2012 ASE/IEEE International Conference on Social Computing and 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust

Flash: Efficient, Stable and Optimal K-Anonymity

Florian Kohlmayer*†‡, Fabian Prasser*†‡,
Claudia Eckert*, Alfons Kemper* and Klaus A. Kuhn†
*Technische Universität München, Department of Computer Science, 85748 Garching, Germany
†Technische Universität München, University Medical Center (Klinikum rechts der Isar), 81675 München, Germany

***Abstract*—K-anonymization is an important technique for the de-identification of sensitive datasets. In this paper, we briefly describe an implementation framework which has been carefully engineered to meet the needs of an important class of k-anonymity algorithms. We have implemented and evaluated two major well-known algorithms within this framework and show that it allows for highly efficient implementations. Regarding their runtime behaviour, we were able to closely reproduce the results from previous publications but also found some algorithmic limitations. Furthermore, we propose a new algorithm that achieves very good performance by implementing a novel strategy and exploiting different aspects of our implementation framework. In contrast to the current state-of-the-art, our algorithm offers algorithmic stability, with execution time being independent of the actual representation of the input data. Experiments with different real-world datasets show that our solution clearly outperforms the previous algorithms.**

I. Introduction

A. Motivation

The amount of data collected about individuals is proceeding at an ever-increasing rate [1]. A number of incidents (e.g., [2]–[4]) have shown that simply removing all directly identifying information (e.g., a person's name) is not sufficient to protect an individual's privacy. In addition, detailed personal data of high quality is often required for analyses. In this context, anonymization is an important building block for balancing an individual's privacy and the need for fine-grained data collections. To this end *k-anonymity* is a wide-spread technique. The basic idea is to protect a dataset against re-identification by generalizing and suppressing the *quasi-identifiers*, which are the attributes that could be used in a linkage attack. This attack tries to link anonymized data to additional identified data, which can result in identity disclosure [4]. A dataset is *k-anonymous* if each data item can not be distinguished from at least $k-1$ other data items [5]. An example dataset with quasi-identifiers *age*, *gender* and *zipcode* as well as a two-anonymous transformation are shown in Figure 1. Without loss of generality we assume a tabular data structure. Alternative methods have been proposed (e.g., differential privacy [6]), but k-anonymization is still considered the option of choice in many domains, e.g., medicine [7].

B. Related Work and Contribution

Generally, there are multiple ways to transform a dataset into a k-anonymous representation, and different algorithms have been proposed. These can be classified along different axes. First, some algorithms implement global recoding (also called full-domain anonymization), whereas others implement local recoding. *Local recoding* means that within a column, different generalization rules can be applied to equal values, whereas *global recoding* means that the same rule is applied. Local recoding is often used by clustering algorithms (e.g., [8]), whereas global recoding is mostly used by algorithms which utilize generalization hierarchies (see Section II for details). Some, but not all, algorithms find an *optimal solution*. Here, optimal describes the solution which results in minimal information loss according to a given metric. Solving this problem has been proven to be NP-hard [9]. There are algorithms (e.g., [10]) which approximate the problem and only find a solution that is within a guaranteed distance to the optimum. Various extensions to the k-anonymity concept exist. The most important ones are ℓ-diversity [11], t-closeness [12] and δ-presence [13]. To derive the appropriate parameters for these algorithms, risk based anonymization [14] can be used. An extensive overview of previous privacy models and algorithms can be found in [15] and [16]. As long as they utilize global recoding with generalization hierarchies, they can all be implemented in our generic framework and with our novel algorithm. Without loss of generality, we focus on the basic k-anonymity problem [5].

Age	Gender	Zipcode
34	male	81667
45	female	81675
66	male	81925
70	female	81931
34	female	81931
70	male	81931
45	male	81931

Age	Gender	Zipcode
<50	$\star$	816$\star\star$
<50	$\star$	816$\star\star$
≥ 50	$\star$	819$\star\star$
≥ 50	$\star$	819$\star\star$
<50	$\star$	819$\star\star$
≥ 50	$\star$	819$\star\star$
<50	$\star$	819$\star\star$

Fig. 1. Example dataset

Approaches which find an optimal solution by applying global recoding with user-defined generalization hierarchies form an important class of k-anonymization algorithms [17]. The main reasons for this are that an optimal solution guarantees minimal information loss, which is important for the usefulness of the result. Furthermore, global recoding delivers the best result in terms of statistical properties. Finally, the utilization of generalization hierarchies allows the definition of different generalization strategies for different use-cases. Li et al. [18] have even shown that such algorithms can, under certain circumstances, guarantee a better degree of privacy than the ones based on clustering and local recoding.

The algorithm of Samarati [19] is an early algorithm from this class. Because it is only able to find an optimal solution for a very limited metric (*Height*: see Section II) we will not

‡Both authors contributed equally to this work

978-0-7695-4848-7/12 $26.00 © 2012 IEEE
DOI 10.1109/SocialCom-PASSAT.2012.52

discuss the algorithm in further detail. Instead, we focus on the two major algorithms, including the current state-of-the-art by El Emam et al. In [20] LeFevre et al. proposed the Incognito algorithm, which is oriented towards dynamic programming. In [17] El Emam et al. described a k-anonymization algorithm, which implements a divide-and-conquer approach, and showed that it outperforms the algorithms presented in [20] and [19]. As our novel algorithm is from the same class, we will describe these algorithms in more detail in Section III.

This article comprises the following scientific contributions: *i)* a description of a generic framework for the efficient implementation of k-anonymization algorithms, *ii)* an evaluation of this framework when implementing the two major previous algorithms, *iii)* the presentation of a novel algorithm which implements a new strategy and makes extensive use of our implementation framework, and *iv)* a comprehensive evaluation with various differently-sized real-world datasets showing that our approach outperforms the two major previous algorithms in terms of performance and algorithmic stability.

In Section II, we present the basic concepts behind optimal k-anonymization with global recoding. Section III reviews the most important related algorithms. Section IV covers the fundamental concepts behind our implementation framework. Our novel algorithm is described in Section V, and Section VI presents an extensive evaluation. Section VII summarizes the results and discusses directions for further improvement.

II. Background

A. Generalization and Monotonicity

Generalization *hierarchies* define a method of iteratively generalizing the values of an attribute. Figure 2 shows generalization hierarchies for the quasi-identifiers in the example dataset. In the two-anonymous version from Figure 1, the *age* is transformed to intervals of length 50 (level 1), the attribute *gender* is suppressed (level 1) and the two least significant digits are dropped from the *zipcode* (level 2).

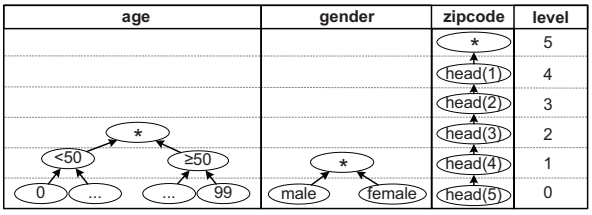

Fig. 2. Generalization hierarchies

Most algorithms which utilize generalization hierarchies operate on a data structure called *Generalization Lattice*. It is shown in Figure 3 for our example dataset and hierarchies. An arrow denotes that a state is a direct *generalization* of a more *specialized* state and can be created by incrementing one of the generalization levels defined by its predecessor. The state with minimal generalization $(0, 0, 0)$ is at the bottom and represents the original input dataset, whereas the state with maximal generalization $(2, 1, 5)$ is at the top. The state that has been applied to anonymize the dataset from Figure 1 is $(1, 1, 2)$.

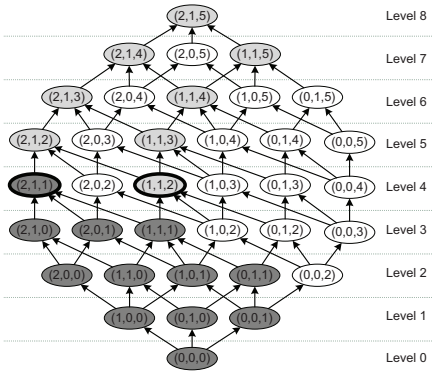

Fig. 3. Lattice and predictive tagging

Monotonicity is a very important property, which enables several optimizations for globally optimal k-anonymity algorithms.

The authors of [20] introduced the notion of *monotonic generalization hierarchies*. In a monotonic generalization hierarchy, the groups at level $l + 1$ are built by merging groups from level l. This allows pruning of large parts of the search space, because all states which are successors of an anonymous state are also anonymous. Furthermore, all predecessors of a non-anonymous state are also non-anonymous. This is because generalization is monotonic for the complete dataset if it is monotonic for each quasi-identifier. An example is shown in Figure 3, where the fact that $(2, 1, 1)$ is non-anonymous implies that all of its predecessors are also non-anonymous (dark gray). Furthermore, all successors of the anonymous state $(1, 1, 2)$ are also anonymous (light gray).

Additionally, the authors of [17] proposed the utilization of *monotonic metrics*. The monotonicity criterion for metrics requires that on each path from the bottom node to the top node of the generalization lattice, the values for information loss increase monotonically. This implies that if a generalization lattice is divided into a set of (potentially overlapping) paths the global optimum can be determined by only comparing the local optima. Furthermore, there is no need to evaluate the metrics for nodes that have been tagged predictively, because they can never be a local or global optimum.

B. Metrics

In this section, we briefly cover the most important monotonic *metrics*. The interested reader is referred to [17], [19]–[23] for further details. One of the earliest monotonic metrics is the *Height Metric*, which is utilized by the algorithm presented in [19]. This metric, as well as the *Precision Metric* [24], measure information loss solely based upon a state itself (i.e., its generalization levels) and is therefore independent of the actual input dataset. In [17], a monotonic version of the *Discernability Metric* [23] has been presented, which estimates information loss based on the equivalence classes induced by a transformation. Another relevant metric is the *Entropy Metric* [21], which compares the original input dataset with the transformed representation. If needed, weights can be assigned to quasi-identifiers to tailor a metric to a specific use case [17].

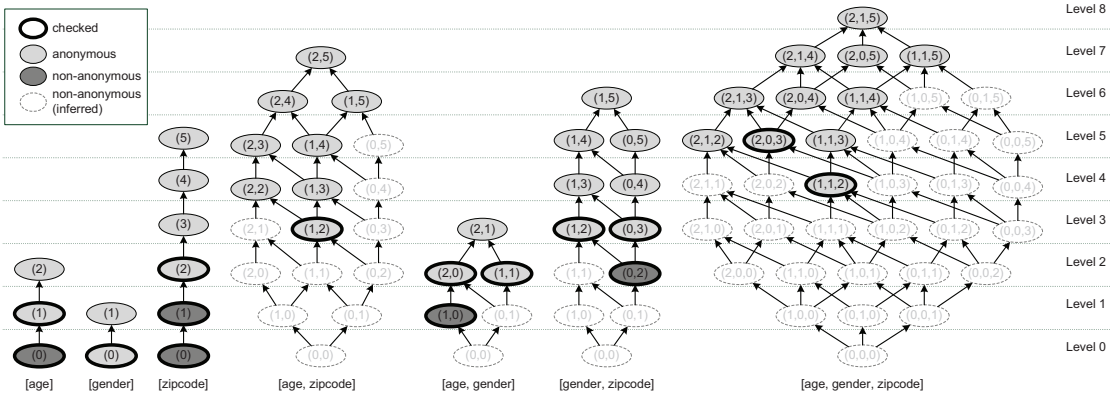

Fig. 4. Example for the Incognito algorithm

III. Previous Algorithms

In this section we quickly review the two major previous algorithms which are relevant to our context. Incognito implements a *horizontal traversal strategy* (traversing the lattice level by level), whereas OLA implements a *vertical traversal strategy* (jumping between levels).

A. Incognito

LeFevre et al. proposed the *Incognito* algorithm [20], which implements an approach related to dynamic programming. The general idea is that if a transformed subset of the quasi-identifiers is not k-anonymous, the transformation of the complete dataset cannot be k-anonymous either. Therefore it constructs generalization lattices for each individual subset of n quasi-identifiers and traverses them by performing a bottom-up, breadth-first search. It utilizes predictive tagging to prune parts of the local search space. The states that have been found to be not anonymous in a subset of size $m < n$ cannot be anonymous in a subset of size $m + 1$. This allows the predictive tagging of states of the generalization lattices constructed in subsequent iterations. The algorithm halts when the lattice for all n quasi-identifiers has been processed.

An example is shown in Figure 4. It shows the lattices built by Incognito for the example dataset. The algorithm starts by focusing on the quasi-identifier *age*. It builds a lattice and checks if the column is anonymous for the state $\langle 0 \rangle$. As this state is not k-anonymous, the algorithm proceeds with the state $\langle 1 \rangle$, which is k-anonymous. This results in predictive tagging of the state $\langle 2 \rangle$. The same procedure is applied to the quasi-identifiers *gender* and *zipcode*. After checking all subsets of size one, Incognito proceeds with all subsets of size two. In this step, it is possible to tag all nodes as non-anonymous that contain substates that were not anonymous in a previous iteration. For example, in the lattice for the columns *age* and *zipcode*, all nodes which define level 0 for *age* and level 0 or 1 for *zipcode* are tagged as being non-anonymous. Therefore, the first state that is processed by Incognito in this lattice is the state $\langle 1, 2 \rangle$. As this state represents a k-anonymous transformation, predictive tagging can be applied to all other nodes in this lattice. The other lattices are processed analogously. For more details the interested reader is referred to [20].

B. OLA

El Emam et al. proposed a k-anonymization algorithm called *Optimal Lattice Anonymization* (OLA) [17] and showed that it outperforms the approaches presented in [19] and [20]. It implements a divide-and-conquer approach. The idea is to decompose a lattice into smaller sublattices and utilize predictive tagging to prune parts of the search space. A sublattice (b, t) is defined by a bottom node b and a top node t and contains b and t as well as all nodes that are generalizations of b and specializations of t. OLA starts by processing the complete lattice. It then constructs sublattices by enumerating all nodes M on level $\lfloor \frac{1}{2}(b.level + t.level) \rfloor$ of the current lattice. If a node $m \in M$ has not been tagged already, it is checked for k-anonymity and predictive tagging is applied. If m is tagged as anonymous, the algorithm proceeds with the lower sublattice (b, m), otherwise it proceeds with the upper sublattice (m, t). This process halts when all sublattices have been enumerated.

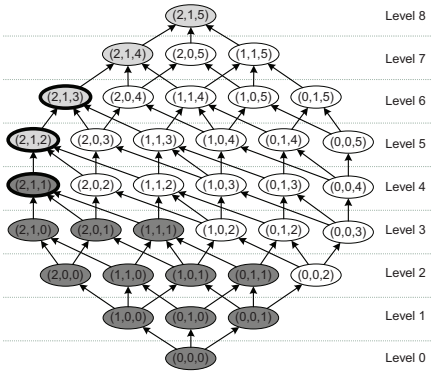

Fig. 5. Example for the OLA algorithm

The first iteration of the algorithm for our example dataset is shown in Figure 5. It starts by enumerating all nodes on level 4, and we assume that it starts with node $\langle 2, 1, 1 \rangle$. As this node has not been processed before, it is checked for anonymity. Because the state does not represent an anonymous transformation, $\langle 2, 1, 1 \rangle$ as well as all of its specializations (dark gray) are tagged as non-anonymous. The algorithm then proceeds to the upper sublattice $(\langle 2, 1, 1 \rangle, \langle 2, 1, 5 \rangle)$, which contains the light gray nodes.

In the subsequent steps, it will construct the sublattices $((2,1,1),(2,1,3))$ and $((2,1,1),(2,1,2))$, effectively checking the nodes $(2,1,1)$, $(2,1,3)$ and $(2,1,2)$ to find the locally optimal two-anonymous node $(2,1,2)$. The algorithm will then further proceed to the next node on level 4 which is $(2,0,2)$ and construct the according lower sublattice, as this state is anonymous. More details are available in [17].

IV. Implementation Framework

This work is based upon a generic framework for the efficient implementation of k-anonymity algorithms. In [25], we presented an efficient implementation of the OLA algorithm on top of it. We quickly review the framework in this section and describe the fundamental ideas behind it:

1) *The process of checking individual states for k-anonymity is the main bottleneck for this class of anonymization algorithms and should be as efficient as possible.*
2) *General purpose database systems are not well suited for k-anonymity algorithms, because they have been designed for much more complex query and transaction processing.*
3) *Given the current trend towards main-memory data management as well as the ability to use data compression techniques, a main-memory based approach is feasible.*

The groundwork of this framework is a carefully-designed memory layout, which enables the efficient application of different generalization strategies to an input dataset. Additionally, the anonymization operators are problem-aware and interwoven with the rest of the algorithm. This allows for several further optimizations. The basic implementation, including the first optimization, can be used for all generalization-based anonymization algorithms which use global recoding. The other optimizations further require monotonic generalization hierarchies.

A. Basic implementation

The framework holds all data in main memory and implements dictionary compression on all data items. Generalization hierarchies are represented in a tabular manner. An example for the generalization hierarchy of the attribute *age* from Figure 2 is shown in Figure 6.

Level 0	Level 1	Level 2
1	<50	⋆
⋮	<50	⋆
49	<50	⋆
50	≥50	⋆
⋮	≥50	⋆
100	≥50	⋆

Fig. 6. Tabular generalization hierarchy

One dictionary $dic_0, ..., dic_{n-1}$ per quasi-identifier is used to map the values contained in the corresponding column onto integer values. By encoding the values from the input dataset before the values contained in the higher levels of the generalization hierarchies it is guaranteed that the original values of a column with m distinct values get assigned the numbers 0 to $m-1$. This allows for an efficient representation of the generalization hierarchies $hier_0, ..., hier_{n-1}$ as two-dimensional arrays. An excerpt of the resulting memory layout for the example dataset is shown in Figure 7. The values of the attribute *age* from column 0 and the relevant values from the generalization hierarchy are encoded in the corresponding dictionary dic_0.

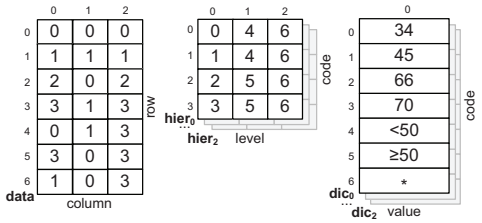

Fig. 7. Example data representation

The associated generalization hierarchy $hier_0$ is represented as a two-dimensional array where the i-th row contains the values for the original data item, which is encoded as i in the dictionary. The j-th column stores the corresponding transformed value at the j-th level of the hierarchy. The other quasi-identifiers are handled analogously. The input dataset itself is represented as a row-oriented integer array ($data$). Additionally, an equal data structure *buffer* is maintained which is used to store a transformed representation of the original data. Based on this memory layout, transforming a value from the input data in cell (row, col) to the value defined on level $level$ of its generalization hierarchy and storing it in the buffer can be implemented by a simple assignment:

$$buffer[row, col] \leftarrow hier_{col}[data[row, col], level] \qquad (1)$$

When checking a state, the algorithm iterates over all rows in the dataset and applies the assignment (1) to each cell. Afterwards, the transformed row is passed to a groupify operator, which computes the equivalence classes by adding the rows to a hash table. Finally, the k-anonymity check is applied by checking whether all classes are of size $\geq k$. Furthermore, a suppression parameter s can be specified. It defines an upper bound for the number of rows that can be suppressed in order to still consider a dataset k-anonymous. This further reduces information loss, as the minimum size k is not enforced for all equivalence classes. Instead, classes of size $< k$ are removed from the dataset as long as the total number of suppressed rows remains under the threshold. This basic implementation is already very efficient and has an amortized runtime complexity of $\mathcal{O}(n * m)$, where m is the number of rows and n is the number of columns.

B. Optimizations

The following section presents some further optimizations that exploit similarities between states which are checked consecutively by an algorithm.

a) Projection: Because the transformed data is materialized in a buffer, it is possible to only transform those parts of the data that actually change. A *projection* can be applied if two consecutive states s_1 and s_2 define the same level of

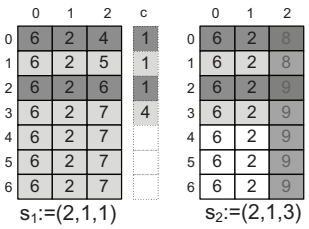

Fig. 8. Roll-up and Projection

generalization for some quasi-identifiers. These columns are already represented in the correct state in the buffer and do not need to be transformed again. An example is shown in Figure 8. Here, the columns 0 and 1 are already in the correct state when moving from the state $(2, 1, 1)$ to $(2, 1, 3)$, leaving only the column 2 subject to transformation.

b) Roll-Up: When an algorithm moves from a state s_1 to a state s_2 which is a generalization of s_1, the equivalence classes for s_2 can be built by merging the equivalence classes of s_1 if the generalization hierarchies are monotonic. This is called *roll-up* and is implemented by storing references to representative rows in the hash table. When a roll-up is performed, we iterate over all representative rows of the previous check and transform and groupify only those. A roll-up is also possible for the transition shown in Figure 8. In this figure, the classes are denoted by different shades of gray. We assume that the previous check resulted in the classes $\{0\}$, $\{1\}$, $\{2\}$ and $\{3,4,5,6\}$. The representatives and sizes (r, c) are (0,1), (1,1), (2,1) and (3,4), as indicated by the additional rightmost column. To compute the classes for s_2, it is only necessary to transform and group four rows, whereas seven rows had to be processed to check the state s_1.

c) Maintaining a buffer of snapshots: A series of roll-up operations can only be performed on a path of non-anonymous nodes. A similar technique can be applied in other state transitions, if a buffer is managed which contains snapshots of the equivalence classes of previous states (called *history*). The equivalence classes for a state s can then be built by merging the classes defined by a more specialized state s' for which a snapshot exists. If multiple suitable snapshots are available, we pick the one that consists of the fewest classes. Furthermore it is only necessary to store snapshots for non-anonymous nodes, because otherwise all generalizations will be tagged predictively. The history has a predefined maximum size which is enforced by a LRU eviction policy and it only stores snapshots which consist of not more than a predefined number of equivalence classes (time-space tradeoff). We again store snapshots as a set of tuples of representative rows and sizes, which results in a very compact representation. This optimization does not fully replace the roll-up optimization but complements it, as it only stores a limited number of snapshots of a predefined maximum size.

d) Putting it all together: The individual optimizations can be combined with each other. The transition from Figure 8 can, for example, benefit from performing a roll-up as well as a projection. There are several limitations for valid state changes which ensure that the buffer is always in a consistent state, however. Transitions are mainly restricted in the context of projections, as these can only be performed successively if the current state allows a roll-up to be performed or when all rows have been transformed previously. For further details about the framework the interested reader is referred to [25].

V. The Flash Algorithm

In this section, we present our novel Flash algorithm. It traverses the lattice in a bottom-up breadth-first manner and constantly generates paths which branch like lightning flashes. It is based upon the following observations:

1) *Predictive tagging can be best exploited if the lattice is traversed vertically and in a binary fashion.*
2) *When traversing a lattice vertically, the execution time becomes volatile in terms of the representation of the input dataset (e.g., the order of the columns). This must be prevented by implementing a stable strategy.*
3) *In order to achieve the best performance, the algorithm should prefer nodes that allow the application of the previously presented optimizations.*

A. Basic Algorithm

As shown in Algorithm 1, Flash iterates over all levels in the lattice, starting at level 0. It enumerates all nodes on each level and calls FindPath($node$) if a node is not tagged. Algorithm 2 shows that this function implements a greedy depth-first search towards the top node. The search terminates when either the top node is reached or the current node does not have a successor that is not already tagged.

Algorithm 1: Outer loop of the Flash algorithm

```
Input: Lattice lattice
1  begin
2      heap ← new min-heap
3      for l = 0 → lattice.height − 1 do
4          foreach node ∈ level[l] do
5              if !node.tagged then
6                  path ← FindPath(node)
7                  CheckPath(path, heap)
8                  while !heap.isEmpty do
9                      node ← heap.extractMin
10                     foreach up ∈ node.successors do
11                         if !up.tagged then
12                             path ← FindPath(up)
13                             CheckPath(path, heap)
```

When a path has been built, the function CheckPath($path, heap$) is called on it. As can be seen in Algorithm 3, it implements a binary search. It starts by checking the node at position $\lfloor \frac{1}{2}(path.size - 1) \rfloor$. Whenever a node is checked for k-anonymity, we also apply predictive tagging within the whole generalization lattice. Depending on the result of the check, the algorithm then proceeds with the lower or upper half of the path. Whenever a node is explicitly checked and determined to be non-anonymous, we add it to a heap. If a node is explicitly checked and determined to

be anonymous, we store a reference to it, as it could be the local optimum. There is no need to check whether a node has already been tagged, because by definition, Algorithm 2 always returns a path of untagged nodes. Furthermore, predictive tagging is always applied in the direction opposite to the one taken by the algorithm. Another important thing to note is that after the search terminates, the variable $optimum$ will always hold a reference to the optimal anonymous node on the path (if there is any). The globally optimal node is determined by comparing the current local optimum with the current global optimum in STORE($optimum$).

Algorithm 2: FINDPATH(NODE)

```
Input: Start node node
Result: Path of untagged nodes path
1  begin
2      path ← new list
3      while path.head() ≠ node do
4          path.add(node)
5          foreach up ∈ node.successors do
6              if !up.tagged then
7                  node ← up
8                  break
9      return path
```

After a path has been checked, the nodes in the heap are utilized to build new paths starting from the successors of the non-anonymous nodes that have been explicitly checked when processing the previous path. The general idea behind this is that these paths increase the potential to apply the roll-up and snapshot optimizations. A somewhat simplified description (see Section V-B) is that, the heap returns the nodes according to their level in the lattice. We start with the minimal element, as this potentially increases the length of the created path. This, in turn, increases the potential to apply predictive tagging to other nodes in the lattice.

Algorithm 3: CHECKPATH(PATH, HEAP)

```
Input: Path path, heap heap
1  begin
2      low ← 0; high ← path.size − 1; optimum ← null
3      while low ≤ high do
4          mid ← ⌊½(low + high)⌋
5          node ← path.get(mid)
6          if CHECKANDTAG(NODE) then
7              optimum ← node; high = mid − 1
8          else
9              heap.add(node); low = mid + 1
10     STORE(optimum)
```

In case of an empty heap, the algorithm proceeds with the outer loop. The algorithm terminates when the outer loop terminates. As explained in the following section, our algorithm further implements a general strategy that induces a total order on all nodes in the lattice. Whenever we iterate over a set of nodes, we follow this order. This includes all steps which are highlighted in gray in the presented pseudocode.

B. Traversal Strategy

An important property of an algorithm that implements a vertical traversal strategy is stability in terms of execution time. This is due to the fact that the order in which nodes are enumerated influences the order in which the nodes are checked. In combination with a vertical strategy and predictive tagging, this leads to different lattice traversals, which leads to a varying number of k-anonymity checks. This finally leads to differences in the algorithms' execution times. In practice, these differences can be, for example, triggered by changing the order of the columns in the input dataset (see Section VI-E). The solution to this problem is to apply a fixed strategy whenever nodes are enumerated. The idea behind our strategy is to prefer nodes with a lower degree of generalization.

A node is a tuple $n = (n_0, ..., n_j)$ with $0 \leq n_i \leq m_i$ for all $0 \leq i \leq j$, where m_i defines the maximum level in the hierarchy of the i-th quasi-identifier. Furthemore $distinct(i, l) = |\{hier_i[x][y] \mid y = l\}|$ returns the distinct number of values on level l of the i-th generalization hierarchy. We define three criteria for each node n. The criterion $c_1(n)$ returns the level of the node in the lattice:

$$c_1(n) = \sum_{i=0}^{j} n_i \tag{2}$$

The criterion $c_2(n)$ is used to differentiate between nodes on the same level and resembles the Precision metric. It returns the average generalization over all quasi-identifiers:

$$c_2(n) = \frac{1}{j} \sum_{i=0}^{j} \frac{n_i}{m_i} \tag{3}$$

Finally, the criterion $c_3(n)$ is utilized to differentiate between nodes on the same level which also describe the same average generalization. It represents the average over the number of distinct values on the current level of each quasi-identifier:

$$c_3(n) = 1 - \frac{1}{j} \sum_{i=0}^{j} \frac{distinct(i, n_i)}{distinct(i, 0)} \tag{4}$$

These three criteria are then combined into a vector in $\mathbb{R}^3$ in the following way:

$$c(n) = \begin{pmatrix} c_1(n) \\ c_2(n) \\ c_3(n) \end{pmatrix} \tag{5}$$

The nodes are traversed according to the totally ordered vector space induced by the relation $\leq$, which defines the lexicographical order, i.e. $c(n_1) \leq c(n_2)$ iff:

- $c_1(n_1) < c_1(n_2)$, or
- $c_1(n_1) = c_1(n_2) \wedge c_2(n_1) < c_2(n_2)$, or
- $c_1(n_1) = c_1(n_2) \wedge c_2(n_1) = c_2(n_2) \wedge c_3(n_1) \leq c_3(n_2)$.

We apply this order when enumerating nodes on a level and when enumerating successors of a node. Finally, the priority function also serves as the key for the entries in the heap. The

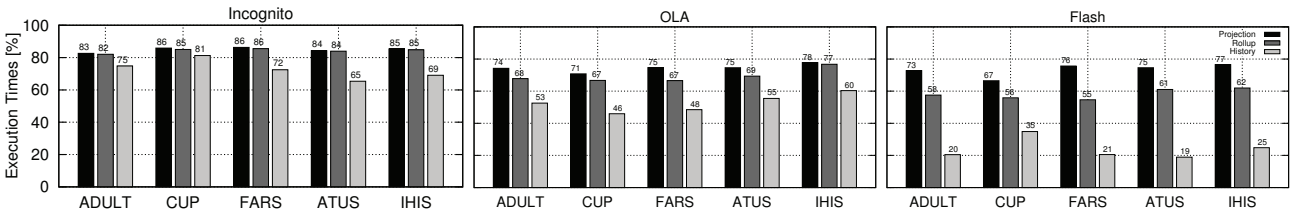

Fig. 9. Effectiveness of optimization levels

strategy has to be implemented carefully. Sorting all levels in the lattice and all pointers to successors prior to the execution of the algorithm is too expensive. We therefore evaluate the priority function lazily and sort a nodes' successors only when needed. Analogously, a level is sorted directly before iterating over it. This allows exclusion of all nodes which have already been tagged from the sorting step.

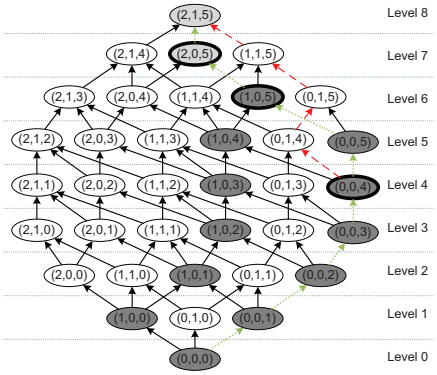

Fig. 10. Example for the Flash algorithm

C. Example

The first two iterations of the Flash algorithm can be seen in Figure 10. It starts by building a path from the bottom node $(0,0,0)$ to the top node $(2,1,5)$. This results in the rightmost path, indicated by the dotted lines. This path is then checked in a binary manner, which results in explicit checks of the nodes $(0,0,4)$, $(1,0,5)$ (non-anonymous) and $(2,0,5)$ (anonymous). The states of the other non-anonymous nodes (dark gray) are determined by applying predictive tagging from $(0,0,4)$ and $(1,0,5)$. The top-node is predictively tagged as being anonymous from the node $(2,0,5)$. After checking the first path, the heap contains the nodes $(1,0,5)$ and $(0,0,4)$. As $(0,0,4)$ is prioritized according to our general strategy, we start to build a new path from $(0,0,4)$, which is denoted by the dashed lines. When the heap is empty, the algorithm builds the next path starting from $(2,0,0)$, as $(0,1,0)$ has already been tagged.

VI. Evaluation

For the evaluation we used five real-world datasets, most of which have already been utilized for benchmarking previous work on k-anonymity.

A. Datasets

The datasets include the 1994 US census database (ADULT), KDD Cup 1998 data (CUP), NHTSA crash statistics (FARS), the American Time Use Survey (ATUS) and the Integrated Health Interview Series (IHIS). The ADULT dataset serves as a de-facto standard for the evaluation of k-anonymity algorithms. An overview over the datasets is shown in Figure 11. They cover a wide spectrum, ranging from about 30k to 1.2M rows (2.52 MB to 107.56 MB) consisting of eight or nine quasi-identifiers. The associated generalization hierarchies have a height between 2 and 6 levels. The number of states in the generalization lattice, which is defined by the number of quasi-identifiers as well as the height of the associated hierarchies, ranges from 12,960 for the ADULT dataset to 45,000 for the CUP dataset.

Dataset	QIs	Records	States	Size [MB]	Init [ms]
ADULT [1]	9	30,162	12,960	2.52	86
CUP [2]	8	63,441	45,000	7.11	334
FARS [3]	8	100,937	20,736	7.19	152
ATUS [4]	9	539,253	34,992	84.03	1,031
IHIS [5]	9	1,193,504	25,920	107.56	1,627

Fig. 11. Evaluation datasets

B. Setup

The benchmarks were performed on a desktop machine equipped with a quad-core 3.1 GHz Intel Core i5 CPU running a 64-bit Linux 3.0.14 kernel. The algorithm was implemented in Java and executed on a 64-bit Sun JVM (1.6.0) with a heap size of 512 MB. We anonymized each of the datasets with $2 \leq k \leq 10$, suppression rates s of 0%, 2% and 4% and the monotonic *Discernability Metric*. We furthermore incrementally enabled the optimizations, resulting in four different configurations. We executed each of these configurations for each algorithm (Incognito, OLA and Flash) and combination of the parameters k and s, which resulted in 108 runs per dataset. The threshold for the maximum size of a snapshot was set to 20% and the size of the history was limited to 200 entries. Regarding the algorithms' runtime behaviour we observed analogous results for other parameters. The results are reported without the time needed for initialization, which includes reading the entire data from disk and performing dictionary encoding (see Figure 11). The execution time of all three algorithms is dominated by the time spent on k-anonymity checks. As these are implemented in the same way for all algorithms, the comparison is fair.

C. Overview

Figure 9 shows the workload averages (geometric mean over all values of k and s) for each dataset and algorithm. The basic

[1] http://archive.ics.uci.edu/ml/datasets/adult
[2] http://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html
[3] http://www-fars.nhtsa.dot.gov/main/index.aspx
[4] http://atusdata.org/index.shtml
[5] http://www.ihis.us/

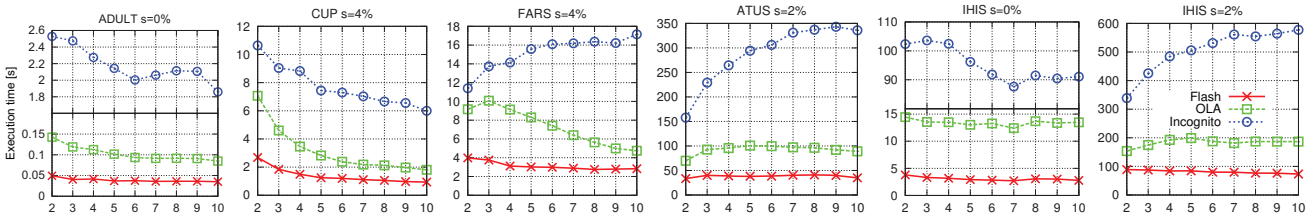

Fig. 12. Execution times for individual configurations and $2 \leq k \leq 10$

implementation, which does not implement any optimizations, defines the 100% baseline. The optimizations are enabled incrementally, and it can be seen that all optimizations have a positive effect on the execution times of all algorithms for all datasets. Although we only show workload averages due to the limited space, it is important to note that none of the optimizations yields any overhead for any configuration. The Flash algorithm benefits the most, as it was explicitly designed to fully exploit the framework. Incognito benefits the least because of its horizontal traversal strategy, which, for example, completely prevents the roll-up optimization. Figure 13 presents a comparison of the algorithms in terms of the geometric mean (logarithmic scale) of the execution times in seconds. Flash outperforms all algorithms and the speedup compared to the previous state-of-the-art (OLA) ranges from about 37% for CUP to a factor of 2.7 for IHIS.

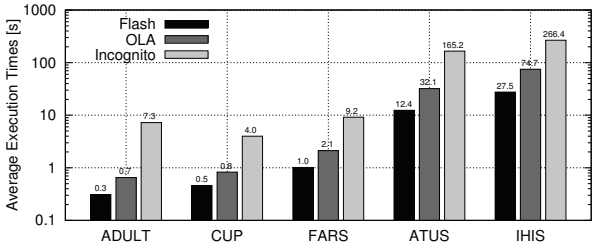

Fig. 13. Comparison of average execution times

D. Details

Figure 12 shows a comparison of the actual execution times of the algorithms for $2 \leq k \leq 10$ on selected configurations and datasets. It can be seen that the execution times of the algorithms do not differ only by a constant factor, but rather follow different trends. In the context of an individual configuration, Flash offers an almost constant execution time and outperforms the other algorithms for each value of k.

	ADULT	CUP	FARS	ATUS	IHIS
s=0%	0.04 – 0.05	0.07 – 0.08	0.10 – 0.17	1.20 – 1.80	2.72 – 3.79
s=2%	0.50 – 1.35	0.83 – 1.92	2.31 – 3.74	33.55 – 40.95	73.22 – 89.04
s=4%	0.74 – 1.70	0.94 – 2.69	2.75 – 3.99	17.11 – 38.65	80.51 – 88.63

Fig. 14. Execution times of the Flash algorithm [s]

As can be seen in the Figures 14 and 16, this is also true for the other configurations. Figure 14 shows the minimum and the maximum of the execution times of Flash for $2 \leq k \leq 10$ on all datasets and suppression rates in seconds. It can be seen that Flash is able to find the optimal solution in well under 4 seconds for all configurations despite ATUS and IHIS with 2% and 4% suppression rates. In these cases, more checks have to be performed to find the optimal solution, which leads to execution times in the order of a minute.

Figure 16 compares the performance of Incognito and OLA to our algorithm. It shows the minimum and maximum factors between the execution times of the other algorithms and Flash over $2 \leq k \leq 10$ for each configuration. It can be seen that Flash outperforms the other algorithms for all configurations on all datasets. The largest performance gain in comparison to the Incognito algorithm exists for a 0% suppression rate with a factor of more than 70 for ATUS ($k = 8$). Flash especially outperforms OLA on the two largest datasets at a 0% suppression rate, with a factor of about 4 for ATUS ($k = 8$) and a factor of about 5 for IHIS ($k = 10$).

	ADULT	CUP	FARS	ATUS	IHIS
	s=0%				
OLA	2.4 – 3.0	1.3 – 1.6	2.0 – 2.5	3.0 – 3.9	3.8 – 4.9
Incognito	51.2 – 61.3	14.8 – 18.2	21.8 – 25.8	55.4 – 72.2	27.0 – 32.9
	s=2%				
OLA	1.7 – 2.2	1.5 – 2.3	1.5 – 2.7	2.1 – 2.6	1.7 – 2.6
Incognito	10.2 – 24.0	5.1 – 8.2	3.9 – 8.2	4.7 – 9.6	3.8 – 7.9
	s=4%				
OLA	1.6 – 2.0	1.9 – 2.6	1.7 – 2.9	1.2 – 2.8	1.4 – 2.4
Incognito	7.1 – 17.9	4.0 – 6.7	2.9 – 6.1	3.5 – 7.5	2.8 – 6.1

Fig. 16. Performance of Incognito and OLA compared to Flash [factor]

The experiments also show that our generic framework is suitable for the efficient implementation of other algorithms for optimal k-anonymity. Furthermore, OLA clearly outperforms the Incognito algorithm, up to an order of magnitude in some cases. We have not included numbers for the *SuperRoots* variant of Incognito, because our experiments revealed that it is only beneficial without suppression and decreases the performance in all other cases. Flash outperforms OLA in all cases, although our implementation of OLA is already highly efficient and includes several optimizations [25].

Dataset	Flash	OLA	Incognito
ADULT	8 – 15	10 – 18	8 – 13
CUP	39 – 55	44 – 79	37 – 43
FARS	20 – 46	22 – 55	20 – 37
ATUS	81 – 155	87 – 163	80 – 146
IHIS	166 – 432	172 – 471	162 – 376

Fig. 17. Memory consumption [MB]

Figure 17 presents a comparison of the algorithms' memory consumption. The snapshots stored by the history optimization dominate the overall memory footprint. Incognito uses the least memory, whereas OLA uses the most. Incognito has the lowest memory usage, because many nodes in the largest generalization lattice (for all quasi-identifiers) are already tagged when the lattice is traversed. It therefore creates significantly fewer snapshots than the other algorithms. Flash is designed to immediately exploit snapshots, which leads to early eviction of many entries in the buffer. It thus shows memory consumption which is in between the two extremes, Incognito and OLA.

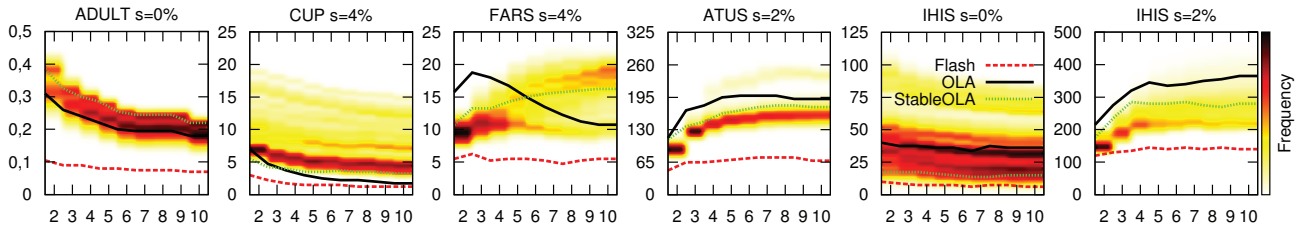

Fig. 15. Distribution of the execution times of OLA for $2 \leq k \leq 10$

Although the Flash algorithm has been designed to leverage our implementation framework as much as possible, it also offers competitive performance in other implementation scenarios. This can be seen by comparing Flash and OLA when all optimizations are disabled and the execution times are basically defined by the number of k-anonymity checks performed by the algorithms. In this case, Flash is up to 30% slower for the ADULT and FARS dataset with a 0% suppression rate. As these are the cases in which the execution times are very low, however, Flash still provides very good performance. In all other cases (including the larger datasets), the performance of Flash and OLA differs only by up to 10% and there is no clear winner. As soon as the optimizations are enabled, Flash outperforms OLA in all configurations by large margins. Furthermore, Flash offers algorithmic stability as is described in the following section.

E. Algorithmic Stability

In contrast to Incognito, Flash and OLA implement a vertical traversal strategy; however, only Flash offers a stable execution time. *Stable* means that the execution time of the algorithm does not depend on the order of the columns in the input dataset or the algorithm that is used to build the generalization lattice. For example, a node $(1, 0, 0)$ represents a different transformation if the first column is swapped with another column in the dataset. This is an important property, as otherwise the behavior of an algorithm is not reproducible without explicit definition of these surrounding conditions.

As this property has not been discussed in any previous work, we assume that the experiments therein have been performed based on a *natural ordering*. First, the order of the columns in the input datasets has been preserved as is. Secondly, the successors of a node are ordered according to which quasi-identifier has been incremented. Finally, the nodes on a level are enumerated in the same way, assuming a breadth-first strategy during the lattice building process. This strategy has also been used for all example lattices in this paper and in our experiments (see, e.g., Figure 3). We were able to closely reproduce the previously published results.

The distribution of OLA's execution times can be determined if a fixed lattice building algorithm is used and OLA is executed for all permutations of the columns in the input datasets. Unfortunately, this is not feasible, as the number of permutations is factorial in the number of columns (e.g., $9! = 362,880$). Instead, we precomputed the anonymity property for all states in the lattices and used this information to simulate the execution of OLA for all permutations. The simulations take all optimizations from our framework into account. Each simulation results in a sequence of k-anonymity checks T, which were performed while processing one permutation. Each check $t \in T$ is defined by the number of active columns (t_c), as induced by a projection, and the number of active rows (t_r), as induced by the roll-up and history optimizations. c is the overall number of columns in the dataset. Based on this information, we developed a cost model which estimates the overall execution time. For one check $t \in T$, the costs for transforming the data are defined as $costs_t(t) = t_c \cdot t_r$, which is the number of cells that had to be transformed. Furthermore, the costs for grouping the data are $costs_g(t) = c \cdot t_r$, because the costs of grouping always depend on the total number of quasi-identifiers in the dataset. In order to fit the resulting costs to the actual times measured in the experiments, we divided the results by a constant factor. The overall costs for a simulation T were defined as:

$$costs(T) = \sum_{t \in T} costs_t(t) + costs_g(t) = \sum_{t \in T} t_c \cdot t_r + c \cdot t_r$$

Figure 15 shows the distribution of the costs for different values of k and selected datasets. The frequency of a cost estimate is represented by colors ranging from white (lowest) to black (highest). The solid lines (OLA) represent the costs of the standard OLA algorithm with natural ordering, which has been used in the other experiments (e.g., in Figure 12). The dotted lines (StableOLA) show the costs of an implementation of OLA, which uses the same strategy as the Flash algorithm and is thus stable. To this end, the nodes on level $\lfloor \frac{1}{2}(b.level + t.level) \rfloor$ for a sublattice (b, t) are enumerated in the order induced by our traversal strategy (see Section V-B). The dashed lines represent the costs of Flash.

As can be seen when comparing Figure 15 to Figure 12, our cost model closely resembles the actual execution times of the algorithms. Except for the FARS dataset at 2% and 4% suppression rates, the trends of the resulting frequencies are very similar to the trends measured in our experiments. In contrast, the costs estimated for StableOLA always follow the overall trend. On average, this variant does not outperform standard OLA. The costs of OLA vary greatly (e.g., by up to a factor of 38 for IHIS with s=0%). The performance with natural ordering is sometimes excellent (e.g. CUP, s=4%) and sometimes quite poor (e.g. IHIS, s=2%). In contrast, the Flash algorithm offers a stable execution time, which outperforms OLA with natural ordering and OLA with the same stable strategy in all cases. Flash even outperforms the best run of OLA (fastest permutation) in 114 out of 135 cases.

VII. Discussion and Future Work

In this paper, we have presented a generic implementation framework for globally-optimal full-domain k-anonymity algorithms. We have shown that it is well suited for the efficient implementation of the two major algorithms from this class, Incognito and OLA. We confirmed that OLA outperforms Incognito in case of natural ordering, but showed that OLA is not stable in terms of execution times. We further presented a novel k-anonymity algorithm, which clearly outperforms Incognito and OLA. Due to its vertical traversal strategy, it fully exploits predictive tagging and has been designed to make exhaustive use of our implementation framework. It offers algorithmic stability by solely relying on a consistent strategy that implies a total order among the nodes in the generalization lattice. Even when it is not implemented on top of our framework, it displays highly competitive performance.

In the original work on OLA, the authors reported a total execution time of 20 s for a subset of the ADULT dataset ($k = 5$, $s = 1\%$) on a comparable testbed [14]. With our framework, OLA was able to compute the optimal solution for the exact same configuration in 550 ms, whereas Flash needed 310 ms (both numbers include initialization). In [20], the authors also published results for the same dataset ($k = 5$, $s = 0\%$) and an implementation of Incognito on top of a relational database system. In contrast to more than 3 minutes in the original work, our implementation of Incognito was able to solve this problem in less than 1s (including initialization) for the exact same configuration. Flash needed about 100 ms. This shows that a large performance gain can be achieved by implementing a dedicated data management framework for k-anonymity algorithms. An additional feature of our framework is that monotonic metrics can be evaluated with nearly no additional cost. As all locally optimal nodes are explicitly checked for k-anonymity, complex metrics, which are based on the data itself or the resulting equivalence classes (e.g, *Entropy*), can directly access the transformed data in the buffer or the equivalence classes in the hash table. It is further possible to include the ℓ-diversity and t-closeness properties into our framework, which allow further balancing of information loss versus the degree of privacy protection.

In future work we are planning to better leverage the capabilities of modern multi-core processors by parallelizing our implementation framework as well as the Flash algorithm. Early experiments with simple intra-operator parallelization within our framework were promising, but the parallelization of the k-anonymity algorithm itself is challenging. In case of limited availability of main memory or very large datasets, a disk-based implementation of a k-anonymity algorithm might be needed. Although this can be implemented on top of a relational database system, we are currently investigating a disk-based version of our framework and are confident that it clearly outperforms any off-the-shelf relational database system. Our generic framework and the implementation of the Flash algorithm are available as open-source software[1].

[1] https://code.google.com/p/flash-anonymizer

VIII. Acknowledgements

This work has been supported by the Graduate School of Information Science in Health (GSISH) and the TUM Graduate School.

References

[1] S. L. Garfinkel, "Database nation - the death of privacy in the 21th century," O'Reilly, 2001.

[2] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Proc. IEEE Symposium on Security and Privacy*, 2008, pp. 111–125.

[3] M. Zeller and T. Barbaro, "A Face Is Exposed for AOL Searcher No. 4417749," The New York Times, Aug. 2006.

[4] L. Sweeney, "Computational disclosure control - a primer on data privacy protection," Ph.D. dissertation, Massachusetts Institute of Technology, 2001.

[5] P. Samarati and L. Sweeney, "Generalizing data to provide anonymity when disclosing information," in *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1998, p. 188.

[6] C. Dwork, "Differential privacy," in *33rd Int. Colloq. on Automata, Languages and Programming*, 2006, p. 112.

[7] F. K. Dankar and K. E. Emam, "The application of differential privacy to health data," in *EDBT/ICDT Workshops: The 5th Int. Workshop on Privacy and Anonymity in the Infrom. Soc.*, 2012.

[8] G. Aggarwal *et al.*, "Achieving anonymity via clustering," *ACM Trans. on Algorithms*, vol. 6, no. 3, 2010.

[9] A. Meyerson and R. Williams, "On the complexity of optimal k-anonymity," in *Proc. 32rd ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, 2004, pp. 223–228.

[10] G. Aggarwal *et al.*, "Approximation algorithms for k-anonymity," in *Proc. Int. Conf. on Database Theory*, Nov. 2005.

[11] A. Machanavajjhala *et al.*, "*L*-diversity: Privacy beyond *k*-anonymity," *ACM Trans. on Knowledge Discovery from Data*, vol. 1, no. 1, 2007.

[12] N. Li *et al.*, "t-closeness: Privacy beyond k-anonymity and l-diversity," in *23rd Int. Conf. on Data Engineering*, 2007, pp. 106–115.

[13] M. E. Nergiz *et al.*, "Hiding the presence of individuals from shared databases," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2007, pp. 665–676.

[14] K. E. Emam, "Risk-based de-identification of health data," *IEEE Security & Privacy*, vol. 8, no. 3, pp. 64–67, 2010.

[15] V. Ciriani *et al.*, "*k* -anonymous data mining: A survey," in *Privacy-Preserving Data Mining*, 2008, pp. 105–136.

[16] B. C. M. Fung *et al.*, "Privacy-preserving data publishing: A survey of recent developments," *ACM Computing Surveys*, vol. 42, no. 4, 2010.

[17] K. E. Emam *et al.*, "A globally optimal k-anonymity method for the de-identification of health data," *J. Amer. Medical Informatics Assoc.*, vol. 16, no. 5, pp. 670–682, 2009.

[18] N. Li *et al.*, "Provably private data anonymization: Or, k-anonymity meets differential privacy," Purdue University, Tech. Rep. TR-2010-27, 2011.

[19] P. Samarati, "Protecting respondents' identities in microdata release," *IEEE Trans. on Knowledge and Data Engineering*, vol. 13, no. 6, pp. 1010–1027, 2001.

[20] K. LeFevre *et al.*, "Incognito: Efficient full-domain k-anonymity," in *Proc. 2005 ACM SIGMOD Int. Conf. on Management of Data*, 2005, pp. 49–60.

[21] A. DeWaal and L. Willenborg, "Information loss through global recoding and local suppression," in *Special issue on SDC 14*. Netherlands Official Statistics, 1999, pp. 17–20.

[22] L. Sweeney, "Datafly: A system for providing anonymity in medical data," in *Proc. IFIP TC11 WG11.3 11th Int. Conf. on Database Security XI: Status and Prospects*, 1997, pp. 356–381.

[23] R. J. Bayardo and R. Agrawal, "Data privacy through optimal k-anonymization," in *Proc. 21st Int. Conf. on Data Engineering*, 2005, pp. 217–228.

[24] L. Sweeney, "Achieving k-anonymity privacy protection using generalization and suppression," *Int. J. of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 571–588, 2002.

[25] F. Kohlmayer *et al.*, "Highly efficient optimal k-anonymity for biomedical datasets," in *Proc. 25th IEEE Int. Symp. on Computer-Based Medical Systems*, 2012.