important constraints here are *query-rate limitation* and *tuple insertion constraint*.

Most web databases enforce certain query-rate limits, i.e., limits on the number of queries one can issue (e.g., from an IP address or a user account) per time period (e.g., each day), in order to prevent overburdening of the backend database and/or third-party crawling of its contents. Hence an adversary must aim to minimize the query cost of a rank-based inference attack, as otherwise it would have to acquire more resources (e.g., more IP addresses, registering more accounts) in order to issue all queries required by the attack.

Tuple insertion constraint, on the other hand, refers to ones ability to *insert* tuples into the database. Some web databases, including many online social networks, do not enforce this constraint - i.e., one can freely insert new tuples (i.e., user accounts) to the database by registering for new accounts (e.g., using a new email address). Nonetheless, there are also others that require users' real identities and use offline authentication to check them. For example, catch22dating, a popular online dating website used in our real-world experiments, requires each user to have an authenticated identity as student of selected universities. For these databases, inserting new/fake tuples becomes extremely difficult, if not impossible. We say that the web database enforces a tuple insertion constraint which prevents an adversary from inserting arbitrary tuples.

2.2 Properties of Ranking Function

There has been significant research in database ranking (e.g., [17,19,20]) which studies the design of ranking function $s(t|q)$, including in cases where the query has IN predicates (e.g., [18,19]). While this paper aims to study *generic* rank-based inferences that work for a broad class of ranking functions, it is important to note that *no* attack will work without assuming certain properties of the ranking function. To understand why, consider a simple example where $s(t|q)$ is generated uniformly at random from $[1, n]$. Since the rank of a tuple has nothing to do with the tuple's (private) attribute values, no adversary can compromise any private information from the returned ranks. Thus, it is the objective of this subsection to define a minimum set of conditions that are satisfied by most if not all ranking functions used in practice. Specifically, we consider *monotonicity* and *additivity*, respectively as follows.

Monotonicity Condition: Intuitively, the monotonicity condition simply states that, for a given query, the relative rank between two tuples which differ only on one attribute should be determined by that attribute alone. Formally, for a point-query interface, if two tuples t and t' differ only on A_i and $t[A_i] = q[A_i]$, then t should have a smaller distance to q than t'. More generally, we have the following definition. Note that in this definition, we consider $q[A_i]$ (resp. $q[B_j]$) to be a set (in the case of point-query, containing just a single value) without introducing ambiguity.

Monotonicity: $\forall q$, $t \in D$, and $i \in [1, m]$ (resp. $j \in [1, m']$), if t and t' share the same value on all attributes except A_i (resp. B_j) and $t[A_i] \in q[A_i]$ while $t'[A_i] \notin q[A_i]$ (resp. $t[B_j] \in q[B_j]$, $t'[B_j] \notin q[B_j]$), there must be $s(t|q) < s(t'|q)$.

Additivity Condition: Intuitively, the additivity condition states that, for two tuples t and t', if t is already ranked higher than t' in query q, then further changing the predicate of q on A_i (resp. B_j) to exactly match t - i.e., making $q[A_i] = t[A_i]$ (resp. $q[B_j] = t[B_j]$) - should not change the relative rank between the two tuples. More formally, we have the following definition:

Additivity: $\forall q$ and $t, t' \in D$, if $s(t|q) < s(t'|q)$, then there must be $s(t|q') < s(t'|q')$, where q' is the same as q on all but one attribute A_i (resp. B_j), on which $q[A_i] = t[A_i]$ (resp. $q[B_j] = t[B_j]$).

Our studies of real-world web databases (in §7) verified this observation, as all websites considered satisfy both conditions.

3. PROBLEM SPACE

In this section, we define the rank-based inference problem studied in the paper. Specifically, we start with defining the objectives of an adversary. Then, we partition the entire problem space into four quadrants along two dimensions: the type of queries supported, and the type of operations an adversary can perform.

3.1 Adversary Model

The objective of an adversary is two-fold: *compromising privacy* and *minimizing query cost*. Privacy-wise, an adversary aims to compromise private attributes of a victim tuple v. Without loss of generality, we assume that the adversary aims to compromise the value of $v[B_1]$ based on prior knowledge of all public attributes of v, i.e., $v[A_1], \ldots, v[A_m]$. In §4, we shall address cases where an adversary aims to compromise all private attributes of v.

To ensure the versatility of our algorithms, we make a conservative assumption that an adversary has *no* prior knowledge of the ranking function other than the fact that it satisfies the monotonicity and additivity conditions defined above. Clearly, all algorithms in the paper still work if an adversary does know the ranking function. While it is possible that prior knowledge of certain ranking functions can enable more efficient attacks than those in the paper, we leave such ranking-function-specific studies to future work.

Given the query-rate limitation discussed in §2, an important goal of the adversary is to minimize the query cost for the attack, as otherwise the website-enforced limit on the number of queries from each user (e.g., IP-address) may stop the attack from being completed. To this end, it is important to note that our key efficiency measure here is the number of requests issued to the web database (hereafter referring to as the *query cost*, including both search queries and requests to insert tuples, if the database does not enforce the aforementioned tuple insertion constraint) - while other measures such as local (CPU or I/O) processing overhead are secondary.

3.2 Two Dimensions

The first dimension we use for partitioning the problem space is the type of queries supported. There are two different cases: (1) Point-Query Interface which requires a point predicate defined in §2 to be specified for *every* attribute. An example here is the friend recommendation offered by many social media websites - each user has to complete his/her own profile to enable the feature, essentially requiring the user to specify a point predicate on every public and private attribute. (2) IN-Query Interface which supports *IN queries* over all attributes. Clearly, here a user can choose "do not care" for an attribute by assigning its entire value domain to the IN condition. Since point queries are special cases of IN, all queries supported by the point-query interface are also supported here.

The second dimension for partitioning the problem space is the adversary power. Specifically, we consider the following two cases:

- *Query-and-Insert (Q&I) Adversary* can not only issue queries but also *insert* tuples to the database. It can also update or delete any tuple it inserted. These adversaries exist for websites which do not enforce the tuple insertion constraint.
- *Query-only (Q-only) Adversary* can query the web database but cannot change it. This is the case when the website enforces the tuple insertion constraint (see §2).

One can see from the definitions that Q&I adversaries are stronger - i.e., any attack launched by a Q-only adversary can also be launched by a Q&I-one, while the opposite is not true. We shall show later in the paper that the ability to *insert* leads to significant differences on the outcome of a rank-based inference attack. Specifically, while a Q&I adversary can *always* accomplish the attack even in the worst-case scenario, the same is not true for Q-only adversaries.

3.3 Problem Definition

Given the two dimensions, we partition the problem space into four quadrants: (1) point query interface with Q&I adversaries, (2) point query with Q-only, (3) IN with Q&I, and (4) IN with Q-only.

Problem Definition (Rank-Based Inference): Given a database D and a victim tuple $v \in D$, find the shortest sequence of queries $q_1, \ldots, q_c$ supported by the interface and a corresponding sequence of tuple sets $T_1, \ldots, T_c$, such that

$$\delta(q_1(D \cup T_1), q_2(D \cup T_2), \ldots, q_c(D \cup T_c)) = v[B_1]. \qquad (1)$$

where $q_i(D \cup T_i)$ is the answer to q_i over the $D \cup T_i$ and $\delta(\cdot)$ is a (deterministic) function for rank-based inferencing. For a Q-only adversary, there must be $T_1 = \cdots = T_c = \emptyset$.

Naturally, the problem could be extended to infer multiple, if not all, private attributes of victim tuple v. In fact, as we shall describe in §4, our algorithm iteratively learns private attribute values one at a time till $v[B_1]$ is inferred. Extending it to infer all attributes is trivial. To better illustrate our ideas and to significantly simplify the notations, in the theoretical discussions in this paper, we focus on the case where $k = 1$ (note that $k = 1$ is actually a conservative worst-case assumption for the attack design), and discuss the straightforward extension to larger k in the experiments section.

Running example of ranking function: All algorithms developed in this paper work for any ranking function satisfying monotonicity and additivity - so does all complexity and lower bound analysis. Nonetheless, when studying the practical performance of attacks and illustrating how different ranking-function designs affect attack effectiveness, it is necessary to consider certain concrete ranking function designs - for this purpose only, we consider the following linear ranking function as a running example:

$$s(t|q) = \sum_{i=1}^{m} w_i^{\mathrm{A}} \cdot \rho(q[A_i], t[A_i]) + \sum_{i=1}^{m'} w_i^{\mathrm{B}} \cdot \rho(q[B_i], t[B_i]), \qquad (2)$$

where $w_i^{\mathrm{A}}, w_i^{\mathrm{B}} \in (0, 1]$ are the *ranking weight* for attribute A_i and B_i, respectively. The distance measure for each attribute, i.e.,

$\rho(q[A_i], t[A_i])$, is a variation of the discrete metric: (1) $\rho(q[A_i], t[A_i]) = 0$ if $t[A_i] \in q[A_i]$ (note that for point queries, this means $t[A_i]$ being equal to the single value in $q[A_i]$), and (2) $\rho(q[A_i], t[A_i]) = 1$ if $t[A_i] \notin q[A_i]$.

Once again, we would like to note that the adversary has *no* knowledge of the ranking function design whatsoever (other than its monotonicity and additivity). This linear ranking function based running example merely provides a concrete basis for the analysis of attack performance in practice.

4. POINT QUERY INTERFACE

We start by considering a point query interface. Specifically, we shall start with reducing rank-based inference to the problem of finding pairs of *differential queries* based on the victim tuple v. Then, we discuss the design of Q&I-Point and Q-Point, our rank-based inference algorithms for Q&I and Q-only adversaries over a point query interface, respectively.

4.1 Goal: Finding Differential Queries

We start by showing that, for the worst-case scenario of $k = 1$, the problem of compromising the private attribute B_1 of victim tuple v can be reduced to finding for each possible value of B_1 except $v[B_1]$, i.e., $\forall \theta \in (V_1^{\mathrm{B}} \backslash v[B_1])$, a pair of *differential queries* q_θ and q'_θ which satisfy three properties: (1) they share the same predicate on all attributes but B_1, (2) $q'_\theta[B_1] = \theta$ while $q_\theta[B_1] \neq \theta$, and (3) q_θ returns the victim tuple v while q'_θ does not - i.e.,

$$\begin{aligned} &\forall t \in D \text{ where } t \neq v, s(t|q_\theta) > s(v|q_\theta), \\ &\exists t \in D \text{ with } t \neq v \text{ such that } s(t|q'_\theta) < s(v|q'_\theta). \end{aligned} \quad (3)$$

The proof of this reduction is straightforward: Due to the additivity condition, we can infer from (3) that the value of victim tuple v on B_1 must *not* be the same as $q'_\theta[B_1]$, i.e., $v[B_1] \neq \theta$. Since we found differential queries q_θ, q'_θ for all $\theta \in V_1^{\mathrm{B}} \backslash v[B_1]$, the only remaining possibility is the correct value of $v[B_1]$.

While this reduction is the basis of our following discussions, it is important to note that reduction in the opposite direction does not hold - i.e., an adversary does *not* have to find all $|V_1^{\mathrm{B}}| - 1$ pairs of differential queries in order to compromise $v[B_1]$. To understand why, consider an example where a Q&I-adversary inserts into the database a dummy tuple t with value 0 on all public and private attributes. Then, upon issuing a query with $q[A_i] = 0$ and $q[B_i] = 0$ on all attributes, the adversary receives v rather than t as the No. 1 result. One can see that the adversary can safely infer $v[B_1] = 0$ without issuing any additional query or identifying the differential queries for any value of B_1.

4.2 Q&I adversary

We develop Algorithm Q&I-Point in this subsection. Specifically, we start with a somewhat surprising finding - for a Q&I adversary, as long as it has the ability to find a query that returns the victim tuple v for a given database, then it can *always* successfully compromise $v[B_1]$ (by finding differential queries for all other values in V_1^{B}). Then, we present Algorithm Q&I-Point and analyze its worst- and average-case query costs.

4.2.1 Reduction to finding a query that returns v

Algorithm Q&I-Point: To construct the reduction to finding one query which returns the victim tuple, we start by assuming an oracle FIND-Q which, upon given input of a database D and the victim tuple v, returns a query q which returns v. We first develop Algorithm Q&I-Point which calls upon this oracle FIND-Q to compromise $v[B_1]$, and then introduce the design of FIND-Q afterwards. The pseudocode of Q&I-Point is shown in Algorithm 1.

Without loss of generality, we assume the output of FIND-Q to always have $q[A_1] = v[A_1], \cdots, q[A_m] = v[A_m]$. The reason here is simple - if q differs from v on any public attribute A_i, we can always change the attribute to $q[A_i] = v[A_i]$ - the new query will still return v due to the additivity condition of the ranking function.

We denote by $V_j^{'B}$ the smallest domain value for private attribute B_j. We start by inserting into the database a tuple t that has the same value for public attributes as v. We set each of t's private attribute to $V_j^{'B}$. Then, we call FIND-Q over the new database to discover q which returns v. Note that if FIND-Q fails to do so - i.e., no query over the database returns v - then we already succeed because, due to the no-duplicate assumption, the only scenario for this to happen is when $v = t$. Given the result q of FIND-Q, we note that q must differ from t on at least one private attribute - again, if $q = t$ and yet returns v, there must be $v = t$. Without loss of generality, suppose that q differs from t on private attributes $B'_1, \ldots, B'_h$ - i.e., $\forall i \in [1, h]$, $q[B'_i] \neq t[B'_i]$.

We now construct $h+1$ queries $q_0, \ldots, q_h$ as follows: all these $h+1$ queries share the exact same value as t on all attributes but $B'_1, \ldots, B'_h$. For those h attributes, we assign to query q_i ($i \in [0, h]$) $q_i[B'_j] = q[B'_j]$ if $j \leq h - i$ and $q_i[B'_j] = t[B'_j]$ otherwise (i.e., if $j > h - i$). The following table shows an example. Note at the two extremes $q_0 = q$ and $q_h = t$.

There are two important observations from the above query sequence: First, unless $v = t$, queries at the two ends must return different results - specifically, q_0 returns v while q_h returns t. The only exception here is when q_h also returns v - but this must mean $v = t$ because q_h exactly matches t - leading to an immediate compromise of $v[B_1]$. Second, every pair of adjacent queries in the sequence differ by exactly one attribute - i.e., query q_i and q_{i+1} differ on B'_{h-i}. Combining two observations, we know two things: (1) there must exist a pair of adjacent queries q_i and q_{i+1} such that q_i returns v while q_{i+1} does not - because otherwise all $h+1$ queries would return v, contradicting Observation 1. (2) this pair of adjacent queries differ on exactly one attribute B'_{h-i}. In other words, they serve as a pair of *differential queries* for value $t[B'_{h-i}]$ in the domain of B'_{h-i}, and prove $v[B'_{h-i}] \neq t[B'_{h-i}]$. Note that the process of finding this pair of differential queries takes at most $h \leq m'$ queries.

Of course, this may not yet achieve the adversarial goal of compromising $v[B_1]$. Nonetheless, note that once we know $v[B'_{h-i}] \neq t[B'_{h-i}]$, we can insert into the database a new t which replaces its value on B'_{h-i} with another value (other than $t[B'_{h-i}]$) in its domain. We can then repeat the exact same process and get one of only two possible outcomes: either (1) we find another pair of differential queries and exclude from consideration a value for one of the private attributes; or (2) an anomaly occurs - either FIND-Q cannot find q or q^h returns v instead of t - meaning $t = v$ and we have compromised $v[B_1]$ already.

One can see that, the worst-case scenario here is for us to repeat the process for $\sum_{i=1}^{m'}(|V_i^{\mathrm{B}}| - 1)$ times - more repetitions is impossible because we would have already excluded all wrong values for $B_1, \ldots, B_{m'}$. Throughout all repetitions, the number of queries issued by Algorithm Q&I-Point (excluding those required by FIND-Q) is $O(m' \cdot \sum_{i=1}^{m'} |V_i^{\mathrm{B}}|)$.

Practical Implications: We now discuss the practical implications of Algorithm Q&I-Point. First, while we shall address the design and theoretical bounds of FIND-Q in detail next, we would like to first point out here that, in practice, FIND-Q is usually a straightforward and efficient process, especially when there are many public attributes. The reason is simple: those public attributes alone are often sufficient to uniquely identify the victim tuple. Since FIND-Q knows $v[A_1], \dots, v[A_m]$, it largely just needs to avoid hitting the few "fake" tuples Algorithm Q&I-Point inserts (by avoiding their private attribute values) in order to find a query that returns v.

The cost of FIND-Q aside, there are three interesting observations we can make regarding Algorithm Q&I-Point. First, its query cost depends on the SUM (not product) of domain size of private attributes. This works to the attacker's advantage in practice as real-world websites often feature only a few private attributes with small domains[2]. Nonetheless, this also means that large-domain attributes such as ZIP code can be very costly to attack. Intuitively, this is caused by nature of the point query interface - as each query here "covers" only one of the many domain values.

The second observation we would like to make is the *anytime* nature of the algorithm. While our problem definition focuses on compromising $v[B_1]$, one can see from the design of Q&I-Point that it indeed learns all private attributes of v. Specifically, every iteration (costing at most m' queries) excludes one value from consideration for one of the private attributes. Thus, even if we interrupt the algorithm at anytime (say running out of query allowance by the database), we would still have learned substantial information about many private attributes. This anytime feature makes the algorithm particularly difficult to thwart in practice.

Third, note from the design of Q&I-Point that all queries it issues (including those by FIND-Q) must have public attribute values equal to those of v. This makes the algorithm fairly resilient against changes to the database during the course of an attack - because the only changes that would affect the execution of Q&I-Point are those that feature tuples with the exact same public-attribute value-combination as v - an extremely unlikely event in practical databases.

Algorithm 1 Q&I-Point

```
1:  Input: $q, v$        Output: $v[B_1]$
2:  $H_v = \varnothing$; $t[A_i] = v[A_i]\,\forall i \in [1, m]$; $t[B_j] = 0\,\forall j \in [1, m']$
3:  while $v[B_1]$ is not yet inferred do
4:      Insert $t$ into $D$
5:      if $q$ does not return $v$ then $q \leftarrow$ FIND-Q$(v, H_v)$
6:      if no such $q$, then return $t[B_1]$        // case: $t = v$
7:      Let $B'_1 \dots B'_h$ be the attributes differing between $t$ and $q$
8:      $i = 0$;        $q_i = q$
9:      for $i = 1$ to $h$ do
10:         $q_i = q_{i-1}$;    $q_i[B'_{h-i+1}] = t[B'_{h-i+1}]$
11:         if $q_i$ and $q_{i-1}$ return different tuples then
12:             $q = q_{i-1}$;    Set $t[B'_i]$ to an unexplored value
13:             break for loop
```

4.2.2 Query Cost Analysis

Algorithm FIND-Q: We now describe the algorithm for finding a query q that returns v for a given database D. The design is mostly straightforward - we randomly generate and issue a query q with $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and each $q[B_j]$ ($j \in [1, m']$) drawn i.i.d. uniformly at random from V_j^{B} - and repeat this process until finding q that returns v. The only note of caution here is that the random generation is done *without replacement*, and *with memory* across different executions of FIND-Q. To understand why, note from the design of Algorithm Q&I-Point that we only insert tuples into the database, and do not tamper with or delete the existing tuple values. Thus, any query which does not return v before cannot return v in the future - justifying the design.

One can see from the design of FIND-Q that it always succeeds. As such, our focus here is to consider its query cost. First, all calls of FIND-Q, altogether, consume a worst-case query cost of $O(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$. While this seems like an outrageously high cost, we make two interesting notes here: First, the worst-case scenario indeed requires these many queries - as proved by the following lower bound result which shows that the cost cannot be improved beyond a constant factor. Second, the real-world query cost for FIND-Q is likely much smaller, as demonstrated by an average-case example study at the end of this subsection.

Lower Bound on Worst-Case Query Cost: The following theorem shows that, in the worst-case scenario, no algorithm can accomplish the attack without issuing $\Omega(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries.

THEOREM 1. *Given any ranking function and victim tuple v, there exists a database D such that no Q&I-adversary can compromise $v[B_1]$ without issuing $\Omega(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries.*

Due to space limit, please refer to [23] for all theorem proofs in the paper. Intuitively, the proof relies on the fact that, in order for an adversary to compromise $v[B_1]$, it must be able to find at least one query which returns v. We could then construct a database D where for any $j \in [1, m']$, every v_j^i ($i \in [1, |V_j^{\mathrm{B}}| - 1]$) shares the same value as v on all attributes but B_j. In addition, each v_j^i takes a unique domain value in V_j^{B} that is different from $v[B_j]$. Due to the existence of these tuples, any query q which differs from v on at least one attribute will not return v due to monotonicity condition.

Running Example Query Cost: We now consider how Q&I-Point performs over the running example of a linear-combination ranking function in Equation 2 and a database where each tuple is generated i.i.d. randomly according to the uniform distribution, while the victim v is chosen uniformly from the database.

THEOREM 2. *In the running example, the expected number of queries Q&I-Point issues to compromise $v[B_1]$ is at most $1/p + \sum_{i=1}^{m'}(|V_i^{\mathrm{B}}| - 1)$, where*

$$p = \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \mathrm{erf}\left(\frac{d^A(v,t)}{\sqrt{\sum_{i=1}^{m'} 2w_i'^2 \cdot \frac{|V_i^{\mathrm{B}}|-1}{|V_i^{\mathrm{B}}|^2}}} \right) \right) \quad (4)$$

where $\mathrm{erf}(\cdot)$ is the standard error function [10], and $d^A(v,t)$ is the distance between v and t on public attributes - i.e., $d^A(v,t) = w_1 \cdot \rho(v[A_1], t[A_1]) + \cdots + w_m \cdot \rho(v[A_m], t[A_m])$, where ρ is the distance function defined in the running example.

Due to space limit, please refer to [23] for all theorem proofs in the paper. Note from (4) why the average-case query cost of FIND-Q (and thereby Q&I-Point) is likely much smaller than its worst-case bound: p is the probability for a query q randomly tested in FIND-Q to return v. One can see that, when there is a large number of public attributes or a small number of private ones - i.e., a larger $d^A(v,t)$ or a smaller w_i', the probability for a tuple t ($t \neq v$) to "overcome" its difference with q on public attributes (with which v has zero difference) by private attribute values is fairly small - leading to a larger p and, ultimately, a smaller query cost.

The query cost required for Q&I-Point to compromise $v[B_1]$ actually *decreases* with a *smaller* weight on the private attributes. This observation seems counter-intuitive because when $w_1' = 0$,

[2]e.g., in the case of dating website discussed in Section 7.3, all private attributes are Boolean - e.g., "whether a user is willing to consider matches of a different race."

no privacy disclosure occurs as the rank becomes independent of $v[B_1]$ - but the worst disclosure occurs when w'_1 takes the smallest positive value! To understand why, note that the smaller private ranking weights w'_i are, the easier it is for an adversary to pinpoint a query that returns v, as the adversary already has prior knowledge of all public attribute values of v. Given that, for a Q&I-adversary, finding a query returning v is (almost) equivalent with compromising $v[B_1]$, we have this seemingly counter-intuitive observation.

4.3 Q-only adversary

Design of Q-Point: For adversaries subject to the tuple-insertion constraint, the feasibility of compromising $v[B_1]$ is not of certainty as in the Q&I adversary case, as shown in the following theorem.

THEOREM 3. *Given any victim tuple v, there exists a ranking function $s(\cdot|\cdot)$ and a database D such that no Q-only adversary can perform a rank-based inference of $v[B_1]$ over D.*

While the detailed proof of the theorem is available in [23], the key idea of it is easy to explain: Consider the linear ranking function in the running example and a database with only two attributes, one public A_1 and one private B_1. If the weighting on A_1 is larger than B_1, and each tuple in the database takes a different value on A_1, then there is no way for a Q-only adversary to infer $v[B_1]$ because the results of every possible query is already determined without knowing the value of B_1 for any tuple. Specifically, a query will always return the tuple that shares its value on A_1, regardless of what values the tuples have on B_1. As such, the inference of $v[B_1]$ from tuple ranks becomes infeasible.

Despite of the worst-case infeasibility, however, in practice it is quite likely for a Q-only adversary to find enough queries to unveil $v[B_1]$, as we shall show in the experimental results. To address these cases, we now develop Algorithm Q-Point for a Q-only adversary to launch a rank-based inference attack over a point query interface. Once again, our goal here is to find a pair of *differential queries* q_θ and q'_θ for each value $\theta \in V_1^{\mathrm{B}} \backslash v[B_1]$. Like in the Q&I-case, without loss of generality, we denote the domain values in V_1^{B} as $0, 1, \ldots, |V_1^{\mathrm{B}}| - 1$.

We start by calling Algorithm FIND-Q to find a query q which returns v. Then, we construct and issue $|V_1^{\mathrm{B}}|$ queries $f_0(q)$, $f_1(q)$, $\ldots$, $f_{|V_1^{\mathrm{B}}|-1}(q)$. While all these queries share the exact same predicates as q on $A_1, \ldots, A_m, B_2, \ldots, B_{m'}$, there is $f_i(q)[B_1] = i$ for all $i \in [0, |V_1^{\mathrm{B}}| - 1]$. Due to the additivity property, at least one of these $|V_1^{\mathrm{B}}|$ queries must return v. If only one does, then our attack on $v[B_1]$ already succeeds - the one which returns v must have the same value as v on B_1. If more than one return v, we can do two things: First, we can exclude from consideration those values corresponding to the queries that do not return v - for those, we have already found their differential queries to support the exclusion. Second, we can proceed to revise q (and correspondingly $f_i(q)$) as follows to continue the exclusion process.

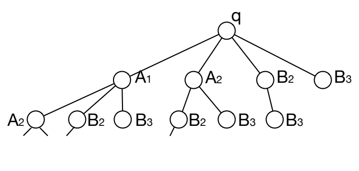

Figure 2: Enumeration Tree in Algorithm Q-Point

Specifically, our query-revision process can be considered performing a breadth-first search over the tree structure depicted in Figure 2, which demonstrates a special case where $m = 2$ and $m' = 3$. In the tree, each node consists of a class of revisions to q. Specifically, a node contains all queries that differ from q exactly on the attributes that appear on the path from the node to the root in the tree. For example, the bottom-left corner node in Figure 2 contains all queries that differ from q on A_1 and A_2.

During the search process, for each node encountered, we enumerate all queries q' in the node and repeat the value-exclusion process described above by issuing $f_i(q')$ for all $i \in [0, |V_1^{\mathrm{B}}| - 1]$. Note that the enumeration can be made more efficient with a pruning-based optimization: If for a query q', none of the $|V_1^{\mathrm{B}}|$ queries $f_i(q')$ returns v, then we can safely exclude from future consideration all queries in the subtree of the current node which only differs from q' on public attribute values. Algorithm 2 summarizes the pseudocode for Algorithm Q-Point.

Performance Analysis: One can see from the design of Q-Point that, in the worst-case scenario, it issues enough queries to determine for every query specifiable through the point-query interface whether it returns v. Thus, Q-Point always accomplishes the attack as long as such an attack is at all feasible over the point-query interface. Nonetheless, the query complexity of Q-Point is $O(\prod_{i=1}^{m} |V_i^{\mathrm{A}}| \cdot \prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ - much higher than Q&I-Point, given that real-world databases often feature more public attributes with large domains.

We consider again the linear ranking function in the running example and a database where each tuple is generated i.i.d. uniformly at random. We have the following result for Q-Point:

THEOREM 4. *In the above scenario, given q produced by FIND-Q, the probability (taken over the randomness of database D) for Q-Point to infer $v[B_1]$ after issuing only $|V_1^{\mathrm{B}}|$ queries is at least*

$$\left(1 - \prod_{t \in D, t \neq v} \frac{1 + \operatorname{erf}\left(\frac{(d^A(v,t) - w'_1)}{\sqrt{2\sum_{i=2}^{m'}(w_i'^2 \cdot (|V_i^{\mathrm{B}}|-1)/|V_i^{\mathrm{B}}|^2)}}\right)}{1 + \operatorname{erf}\left(\frac{d^A(v,t)}{\sqrt{2\sum_{i=2}^{m'}(w_i'^2 \cdot (|V_i^{\mathrm{B}}|-1)/|V_i^{\mathrm{B}}|^2)}}\right)}\right)^{|V_1^{\mathrm{B}}|-1} \tag{5}$$

Similar to the Q&I case, the attack is more (likely to be) efficient with a smaller $|V_1^{\mathrm{B}}|$. Nonetheless, an interesting observation here is that, contrary to the Q&I case, now the larger w'_1 is, the more efficient the attack is likely to be. On the other hand, the efficiency also increases with a larger database size $|D|$ and a smaller weight on other private attributes w'_i (as $\operatorname{erf}(x)$ has a larger derivative when x is close to 0).

Algorithm 2 Q-Point

```
1:  Input: $v$        Output: $v[B_1]$
2:  while some query returns $v$ do
3:      $q \leftarrow$ FIND-Q($v, H_v$); Construct enumeration tree $T_q$ for $q$
4:      for $i = 1$ to $m + m'$ do
5:          for each query node $q'$ in level $i$ of $T_q$ do
6:              Construct queries $f_0(q') \ldots f_{|V_1^B|-1}(q')$
7:              if none return $v$ then prune subtree($q'$)
8:              if only $f_j(q')$ returns $v$ then return $j$ as $v[B_1]$
9:              Exclude query nodes $f_k(q')$ that does not return $v$
10: return failure
```

5. IN QUERY INTERFACE

5.1 Q&I Adversary

For Q&I adversaries, the feasibility of rank-based inference attack is established for point-query interface in §4. Since point-query interface is a special case of IN, the attack feasibility here is already established. Thus, our focus here is to study how the additional power of IN queries further empowers Q&I adversaries.

Recall from §4 that, for Q&I adversaries, rank-based inference can be fairly efficiently reduced to the task of FIND-Q - i.e., identifying a (now IN) query returning victim v. We shall start by showing that, despite of the larger space of queries, the reduction still holds - leading to the design of Algorithm Q&I-IN. Then, we show that, while FIND-Q for IN has the same worst-case query cost as in Q&I-Point, the query cost in practice is likely much smaller.

5.1.1 Reduction to finding an IN query that returns v

We start by showing that, so long as a Q&I adversary can call upon FIND-Q to identify an *IN query* q that returns v, it can always infer $v[B_1]$ within $O(m' \cdot \sum_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries. The reduction in §4 cannot be directly used here as it relies on the ability to find a *point* query returning v - which we do not want FIND-Q to do over an IN-query interface due to high query cost associated with it.

To enable the reduction to finding an IN query, the only difference from point-query case (§4.2.1) is that now the input q might have ranges like $\{0, 1, 2\}$ specified as predicates on B_i, instead of a single value as in the point-query case (which we denoted as 0). Fortunately, this change does not alter the key design of reduction construction. What we do now is to define $B'_1, \ldots, B'_h$ as those attributes on which the inserted tuple t has a value that differs from the set specified in q. This could be that $t[B'_i]$ falls outside of the range specified in $q[B'_i]$ (e.g., when $t[B'_i] = 3$ while $q[B'_i] = \{0, 1, 2\}$; or that $t[B'_i]$ is in the range but not the only element of $q[B'_i]$ (e.g., when $t[B'_i] = 0$ and $q[B'_i] = \{0, 1, 2\}$. Here is an example of the sequence of queries we construct:

	$A_1, \ldots, A_m$	B'_1	B'_2	B'_3	B_{others}
q	{0}	{0,1}	{1,2}	{1}	{0}
q_0	{0}	{0,1}	{1,2}	{1}	{0}
q_1	{0}	{0,1}	{1,2}	{0}	{0}
q_2	{0}	{0,1}	{0}	{0}	{0}
q_3	{0}	{0}	{0}	{0}	{0}
t	0	0	0	0	0

Once again, there must exist a pair of adjacent queries q_i and q_{i+1} such that q_i returns v while q_{i+1} does not. The remaining inference process follows §4.2.1. For example, if q_1 returns v but q_2 does not, then we can safely infer that $v[B'_2] \neq 0$ due to the additivity condition. Similarly, if q_2 returns v while q_3 does not, we can infer that $v[B'_1] \neq 0$. Thus, just like in the Q&I-Point case, excluding the query cost of FIND-Q, a Q&I-adversary requires at most $O(m' \cdot \sum_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ queries to compromise $v[B_1]$.

5.1.2 Efficiency Enhancement in Q&I-IN

Given that the reduction still holds, we are now ready to study how IN queries empower an adversary to quickly accomplish FIND-Q and find a query that returns v. In the following, we describe a concrete example which demonstrates the significant saving brought by IN queries, followed by the design of Algorithm Q&I-IN.

Example of significant query savings: To understand why IN queries significantly reduce the query cost, consider a simple example where: (1) the number of public attributes m is sufficiently large, so each tuple in the database has a unique value combination for the m public attributes; and (2) the number of private attributes m' is even larger, so the probability for a randomly generated point query to return v is extremely small.

The first observation from this example is that FIND-Q over a point-query interface actually requires an extremely large number of queries. Specifically, note from Theorems 1 and 2 that, for a given m, the query cost can be made arbitrarily large with an increasing m'. On the other hand, if IN queries are available, the attack query cost - more specifically, the number of queries required to find one query returning v - is exactly 1 because an IN query q with $A_i = v[A_i]$ for $i \in [1, m]$ and $B_j = V_j^{\mathrm{B}}$ (essentially "*", i.e., do-not-care) for $j \in [1, m']$ always returns v.

One can see from the example that the usage of IN queries significantly reduces the attack query cost because of a simple reason: the ability for an adversary to *eliminate* all private attributes from a query specification makes it much easier for FIND-Q to unveil the victim tuple from the database, so that the adversary can compromise the private attributes one at a time using the above-described reduction. In other words, with IN queries, an adversary no longer has to get lucky and guess multiple private attributes correctly at the same time (e.g., in order to have v returned by a point query).

Design of Q&I-IN: Algorithm 3 depicts the pseudocode for Algorithm Q&I-IN, which enables a Q&I-adversary to launch our rank-based inference attack on $v[B_1]$ over an IN query interface. With the algorithm, we start with a query q which has $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and $q[B_j] = V_j^{\mathrm{B}}$ for all $j \in [1, m']$. Then, if q does not return v, we gradually replace predicates on B_i with point predicates (i.e., $B_i = v$ where $v \in V_i^{\mathrm{B}}$). Specifically, we perform what is essentially a *breadth-first search* process which enumerates all value combinations for B_1, $\{B_1, B_2\}$, $\{B_1, B_2, B_3\}$, ..., $\{B_1, \ldots, B_{m'}\}$ in order. For example, when $V_1^{\mathrm{B}} = V_2^{\mathrm{B}} = \{0, 1\}$, the queries we issue are $B_1 = 0$, $B_1 = 1$, $B_1 = 0$ AND $B_2 = 0$, $B_1 = 0$ AND $B_2 = 1$, $B_1 = 1$ AND $B_2 = 0$, $B_1 = 1$ AND $B_2 = 1$, ..., where each query also includes $q[A_i] = v[A_i]$ for all $i \in [1, m]$ and $q[B_j] = V_j^{\mathrm{B}}$ for all unspecified B_j. When we find a query that returns v, we launch the above-described reduction process to complete the attack of $v[B_1]$.

One can see from the algorithm design that, just like in the point-query case, we guarantee a successful attack. But the worst-case query cost for Q&I-IN is also just like Q&I-Point - i.e., $O(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$. As we shall demonstrate in the following worst-case analysis, this query cost still cannot be improved beyond a constant factor.

Algorithm 3 Q&I-IN

```
1: Input: $v$        Output: $v[B_1]$
2: Initialize starting query $q$: $q[A_i] = v[A_i] \forall i \in [1, m]$ and
   $q[B_j] = V_j^B \forall j \in [1, m']$
3: Iteratively convert $q$ to a point query till it returns $v$
4: $v[B_1] \leftarrow$ Q&I-Point($q, v$)
```

5.1.3 Query cost analysis

The main result here is that, while the availability of an IN query interface does *not* help a Q&I adversary at all in the worst-case scenario, it does have the potential to significantly reduce the query cost in practice - especially when the number of public attributes is large. To understand why the worst-case scenario remains unchanged, consider the construction in the proof of Theorem 1 which inserts to the database $\sum_{i=1}^{m'}(|V_i^{\mathrm{B}}| - 1)$ tuples described in §4. Given the worst-case assumption that, when there is a draw (i.e., $s(t_1|q) = s(t_2|q)$), any inserted tuple will be returned before the victim v, one can see that the adversary gets no help from IN queries - because as long as a query q contains an IN predicate, say on B_i, it is impossible for q to return the victim tuple v as there must exist an inserted tuple which matches q on B_i, has the exact same value combination as v on all other attributes, and therefore will be returned ahead of v in the answer to q. Thus, the worst-case query cost Q&I-IN remains $\Omega(\prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$ - same as Q&I-Point.

THEOREM 5. *In the running example, the expected number of queries FIND-Q requires for finding a query that returns v is* 1 *if*

$\min_{t \in D, t \neq v} d^A(v,t) > 0$, and at most $\sum_{h=1}^{m'-1}(c_{h+1} \cdot (1-(1-p(h))^{c_h})$ otherwise, where $c_h = \sum_{i=1}^{h} \prod_{j=1}^{i} |V_i^{\mathrm{B}}|$ and

$$p(h) = \prod_{t \in D, t \neq v} \left(\frac{1}{2} + \frac{1}{2} \cdot \mathrm{erf} \left(\frac{d^A(v,t)}{\sqrt{2 \sum_{i=1}^{h} w_i'^2 \cdot \frac{|V_i^{\mathrm{B}}|-1}{|V_i^{\mathrm{B}}|^2}}} \right) \right). \quad (6)$$

One can see from the theorem the substantial promise for IN queries to significantly reduce the query cost - not only the query cost can be cut to 1 when no other tuple shares the same public-attribute value-combination as v, but the value of $p(h)$ - i.e., the probability for a query with h point-predicates on private attributes to return v - actually decreases with h. As such, the query cost is likely much smaller than Q&I-Point, especially when the number of public attributes m is large (which leads to a large $d^A(v,t)$).

5.2 Q-only adversary

Just like the availability of IN queries does not help reduce the worst-case query cost for Q&I-adversaries, it cannot change the (in)feasibility result for Q-only adversaries either. To understand why, consider a database with the aforementioned linear ranking function and all tuples sharing the same value on B_1. Clearly, the returned tuples will be of the same order regardless of what range the query specifies on B_1. Thus, there is no way for a Q-only adversary to infer which value in $v[B_1]$ all tuples take - proving that Q-only adversaries cannot guarantee the success of rank-based inference even for IN query interfaces. Nonetheless, as we shall show in this subsection and in the experimental results, the availability of IN queries does help with reducing the query cost in practice, especially when the number of public attributes is large.

Algorithm 4 depicts the pseudocode for Algorithm Q-IN, which enables a Q-only adversary to launch our rank-based inference attack on $v[B_1]$ over an IN query interface. We start with calling Algorithm FIND-Q to find one query q which returns v. Note that, according to the design of FIND-Q, q always has $q[A_i] = v[A_i]$ for all $i \in [1,m]$. After obtaining q, Algorithm Q-IN issues $|V_1^{\mathrm{B}}|$ queries $f_0(q), \ldots, f_{|V_1^{\mathrm{B}}|-1}(q)$ defined in the same way as in §4 - i.e., while all these queries are exactly the same as q on $A_1, \ldots, A_m, B_2, \ldots, B_{m'}$, there is $f_i(q)[B_1] = i$ for all $i \in [0, |V_1^{\mathrm{B}}|-1]$. Similar to the discussion in Algorithm Q-Point, one can see that at least one of these queries must return v, and the attack is already successful if only one of them does. If more than one returns v, we can exclude from consideration those values corresponding to queries that do not return v, and then gradually revise q according to the following procedure.

Specifically, we start with revising q to $q_1, \ldots, q_m$ by changing the predicate of q_i on A_i to (A_i IN V_i^{A}). For each q_i which returns v, we repeat the above process and issue $f_j(q_i)$ for each $j \in [0, |V_1^{\mathrm{B}}|-1]$ that is not yet excluded as a possible value of $v[B_1]$. Once again, this either directly reveals $v[B_1]$ or further excludes additional values from consideration. If we still cannot pin down $v[B_1]$ after enumerating $q_1, \ldots, q_m$, we consider the process by setting an additional public attribute to its entire domain. For example, if q_1 returns v, we construct $q_{1,x_1}, \ldots, q_{1,x_h}$, such that (1) $q_{x_1}, \ldots, q_{x_h}$ also return v, and (2) $q_{1,i}$ is the same as q_1 on all attributes but A_i, for which there is $q_{1,i}[A_i] = V_i^{\mathrm{A}}$. We repeat this value-exclusion process until finding the exact value of $v[B_1]$, or when we have exhausted all combinations of public attributes - at which time we move back to Algorithm FIND-Q, find another query q which returns v, and attempt the revision process again.

One can see from the design of Algorithm Q-IN that its worst-case query cost is the same as Q-Point, i.e., $O(\prod_{i=1}^{m} |V_i^{\mathrm{A}}| \cdot \prod_{i=1}^{m'} |V_i^{\mathrm{B}}|)$.

Algorithm 4 Q-IN

```
1: Input: $v$        Output: $v[B_1]$
2: while some query returns $v$ do
3:     $q \leftarrow$ FIND-Q($v, H_v$)
4:     for $i = 0$ to $m$ do
5:         for each $\binom{m}{i}$ combination of $C$ of $\{A_1, \ldots, A_m\}$ do
6:             $q' \leftarrow q$;    $q[A_{i'}] = V_{i'}^A \quad \forall A_{i'} \in C$
7:             Construct queries $f_0(q') \ldots f_{|V_1^B|-1}(q')$
8:             if only $f_j(q')$ returned $v$ then return $j$ as $V[B_1]$
9:             Exclude query nodes that did not return $v$
```

For the running example and a database where each tuple is generated i.i.d. uniformly at random, we have the following results:

COROLLARY 1. *In the above scenario, given q from FIND-Q which (1) has point-predicates on $S \subseteq \{A_1, \ldots, A_m\}$, (2) has point-predicates on B_1 and $S' \subseteq \{B_2, \ldots, B_{m'}\}$, and (3) returns v, the probability (taken over the randomness of database D) for Q-IN to infer $v[B_1]$ after issuing only $|V_1^{\mathrm{B}}|$ queries is at least*

$$\left(1 - \prod_{t \in D, t \neq v} \frac{1 + \mathrm{erf}\left(\frac{(d^S(v,t) - w_1')}{\sqrt{2 \sum_{i:B_i \in S'} (w_i'^2 \cdot (|V_i^{\mathrm{B}}|-1)/|V_i^{\mathrm{B}}|^2)}} \right)}{1 + \mathrm{erf}\left(\frac{d^S(v,t)}{\sqrt{2 \sum_{i:B_i \in S'} (w_i'^2 \cdot (|V_i^{\mathrm{B}}|-1)/|V_i^{\mathrm{B}}|^2)}} \right)} \right)^{|V_1^{\mathrm{B}}|-1} \quad (7)$$

The corollary follows directly from Theorem 4. We can observe from the theorem the substantial promise for IN queries to significantly reduce the query cost - specifically, note that the smaller S or S' is, the higher this expected ratio will be. As such, the overall query cost is likely much smaller than Q-Point.

6. DISCUSSION

6.1 Non-discrete Attributes

In this paper, we considered the attributes to be discrete as they are widely used. Further, their attribute domain can be easily inferred such as from the list of values from web UI controls such as dropdowns. Our proposed techniques can be directly extended to handle non-discrete attributes by (essentially) testing the binary correctness of each discretized range (in case of numeric data) or keyword (in case of text data). For numeric data, we can approach the problem by partitioning the numeric domain into discrete ranges and then applying our algorithm over the discretized data. The discretization process can be done in a recursive fashion (from coarse to fine) so we can infer the private value to an arbitrary precision. While these baseline approaches may reveal certain private information about non-discrete attributes, we note that their design cannot address the various complex issues that might require different problem definitions. For example, in the case of numeric attributes, success could be defined as inferring the private value exactly or as bounding it to a narrow range. This distinction is important as it is possible to design interfaces where the precision to which an attribute can be inferred can be restricted (say to 0.01). If the query interface allows a range to be specified for each attribute, and the ranking function simply assigns a score of 0 or 1 based on whether the correct value lies within that range, then we can infer the attribute to arbitrary precision. However, if the range can only be specified to a particular precision, then the attack precision also gets restricted. Refer to [23] for further discussion.

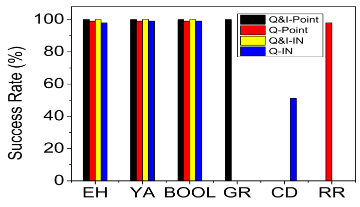

Figure 3: Attack Success Rate

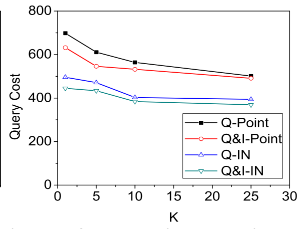

Figure 4: Varying Maximum number of tuples returned k

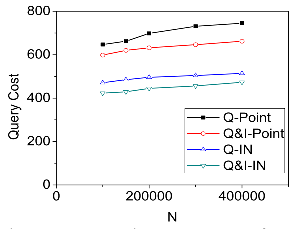

Figure 5: Varying Number of Tuples, n

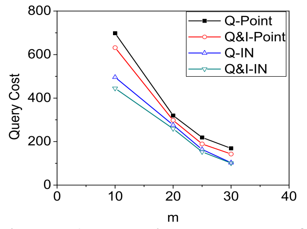

Figure 6: Varying Number of Public Attributes, m

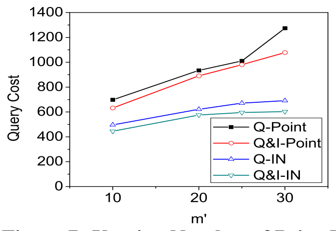

Figure 7: Varying Number of Private Attributes, m'

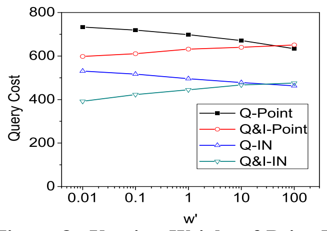

Figure 8: Varying Weight of Private Attribute, w'_1

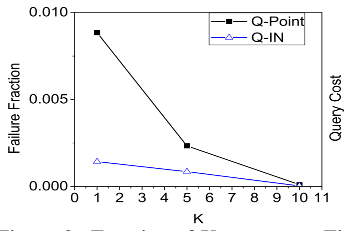

Figure 9: Fraction of Uncompromised Accounts

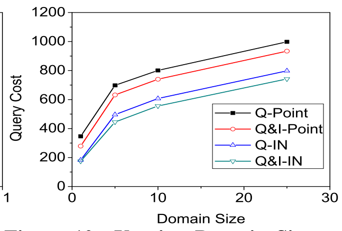

Figure 10: Varying Domain Size of Inferred Attribute

6.2 Defense Against Rank-based Inference

Since our main objective here is to unveil a novel rank-based inference attack on web databases, a comprehensive discussion of defense methodologies is beyond the scope of this paper. Please refer to [23] for a more detailed discussion. Besides the ultimate solution of excluding private attributes from being considered in ranking, an obvious defense methodology is to enforce more stringent *interface constraints* discussed in the paper - e.g., requiring a user to answer a CAPTCHA challenge before issuing each query, performing rigid authentication for each tuple insertion/update operation etc. Another possible strategy here is to *delay* any new tuples from appearing in query answers which dramatically affects Q&I adversaries. Another category of defense is to adjust the assignment of public/private attributes and/or the design of ranking function. For example, the database owner can be make more attributes as private thereby increasing query cost for the adversary. However, it is important to note that all defense strategies discussed are essentially making a tradeoff between privacy protection and the convenience of bona fide users, and therefore must be designed and implemented carefully (e.g., after user studies).

7. EXPERIMENTAL RESULTS

7.1 Experimental Setup

Hardware and Platform: All our experiments were performed on a quad-core 2 GHz AMD Phenom machine running Ubuntu 14.04 with 8 GB of RAM. The algorithms were implemented in Python.

Offline Datasets: To verify the correctness of our results, we started by testing our algorithms locally over two real-world and one synthetic dataset. We have full access to these datasets, along with full control of the ranking function used. One dataset is from *eHarmony* [3], a prominent online dating service [21] and consists of anonymized profile information of 500K users. Each user has 56 attributes, of which more than 30 are boolean. The second dataset is *Yahoo! Autos*, which contains 200K used cars for sale in the Dallas-Fort Worth area with 32 Boolean attributes and 6 categorical attributes, the domain cardinalities of which vary from 5 to 447. The third dataset is a synthetic Boolean i.i.d. dataset with 200K tuples and 40 attributes, each following the uniform distribution.

The public and private attributes were randomly chosen from the set of available attributes. By default, we randomly picked 20 attributes for testing, designated $m = 10$ of them as public and $m' = 10$ as private, while varying m and m' between 10 and 30 in various tests. Target attribute B_1 was chosen uniformly at random from all private attributes per iteration with its cardinality varying between 2 to 122 (with an average of 13). By default, we used the ranking function from (2) with all weights set to 1.

Online Demonstration: In order to demonstrate the success of our attacks over real-world websites, we selected three high-profile real-world websites - Renren.com, Amazon Goodreads, Catch22Dating - and conducted live experiments using our algorithms. We would like to note that, without a partnership with these websites, we do not possess/assume any knowledge of their ranking function (other than the monotonicity and additivity properties defined in §2, which we verified through the correctness of our experiment outputs). The results of the online experiments can be found in §7.3.

Performance Measures: As discussed earlier in §3, we measure efficiency through query cost, i.e., the number of queries required for each attack - consistent with prior work [11, 12].

7.2 Experiments over Real-World Datasets

Empirical Evaluation of Attack Success Rate: Figure 3 shows the attack success rate of all our algorithms over 3 offline datasets and the relevant algorithm (based on the problem subspace the website falls) over 3 online datasets. As expected, Q&I-adversary has 100% success rate for all datasets. For Q-only adversaries over real-world datasets, we were able to achieve a success rate of almost 100%. The same holds for online tests except CD - the main reason here is that CD allows NULL value on the private attribute we are targeting, leading to failed attacks.

In the following discussion of offline experiments, we focus on results over eHarmony. Due to space limitations, please refer to [23] for results on Yahoo! Autos and the synthetic dataset (which are largely similar).

Query Cost versus k: We first investigated the performance of our algorithms for different values of k. Figure 4 shows that query cost decreases with higher values of k as expected. Extending our algorithms for $k > 1$ is straightforward. First, we seek to find a query that returns v in top-k (not just top-1). Second, we extend the notion of differential queries (see §4.1) such that the v has a higher rank for query q'_θ than for q_θ. The query cost of our algorithms can be broadly categorized into two parts - the query cost to identify a query q that returns the victim tuple and the query cost required to construct additional queries from q through which the private attribute is inferred. When the value of k increases, the former query cost falls dramatically. Further, the figure also shows that when IN-queries are available (Q&I-IN and Q-IN), the query cost is lower than the cases where only point queries are allowed (Q&I-Point and Q-Point), consistent with our discussions in §5.

Query Cost versus Database Size, n: Figure 5 depicts the impact of database size on query cost when $k = 1$ (which is henceforth used as the default setting unless otherwise specified). As expected, the increase in database size do not have any major impact and only results in a slight increase in overall query cost. This is due to the fact that the number of queries needed to identify a randomly chosen tuple increases much more slowly than the database size.

Query Cost versus m, m': In our next experiments, we investigate how varying the number of public and private attributes affect the query cost. The results of these experiments are shown in Figures 6 and 7. As expected, when the number of public attributes increase, the query cost drops significantly. When the number of public attributes are limited, their values are not adequate to distinctly identify a random tuple. Hence, we need to resort to using randomly chosen values for the private attributes which increases query cost. However, when m increases, most tuples become uniquely identified based on their public attributes only. For a fixed m, the query cost increases with increasing m' - when the public attributes are inadequate for uniquely identifying the victim tuple, our algorithms resort to issuing queries where the private attributes are chosen randomly from their respective value domains. But the number of such possible queries increases with higher m' - hence the phenomenon.

Query Cost versus Ranking Weights: In this experiment, we fixed the weight of all public attributes to 1 and varied the weights of private attributes w'_i between 0.01 and 100. The results shown in Figure 8 are consistent with our theoretical results from Sections 4 and 5. When the weights over private attributes decrease, the query cost for Q&I adversaries also decreases. This is due to the fact that identifying the query q that returns the victim tuple v becomes much easier for this case. The opposite holds for Q-only adversaries where increasing the weights decreases the query cost.

Other Experiments: In order to identify the fraction of tuples in a database that could be successfully compromised using our algorithms, we randomly chose 100K tuples and tried to compromise them. Recall that the Q&I adversary based algorithms are always guaranteed to succeed. Figure 9 shows that the Q-only algorithms are able to compromise almost all the tuples. Even with a highly restrictive interface of $k = 1$, Q-Point compromises more than 99% of the tuples. We then adapted our inference algorithms so that they seek to infer all $m' = 25$ private attributes. Figure 11 shows the result. While the overall query cost seems high, the amortized query per private attribute varies between 35 and 60. Figure 10 shows how varying the domain size of the private attribute affects the query cost. Consistent with our analysis in [23], query cost increases with larger domain size.

7.3 Online Demonstration

In the online experiments, we sought to compromise private attributes of user profiles from Amazon Goodreads (GR), Catch22-Dating (CD) and Renren.com (RR) respectively. A detailed description of the procedure we used and its correctness can be found in [23]. Note that since we have no connection with these websites

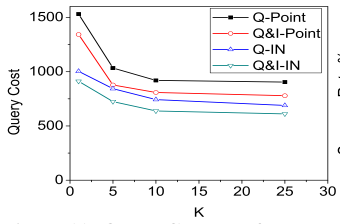

Figure 11: Query Cost to Infer all Private Attributes

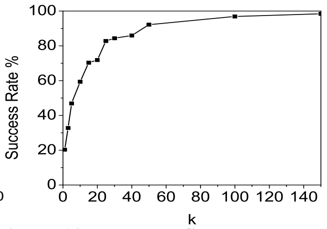

Figure 12: Renren: Success Rate vs. k

and thus do not have access to the ground truth, we limit the scope to a small-scale proof-of-concept.

Renren: Renren (RR) [4] is a major Chinese social networking website (similar to Facebook) with more than 160 million users. The user profile consists of details like demographics, education and work affiliation. The website supports extensive privacy settings - allowing a user to specify any subset of profile attributes as public, private or only visible to friends. In our experiments, we focused on one attribute `hometown province` with a domain size of 34 - which is set to private by all users we target. Renren has a search interface that accepts keyword queries on a user's public attributes such as profile name. It displays appropriate information depending on who issued the query - i.e. everyone can see public attributes, only friends can see attributes marked as visible to friends and none can see the private attributes. The results are ordered based on a ranking function that takes into account the entire profile regardless of privacy settings (as shown in §1).

Renren enforces tuple insertion constraint and also allows NULL values. We conducted our attack using Q-Point algorithm. One can see from Table 2 that we were able to successfully infer the private attribute for 75 out of 76 profiles with an average query cost of 20 per profile. We also conducted an experiment to measure the success rate of our attacks by varying k. The search interface of Renren, has a large value of k (ranging in hundreds). In our experiment, we artificially truncated the results for different values of k and verified if we can infer the private attribute. Figure 12 shows that for k as little as 50, we achieve a success rate of 92%.

Catch22Dating: Catch22Dating (CD) [2] is an online dating website where users create profiles that are then matched to other users. The public attributes here capture the demographic information of a user, whereas the private attributes specify a user's matching preferences - e.g., the one private attribute we focus on is Boolean "*Is it OK if your matches have been married before*" (henceforth referred to as `Married`). The search interface of Catch22Dating has an option called "Both Perspectives", which enables the ranking function to take into account both public and private attributes of all profiles on the website. It does enforce the tuple insertion constraint by requiring Student ID from selected universities during user registration. It also allows IN queries to be specified (e.g., one can set an attribute to be "do not care" in the query). Hence, we model the adversary as Q-only operating over an IN-interface.

The website allows NULL values on almost all attributes. As a result, our Q-IN attack might fail simply because the user specified NULL as the attribute value. One can see from Table 2 that out of the 120 users we attacked, we compromised the private attribute `Married` for 61 of them. For the other 60, either the user did not specify whether he/she would like to accept matches who have been married, or Q-IN attack fails on these users.

Amazon Goodreads (GR) [1] a social cataloging site where the users can connect to each other and share their experience/opinions about books. The user profile consists of demographic information such as `user name` which is always public, and attributes such as `zipcode` which can be set as private. Regardless of a user's choice on location privacy, the ranking function used in the website's "user search" interface ranks each user according to its (geographic) distance from the location of the user performing the search. Goodreads allows free and instant account registration - i.e., there is no tuple insertion constraint - but no range query. Hence we use the Q&I-Point algorithm.

We started with registering 10 fake accounts with randomly generated ZIP codes, and launched Q&I-Point over it to verify the correctness of our algorithm. Then, to enable verification on real accounts, we identified 53 "special" users at Goodreads who have their ZIP code hidden but chose to reveal their city/state (in US). We launched Q&I-Point successfully on all these users, and then verified that every ZIP code we compromised indeed belongs to its corresponding city/state revealed by the user.

Table 2: Summary of Online Experiments

	#Accounts Attacked	#Success	Avg Cost (Success)	Avg Cost (Failure)
CD	120	61	60	660
GR	53	53	455	N/A
RR	237	229	19	34

8. RELATED WORK

Database Ranking: The area of ranking has been extensively studied in the context of deterministic [19] and probabilistic [20] data. Processing top-k query when the ranking score is a combination of scores of individual attributes was studied in [15]. A popular ranking function is nearest neighbor [17] where the tuples are ordered based on the distance between tuple t and the given query q. Other categorizations such as monotone, generic or no ranking (such as Skyline queries) has also been studied [19]. Recently, there have been studies on learning the rank of a tuple [25].

Inference Control: Prior work on privacy inference [16] studied the problem of inferring individual tuple values [8] from aggregates such as SUM, MIN, MAX, etc. The field of inference control [13] seeks to prevent such attacks by through query auditing, controlling the number of tuples that match a query or modify query responses using perturbation, distortion etc [9]. Recently, [14] has showed that it is possible to infer the location of a user in a Location based Social Network (LBSN) if the ranking function returns the distance between the query and the victim tuple.

9. FINAL REMARKS

In this paper, we identified a novel problem of rank-based inferencing over databases that use ranked retrieval model. We introduced a taxonomy of the problem space into four important subspaces based on varying interface designs and adversarial capabilities. For each problem subspace, we developed nontrivial attacking algorithms and conducted theoretical analysis of their feasibility and performance. We verified the effectiveness of the attacks using a comprehensive set of experiments on real-world datasets and online demonstrations on high-profile real-world websites.

It is our hope that the paper initiates a new topic of research on the privacy implications of database ranking; and future research will address the many open problems, e.g., how to design effective defensive strategies that thwart the rank-based inference of private attributes yet maintain the utility of ranking functions.

10. ACKNOWLEDGMENTS

The authors were partially supported by National Science Foundation under grants 0915834, 1018865, 0852674, 1117297, 1343976, Army Research Office under grant W911NF-15-1-0020 and a grant from Microsoft Research.

11. REFERENCES

[1] Amazon goodreads. `https://www.goodreads.com/`.

[2] Catch22dating. `http://www.catch22dating.com/`.

[3] Eharmony. `http://www.eharmony.com`.

[4] Renren. `http://www.renren.com`.

[5] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *In CIDR*. CIDR, 2003.

[6] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS*, 27(2), 2002.

[7] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. VLDB, 2004.

[8] F. Chin. Security problems on inference control for sum, max, and min queries. *JACM*, 33(3):451–464, 1986.

[9] F. Y. Chin and G. Ozsoyoglu. Auditing and inference control in statistical databases. *TSE*, 8(6):574–582, 1982.

[10] D. R. Cox. *Principles of statistical inference*. Cambridge University Press, 2006.

[11] A. Dasgupta, G. Das, and H. Mannila. A random walk approach to sampling hidden databases. In *SIGMOD*, 2007.

[12] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das. Unbiased estimation of size and other aggregates over hidden web databases. In *SIGMOD*, pages 855–866. ACM, 2010.

[13] J. Domingo-Ferrer. A survey of inference control methods for privacy-preserving data mining. In *Privacy-preserving data mining*, pages 53–80. Springer, 2008.

[14] M. L. et al. All your location are belong to us: Breaking mobile social networks for automated user location tracking. MobiHoc, 2014.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[16] C. Farkas and S. Jajodia. The inference problem: a survey. *ACM SIGKDD Explorations Newsletter*, 4(2):6–11, 2002.

[17] X. Geng, T.-Y. Liu, T. Qin, A. Arnold, H. Li, and H.-Y. Shum. Query dependent ranking using k-nearest neighbor. In *SIGIR*, pages 115–122. ACM, 2008.

[18] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. *Range queries in OLAP data cubes*, volume 26. 1997.

[19] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

[20] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *VLDB*, 2009.

[21] B. McFee and G. R. Lanckriet. Metric learning to rank. In *ICML*, pages 775–782, 2010.

[22] A. Motro. Vague: A user interface to relational databases that permits vague queries. *ACM TOIS*, 6(3):187–214, 1988.

[23] M. F. Rahman, W. Liu, S. Thirumuruganathan, N. Zhang, and G. Das. Rank-based inference over web databases. *arXiv preprint arXiv:1411.1455*, 2014.

[24] Y. Rui, T. S. Huang, and S. Mehrotra. Content-based image retrieval with relevance feedback in mars. In *ICIP*, 1997.

[25] S. Thirumuruganathan, N. Zhang, and G. Das. Rank discovery from web databases. *VLDB*, 2013.