THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, Singapore 117590

TRB2/08

CASTLE: Continuously Anonymizing Data Streams

Jianneng Cao, Barbara Carminati, Elena Ferrari and Kian Lee Tan

February 2008

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

CASTLE: Continuously Anonymizing Data Streams

Jianneng Cao[1], Barbara Carminati[2], Elena Ferrari[2], Kian Lee Tan[1]
[1]School of Computing, National University of Singapore [2]DICOM, University of Insubria, Italy

***Abstract*— Most of existing privacy preserving techniques, such as *k*-anonymity methods, are designed for static data sets. As such, they cannot be applied to streaming data which are continuous, transient and usually unbounded. Moreover, in streaming applications, there is a need to offer strong guarantees on the maximum allowed delay between incoming data and the corresponding anonymized output. To cope with these requirements, in this paper, we present *CASTLE* (*Continuously Anonymizing STreaming data via adaptive cLustEring*), a cluster-based scheme that anonymizes data streams on-the-fly and, at the same time, ensures the freshness of the anonymized data by satisfying specified delay constraints. We further show how CASTLE can be easily extended to handle l-diversity. Our extensive performance study shows that CASTLE is efficient and effective wrt the quality of the output data.**

***Index Terms*— Data stream, Privacy-preserving data mining, Anonymity.**

I. Introduction

Data streams are common to many application environments, such as, for instance, telecommunication, market-basket analysis, network monitoring, sensor networks, and so on. Such streams may contain many private information that must be carefully protected. As an example, consider Amazon.com. In a single day, it records hundreds of thousands of online sales transactions, which are received in the form of streaming data. Suppose the sales transaction stream has the schema $S(tid, cid, goods)$, whereas a relation C containing the information about Amazon customers is stored on disk, with schema $C(cid, name, sex, age, zipcode, address, telephone, credit\text{-}card\#)$. Suppose moreover that, to analyze customers' buying behavior, a mining algorithm operates on a data stream obtained by joining S with relation C. To protect users' privacy, attributes that explicitly identify the customers (such as $name$, $address$ and $telephone$) are projected out of the join result. However, in many cases, the remaining data may still be vulnerable to *linking attacks*: data can be exploited to re-identify individuals by linking or matching the data to other data sources, such as for instance a voter registration table containing the customers $gender$, $birthday$ and $zipcode$.

A well-known technique to anonymize data is *k*-anonymity [23]. A data set T is said to satisfy the *k*-anonymity property with respect to attribute set *QI*, if each combination of values of *QI* in *T* occurs at least *k* times; here, the attributes in *QI* are termed as *quasi-identifier* attributes and can be used to link with external information. A well-known technique to achieve *k*-anonymity, exploited by many of the methods proposed so far (e.g., [10], [14], [19], [20], [25], [27]), is generalization. Generalization implies that a quasi-identifier attribute value is replaced by a less specific but semantically related value. For instance, a value in the Sex domain $\{male, female\}$ can be generalized to $person$. However, traditional *k*-anonymity schemes (see Section VI for a survey) are not suitable for streaming data. The main reason is that these methods cannot be directly applied on streaming data, because they are designed for static data sets. First, these techniques typically assume that each record in a data set is associated with a different person, that is, that each person appears in the data set only once. Although this assumption is fine in a static setting, this is not realistic for streaming data. Second, data streams have a *temporal dimension*, in that they arrive at a certain rate, they are dynamically processed, and the result is output with a certain delay. In some applications, the output data are immediately used to trigger appropriate procedures. For example, in a sensor network application the output stream can be used to real-time react to some anomalous situations, and the time to react is very crucial. Therefore, the application receiving the output stream should have strong guarantees on the *maximum delay* of the output data. This is not easily obtainable with traditional *k*-anonymity schemes. Indeed, the unique way to exploit *k*-anonymity schemes devised for static data sets to *k*-anonymize data streams works in *batches* as follows: (a) as tuples arrive, they are buffered; (b) once a sufficient number of tuples have arrived, the algorithm can be applied on the buffered content in order to anonymize the data; (c) output the buffered (anonymized) data as a data stream. However, this method has three main shortcomings. First, it implies a huge storage overhead, in that it requires at least double the amount of buffer: while the anonymizing algorithm operates on one buffer, the other buffer receives the input data stream. Second, this method causes a delay between the data arrival time (in the input data stream) and the anonymized data output time (in the output stream). Unfortunately, this delay is expected to be large since it does not make sense to anonymize a small number of tuples. As such, for application with strict time constraints this method cannot be applied. Third, the quality of the anonymized data may not be as good since the data size to be anonymized is restricted to the buffer size only.

As we will further explain in Section II also the naive solution of joining S with a *k*-anonymized version of the relation C is not adequate, mainly because *(a)* the dimensions (i.e., quasi-identifier attributes) to be anonymized may come from both C and S; *(b)* linking attacks are still possible due to the possible inferences that can be performed by inspecting the sequence of anonymized tuples given in output.

Recently, the problem of anonymizing dynamic datasets , i.e., datasets where new tuples are inserted as well existing ones are deleted, has started to be investigated [11], [24], [28], [29]. Anonymizing data streams and anonymizing dynamic datasets have some similarities, in that they both suffer from potential inferences due to dynamic updates. However, as it will be discussed in Section VI, the inferences that may arise when anonymizing dynamic datasets are totally different from those

A preliminary version of this paper, titled: *CASTLE: a delay constrained scheme for k_s-anonymizing data streams*, has been accepted as a poster at ICDE'08.

that can take place during anonymization of data streams, that is, they are possible under different assumptions. This makes the methods proposed for secure anonymization of dynamic datasets not suitable for data streams.

To cope with all the above-discussed requirements, in this paper, we present *CASTLE* (*Continuously Anonymizing STreaming data via adaptive cLustEring*), a cluster-based scheme that k-anonymizes streams on the fly and, at the same time, ensures the freshness of anonymized data by satisfying specified delay constraints. Moreover, we propose an extension of CASTLE to support l-diversity [22] on data streams. To the best of our knowledge, this is the first reported work that considers k-anonymity and l-diversity on data streams.

The basic idea of the proposed approach is to exploit quasi-identifier attributes to define a metric space: tuples are modeled as points in this space. CASTLE groups incoming tuples in clusters and releases all tuples belonging to the same cluster with the same generalization. CASTLE supports both numerical and categorical attributes anonymization, by generalizing the latter through domain generalization hierarchies, and the first through intervals. Clustering of tuples is further constrained by the need to have fresh anonymized data. To cope with this requirement, CASTLE ensures that the delay between a tuple's input and its output is at most equal to a given input parameter δ. We refer to this constraint as *delay constraint*. When a tuple is going to expire (i.e., its delay is equal to $\delta - 1$), CASTLE immediately releases it. Obviously, it could be the case that an expiring tuple does not belong to a cluster with size at least k. To manage this case, CASTLE implements a merge and split technique to obtain a cluster with size at least k and whose generalization minimizes the information loss. Additionally, to reduce information loss, CASTLE exploits a strategy that allows the reuse of clusters. When a cluster is anonymized and output all its tuples, CASTLE still keeps it (a.k.a. the corresponding generalization) in memory to anonymize newly arriving tuples, if necessary. However, we found that adopting a naive reuse strategy is flawed *even if it strictly follows the definition of k-anonymity on static data sets*, since it is vulnerable to inference attacks that exploits the sequence of anonymized tuples returned as output. In the paper, we present the reuse strategy employed by CASTLE to avoid such privacy breaches.

The remainder of this paper is organized as follows. Section II defines k-anonymity and delay constraints for data streams. In Section III, we overview CASTLE, whereas Section IV presents CASTLE algorithms and formal results about their security, and complexity. Section V reports experimental results on the effectiveness and efficiency of CASTLE. Section VI reviews related work, whereas Section VII concludes the paper. Proofs of the main formal results presented in the paper are reported in Appendix.

II. Anonymizing data streams

As discussed in the introduction, the traditional k-anonymity definition and the algorithms to achieve it are not adequate for data streams. Let us see in more details why. Consider once again the Amazon.com example. A naive solution to achieve stream anonymity is to apply standard k-anonymization techniques to relation C and then joining the anonymized version with stream S. As remarked in the introduction, the first problem with this solution is that the attributes to be anonymized may come from both C and S. Thus, the solution of anonymizing only C is not useful since the resultant joined stream must be further anonymized.

CID	Name	Sex	Zipcode	Age
01	Mike	M	53708	31
02	Alice	F	53715	21
03	John	M	53703	28
04	Bob	M	53706	26
05	Beth	F	53703	24
06	Carol	F	53706	22

TABLE I
CUSTOMER TABLE

CID	Name	Sex	Zipcode	Age
01	Mike	M	[53703-53708]	[26-31]
03	John	M	[53703-53708]	[26-31]
04	Bob	M	[53703-53708]	[26-31]
02	Alice	F	[53703-53715]	[21-24]
05	Beth	F	[53703-53715]	[21-24]
06	Carol	F	[53703-53715]	[21-24]

TABLE II
3-ANONYMIZED CUSTOMER TABLE

Further, applying the naive solution to data streams results in what has been called in [9] *weak k-anonymity*. Weak k-anonymity is less secure than k-anonymity and provides strong security guarantees only under the assumption that an adversary does not know a priori whether an individual is in the output stream or not, which appears too strong for most of the application environments related to data streams.

As an example, assume Table I is a portion of the customer table, in which quasi-identifier attributes are *Sex, Zipcode* and *Age*. Table II is a 3-anonymized version of Table I. Suppose that Beth is a neighbor of Bob and knows the detailed information of Bob's quasi-identifiers. Suppose, moreover, that Beth knows Bob has bought something from a store, and finds an anonymized record in the output stream published by this store, i.e., *(M, [53703-53708], [26-31], sex video)*. In addition, no other record with the same generalized quasi-identifiers is found. So Beth is sure that this record refers to Bob, not to John or Mike, and knows that Bob bought sex video. Bob's privacy is therefore violated.

In addition, applying the naive solution requires a further anonymization step if we want to support l-diversity [22]. l-diversity requires that the tuples with the same generalized values should have at least l distinct values for each different sensitive attribute. Since the resultant joined stream only k-anonymizes quasi-identifier attributes from C, it should be further processed to make the sensitive attributes from S l-diverse.

For example, consider once again Tables I and II. Let $k = 3$ and $l = 2$. Assume Bob bought sex video, John bought vegetable, and Carol bought a book. By joining these three transactions with Table II three tuples are generated: (M, [53703-53708], [26-31], sex video), (M, [53703-53708], [26-31], vegetable), and (F, [53703-53715], [21-24], book). The first two together are 2-diverse. But the last one alone cannot be. So the three tuples should be anonymized again to ensure l-diversity and the following three tuples are generated: ([M-F], [53703-53715], [21-31], sex video), ([M-F], [53703-53715], [21-31], vegetable), ([M-F], [53703-53715], [21-31], book).

Further the assumption made by traditional k-anonymity schemes that each record in a data set is associated with a different person, that is, that each person appears in the data set only once works for static data sets but it is not realistic for streaming data. For instance, consider again the example of Amazon.com and assume that one customer purchased $n \geq k$ items. After joining these transactions with customer relation C, n tuples with the same quasi-identifiers will appear in the resulting stream. By traditional k-anonymization schemes they can be output immediately since $n \geq k$. However, the quasi-identifiers can be linked with voting list. Thus, the customer is re-identified.

Therefore, we strongly believe that solutions specifically conceived to take into account the characteristics of data streams are needed. In this section, we start by revising the standard *k*-anonymity principle to take into account the continuity of data streams, and to relax the assumption that each record in the data set is associated with a different person. Then, we present the information loss metric used throughout the paper. In Section III we present our framework to anonymize data streams.

A. k_s-anonymity of data streams

In the following, we model a data stream as an infinite append-only sequence of tuples with an incremental order that stores, together with standard attributes, also information about when the data have been collected. This is usually modeled as an additional attribute storing the time of origin of the corresponding tuple, or the position of the tuple inside the stream. Without loss of generality, we will consider the tuple position throughout the paper. Thus, given a tuple t in a stream S, we denote with $t.p$ the attribute of t storing the position of tuple t.

Definition 1 (Data stream k_s-anonymity): Let $S(p, pid, a_1,\ldots,a_j, q_1,\ldots,q_n)$ be a stream, where QI=$\{q_1,\ldots,q_n\}$ are quasi-identifiers, pid is the person's identity, p is the tuple's position, and $a_1, \ldots, a_j$ are the remaining attributes. Let S_{out} be the anonymized stream generated from S where p and pid have been pruned. We say that S_{out} is *k_s-anonymized*, if both the following conditions hold:

- For each tuple $t \in S$, there exists in S_{out} the corresponding anonymized tuple $\bar{t}$.
- Given a tuple $t \in S_{out}$ we define qi as the corresponding *QI group*, where $qi = \{\bar{t} \in S_{out} | t.q_j = \bar{t}.q_j, j \in [1, n]\}$. Given a QI group qi, let $DP(qi)$ be the set of distinct persons to which tuples in qi refer to. For each possible distinct $qi \in S_{out}$, $|DP(qi)| \geq k$.

A relevant property of k_s-anonymized data is their *freshness*. This can be considered as the maximum allowed time of a tuple staying in the memory before it is output, which can be formally defined as follows.

Definition 2 (Delay constraint): Let A be a *k*-anonymization scheme that takes as input a data stream S_{in} and generates in output a data stream S_{out}, and let δ be a positive integer. We say that A satisfies the *delay constraint* δ if and only if for each new tuple $t \in S_{in}$ with position $t.p$, all tuples with position less than $t.p - \delta$ have already been output by A.

By the definition, when a new tuple t arrives, the tuple t' with position $t'.p = t.p - \delta$ can still stay in the memory. However, when the next new tuple with position $t.p + 1$ comes, t' should already have been output. Therefore, once t has arrived, t' is *expiring* and need to be output. In the following, we say that a tuple *is going to expire* if it must be output before the next new tuple arrives. Note that the δ parameter can be tuned on the basis of the application domain and its temporal requirements, making it possible to trade off between the allowed delay and the obtained information quality.

B. Information loss metrics

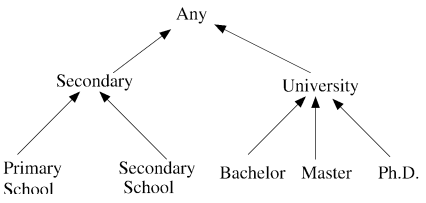


Fig. 1. Domain generalization hierarchy of Edu

As we are dealing with streaming data, we need an information loss metric that can be calculated in an incremental manner. We adapt the general information loss metric presented by Iyengar [17] to streaming data. Let *QI*=$\{q_1, \ldots, q_n\}$ be the set of quasi-identifier attributes. Consider first a categorical attribute q_i and let DGH_i be the domain generalization hierarchy for q_i. Given a node *v* in DGH_i, the information loss of *v* is defined as follows:

$$VInfoLoss(v) = \frac{|S_v|-1}{|S|-1}$$

where S_v is the set of leaf nodes of the subtree rooted at *v* in DGH_i and *S* is the set of all the leaf nodes in DGH_i. Intuitively the information loss of a leaf node is 0 according to the formula above. For example, considering the domain generalization hierarchy in Figure 1, the information loss of value "University" is 2/4.

In contrast, given a continuous attribute a and an interval $I = [l, u]$ from the domain $[L, U]$ of a, used to generalize a's value, the information loss associated with *I* is defined as follows:

$$VInfoLoss(I) = \frac{u-l}{U-L}$$

Hence, we define the information loss of a tuple generalization $g = (v_1, ..., v_n)$ as follows:

$$InfoLoss(g) = \frac{1}{n} \sum_{i=1}^{n} VInfoLoss(v_i)$$

Example 1: Let us consider the stream Customer(*p*,*pid*, $a_1,\ldots,$ a_j, Age, Edu), where Age and Edu are the quasi-identifier attributes. With respect to the domain generalization hierarchy in Figure 1 and assuming the interval [0,100] as the domain of the Age attribute, the tuple generalization g=$([1, 40], University)$ has the following information loss: $InfoLoss(g)$ = 1/2*(2/4 + 39/100)=0.445.

Given a data stream S k_s-anonymized up to position P, we can define the average information loss of S up to P as follows:

$$AvgLoss(S, P) = \frac{1}{P} \sum_{t_i \in S, t_i.p \leq P} InfoLoss(t_i)$$

III. THE CASTLE FRAMEWORK

In the following section, before presenting and discussing CASTLE in more details, we introduce the notion of k_s-anonymized cluster.

A. k_s-anonymized clusters

The basic idea of the proposed approach is to exploit quasi-identifier attributes to define a metric space, such that tuples can be considered points in this space. According to this strategy, clusters can be defined as n-dimensional intervals, where n is the number of quasi-identifier attributes. The formal definition of cluster is given below.

Definition 3 (Cluster over a data stream): Let $S(p, pid, a_1,\ldots, a_j, q_1, \ldots, q_n)$ be a stream where QI=$\{q_1, q_2, \ldots, q_n\}$ are the quasi-identifier attributes. Let S' be a set of tuples in S. A cluster C over S' is defined as a set of intervals, called *range intervals*, in the quasi-identifier attribute domains. More precisely, for each $q_i \in$ QI, the corresponding range interval r_i is defined as follows:

- if q_i is a continuous attribute, r_i is the minimal sub-interval of q_i's domain that contains all q_i's values of tuples in S';
- if q_i is a categorical attribute, let $Leaves(DGH_i)$ be the set of leaves of DGH_i generated by a left-most traversal of all the leaves in DGH_i. Let L_{q_i} be the smallest subset of $Leaves(DGH_i)$ containing all values of q_i of tuples in S', r_i's bounds are the left-most and the right-most values of L_{q_i}, respectively.

Given a cluster C we denote with $C.size$ the number of distinct persons to which the tuples in C refer to. This can be easily calculated by considering the number of distinct values of the pid attribute. Moreover, we denote with $C.r_i$ the i-th range interval of C, and with $C(r_1,\ldots,r_n)$ the cluster together with its range intervals.

Example 2: Consider the stream Customer(p, pid, $a_1, \ldots, a_j$, Age, Edu), where Age and Edu are the quasi-identifier attributes, and the DGH_{Edu} is presented in Figure 1. Let us consider the following tuples:[1] (pid_1,25, Bachelor), (pid_2,26,Master), and (pid_3,30,Ph.D.). According to Definition 3, cluster C defined over these three tuples has [25,30] as Age range interval and [Bachelor, Ph.D.] as Edu range interval. Therefore, it is denoted as C([25,30], [Bachelor,Ph.D]) with size equal to 3. If we further add the tuple (pid_1,25,Bachelor) to C, it does not change its range intervals nor its size (which is still equal to 3).

Once a cluster C reaches the size of at least *k*, it implies that there exists at least *k* distinct individuals to which tuples in C refer to. Moreover, since all tuples are contained in the same cluster, each quasi-identifier attribute of all of them is enclosed into the same range interval. Intuitively, if all these tuples are given in output by generalizing the value of each quasi-identifier in the same way, we satisfy properties of Definition 1. According to our approach, the way by which quasi-identifiers attributes are given in output is determined on the basis of cluster range intervals and formally defined as follows.

Definition 4 (Cluster generalization): Let $C(r_1,\ldots,r_n)$ be a cluster. The corresponding cluster generalization (or simply generalization), denoted as G=($g_1,\ldots,g_n$), is defined such that, for each $r_i, i \in [1, n]$, g_i is computed as follows:

- if r_i is defined on a continuous attribute q_i, $g_i = r_i$;

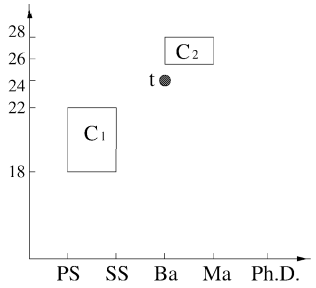


Fig. 2. Cluster selection

- if r_i is defined on a categorical attribute q_i, g_i is set equal to the lowest common ancestor wrt DGH_i of the bounds of r_i.

Moreover, we say that a tuple t is output with C's generalization, if each quasi-identifier attribute $q_i, i \in [1, n]$, of t is replaced by the corresponding value in the tuple generalization associated with C.

Example 3: Consider the cluster of Example 2, that is, C([25,30], [Bachelor,Ph.D]). According to the domain generalization hierarchy presented in Figure 1, the generalization associated with C is ([25,30],University), since University is the lowest common ancestor of Bachelor and Ph.D.

The information loss of a cluster C, denoted in what follows as *InfoLoss*(C), is the information loss of the corresponding generalization (see Section II-B). We can now state when a cluster is k_s-anonymized.

Definition 5 (k_s-anonymized cluster): Let $C(r_1,\ldots,r_n)$ be a cluster, and ($g_1,\ldots,g_n$) be the corresponding tuple generalization. If at a given time instant t_j, $C.size$ is greater than or equal to *k* and all tuples in C are output with C's generalization ($g_1,\ldots,g_n$), we say that, starting from j, C is a k_s-anonymized cluster.

Example 4: Consider the stream Customer(p, pid, $a_1, \ldots, a_j$, Age, Edu) and the cluster of Example 2, that is, C([25,30], [Bachelor,Ph.D]) with size equal to 3. Let assume $k \leq 3$. If, at a given instant t_j, all tuples in C are output with its generalization,[2] C will start to be k_s-anonymized.

B. CASTLE

In this section, we present CASTLE. We start by giving a general overview of the underlying approach and by showing how it adapts to the data stream distribution. Detailed algorithms are presented in Section IV.

1) Scheme overview: Initially, no clusters are in memory. When CASTLE receives the first tuple, it generates a cluster over it. Then, for every newly arriving tuple t, CASTLE selects, among all the existing clusters, the one to which t can be assigned, that is, the one whose range intervals enclose t's attribute values. However, it could be the case that no cluster can contain the new tuple, that is, the values of quasi-identifier attributes of t are not contained into the range intervals of any cluster.

[1]For simplicity, here and in the following we only consider the pid and quasi-identifier attributes. Moreover, where not relevant, pid attribute is omitted.

[2]We will see in the following section that the event that triggers the output of all tuples in a cluster with size at least *k* is the expiration of one of the tuples the cluster contains.

Example 5: Consider clusters C_1 and C_2 in Figure 2. Suppose that a new tuple t = (24,Bachelor) arrives. As graphically depicted in Figure 2, tuple t cannot be enclosed neither in C_1([18,22],[Primary School,Secondary School]) nor in C_2([26,28], [Bachelor,Master]).

When a new tuple can not be assigned to any existing cluster, there is the need to enlarge one of them in order to accommodate the new tuple. Cluster enlargement implies the enlargement of its range intervals and, as a consequence, an increase of the information loss. To minimize information loss, when selecting the cluster where a new tuple is pushed, CASTLE chooses the one that requires the smallest enlargement. Cluster enlargement is formally defined as follows.

Definition 6 (Cluster enlargement): Let S(p, pid, a_1,..., a_j, q_1, ..., q_n) be a stream where QI=$\{q_1,\ldots,q_n\}$ are the quasi-identifier attributes. Let $C(r_1, r_2, ..., r_n)$ be a cluster defined over tuples in $S' \subset S$, and let $(g_1, g_2, ..., g_n)$ be the associated generalization. Let t be a tuple in $S \setminus S'$. The enlargement of C w.r.t. t is defined as follows:
$Enlargement(C, t) = \sum_{i=1}^{n}(InfoLoss(\tilde{g}_i) - InfoLoss(g_i))$ [3]
where $(\tilde{g}_1, \tilde{g}_2, ..., \tilde{g}_n)$ is the tuple generalization associated with C calculated with the new range intervals $(\tilde{r}_1, \tilde{r}_2, ..., \tilde{r}_n)$, computed such that C contains tuples in $S' \cup \{t\}$.

Example 6: Consider clusters C_1 and C_2 in Figure 2. To enclose tuple t into cluster C_1 its range intervals should be enlarged to [18,24] and [Primary School,Bachelor], respectively, which implies that the new generalization associated with C_1 is ([18,24],Any). Thus, the enlargement of C_1 due to absorbing t is $Enlargement(C_1, t)$= $1/2*(4/4+6/100)-1/2*(1/4+4/100) = 0.509$. In contrast, the range intervals of C_2, enlarged to enclose t, are [24,28] and [Bachelor,Master], respectively, which correspond to the new generalization associated with C_2: ([24,28], University). This implies that the enlargement of C_2 due to absorbing t is $Enlargement(C_2, t)$=$1/2*(2/4+6/100)-1/2*(2/4+2/100) = 0.02$. Thus, t is pushed into C_2.

Therefore, a tuple is pushed into the cluster which requires the smallest enlargement to include the tuple. However, to prevent clusters from becoming too big, which implies generalizations with poor information quality, if pushing a new tuple to any existing cluster makes the information loss of the cluster greater than a predefined threshold τ, CASTLE generates a new cluster over the new tuple (see Section III-B.3 for more details).

To satisfy delay constraints, when a new tuple arrives, CASTLE checks whether a tuple in some clusters is going to expire. In such a case, the corresponding anonymized tuple must be immediately output. Here, there are two main cases. The first is when cluster C, hosting the expiring tuple, has already a size greater than or equal to *k*. In this case CASTLE simply outputs all the tuples in C with its generalization, and starting from the instant of outputting, it considers C as a k_s-anonymized cluster. The second case is when the cluster C hosting the expiring tuple has size less than *k*. To immediately output the expiring tuple, CASTLE performs some merge operations. In particular, CASTLE merges C with some of its neighboring clusters such that the size of the resultant cluster is greater than or equal to *k*. More precisely, among all the neighboring clusters, CASTLE selects those that result in minimum enlargement to C. Then, all the tuples contained in the new cluster can be output with its generalization (see Section IV for more details on the merge operation).

In both the cases, before outputting the cluster's tuples, CASTLE verifies whether the cluster can be split into smaller sub-clusters. Indeed, according to the adopted information loss metric, the smaller the cluster is (i.e., its range intervals), the smaller its information loss will be. Therefore, if C's size is at least *2k*, before outputting the tuples, CASTLE splits it into two or more sub-clusters, each with size at least k (see Section IV for more details on the split operation).

2) Reuse of k_s-anonymized clusters: To increase the quality of anonymized data, we have enhanced CASTLE with a strategy that enables to *reuse* k_s-anonymized clusters (i.e., their generalization). According to this approach, after a cluster becomes k_s-anonymized, it is not deleted from memory rather its generalization is kept. In such a way, the generalization can be used later on to output expiring tuples contained in the corresponding k_s-anonymized cluster.

It is relevant to note, however, that a k_s-anonymized cluster can not be enlarged to absorb new tuples. Indeed, if one of its range is enlarged, the generalization changes and, as a consequence, it can no more be considered k_s-anonymized. The following example better clarifies this point.

Example 7: Consider the stream Customer(p,a_1,..., a_j, Age, Edu) and assume that at a given instant t_i cluster C_2([26,28], [Bachelor,Master]) becomes k_s-anonymized. This means that in S_{out} there exist at least *k* tuples[4] given in output with its generalization. Suppose now that C_2 absorbs tuple t = (24,Bachelor). As shown in Example 6, C_2's new range intervals are [24,28] and [Bachelor,Master]. Let us assume that we still consider C_2 k_s-anonymized, and, suppose, that it does not absorb any more tuple. When CASTLE acquires the tuple having position $p_{i+\delta-1}$, t is going to expire. Since C_2 is considered k_s-anonymized, CASTLE outputs t with C_2's new generalization, that is, ([24,28], University). However, this breaks the k_s-anonymity property (cfr. Definition 1), in that t is the only tuple output so far with generalization ([24,28], University).

To avoid enlargements of k_s-anonymized clusters, when a new tuple arrives, CASTLE selects the cluster where pushing it by choosing only among the set of *non-k_s*-anonymized clusters. Additionally, even if a k_s-anonymized cluster can absorb a new tuple without changing its range intervals, pushing a new tuple into a *non-k_s*-anonymized cluster, instead of an already k_s-anonymized cluster, gives to the first more possibilities to become k_s-anonymized. It is relevant to note that this strategy leads to overlaps between a k_s-anonymized and a *non-k_s*-anonymized clusters. Indeed, it could be the case that, when a new tuple t arrives, a new cluster C is created over it, even if t can be enclosed, without enlargement, by a k_s-anonymized cluster KC. This implies that between C and KC there exists an overlap. The main advantage of overlaps among k_s-anonymized and *non-k_s*-anonymized clusters is that the generalizations of k_s-anonymized clusters can be used for anonymizing expiring tuples that are enclosed in a *non-k_s*-anonymized cluster. Therefore, every time a tuple t inside a *non-k_s*-anonymized cluster is going to expire, CASTLE verifies whether t also falls in a k_s-anonymized

[3] In the case C contains only one tuple $\bar{t}$, we say that $Enlargement(C, t)$ returns the distance between $\bar{t}$ and t.

[4] More precisely, according to the definition of k_s-anonymity, in S_{out} there exist at least *k* tuples, all referring to distinct persons.

cluster KC. If this happens, t is immediately output with KC's generalization. Thus, the reuse strategy avoids some clusters merges and therefore improves information quality.

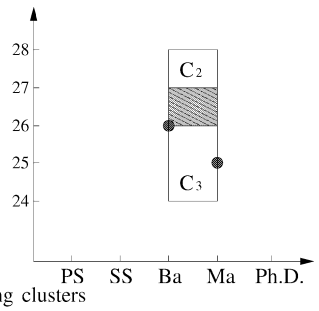


Fig. 3. Overlapping clusters

Example 8: Let us consider the clusters in Figure 3, and suppose that, at a given instant t_i, cluster C_2([26,28], [Bachelor,Master]) becomes k_s-anonymized. Moreover, suppose that, after some time, a new cluster C_3([24,27], [Bachelor,Master]) is generated (see Figure 3). In case a tuple t = (26,Bachelor) of C_3 is going to expire and C_3 still has size less than k, t can be given in output with C_2's generalization, that is, ([26,28], University).

However, cluster overlapping, if not carefully managed, can be exploited by a potential attacker to infer additional knowledge about a tuple value or even guess its exact value, by analyzing clusters' intersections, as the following example shows.

Example 9: With reference to Figure 3, suppose that at a given instant t_j both clusters C_2([26,28], [Bachelor, Master]) and C_3([24,27], [Bachelor,Master]) are k_s-anonymized. Suppose moreover that a tuple $\tilde{t}$ = (25,Master) arrives after instant t_j and that, after some time, it is going to expire. CASTLE outputs this tuple with C_3's generalization, i.e., ([24,27], University). However, by tracing the output stream, an attacker is able to infer that $\tilde{t}.Age$ belongs to $[24, 25]$, instead of $[24, 27]$. Indeed, an attacker knows the existence of C_2, by tracing the anonymized output stream. Moreover, s/he knows that $\tilde{t}.Age$ is not in $[26, 27]$, otherwise it would have been given in output with C_2's generalization, since this generalization preserves more information wrt C_3 generalization. Thus, an attacker can infer that $\tilde{t}.Age$ belongs to $[24, 25]$.

To overcome this problem, a first naive solution is to avoid overlaps among clusters. However, this solution has the drawback of diminishing the possibility of anonymizing expiring tuples by reusing the generalization of k_s-anonymized clusters. Therefore, we adopt a different solution: if an expiring tuple falls into the overlap of two or more k_s-anonymized clusters, CASTLE randomly selects one k_s-anonymized cluster and anonymizes the tuple with its generalization. This avoids the security flaws previously discussed (see Section IV-C for a formal proof).

Example 10: Consider again Example 9 and assume that the strategy discussed above is adopted. When a potential attacker sees the generalization of tuple $\tilde{t}$, he/she cannot infer any additional knowledge about $\tilde{t}.Age$. Indeed, he/she may infer that $\tilde{t}.Age$ could be 24, 25 or any value in the intersection between C_2 and C_3 Age range intervals, that is, 26 and 27. Thus, what he/she can infer is that the value of the Age attribute is in the range [24,27], which is exactly the generalization by which $\tilde{t}$ has been output, i.e., C_3's generalization.

3) Adaptability to data stream distribution: In order to adapt CASTLE to data stream distribution we enhanced it with several strategies.

The first is related to the value of τ. We recall that CASTLE exploits a threshold τ to decide whether pushing a new tuple into any existing cluster or generating a new cluster over the new tuple. In order to adapt to the data stream distribution, we do not consider a predefined and fixed τ. Instead, τ is set to the average information loss of the most recent k_s-anonymized clusters (see Section V for a discussion on how setting the number of μ most recent clusters). Let us see the benefits of this adaptive τ. When a data stream contains well clustered tuples, it is possible to generate over them small clusters with small information loss. Therefore, τ will assume a small value. As a consequence, a new tuple is pushed into a cluster only if it is very close to it. This ensures that if tuples in a stream are well clustered, only clusters with small range intervals are formed. On the contrary, if a data stream contains sparsely distributed tuples, clusters with big range intervals are generated. This implies that τ will have a high value. The benefit of this is that the chance of a new tuple to be pushed into an existing cluster is increased, since the enlargement allowed by τ is increased too. As a consequence, each cluster has increased possibility of reaching the size of k and the number of cluster merging operations is reduced.

Another adaptivity is obtained by constraining the maximum number of *non-*k_s-anonymized clusters that can be in memory. This number is limited by a given parameter β. This parameter co-operates with τ to handle the varying of data distribution. More precisely, suppose that, at the beginning, all tuples are well clustered, so τ has a small value. If the newly arriving tuples are sparely distributed, many small-size clusters will be formed since the small τ only allows a small cluster enlargement. The large number of clusters increases the overhead of searching for the best cluster into which a new tuple should be pushed, and it also increases the overhead of merging clusters when tuples expire. For this reason, we adopt β to prevent the generation of a possibly large number of clusters and thus limit the overhead. β can be set by taking into account the available computational and storage resources. Therefore, a new cluster is generated only if the number of *non-*k_s-anonymized clusters is fewer than β. Otherwise, a tuple is pushed to an existing cluster, selected among the ones which require the minimum enlargement to enclose the tuple.

4) Management of cluster outliers: In order to improve data quality of anonymized tuples, CASTLE manages *outliers*, that is, tuples that are faraway from other tuples. Indeed, the presence of few outlier tuples can drastically increases the information loss of the k_s-anonymized stream.

According to the clustering approach presented in Section III-B.1, accommodating an outlier tuple into existing clusters could require big enlargements of their ranges, which obviously implies an information loss increase. However, these big enlargements are not always performed due to the τ threshold (indeed, as pointed out in the previous section, well-clustered tuples imply small τ value). In this case, the outlier tuple is managed by creating over it a new cluster. It is reasonable that this cluster will contain only few tuples (i.e., few other outliers), otherwise, the tuple is not an outlier. A cluster with only few tuples implies merge operations when one of them is going to expire. Thus, in both the cases, that is, in the case: (a) the outlier is absorbed

by existing clusters; or (b) it is enclosed into a new one, the information loss increases. To overcome this drawback, CASTLE handles outliers by suppressing them. Suppressing a tuple means that the tuple is anonymized with the most generalized values, whose information loss is obviously 1. However, if the tuple is not suppressed, its expiration will require to anonymize tuples with the generalizations of enlarged clusters, in case (a) or merge operations in case (b). In both the case, this implies that at least *k* tuples are anonymized according to a possibly big cluster. The overall information loss of the anonymized stream is more impacted by the information loss of the *k* tuples anonymized by a big cluster, rather than by a single (or few) tuple whose information loss is 1.

More precisely, CASTLE suppresses expiring tuples contained into clusters considered outlier. To judge a cluster as outlier, we consider its size. Intuitively, the longer the cluster exists, the bigger its size should be. Thus, only clusters with *'small'* size should be considered as outliers. However, based on the data stream distribution and the τ and β values, it is possible that all clusters are small in size at some moment. Therefore, to judge a cluster enough small to be an outlier, we compare its size with the sizes of all other clusters. If the cluster size is smaller than the majority of the other clusters, it is judged an outlier (see Section IV for more details).

IV. CASTLE ALGORITHMS AND ANALYSIS

In the following, we present the CASTLE algorithms, implementing the techniques illustrated in the previous section. Then, we illustrate their extension to achieve l-diversity. Moreover, we analyze their security and complexity.

A. Algorithms

The main algorithm is Algorithm 1, which continuously processes the incoming data stream by producing in output a flow of k_s-anonymized tuples.

Algorithm 1: *CASTLE*(S, k, δ, β)

```
1  Let Γ be the set of non-k_s-anonymized clusters, initialized to be empty;
2  Let Ω be the set of k_s-anonymized clusters, initialized to be empty;
3  Let τ be initialized to 0;
4  while S is non-empty do
5      Let t be the next tuple from S;
6      Let C be the cluster returned by best_selection(t);
7      if C = NULL then
8          Create a new cluster on t and insert it into Γ;
9      else
10         Push t to C;
11     Let t' be the tuple with position equal to t.p − δ;
12     if t' has not yet been output then
13         delay_constraint(t');
```

The algorithm takes as input the stream S to be anonymized, and the parameters: k, δ and β. At the beginning, the set of *non-*k_s-anonymized clusters (i.e., Γ) as well as the set of k_s-anonymized clusters (i.e., Ω) are empty. Then, every time a tuple t arrives (step 5), Algorithm 1 calls function $best_selection()$ (step 6), to select from Γ the best cluster into which t is pushed (step 10). If such a cluster does not exist, Algorithm 1 creates a new cluster for t (steps 7-8). Then, Algorithm 1 verifies whether the arrival of the new tuple t forces a tuple t' with position $t.p - \delta$ to expire (step 11). If this is the case, Algorithm 1 calls procedure $delay_constraint()$ (step 13). In the following, we illustrate the procedures/functions used by Algorithm 1.

Function `best_selection(t)`

```
1  Let E be a set initialized empty;
2  foreach C_j ∈ Γ do
3      Let e be Enlargement(C_j, t);
4      Insert e into E;
5  Let min be the minimum element in E;
6  Let SetC_min be the set of clusters C̃ in Γ with Enlargement(C̃, t)
   equals to min;
7  foreach C_j ∈ SetC_min do
8      Let ĪL_{C_j} be the information loss of C_j after pushing t into it;
9      if ĪL_{C_j} ≤ τ then
10         Insert C_j into SetC_ok;
11 if SetC_ok is empty then
12     if |Γ| ≥ β then
13         Return any cluster in SetC_min with minimum size;
14     else
15         Return NULL;
16 else
17     Return any cluster in SetC_ok with minimum size;
```

Function $best_selection()$

We recall that according to the reuse strategy, a new tuple is always pushed into a *non-*k_s-anonymized cluster, by selecting one among those which require the minimum enlargement. Thus, to find out the best *non-*k_s-anonymized cluster where a new tuple t should be inserted, $best_selection()$ calculates the enlargement implied by the insertion of t in each cluster in Γ (steps 1-4). Then, it selects from Γ only the clusters requiring the minimum enlargement (steps 5-6). According to the adaptability strategy described in Section III-B.3, $best_selection()$ selects from the returned clusters only those whose information loss is less than or equal to τ (steps 7-10). Among those, $best_selection()$ chooses the cluster with the minimum size (step 17). By contrast, if no clusters have information loss less or equal to τ, it implies that a new cluster should be created over t. However, it is necessary to verify whether the constraint on the maximum number of *non-*k_s-anonymized clusters is satisfied. Thus, there could be two different cases. The first case is when the number of *non-*k_s-anonymized clusters is greater than or equal to β (step 12). In such a case, it is not possible to create a new cluster, thus among clusters requiring the minimum enlargement, i.e., $SetC_{min}$, it is returned one among those with the minimum size (step 13). Otherwise, the function returns a NULL value (step 15), which triggers in Algorithm 1 the generation of a new cluster over t.

Procedure $delay_constraint()$

When a tuple t is expiring, Algorithm 1 calls procedure $delay_constraint()$, whose main goal is to give in output t. According to the proposed approach, this can be achieved in several ways. First, $delay_constraint()$ verifies whether the expiring tuple can be output with the generalization of the *non-*k_s-anonymized hosting cluster (steps 1-3). Obviously, since the hosting cluster is *non-*k_s-anonymized,[5] the procedure first verifies whether the cluster hosting the tuple has size greater than or equal to k. In this case, $delay_constraint()$ calls procedure $output_cluster()$ (step 3). Otherwise, $delay_constraint()$ verifies whether it is possible to apply on the expiring tuple the reuse strategy illustrated in Section III-B.1 (steps 5-9). The reuse

[5]We recall that tuples are only pushed into *non-*k_s-anonymized clusters.

Procedure `delay_constraint(t)`

1 Let C be the *non-k_s*-anonymized cluster to which t belongs to;
2 **if** $C.size \geq k$ **then**
3 output_cluster(C);
4 **else**
5 Let KC_{set} be the k_s-anonymized clusters in Ω containing t;
6 **if** KC_{set} *is not empty* **then**
7 Let $\overline{KC}$ be a cluster randomly selected from KC_{set};
8 Output t with $\overline{KC}$ generalization;
9 Return;
10 Let m be an integer set to 0;
11 **foreach** $C_j \in \Gamma$ **do**
12 **if** $C.size < C_j.size$ **then**
13 $m = m + 1$;
14 **if** $m > \frac{|\Gamma|}{2}$ **then**
15 Suppress tuple t;
16 Return;
17 **if** $\sum_{i=1}^{|\Gamma|} C_i.size < k$ **then**
18 Suppress t;
19 Return;
20 MC =$merge_clusters(C, \Gamma \setminus C)$;
21 output_cluster(MC);

strategy can be applied if there exist one or more k_s-anonymized clusters, i.e., KC_{set}, containing the expiring tuple. If this the case, to overcome the inference problem described in Section III-B.2, the $delay_constraint()$ procedure randomly selects a cluster in KC_{set} and gives in output the tuple with its generalization.

If the reuse strategy cannot be applied, i.e., KC_{set} is empty, the $delay_constraint()$ procedure verifies whether t is in contained into an outlier cluster, so to suppress it (steps 10-16). As introduced in Section III-B.4, to judge a cluster as an outlier, CASTLE verifies whether the cluster's size is smaller than the one of the majority of the other clusters. More precisely, the $delay_constraint()$ procedure scans all *non-k_s*-anonymized clusters, by checking their sizes. If the cluster's size is less than the one of $\frac{|\Gamma|}{2}$ existing clusters, the cluster is regarded as an outlier. In this case, CASTLE suppresses the tuple t, that is, it gives t in output with the most generalized values.

As the last alternative, procedure $delay_constraint()$ verifies whether a merge among C and some of the other *non-k_s*-anonymized clusters is possible. Notice that, if the total size of all clusters in Γ is fewer than k (step 17), a merge operation would not generate a cluster with size at least k. Therefore, the only way to output the expiring tuple is suppressing it (step 18). Otherwise, the merge can take place (step 20), and after the merge, the merged cluster is output (step 21).

The $merge_clusters()$ function[6] receives as input the cluster to be merged, i.e., C, and the set of *non-k_s*-anonymized clusters excluding C itself. The procedure, for every *non-k_s*-anonymized cluster C_i, calculates the enlargement of C due to the possible merge with C_i. Then, it selects the cluster, which brings the minimum enlargement to C, and merges C with it. This process continues until C's size is at least *k*. Then, the resulting cluster is given in output.

Procedure $output_cluster()$

When outputting a cluster C, in order to minimize information loss, procedure $output_cluster()$ verifies if C can be split, i.e., whether its size is at least $2k$ (step 1). If this is the case, it calls function $split()$ that splits C into sub-clusters, each with size at least k. Then, all tuples of the new created clusters (or of C, respectively) are given in output with the generalization of the corresponding cluster (step 6). Moreover, τ is updated to be the average information loss of the most recent k_s-anonymized clusters including the new ones (step 7). However, $output_cluster()$ does not store all these new clusters into Ω, the set of k_s-anonymized clusters. Indeed, to minimize information loss, only clusters with good information quality are reused. For this reason, $output_cluster()$ inserts a new k_s-anonymized cluster C_i into Ω only if its information loss is less than τ (steps 8-11). In addition, C_i is deleted from Γ (step 12).

[6]We do not report here the code of this procedure due to its simplicity.

Procedure `output_cluster(C)`

1 **if** $C.size \geq 2k$ **then**
2 Let SC be the set of clusters returned by $split(C)$;
3 **else**
4 SC := $\{C\}$;
5 **foreach** $C_i \in SC$ **do**
6 Output all tuples in C_i with its generalization;
7 Update τ according to *InfoLoss*(C_i);
8 **if** InfoLoss(C_i) $< \tau$ **then**
9 Insert C_i into Ω;
10 **else**
11 delete C_i;
12 delete C_i from Γ;

Function $split()$

Function `split(C)`

1 Initialize SC to be empty;
2 Let BS be the set of buckets created by grouping tuples in C by pid attribute;
3 **while** $|BS| \geq k$ **do**
4 Randomly select a bucket $\bar{B}$ from BS, and pick one of its tuples $\bar{t}$;
5 Create a new subcluster C_{new} over $\bar{t}$;
6 **if** B *is empty* **then**
7 delete $\bar{B}$;
8 Let H_{k-1} be a heap with $k-1$ nodes, each with an infinite distance to $\bar{t}$;
9 **foreach** *bucket in* $BS \setminus \bar{B}$ **do**
10 Pick one of its tuples t, and calculate t's distance to $\bar{t}$;
11 **if** t *is closer to* $\bar{t}$ *than the root of* H_{k-1} **then**
12 t replaces the root, and H_{k-1} is adjusted accordingly;
13 **foreach** *node in* H_{k-1} **do**
14 Let $\tilde{t}$ be the tuple in the node;
15 Insert $\tilde{t}$ into C_{new};
16 Let B_j be the bucket containing $\tilde{t}$;
17 Delete $\tilde{t}$ from B_j;
18 **if** B_j *is empty* **then**
19 delete B_j;
20 Add C_{new} to SC;
21 **foreach** $B_i \in BS$ **do**
22 pick a tuple t_i in B_i;
23 find t_i's nearest cluster in SC, and add all the tuples in B_i to it;
24 delete B_i;
25 return SC;

The splitting technique exploited in CASTLE is based on the KNN algorithm proposed [4], which we adapt to the definition of k_s-anonymity. In general, the *split()* function randomly selects a tuple t from a cluster C and creates over it a new cluster C_{new}. Then, it populates the new cluster C_{new} with other tuples of C, obviously t excluded, until C_{new} has size equal to k. This process of creating and populating a subcluster is repeated until C's size is less than *k*. Then, the remaining tuples of C are accommodated into one of the new generated clusters.

The definition of k_s-anonymity (cfr. Definition 1) requires that, for each possible QI group, there exist at least k distinct individuals referring to its tuples. To be compliant with this definition, clusters generated by function *split()* must contain tuples referring to at least k distinct individuals. Therefore, in selecting a tuple of C to be pushed into C_{new}, the $split()$ function considers only those tuples having pid different from $t.pid$. Among these, it selects the ones that require minimum C_{new}'s enlargement, so that the information loss is kept low.

Let us see in more details how $split()$ works. First, the function groups all tuples in C by their pid values, creating so a set of buckets BS (step 2), each one containing only tuples with the same value for attribute pid. Then, it randomly selects a tuple $\bar{t}$, i.e., it randomly selects a bucket $\bar{B}$ and picks one of its tuples (step 4) and creates a new cluster over it (step 5). To choose the k-1 tuples nearest to $\bar{t}$ to be pushed into the new generated subcluster, $split()$ sorts the buckets according to their distance to $\bar{t}$.[7] To speed-up this step, we exploit a heap structure of k-1 nodes, where each one contains a tuple belonging to a different bucket in BS. The heap is generated in lines 8-12, by updating its nodes based on their distances to $\bar{t}$ (step 7).

Then, $split()$ creates the new subcluster C_{new} by inserting into it the tuples stored into the heap nodes (steps 13-15). This ensures that the new generated cluster C_{new} contains k tuples of k different buckets, that is, k tuples referring to k distinct individuals, i.e., and this satisfies the k_s-anonymity definition. The above steps are repeated until the number of buckets are greater than or equal to k, which implies that there exist at least k further tuples referring to k distinct individuals, thus, another subcluster can be generated (step 3). Finally, each of the remaining tuples in buckets in BS is pushed into one of the new generated sub-clusters (steps 21-23). More precisely, a tuple is pushed into the cluster that requires the minimum enlargement to enclose it.

B. *l*-diversity

Recently, Machanavajjhala et al. [22] proposed the *l*-diversity principle to avoid possible inference attacks on k-anonymized data. Ensuring *l*-diversity requires that all tuples with the same generalization, i.e., all tuples belonging to the same QI group, have at least l distinct values for the sensitive attribute. In this section we illustrate how our scheme for k_s-anonymizing streaming data can be easily extended for k_s-anonymizing and *l*-diversifying streaming data. We recall that CASTLE outputs a tuple with the generalization associated with one of the k_s-anonymized cluster it belongs to. Thus, in order to ensure that tuples with the same generalization, i.e. tuples belonging to the same cluster, have at least l distinct values for the sensitive attribute, we need to modify the definition of k_s-anonymized cluster, to include the *l*-diversity principle. In doing that, we consider a single sensitive attribute a_s.

Moreover, given a cluster C, we denote with $C.diversity$ the number of distinct values of a_s for tuples in C.

Definition 7 (k_s-anonymized and l-diversified cluster): Let $C(r_1,\ldots,r_n)$ be a cluster, and $(g_1,\ldots,g_n)$ be the corresponding generalization. If, at a given time instant t_j: *(1)* $C.size$ is greater than or equal to k, *(2)* $C.diversity$ is greater than or equal to l, and *(3)* all tuples in C are output with C's generalization $(g_1,\ldots,g_n)$, we say that, starting from t_j, C is a k_s-anonymized and l-diversified cluster.

In order to ensure that tuples are given in output only if they belong to a k_s-anonymized and l-diversified cluster, it is necessary to slightly modify CASTLE algorithms. We recall that in Algorithm 1 tuples are given in output only when procedure *delay_constraint()* is called. The procedure verifies whether one of the following cases is satisfied: (a) the expiring tuple belongs to a cluster that is ready to become anonymized; (b) the expiring tuple belongs to an outlier cluster; or (c) if no one of the previous conditions are satisfied, *delay_constraint()* verifies whether a merge is possible, that is, whether merging some of the existing clusters generates a new k_s-anonymized cluster. Thus, the new definition of k_s-anonymized and l-diversified cluster, requires to modify cases (a) and (c). In particular, the latter one simply entails to update the instruction that verifies whether a merge operation is applicable (see line 17 of *delay_constraint()*). Besides the condition that $\sum_{i=1}^{\Gamma} C_i.size < k$, it must also be checked that there exist at least l distinct values of a_s among all clusters in Γ.

In contrast, to properly handle case (a), it is necessary to replace the condition according to which a cluster C is judged ready to be anonymized (see line 1 of *delay_constraint()*). Indeed, to take into account also *l*-diversity, the conditions to be satisfied are now two: $C.size \geq k$ and $C.diversity \geq l$. It is relevant to notice that, when a cluster is ready to be anonymized, it is passed to the $output_cluster()$ procedure that, before giving in output all its tuples, verifies whether a split is possible, that is, whether $C.size \geq 2k$. If this is the case, it calls $split()$, whose aim is to generate k_s-anonymized sub-clusters. However, due to the new definition of anonymized cluster (cfr. Definition 7), the *split()* function needs to be redefined.

Indeed, the *split()* function in Section IV aims to generate sub-clusters with size at least equal to k. In contrast, the *l*-diversity principle requires to take in account during sub-clusters generation also the sensitive attribute a_s. That is, the subclusters returned by the $split()$ function must have size and diversity at least equal to k and l, respectively. To satisfy both these conditions, we have designed a different $split()$ function (see $split^l$ below) that, differently from the previous one, should be called only if the cluster C to be split has size and diversity at least equal to 2k and l, respectively (this implies to update line 1 of $output_cluster()$).

Function $split^l$

Let us first introduce the basic idea of the new split function, by assuming that each tuple in the data stream refers to a distinct individual, that is, no tuples referring to the same individual are present in the data stream. We illustrate, then, how we can relax this assumption. Given a cluster C, where $C.size \geq k$ and $C.diversity \geq l$, the first step to generate a k_s-anonymized and l-diversified subcluster is to group C's tuples into distinct buckets BS on the basis of the values of the sensitive attribute a_s. Then, a new subcluster C_{new} is generated by randomly selecting from each bucket a tuple to be inserted into it. This is repeated until C_{new} has absorbed k tuples, that is, until it satisfies the k_s-anonymity definition.[8] Note that since k is greater than or

[7] Note that all tuples in a bucket have the same distance to $\bar{t}$. Indeed, tuples with the same *pid* have the same position in the n-dimensional space, in that, tuples having the same *pid* refer to the same individual, thus they all have the same quasi-identifier attributes.

[8] Since we are assuming that each tuple in the data stream refers to a distinct individual, k_s-anonymity is equal to k-anonymity.

Function `split(C, $a_s$)`

1 Let BS be the set of disjoint buckets generated by $generate_buckets(C, a_s)$;
2 **if** $|BS| < L$ **then**
3 return $\{C\}$;
4 Let SC be a set of subclusters, initialized empty;
5 **while** $|BS| \geq L$ *and* $sum = \sum_{B_i \in BS} B_i.size \geq k$ **do**
6 Randomly select a $\bar{B}$ from BS;
7 Randomly select a tuple $\bar{t}$ from $\bar{B}$;
8 Generate a subcluster C_{sub} over t;
9 Delete $\bar{t}$ from $\bar{B}$;
10 **foreach** $B_j \in BS$ **do**
11 **foreach** *tuple* $t_i \in B_j$ **do**
12 Let e_i be $Enlargement(C_{new}, t_i)$;
13 Sort tuples of B_j by ascending order of their enlargement e_i;
14 Let T_j be the set of the first $k \times \frac{B_j.size}{sum}$ tuples in B_j;
15 Insert T_j into C_{new};
16 Delete T_j from B_j;
17 **if** $B_j.size = 0$ **then**
18 Delete B_j from BS;
19 Add C_{sub} to SC;
20 **foreach** $B_i \in BS$ **do**
21 **foreach** *tuple* $t_i \in B_i$ **do**
22 Let SC_n be t_i's nearest subcluster in SC;
23 Insert t_i into SC_n;
24 Delete B_i;
25 **foreach** *subcluster* $SC_i \in SC$ **do**
26 **foreach** *tuple* $t \in SC_i$ **do**
27 Let G_t be the set of tuples in C, such that $G_t = \{\bar{t} \in C | t.pid = \bar{t}.pid\}$;
28 Insert G_t into SC_i;
29 Delete G_t from C;
30 Return SC;

equal to l, each bucket in BS participates with at least a tuple in the new subcluster. This ensures that the new created subcluster contains l distinct values of the sensitive attribute, and thus it also satisfies the *l*-diversity principle. However, relaxing the condition that each tuple in the data set is associated with a different person requires to slightly modify the basic strategy. Indeed, in this case it may happen that two or more tuples selected from two distinct buckets have the same value for attribute pid. Thus, even if C_{new} absorbs k tuples from different buckets in BS it could be still not k_s-anonymized, in that its size could be still less than k.[9] To overcome this, we consider only buckets without overlaps wrt pid. This is done by selecting only one tuple from C for each distinct pid value, and grouping only these selected tuples into buckets according to the value of their sensitive attribute. Avoiding overlaps ensures that the k tuples selected from buckets in BS have different pids, thus the size of the new generated subcluster C_{new} is equal to k. It is important to note that selecting only a subset of tuples of C does not mean that the other tuples are not considered. By contrast, when all possible subclusters are generated, each tuple t previously not selected from C is inserted into one of the new subclusters. In particular, a tuple t is inserted into the unique subcluster that contains a tuple with the same pid of t.

Let us see now in more details how $split^l()$ works. As first step, it generates the buckets BS without overlaps. This is done by calling function *generate_buckets()*. Then, if the number of disjoint buckets is less than l, this means that the split is not possible. Thus, the input cluster C is returned (line 3).

[9]We recall that with cluster size we denote the number of tuples with distinct value of the pid attribute.

Otherwise, until the number of buckets is greater than l and the number of tuples into BS is greater than k, a k_s-anonymized and *l*-diversified subcluster C_{new} is generate (line 5). In this the case, the function selects, from each bucket B_j in BS a subset of tuples T_j proportional to the bucket size (line 9). More precisely, to reduce the information loss, the function selects the T_j tuples in B_j which requires the minimum enlargement to C_{new}. Such tuples are then inserted into C_{new}. Finally, when one or both the conditions of the while cycle are no more satisfied, remaining tuples are accommodated into one of the new created subclusters.

C. Formal results

In this section, we analyze the security and complexity of CASTLE.

1) CASTLE security analysis: We start by proving that CASTLE generates k_s-anonymized data streams.

Theorem 1: Let $S(p,pid, a_1,\ldots, a_j, q_1, \ldots, q_n)$ be a stream where QI=$\{q_1,\ldots,q_n\}$ are the quasi-identifier attributes. Let S_{out} be the stream generated by CASTLE, S_{out} is k_s-anonymized.

The formal proof of the above theorem is reported in Appendix.

Proving that CASTLE generates k_s-anonymized data streams is not enough. As shown by Example 9, overlaps in cluster generalizations can be exploited by a potential attacker to infer additional knowledge about a tuple value, or even guess its exact value. We have therefore to prove that the reuse strategy implemented by CASTLE avoids these security flaws. Before doing that, we introduce the adversary model we are considering.

We assume that the attacker knows the deployed generalization principle (that is, the algorithm according to which tuples are generalized by CASTLE). Additionally, he/she knows the victim and therefore he/she knows the values of his/her quasi-identifier attributes. Moreover, we assume that the attacker knows whether a tuple related to the victim is or is not in S_{out}, i.e., the k_s-anonymized stream generated by CASTLE. With respect to the scenario introduced in Section I, for instance, an attacker knows that the victim has bought something in the store, which implies that the stream S_{out} generated by CASTLE will contain an anonymized tuple referring to this transaction. In addition to this background knowledge, which is common to traditional k–anonymity schemes, we assume that the attacker has been keeping track of all the anonymized tuples generated by CASTLE, i.e., all tuples in S_{out}. More precisely, we assume an attacker knows all the generalizations $TG = \{G_1, \ldots, G_m\}$ according to which tuples in S_{out} have been given in output so far.

To better point out the security flaws that may arise due to cluster generalization overlaps, we need to introduce some notions.

In general, given a cluster generalization $G = (g_1, \ldots, g_n)$, the generalization g_j of the quasi identifier attribute q_j gives information about the range in which the real value of q_j for tuples generalized by G falls. For instance, given the domain generalization hierarchy presented in Figure 1, if the value given in output by CASTLE for the quasi-identifier attribute *Edu* is University, we can infer that the corresponding non-anonymized tuple can have as value for the *Edu* attribute one of the element in the set {Bachelor, Master, Ph.D.}. Similarly, if the value given in output by CASTLE for the quasi-identifier attribute *Age* is [25-30], then the *Age* attribute of the corresponding non-anonymized tuple has a value in the interval [25,30]. In what follows, given a cluster

generalization $G = (g_1, \ldots, g_n)$, we denote with $values(G.g_i)$ the set of values implied by the generalization g_i of attribute q_i. More precisely, if q_i is a continuous attribute, $values(G.g_i)$ contains all the values in the interval corresponding to $G.g_i$'s value. In contrast, if q_i is a categorical attribute, $values(G.g_i)$ contains all the leaves of the subtree in DGH_i rooted at $G.g_i$'s value, where DGH_i denotes the domain generalization hierarchy for attribute q_i.

The k_s-anonymity property ensures that, even if an attacker knows $values(G.g_i)$, $\forall i \in [1, n]$, he/she is not able to link the anonymized tuple to an individual with a probability greater than $\frac{1}{k}$. However, as reported in Example 9, if an attacker, by tracing the generalizations output so far, is able to shrink the possible real values associated with $G.g_i$ to a subset of $values(G.g_i)$, he/she could infer that the anonymized tuple refers to an individual with a probability greater than $\frac{1}{k}$, as the following example better clarifies.

Example 11: Let us consider again Example 9, and assume that an attacker has the following background knowledge:

- he/she knows that the anonymization method always try to put a tuple in the smallest available cluster, to minimize information loss.
- he/she knows the quasi identifier values of the victim, i.e., $vic.Age$=24 and $vic.Edu$=Bachelor;
- he/she knows all the cluster generalizations TG according to which tuples in S_{out} have been given in output so far. According to Example 9, $TG = \{G_1, G_2\}$, where G_1 = ([26,28],[University]) and G_2 = ([24,27],[University]);
- he/she sees in S_{out} a tuple t = ([24,27],[University], $sexbook$), whose generalization G_t = ([24,27],[University]) encloses the victim quasi-identifiers.

By analyzing the generalizations output so far, the attacker can conclude that if t has not been generalized by G_1, which is smaller and thus minimizes the information loss, it is due to the fact that at least a quasi identifier q_i of t does not belong to $values(G_1.q_i)$. In the example, the only quasi identifier for which this assumption might hold is Age because the generalization for Edu is the same in both G_1 and G_2. Thus, an attacker can reduce the possible real values of the Age attribute in t by calculating the values that are in G_t and not in G_1. In this example, these values are $t.Age$=[24,25]. Moreover, an attacker knows that no tuple t' with generalization Age = [24,25] and Edu = University has been given in output by CASTLE. This implies that an attacker can infer that tuple t refers to the victim with a probability greater than $\frac{1}{k}$. For instance, with reference to the Costumer Table I, the attacker can infer that t = ([24,27],[University], $sexbook$) is related to Beth.

To avoid this possible inference attack CASTLE adopts the cluster reuse strategy introduced in Section III-B.2. As the following theorem states, this strategy ensures that an attacker cannot infer any additional knowledge on a tuple t other than the set of real values associated with the generalization according to which it has been given in output.

Theorem 2: Let S(*p*,*pid*, a_1,..., a_j, q_1, ..., q_n) be a stream where QI=$\{q_1,\ldots,q_n\}$ are the quasi-identifier attributes. Let S_{out} be the stream generated by CASTLE, and $TG = \{G_1, \ldots, G_n\}$ be the generalizations corresponding to the tuples in S_{out}. Let t be a tuple and let G_t be its generalization performed by CASTLE. For each $q_i \in QI$ an attacker is not able to infer that the real value of $t.q_i$ belongs to a subset of $values(G_t.q_i)$.

The formal proof is reported in Appendix.

Finally, it is important to note that we are assuming a setting where the adversary does not have any temporal background knowledge (e.g., he/she may know that Bob buys an item but not that Bob buys an item at time 10). Considering this further kind of background knowledge is a challenging issue, that we plan to investigate in the future, and that may require the definition of alternative techniques to achieve k_s-anonymity.

Parameter	Description	Estimated values
k	value of k in k_s-anonymity	100∼1,000
D	number of quasi-identifier attributes	6∼10
δ	delay constraint	5,000∼30,000
β	maximum number of *non*-k_s-anonymized clusters	50
NC_{k_s}	maximum number of k_s-anonymized clusters	
C_d	time required to enlarge a single dimension of a cluster	
C_p	time required to replace the root of a heap	$\log^k$
S_g	space required to store in memory the generalization of a qi attribute	
S_t	space required to store in memory a tuple	

TABLE III
PARAMETERS USED IN THE COMPLEXITY ANALYSIS

2) CASTLE complexity: In order to analyze the efficiency of the proposed approach, we have estimated the time and space complexity of CASTLE algorithms. In doing that we make use of the set of parameters summarized in Table III. The third column of Table III shows the estimated value, if any, of the corresponding parameter. These are the values obtained by the tuning performed in our experiments (see Section V).

Time Complexity

Estimating the time complexity of CASTLE requires to analyze the main operations it carries out, that is: (1) selecting the best cluster where pushing a new tuple; (2) merging clusters to generate a new k_s-anonymized cluster; (3) splitting a cluster to generate new k_s-anonymized subclusters. This requires to analyze the time complexity of the functions implementing these tasks, namely, *best_selection()*, *merge_clusters()* and *split()*. Let us see in more details their time complexity.

best_selection(). We recall that to decide which cluster must absorb a new tuple t, *best_selection()* calculates for each *non*-k_s-anonymized cluster $C_j \in \Gamma$, the enlargement required to make C_j able to absorb t, that is, $Enlargement(C_j, t)$. According to Definition 6, the enlargement is computed by considering the difference of the information loss between the new cluster generalization, generated due the enlargement, and the old one. This is done, by considering each dimension of the cluster, i.e., each QI attribute, that has been enlarged. In the worst case, to absorb a new tuple, a cluster is enlarged over all its dimensions, that is, all the generalization of its QI attributes are enlarged. Thus, assuming that the cost of calculating the enlargement over a dimension is C_d, calculating $Enlargement(C_j, t)$ requires $C_d \cdot D$. Moreover, since by construction Γ contains at maximum β *non*-k_s-anonymized clusters, we can estimate the worst case time complexity of *best_selection()* as:

$$C^w_{best_selection}=C_d \cdot D \cdot \beta$$

merge_clusters(). Let C be the cluster received as input by *merge_clusters()*. We recall that to generate a new k_s-anonymized cluster by merging C with one or more *non-*k_s-anonymized clusters in Γ, the function scans all the clusters in Γ and calculates the enlargement of C due to the possible merge with it. Then, it selects among clusters in Γ only the one that brings the minimum enlargement to C. These steps are performed until C's size is at least *k*.

To have an upper bound the of the time complexity of this function, we need to consider the worst case. According to the adopted cluster merge technique this happens when C is merged with all the existing clusters in Γ, C excluded, to obtain a cluster with size at least *k*. This implies that, in the worst case, the function iterates the above-mentioned steps $\Gamma - 1$ times, and that in each iteration $\Gamma - 1$ enlargement are calculated. Moreover, at each iteration Γ is decremented by one. Thus, since Γ contains at most β *non-*k_s-anonymized clusters and the estimated time for calculating an enlargement is $C_d \cdot D$, the time complexity of *merge_clusters()* is, in the worst case:

$$C^w_{merge}=\sum_{i=1}^{\beta-1}(C_d \cdot D)(\beta - i) = \frac{\beta\cdot(\beta-1)}{2} \cdot (C_d \cdot D)$$

split(). Given in input a cluster C, *split()* randomly selects a tuple t from C, it forms a sub-cluster C_{new} over t, and let C_{new} absorb *k*-1 tuples of C. The above steps are then repeated until C's size is less than *k*.

In doing that the *split()* function generates a set of buckets BS over tuples of C according to their pid values, and let C_{new} absorb a different tuple from *k*-1 nearest buckets. More precisely, to speed up this step a heap structure is exploited. At the beginning, the heap is initialized with $k - 1$ nodes marked with an infinite distance from t. Then, for each bucket in BS, it selects its first tuple t and calculates the distance between $\widetilde{t}$ and t. If t is closer to $\widetilde{t}$ than the root of the heap, t replaces the root and the heap is adjusted accordingly. Let C_p be the cost of replacing the root heap, we can estimate the cost of the heap adjustment as: $|BS| \cdot (C_d \cdot D + C_p)$, where $C_d \cdot D$ is the cost of calculating a distance between two tuples. In order to consider the worst case, we assume that each tuple has a distinct pid. Thus, let n be the number of tuples in C, the number of buckets that are created over them in the worst case is n. This implies that the heap adjustment cost is $n \cdot (C_d \cdot D + C_p)$. In general, the heap adjustment is performed for each possible subcluster that can be generated over C, which, in the worst case, are $\lfloor n/k \rfloor$. Moreover, in each iteration the size of C is decremented of *k*, since a new subcluster is generated. Thus, the total cost of generating $\lfloor n/k \rfloor$ clusters is given by $\sum_{i=0}^{\lfloor n/k \rfloor - 1}(n - i \cdot k)(C_d \cdot D + C_p)$. Finally, the remaining tuples in C are inserted into one of the new generated subcluster. Thus, for each one of the remaining $n - k \cdot \lfloor n/k \rfloor$ tuples, it is calculated the enlargement of each of the generated subcluster. This implies a cost of $|n/k| \cdot (C_d \cdot D)$. Therefore, in the worst case, the overall time complexity of the *split()* function is:

$$\begin{aligned} C^w_{split}=&[\sum_{i=0}^{\lfloor n/k \rfloor - 1}(n - i \cdot k)(C_d \cdot D + C_p)] + \\ &(n - k \cdot \lfloor n/k \rfloor)(\lfloor n/k \rfloor \cdot (C_d \cdot D)) \\ \approx& \frac{n^2}{2k}(C_d \cdot D) \end{aligned}$$

It is relevant to note that since C_p cannot require more than log^k (the hight of heap) comparisons, C_p is much smaller than $C_d \cdot D$. Therefore, the total cost is roughly $\frac{n^2}{2k}(D \cdot C_d)$. If tuples are uniformly distributed among *non-*k_s-anonymized clusters, then each cluster has δ/k tuples and the cost is: $\frac{\delta^2}{2k^3}(C_d \cdot D)$. However, in the worst case, that is, tuples with highly skew distribution, a cluster may contain almost all the δ tuples. In this case the cost is: $\frac{\delta^2}{2k}(C_d \cdot D)$.

Let us now discuss the overall time complexity of CASTLE. In doing that, we estimate how many times *best_selection()*, *merge_clusters()* and *split()* are called by CASTLE. Let S_{in} be the stream given in input to CASTLE, where with $|S_{in}|$ we denote the number of tuples in S_{in}.

best_selection() is called for each of the $|S_{in}|$ tuples, in that, all of them are accommodated into a cluster. Thus, the overall time complexity of *best_selection()* is

$$\begin{aligned} C_{ToT_best_selection}=&|S_{in}| \cdot C^w_{best_selection} \\ =& |S_{in}| \cdot (C_d \cdot D) \cdot \beta \end{aligned}$$

Moreover, since β is a given in input to CASTLE and, according to the estimation given in Table III its value does not exceed 100, we can estimate this cost as: $O(|S_{in}|)$.

We recall that a merge is performed when a tuple is expiring a no other way to output the tuple is possible (see Section IV). Thus, we can say that in the worst case *merge_clusters()* is called for each one of the $|S_{in}|$ tuples. However, every time a merge is performed the number of tuples in $|S_{in}|$ that still have to be evaluated by CASTLE decreases of at least *k* tuples (this is the case when all the tuples have a distinct *pid*). Thus, the total number of possible merge operations is, in the worst case, $\frac{|S_{in}|}{k}$, which implies that the overall time complexity of *merge_clusters()* is:

$$\begin{aligned} C_{ToT_merge_clusters}=&\frac{|S_{in}|}{k} \cdot C^w_{merge} \\ =& \frac{|S_{in}|}{k} \cdot \frac{\beta\cdot(\beta-1)}{2} \cdot (C_d \cdot D) \end{aligned}$$

Moreover, using the same reasoning we have made before regarding the β parameter, we can estimate this cost as O($\frac{|S_{in}|}{k}$).

Finally, the *split()* function is called when an expiring tuple is contained into a *non-*k_s-anonymized with size at least 2k. In the worst case, we can say that this happens for all tuples in S_{in}. However, similarly to the merge operation, ever time a split is performed the number of tuples decreases of at least *2k* tuples. Therefore, the number of possible split over $|S_{in}|$ tuples is $\frac{|S_{in}|}{2k}$, which implies that the overall time complexity of *split()* is:

$$\begin{aligned} C_{ToT_split}=&\frac{|S_{in}|}{2k} \cdot C^w_{split} \\ =& \frac{|S_{in}|}{2k} \cdot \frac{n^2}{2k} \cdot (C_d \cdot D) \\ \leq& \frac{|S_{in}|}{2k} \cdot \frac{\delta^2}{2k} \cdot (C_d \cdot D) \end{aligned}$$

Therefore, the overall time complexity of CASTLE is bounded by the time complexity of *split()*, i.e., $O(\frac{|S_{in}| \cdot \delta^2}{k^2})$. However, since $|S_{in}|$ is much more greater than both k and δ, we can conclude that the overall time complexity of CASTLE is $O(S_{in})$.

Space Complexity

To estimate the space complexity of CASTLE we need to consider the information stored in memory when executing CASTLE. This consists of three main components: (1) the data stream tuples (i.e., tuples in the *non-*k_s-anonymized clusters), (2) the information about *non-*k_s-anonymized and k_s-anonymized clusters.

Regarding the first component, it is important to note that delay constraints ensures that, at any instant, there are at most δ tuples in memory. Thus, let S_t be the space required to store a tuple, the first component requires $\delta \cdot S_t$ space.

Let us consider component (2). For each cluster, either *non-k_s*-anonymized or k_s-anonymized cluster, CASTLE stores the corresponding generalization. Thus, let S_g be the space required to store the generalization of a quasi-identifier attribute, storing the generalization of a cluster requires $(D \cdot S_g)$ space.

Moreover, it is relevant to note that the number of *non-k_s*-anonymized clusters in memory is at maximum β. Whereas, we assume that the number of k_s-anonymized that CASTLE keeps in the memory is equal to NC_{k_s} at maximum, where NC_{k_s} is a threshold that can determined based on the available memory.

Therefore, the total space required by CASTLE is:

$$S_{cost}=\delta \cdot S_t + (\beta + NC_{k_s})(D \cdot S_g) \ .$$

V. Performance evaluation

We have implemented CASTLE and have conducted several experiments. We report our results in this section. Our experiments have been designed with two objectives in mind. First we would like to verify that the proposed method is able to continuously anonymize a data stream while keeping the data useful (i.e., with a low information loss and an acceptable accuracy in answering aggregate queries). Second, to illustrate the effectiveness of CASTLE, we compare it with the approach presented in [4], which is the one comparable to our approach since it k-anonymizes the data set by a single pass on them. For these experiments, we used both synthetic and real world data. In particular, we have adopted the Adult data set from UC Irvine Machine Learning Repository [8], which has become a standard for studying k-anonymity. We refer to this as *UCI-Adult*. Moreover, in order to have a better simulation of a data stream, we have also considered the data set used in [14]. We refer to this as *SFU-Adult*. The experiments were conducted on an Intel Pentium IV 2.4GHz with 1 GB RAM.

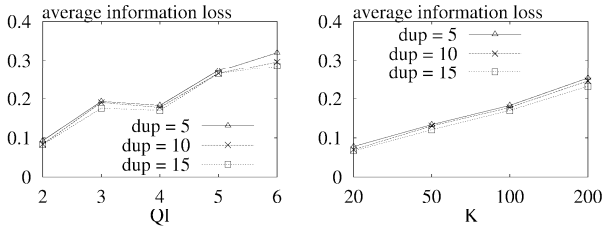
We configure *UCI-Adult* by removing tuples with missing values. Thus it contains 30,162 tuples. *SFU-Adult* is configured by adding 15,060 extra tuples to *UCI-Adult*, so it has 45,222 tuples. For k-anonymization, our quasi-identifier attributes are selected from the following attributes:*age, final-weight, education-number, capital-gain, capital-loss, hours-per-week, education, martial-status, occupation, nation*. The first six of them are continuous, and the left four are categorical. We adopt from [14] the hierarchies for categorical attributes, and the intervals for continuous attributes. In the following, we evaluate CASTLE using the metric described in Section II-B.

A. Tuning CASTLE

In this section, we evaluate CASTLE with different parameters. The parameters that affect the performance of CASTLE are: δ, *k*, the number of quasi-identifier attributes, μ, β, and the data distribution.

Effects of β and μ

CASTLE's adaptability to data distribution is controlled by two parameters: μ, the number of most recent k_s-anonymized clusters on which τ is calculated, and β, the threshold for controlling the maximum number of clusters in the memory. Figure 4 presents the average information loss of k_s-anonymized tuples by simultaneously varying β and μ. We have set $k = 100$, $\delta = 10000$ and have used 10 quasi-identifier attributes. We run the experiments on both the *SFU-Adult* (Figure 4(a)) and *UCI-Adult* (Figure 4(b)) data sets. In both data sets $\beta = 50$ minimizes the information loss. In the experiment of *SFU-Adult*, $\mu = 100$ and $\mu = 150$ yield the best information quality; in the experiment of *UCI-Adult*, $\mu = 100$ outperforms $\mu = 150$. In the following experiments, we shall use $\beta = 50$ and $\mu = 100$ as the default values.



(a) varying QI (b) varying K

Fig. 7. Information loss on transaction stream

Effects of *quasi-identifiers*

A further experiment measures how the number of quasi-identifier attributes affects the average information loss. The experiment has been conducted on the *UCI-Adult* and *SFU-Adult* data sets with $k = 100$, $\delta = 10000$. Figure 5(a) reports how the average information loss varies by increasing the number of quasi-identifier attributes. As expected, the information loss increases when increasing the number of quasi-identifier attributes.

Effects of *k*

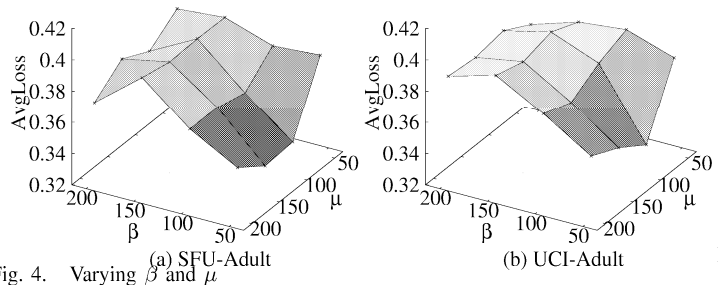
Figure 5(b) shows how the average information loss decreases by increasing the value of *k*. In this experiment we have considered 10 quasi-identifier attributes, $\delta = 10000$, and both *UCI-Adult* and *SFU-Adult* data sets. The results are expected as a larger k implies that we need a larger cluster to anonymize data - this translates to greater loss in information. Moreover, it may require more merging to take place.

Effects of data distribution

We have conducted several experiments to evaluate how CASTLE scales with different data distributions. In order to do that, we have evaluated CASTLE on synthetic data sets following power law distribution generated by means of genzipf [16]. More precisely, power law distribution is generated using the following two formulas: $p(i) = \frac{c}{i^{\alpha}}, i = \{1, \ldots, N\}$ and $\sum_{i=1}^{N} p(i) = 1$. Figure 6 reports the average information loss with different α values. The experiments have been conducted with β, μ and *k* fixed and by varying δ (i.e., $\beta = 50$, $\mu = 100$, and *k*={100,400,700, 1000}). More precisely, we have evaluated the behavior of CASTLE wrt data distribution with different *k* values (cfr. Figures 6(a), ... 6(d)). From the results, it is clear that CASTLE is very effective for clustered data. This is promising as real data are typically clustered.

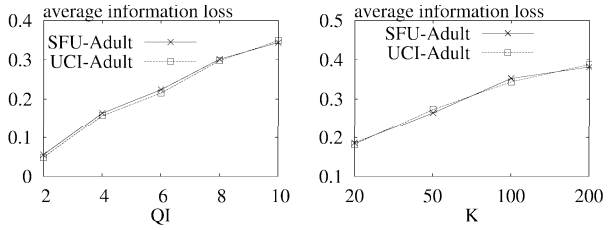
Effect of multiple tuples referring to the same person

In the previous experiments we have considered the UCI-Adult and SFU-Adult datasets, by assuming that each tuple refers to a distinct person, that is, each tuple has a different pid. However, since in a data stream multiple tuples may refer to the same person, we have conducted further experiments to evaluate how this duplication of pids may affect the information loss. In particular, to generate a dataset with duplications we exploit the IBM Quest Synthetic Data Generation Code [3]. By means of this code, we have generated a set of tuples T_{tran} with schema $(pid, tid, list\ of\ items)$, where pid is a customer's id, tid represents transaction id, and *list of items* are the items a customer has bought. Then, the considered dataset is obtained by joining T_{tran}



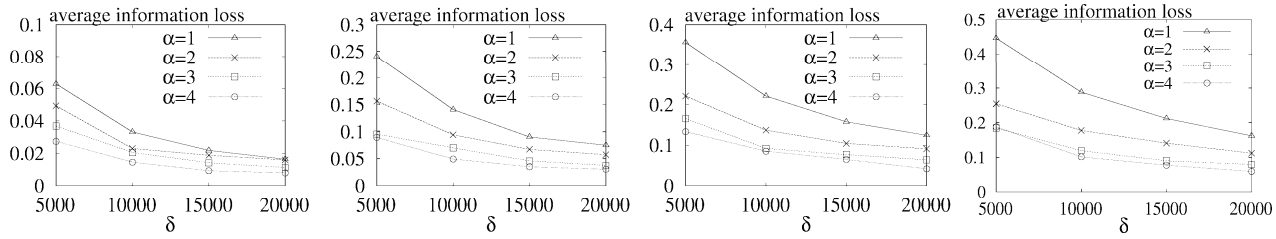
(a) SFU-Adult (b) UCI-Adult

Fig. 4. Varying β and μ



(a) varying QI (b) varying k

Fig. 5. Varying k and quasi-identifiers



(a) k =100 (b) k = 400 (c) k = 700 (d) k = 1000

Fig. 6. Information loss on power-law synthetic data

and UCI-Adult on their *pid*. [10] As a result, tuples in the dataset has the schema $(pid, tid, QI, list\ of\ items)$. Moreover, to simulate the presence of more transactions referring to the same person, when a tuple t of the dataset is pushed into a cluster, CASTLE splits it into a set of tuples with schema $(pid, tid, QI, item)$. For example tuple $(pid_1, tid_1, QI, item_1, item_2, \ldots, item_n)$ is split into $(pid_1, tid_1, QI, item_1)$, ..., $(pid_1, tid_1, QI, item_n)$, obtaining thus n tuple referring to pid_1. According to this approach, we have generated three different synthetic datasets, by varying the number of items in the $list\ of\ items$. The experiments are conducted by setting δ equal to 10000, and varying k and the number of quasi-identifier attributes. In particular, in Figure 7(a) we set $k = 100$ and vary the size of quasi-identifiers, where dup=5, dup=10, and dup=15 indicates the synthetic dataset with 5, 10, and 15 items in the $list\ of\ items$, respectively. In general, when the size of quasi-identifier increases, the information quality degrades. In Figure 7(b) we set the size of quasi-identifiers to be 4, and vary k. We can observe that when k increases, the information loss is uniformly increased.

B. Utility

In this section, we examine the effectiveness of CASTLE by considering its accuracy in answering aggregate queries. In order to do this, we evaluate the same aggregate queries over S_{out} and the non-anonymized stream and compare the results. In particular, since CASTLE does not give in output the tuples in the same order as they are in S_{in}, to ensure a fair comparison, we do not compare results of the aggregate queries over S_{out} and S_{in}. In contrast, we slightly modify CASTLE such that in addition to S_{out} it gives in output also the non generalized stream, hereafter S_{non}, where each tuple t has the same position as the corresponding generalized tuple $\bar{t}$ in S_{out}.

To evaluate the utility of CASTLE we exploit the metric introduced in [29], according to which the evaluated queries are of the form:

[10]Since the T_{tran} is bigger than the UCI-Adult dataset, which consists of 30,162 tuples, we join tuples of T_{tran} with $pid > 30,162$ with randomly selected tuples in UCI-Adult.

SELECT COUNT(*) FROM S_{out}
WHERE $pred(q_1)$ AND...AND $pred(q_n)$ AND $pred(A_s)$;

where:

- $q_1, \ldots, q_n$ are the quasi-identifier attributes, and A_s is the sensitive attribute.
- If q_i (resp., A_s) is a continuous attribute, $pred(q_i)$ (resp., $pred(A_s)$) is $q_i \in R_i$ (resp., $A_s \in R_s$), where R_i is a randomly generated interval in the domain of q_i (resp., A_s) with length $|q_i|\theta^{\frac{1}{n+1}}$ (resp., $|A_s|\theta^{\frac{1}{n+1}}$). $|q_i|$ (resp., $|A_s|$) is the size of domain q_i (resp., A_s) and θ is the *expected selectivity*.
- If q_i (resp., A_s) is a categorical attribute, $pred(q_i)$ (resp., $pred(A_s)$) is $q_i \in R_i$ (resp., $A_s \in R_s$), where R_i is any node in the domain hierarchy of q_i (resp., A_s), such that $\frac{|Leaves(R_i)|}{|Leaves(q_i)|} \approx \theta^{\frac{1}{n+1}}$ (resp., $\frac{|Leaves(R_i)|}{|Leaves(A_s)|} \approx \theta^{\frac{1}{n+1}}$). [11]

These queries have been evaluated both over the non-generalized stream S_{non} and over the generalized stream S_{out}. Given a query Q, let act be the query result on S_{non}, and est the result on S_{out}. est is computed as the sum of the probability that generalized tuple $t \in S_{out}$ satisfies the WHERE clause in Q. Thus, est is computed by then following formula:

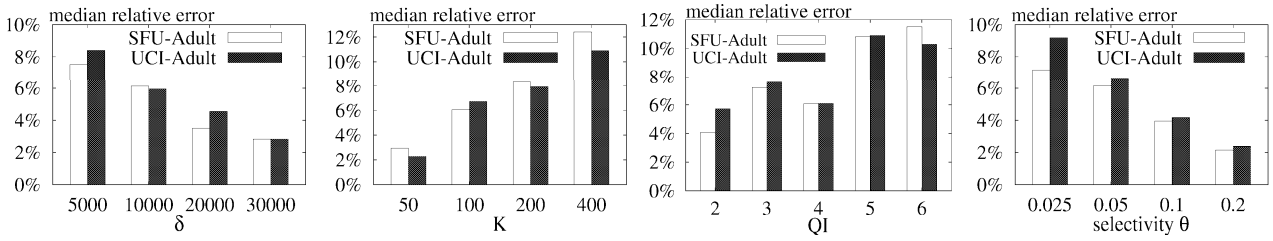
$$est = \textstyle\sum_{t \in S_{out}} P(t), \text{ where}$$
$$P(t) = P(t.A_s \in R_s) \times \textstyle\prod_i^n P(t.q_i \in R_i).$$

In particular,

- if q_i (resp., A_s) is a continuous attribute, $P(t.q_i \in R_i) = \frac{|t.q_i \cap R_i|}{|t.q_i|}$ (resp., $P(t.A_s \in R_s) = \frac{|t.A_s \cap R_s|}{|t.A_s|}$);
- if q_i (resp., A_s) is a categorical attribute, $P(t.q_i \in R_i) = \frac{|leaves(t.q_i) \cap leaves(R_i)|}{|leaves(t.q_i)|}$ (resp., $P(t.A_s \in R_s) = \frac{|leaves(t.A_s) \cap leaves(R_s)|}{|leaves(t.A_s)|}$).

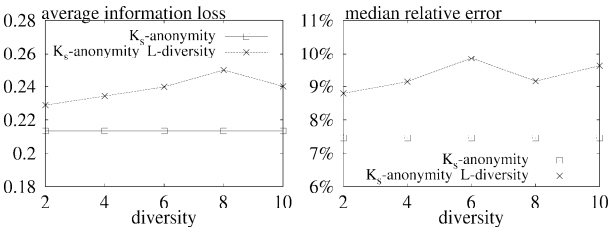
Thus, similar to [29], we measure $|act - est|/|act|$ as the *relative error* of Q, and consider the median relative error of all the queries as the *workload error*. We run 10000 queries for each workload, i.e., *SFU-Adult*, and *UCI-Adult* datasets. Moreover,

[11]Here, $Leaves(R_i)$ is the set of leaves of the subtree rooted at R_i, and $Leaves(q_i)$ is the set of leaves in the domain hierarchy of q_i.



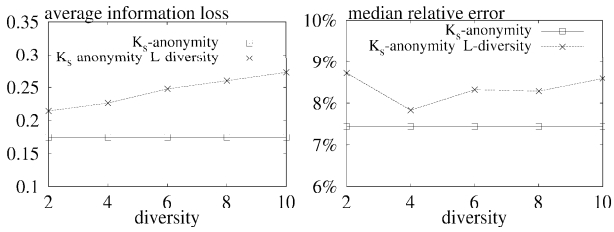
(a) varying δ (b) varying k (c) varying QI (d) varying selectivity

Fig. 8. Data utility



(a) information loss (b) utility

Fig. 9. k_s-anonymity and l-diversity: Age



(a) information loss (b) utility

Fig. 10. k_s-anonymity and l-diversity: occupation

since *est* is calculated with the assumption that values of each attribute q_i (resp., A_s) are uniformly distributed in R_i, we do not consider those attributes of the workload that are non-uniformly distributed, that is, attribute *capital-gain* and *capital-loss* in *UCI-Adult* datasets.[12]

Figure 8 reports the experiments. In particular, in Figure 8(a) we have considered 4 quasi-identifier attributes and 1 sensitive attribute, set $k = 100$, and selectivity $\theta = 0.05$, by varying δ. As expected, when δ increases the utility is better. In Figure 8(b) we consider 4 quasi-identifier attributes and set $\delta = 10000$ and $\theta = 0.05$ by varying k. When k increases, utility degrades since a larger k requires a cluster containing more tuples, which implies that its generalizations are enlarged. Figure 8(c) presents the result of varying QI (i.e., the quasi-identifiers) when k=100, $\delta = 10000$ and $\theta = 0.05$; the workload error does not change uniformly as the size of QI increases. However, in general, utility is reduced when the size of QI increases. Figure 8(d) is the result of varying selectivity θ with 4 quasi-identifier attributes, k=100 and $\delta = 10000$. A bigger θ yields a better accuracy since a bigger θ indicates bigger R_i and allows a larger generalization of q_i (resp., A_s). In all these experiments, the median relative error of CASTLE is less than 13%, indicating high utility of anonymized streams.

C. k_s-anonymity and l-diversity

We evaluate the information quality of k_s-anonymous and l-diverse data on the UCI-Adult data set. We fix $\delta = 10000$, $k = 100$, and the size of quasi-identifiers to 5. In Figure 9 the sensitive attribute is *age*, whereas in Figure 10 *occupation* is used as the sensitive attribute. When the diversity increases, a cluster should contain potentially more tuples, thus the generalization of the cluster increases. Therefore, both information quality and utility decrease when the diversity increases. We know that when k increases and/or diversity increases, the security of the anonymized data stream is better. However, based on all the above experiments, we can conclude that the information loss/utility and the security is a trade-off: to better preserve the privacy we may need to degrade information quality/utility; to preserve more information/utility, we need to reduce k and/or diversity.

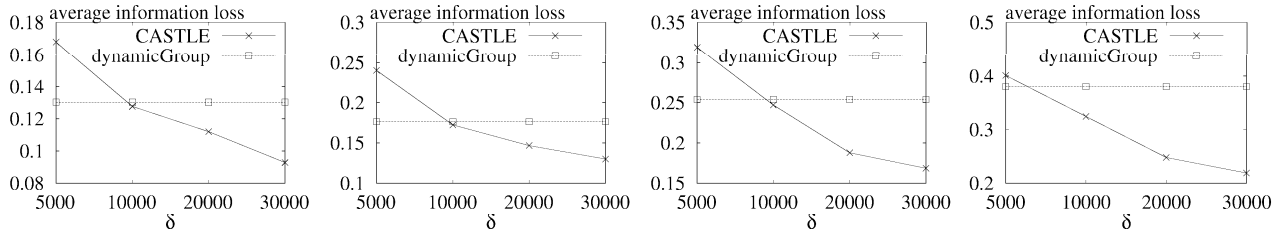
[12]Attribute *capital-gain* has the range [0,99999] in *UCI-Adult*(30162 tuples), but its value in 27624 tuples is 0; attribute *capital-loss* has the range [0,4356] in *UCI-Adult*, but its value in 28735 tuples is 0.

D. Comparative study

Finally, we compare the performance of CASTLE wrt the approach presented in [4], hereafter called *dynamicGroup*. We performed the comparison wrt the average information loss and the number of k-anonymized tuples.

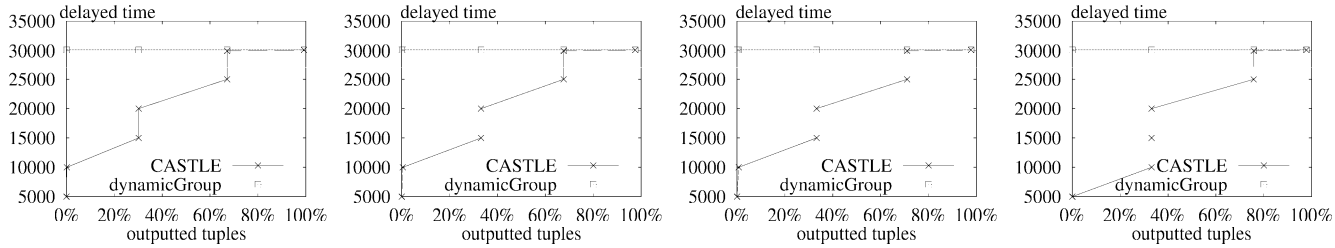
Recall that *dynamicGroup* can only process continuous attributes. Thus, the experiment has been performed with the 6 continuous quasi-identifier attributes of the *UCI-Adult* data set. Moreover, *dynamicGroup* only outputs the anonymized data when the process is complete (i.e., after scanning the entire data set). For a fair comparison, CASTLE anonymizes the stream data up to its end as follows: after the last tuple from the stream is pushed to a cluster, CASTLE outputs all the clusters whose size is not less than k. Then CASTLE generalizes all the tuples which fall in k_s-anonymized clusters in Ω. Finally, CASTLE merges all the remaining non-anonymized tuples to form a cluster and outputs it. *dynamicGroup* uses historical data to build some clusters. We take the first n tuples in UCI-Adult as the historical data, and all the remaining tuples as the streaming data. We vary n from 2000 to 8000, and select the best one for *dynamicGroup*. The best n is 8000.

Figure 11 reports how the average information loss of CASTLE and *dynamicGroup* vary with different δ values. We have considered several values of k (see Figures 11(a),(b),(c), and (d)). It is important to note that *dynamicGroup* does not consider the delay constraint. Thus, it can retain tuples till the end of the process, which obviously influences the information loss. For a fair comparison between CASTLE and *dynamicGroup* we must consider only the average information loss of CASTLE with δ set to *infinity*. As shown in Figure 11, as δ increases, the information



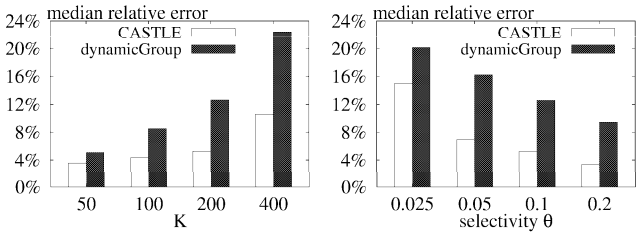
(a) k = 50 (b) k = 100 (c) k = 200 (d) k = 400

Fig. 11. Information loss on UCI-Adult



(a) k =50, δ = 10000 (b) k=100, δ = 10000 (c) k=200, δ = 10000 (d) k=400, δ = 10000

Fig. 12. Output percentage on UCI-Adult



(a) varying k (b) varying selectivity

Fig. 13. Utility on UCI-Adult

loss of CASTLE decreases. Moreover, when δ is 10000, CASTLE outperforms *dynamicGroup*.

Figure 12 shows an additional advantage of CASTLE over *dynamicGroup*. We present the number of output tuples that are returned progressively over time. Here, delayed time represents the order of the most recently arrived tuple. CASTLE is able to output anonymized data progressively. In fact, when the last tuple (30,162) arrives, over 97% of all the tuples have already been output. However, in *dynamicGroup*, anonymized data are only output at the end. This is not acceptable in a data stream context.

We also compare CASTLE with dynamicGroup wrt their effectiveness on utility. In Figure 13 we consider 4 continuous quasi-identifier attributes (without *capital-gain* and *capital-loss*) and 1 sensitive attributes, set $\delta = 20000$. In Figure 13(a) we vary k, by setting $\theta = 0.1$, whereas in Figure 13(b) we vary the selectivity θ, by setting k=200. In both experiments CASTLE outperforms dynamicGroup.

VI. Related work

There are two main fields related to the work reported in this paper, that is, data stream management and privacy preserving data management. Regarding the first field, it is relevant to note that so far the data stream community has been mainly focused on performance issues (e.g., continuous query processing, data stream mining, see [15] for a survey), with the results of the proposal of several data stream processing systems, developed both academically (e.g., Aurora [2], Borealis [1], STREAM [7], TelegraphCQ [13]) and commercially (e.g., StreamBase [26]). Since we can consider these issues orthogonal to our work, in this section we focus on work on privacy preserving data management.

Several techniques have been proposed and investigated to protect the privacy of data to be mined. Among these approaches, we can find methods based on perturbation [4], [6], [18] and k-anonymity [10], [14], [19], [20], [25], [27]. According to perturbation-based techniques data privacy is ensured by releasing perturbed individual data or perturbed queries results, where perturbation is performed, for example, by swapping [18], condensation [4], or adding noise [6]. However, perturbation-based methods have the weakness of tampering the integrity of released data. In contrast, the records within a k-anonymized data set remain true. In the recent years, several k-anonymity methods have been proposed [10], [14], [19], [20], [25], [27]. Recently other anonymization techniques have been proposed to avoid possible inference attacks on k-anonymized data. For instance, Machanavajjhala et al. [22] proposed the l-diversity principle. Ensuring l-diversity implies that all tuples with the same generalization values should have at least l distinct values for each different sensitive attribute. Along the same line is t-closeness [21], which requires that the distribution of sensitive attributes inside tuples with the same generalization values is analogous to the distribution of the whole data set (i.e., the distance between the two distributions should be no more that t).

However, as pointed out in Section II, all these methods are not adequate for data streams. In addition to the motivations presented in Section II, we have to take also into account that all these methods require to scan the data to be anonymized multiple times, and output the anonymous data set in the final step. This assumption is unsuitable in data stream scenario, in that, for streaming data the algorithm can only scan the data in one pass and executes in a pipeline manner.

Among all the k-anonymity methods proposed so far, CASTLE has some similarities with those exploiting cluster-based tech-

niques. In particular, cluster-based algorithms for k-anonymizing static data sets have been proposed in [4], [5], [12]. [5] presents a theoretical framework for grouping tuples in clusters of at least k tuples and publishing tuples by replacing the quasi-identifier attributes with the center of the corresponding cluster. However, this approach does not support domain generalization hierarchies and manages categorical attributes like simple discrete values, thus losing relevant semantic information that a domain generalization hierarchy can provide. In contrast, the approach presented in [12] adopts a distance metric similar to the one we adopt, but, like the approach in [5], it only considers static data sets and no delay constraints are enforced. An approach similar to [12] has been also proposed for the k-anonymization of incremental data sets [28]. However, this approach does not consider the possible inferences that an attacker can do by analyzing multiple releases of the same table, which makes the approach potentially insecure and thus inapplicable. Another cluster-based method related to ours is [4], where a cluster-based scheme for k-anonymizing data is presented. Like the scheme presented in this paper, the approach in [4] requires a single pass over the data, by generating clusters on-the-fly. More precisely, it creates some clusters using historical data and pushes a new tuple to the nearest cluster (i.e., the cluster having its center to the minimum distance from the tuple). Then, after all the tuples from the data set have been processed, tuples in each cluster are generalized and output. Since a data stream is an unbounded flow of tuples, this approach can hardly be applied, due to memory limitation. Moreover, it does not support categorical attributes nor delay constraints.

It is interesting to note that most of the k-anonymity schemes proposed so far have been devised under the assumption that the data set to be anonymized is static, that is, the data set once has been anonymized does not change. However, recently, the problem it is emerged of anonymizing data sets that evolve over time, i.e., data sets where new tuples can be inserted as well existing ones can be deleted. Obviously, tuple insertions and deletions require to release a new k-anonymization of the updated data set. As shown in [11], [24], [29], multiple releases of k-anonymized data sets might be subject to potential inferences that may tamper the k-anonymity of the data. In this sense, anonymizing data streams and anonymizing dynamic data sets have some similarities, in that they both suffer from potential inferences due to dynamic updates. However, it is important to note that the inferences that may arise when anonymizing dynamic data sets and those that might happen during anonymization of data streams are different, as the following discussion clarifies. Anonymizing a dynamic data set requires to publish multiple k-anonymized releases of a table. Here, the inference over the real values of a tuple t is possible since t could be present in more than one of the k-anonymized tables. Therefore, an attacker, by analyzing the generalization of t in the different releases may be able to narrow down its generalization or even to guess the real values of the quasi identifier attributes of t, and thus to link t with a victim, as shown in [11], [24], [29]. When anonymizing data stream this inference does not take place, in that a tuple in the stream is anonymized only once. In contrast, when anonymizing data streams the possible inferences are due to the fact that the attacker is able to inspect the sequence of anonymized tuples given in output (as discussed in Section IV-C.1). Therefore anonymization methods devised for dynamic data sets cannot be directly applied to anonymize data streams.

VII. Conclusions

In this paper we have presented CASTLE a cluster-based framework to k-anonymize data streams. Relevant features of CASTLE are the enforcement of delay constraints, its adaptability to data distributions and the use of a cluster reuse strategy that improves the performance without compromising security. Performance evaluation reported in this paper have shown that CASTLE is efficient and effective. We plan to extend this work along several directions. A first direction is related to the t-closeness principle [21]. Ensuring this principle for data streams arises challenging issues, mainly due to the fact that the data distribution of a stream is unknown a priori. Additionally, in the current paper we have investigated inference problems that could arise by tracing a k-anonymized data stream. As an extension, we would like to investigate how the proposed solution should be extended to consider other forms of background knowledge, such as for instance temporal background knowledge. Finally, we will study optimization techniques to improve the efficiency of our system.

References

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S.B. Zdonik. The design of the borealis stream processing engine. In Proc. of *Conference of Innovative Data System Research (CIDR'05)*, pages 277–289, Asilomar, USA, 2005.

[2] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S.B. Zdonik. Aurora: a new model and architecture for data stream management. In *VLDB Journal*, 12(2):120–139, 2003.

[3] R. Agrawal, R. Srikant. Fast algorithms for mining association rules in large databases. In Proc. of *VLDB*, pages 478-499, 1994.

[4] C. C. Aggarwal and P. S. Yu. A condensation approach to privacy preserving data mining. In Proc. of *EDBT*, pages 183–199, 2004.

[5] G. Aggarwal, T. Feder, K. Kenthapadi, S. Khuller, R. Panigrahy, D. Thomas, and A. Zhu. Achieving anonymity via clustering. In Proc. of *PODS*, pages 153–162, 2006.

[6] R. Agrawal and R. Srikant. Privacy-preserving data mining. In Proc. of *SIGMOD*, pages 439–450, 2000.

[7] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In Proc. of *SIGMOD*, page 665, 2003.

[8] UCI repository. www.ics.uci.edu/~mlearn/MLSummary.html.

[9] M. Atzori Weak k-anonymity: a low-distortion model for protecting privacy, In Proc. of International Security Conference, pages 60-71, 2006.

[10] R. J. Bayardo and R. Agrawal. Data privacy through optimal k-anonymization. In Proc. of *ICDE*, pages 217–228, 2005.

[11] J.W. Byun, Y. Sohn, E. Bertino, and N. Li. Secure Anonymization for Incremental Datasets. In Proc. of the *VLDB Workshop on Secure Data Management*, pages 48–63, 2006.

[12] J.W. Byun, A. Kamra, E. Bertino, and N. Li. Efficient k-Anonymization Using Clustering Techniques In Proc. of *Database Systems for Advanced Applications (DASFAA)*, pages 188-200, 2007.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M.A. Shah. TelegraphCQ: continuous dataflow processing for an uncertain world. In Proc. of *Conference of Innovative Data System Research (CIDR'03)*, Asilomar, USA, 2003.

[14] B. C. M. Fung, K. Wang, and P. S. Yu. Top-down specialization for information and privacy preservation. In Proc. of *ICDE*, pages 205–216, 2005.

[15] L. Golab and M. Ozsu. Issues in data stream management. In *SIGMOD Record*, 32(2):5-14, 2003.

[16] genzipf. www.csee.usf.edu/~christen.

[17] V. S. Iyengar. Transforming data to satisfy privacy constraints. In Proc. of *KDD*, pages 279–288, 2002.

[18] S. Kim and W. Winkler. Masking microdata files. In Proc. of *Section on Survey Research Methods*, pages 114–119, 1995.

[19] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Incognito: Efficient full-domain k-anonymity. In Proc. of *SIGMOD*, pages 49–60, 2005.

[20] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Mondrian Multidimensional K-Anonymity. In Proc. of *ICDE*, page 25, 2006.
[21] N. Li, and T. Li. *t*-closeness: privacy beyond *k*-anonymity and *l*-diversity. In Proc. of *ICDE*, pages 106–115, 2007.
[22] A.Machanavajjhala, J. Gehrke, D. Kifer, M. Venkitasubramaniam. l-Diversity: privacy beyond k-anonymity. In Proc. of *ICDE*, page 24, 2006.
[23] P. Samarati and L. Sweeney. Generalizing data to provide anonymity when disclosing information. In Proc. of *PODS*, page 188, 1998.
[24] J. Pei, J. Xu, Z. Wang, W. Wang, K. Wang Maintaining K-Anonymity against Incremental Updates In Proc. of *SSDBM*, page 5, 2007.
[25] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Transaction Knowledge Data Engineering (TKDE)*, 13(6):1010–1027, 2001.
[26] StreamBase Home Page. http://www.streambase.com//.
[27] L. Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *International Journal on Uncertainty, Fuzziness, and Knowledge-Based Systems*, pages 571–588, 2002.
[28] T.M. Truta, A. Campan k-anonymization incremental maintenance and optimization techniques. In Proc. of *SAC*, pages 380–387. 2007.
[29] X. Xiao, Y. Tao. m-invariance: towards privacy preserving re-publication of dynamic datasets. In Proc. of *SIGMOD*, pages 689–700, 2007.
[30] X. Xiao, Y. Tao. Anatomy: Simple and Effective Privacy Preservation. In Proc. of *VLDB*, pages 139–150, 2006.

APPENDIX

Proof: of Theorem 1

According to Definition 1, proving the theorem requires to prove that:

1) For each tuple t in the input stream S, there exists in S_{out} the corresponding anonymized tuple $\bar{t}$.
2) For any possible distinct QI group $qi \in S_{out}$, the number of distinct individuals to which tuples in qi refer to is greater than or equal to k, that is, $|DP(qi)| \geq k$.

Let us first consider point 1 and let t be a tuple in the input stream S. According to Algorithm 1, at most at time $t.p + \delta$ the tuple is passed to procedure *delay_constraint()* (step 13 of Algorithm 1). We prove that this procedure always output an anonymized tuple corresponding to t. Different cases may arise. The first is when the non k_s-anonymized cluster hosting t has size greater than or equal to k. In this case, the procedure calls *output_cluster()* (step 3) which gives in output a generalized tuple for each tuple in C, thus the thesis holds. If the non k_s-anonymized cluster hosting t has size less than k, procedure *delay_constraint()* verifies whether the reuse strategy can be applied, in this case t is given in output with the generalization of one of the k_s-anonymized clusters hosting it, thus the thesis holds (step 8 of procedure *delay_constraint()*). If the reuse strategy cannot be applied, then *delay_constraint()* verifies whether the cluster containing t is an outlier, in this case all its tuples are suppressed, that is, they are given in output with the most generalized values, thus the thesis holds (step 15 of procedure *delay_constraint()*). Finally, if the cluster hosting t is not an outlier, *delay_constraint()* verifies if it is possible to merge it with other clusters. If this is not possible, all its tuples are suppressed (step 18 of procedure *delay_constraint()*). Otherwise, all the tuples in the cluster resulting from the merge operation are given in output with its generalization (step 21 of procedure *delay_constraint()*). In both the cases, the thesis holds.

Let us consider now point 2. To prove it, we need to consider when a new QI group is formed or modified into S_{out}. Therefore, we consider when a new tuple in inserted into S_{out}. CASTLE outputs a tuple in three possible ways.

(1) – According to the first one the tuple t is output together with all the other tuples contained into the cluster C to which it belongs to (line 6, procedure $output_cluster()$). In this case, CASTLE outputs all tuples of cluster C with the same generalization, which implies that all tuples in C have the same values for QI attributes. This implies that a new QI group is formed into S_{out}, whose values of the quasi identifier attributes are given by the generalization of C. Moreover, CASTLE outputs all tuples of cluster C only if the cluster size is greater than or equal to k. This condition ensures that there are at least k distinct individuals to which tuples in C refer to. This ensures that when a new QI group qi is formed into S_{out}, the number of distinct individual to which tuples in qi refer to is greater or equal to k, that is, $|DP(qi)| \geq k$.

(2) – A tuple t can also be individually output with the generalization of a k_s-anonymized cluster KC, by exploiting the reuse strategy (line 8, procedure $delay_constraint()$). In this case, t belongs to the QI group in S_{out} having the quasi identifier attributes equal to KC generalization. Since KC is k_s-anonymized, this implies that at a given instant KC's size has been greater than or equal to k and all its tuples have been given in output with KC generalization. This condition ensures that in the QI group qi having the quasi identifier attributes equal to KC generalization there are at least k distinct individuals to which tuples in qi refer to. Thus, $|DP(qi)| \geq k$.

(3)– Furthermore, a tuple can be also suppressed (step 18, procedure $delay_constraint()$), which implies that the tuple is given in output with the most generalized values. In this case, the k_s-anonymity definition does not require that $|DP(qi)| \geq k$.

■

Proof: of Theorem 2

For simplicity, let us consider a data stream S with a single quasi-identifier attribute q. Let us consider a tuple t k_s-anonymized with generalization G_t. In this case, an attacker could infer a set of possible value $\chi \subset values(G_t.q)$ for attribute q, only if there exists a set of tuples $\mathcal{T} \subseteq S_{out} \setminus \{t\}$ k_s-anonymized with a generalization G' that *overlaps* G_t. Two generalization G and G' overlap if there exists at least a quasi identifier attribute q_j such that: $Values(G.q_j) \cap Values(G'.q_j) \neq 0$. Let us assume that an attacker infers a set of possible value $\chi \subset values(G_t.q)$ for attribute $t.q$, and prove that a contradiction arises. Let us consider a tuple $\tilde{t}$ in $S_{out} \setminus \{t\}$ output with generalization $G_{\tilde{t}}$ such that $G_t.q$ overlaps $G_{\tilde{t}}.q$. Let us first consider the case that q is a continuous attribute. χ can be inferred by analyzing the overlap between $G_{\tilde{t}}$ and G_t, and can be formalized as follows: χ= $values(G_t.q) \setminus values(G_{\tilde{t}}.q)$. Thus, χ consists of the values in $values(G_t.q)$ which are not in $values(G_{\tilde{t}}.q)$. However, if there exists an intersection between $values(G_t.q)$ and $values(G_{\tilde{t}}.q)$, this implies that CASTLE anonymizes $t.q$ by randomly selecting between $G_{\tilde{t}}.q$ and $G_t.q$ (see line 7, procedure $delay_constraint()$). Thus, the possible real values of $t.q$ that can be inferred are, in addition to those in χ, also those in $values(G_{\tilde{t}}.q)$. Thus, χ= $(values(G_t.q) \setminus values(G_{\tilde{t}}.q)) \cup (values(G_t.q) \cap values(G_{\tilde{t}}.q))$, which implies that χ = $values(G_t.q)$, thus a contradiction arises. We omit the proof when q is a categorical attribute, since it is very similar to the proof given for continuous attributes.

■