A Framework to Enforce Access Control over Data Streams

Barbara Carminati, Elena Ferrari
DICOM, University of Insubria
Varese, Italy
{barbara.carminati, elena.ferrari}@uninsubria.it

Jianneng Cao, Kian Lee Tan
National University of Singapore, Singapore
{caojiann, tankl}@comp.nus.edu.sg

Although access control is today a key component of any computational system, it is only recently that mechanisms to guard against unauthorized access to streaming data have started to be investigated. To cope with this lack, in this paper we propose a general framework to protect streaming data, which is as much as possible independent from the target stream engine. Differently from RDBMSs, up to now a standard query language for data streams has not yet emerged and this makes more difficult the development of a general solution to access control enforcement. The framework we propose in this paper is based on an expressive role-based access control model proposed by us. It exploits a query rewriting mechanism, which rewrites user queries in such a way that they do not return tuples/attributes that should not be accessed according to the specified access control policies. Further, the framework contains a Deployment module able to translate the rewritten query in such a way that it can be executed by different stream engines, therefore overcoming the lack of standardization. In the paper, besides presenting all the components of our framework, we prove the correctness and completeness of the query rewriting algorithm, and we present some experiments that show the feasibility of the developed techniques.

Categories and Subject Descriptors: D.4.6 [Security and Protection]: Access controls; H.2.7 [Database Administration]: Security, integrity, and protection

General Terms: Security

Additional Key Words and Phrases: Data stream, Access control, Secure query rewriting

1. INTRODUCTION

Data Stream Management Systems (DSMSs) have been increasingly used to support a wide range of real-time applications (e.g., battlefield and network monitoring, telecommunications, financial monitoring, sensor networks). In many of these applications, there is a need to protect sensitive data from unauthorized accesses.
For example, in battlefield monitoring, the position of soldiers should only be accessible to the battleground commanders. Even if data are not sensitive, it may still be of commercial value to restrict their accesses. For example, in a financial monitoring service, stock prices are delivered to paying clients based on the stocks they have subscribed to. Hence, there is a need to integrate access control mechanisms into DSMSs. As a first step in this direction, in [Carminati et al. 2007b] we have presented a role-based access control model specifically tailored to the protection of data streams. Objects to be protected are essentially views (or rather queries) over data streams. The model supports two types of privileges - a read privilege for operations such as selection, projection and join, and aggregate privileges for operations such as min, max, count, avg, and sum. In addition, to deal with the intrinsic temporal dimension of data streams, two temporal constraints have been introduced - general constraints, that allow access to data during a given time bound, and window constraints, that support aggregate operations within a specified time window.

The second important issue to be addressed is related to access control enforcement. This issue is further complicated by the fact that, differently from RDBMSs, a standard query language for DSMSs has not yet emerged. Nonetheless, one of our goal is to develop a framework which is as much as possible independent from the target stream engine. Therefore, to overcome the problem of the lack of standardization in current DSMSs, in this paper we define a core query model, on which our access control mechanism is based, which formalizes the set of operators that are common to most of the stream query languages proposed so far (e.g., [Abadi et al. 2005; Abadi et al. 2003; Arasu et al. 2003; Cranor et al. 2003; Chandrasekaran et al. 2003]).

One of the key decisions when developing an access control mechanism is the strategy to be adopted to enforce access control. In this respect, three main solutions can be adopted: pre-processing, post-processing, and query rewriting. Pre-processing is a naïve way to enforce access control according to which the streams are pruned from the unauthorized tuples before entering the user query. The main drawback of this simple strategy is that it works well only for very simple access control models which, differently from our, do not support policies that apply to views. We believe that this is an essential feature to be supported, because it allows the specification of very useful access control policies. For instance, if pre-processing is adopted it is not possible to enforce a policy authorizing a captain to access the average heart beats of his/her soldiers, but only during the time of the certain action and/or of those soldiers positioned in a given region. In contrast, post-processing first executes the original user query, then, it prunes from the result the not authorized tuples, before delivering the resulting stream to the user. Like pre-processing, this strategy has the drawback that it does not support access control policies defined over portions of combined streams. In addition, as we will show in Section 7.2, it may waste computation, since queries are evaluated even if they are denied by the specified access control policies.

For the above reasons, we adopt the query rewriting approach. Thus, unlike conventional RDBMSs, our access control mechanism operates at query definition time, and hence avoids run-time overhead. This strategy fits very well in the data stream
scenario where queries are continuous and long running. Queries are registered into the stream engine and continuously executed on the incoming tuples. Whenever a user submits a query, the Query Rewriter checks the authorization catalogs to verify whether the query can be partially or totally executed, or should be denied. In case of partially authorized queries, the specified query is rewritten in such a way that it results only authorized data. On support of the query rewriting task, we design a set of novel secure operators (namely, Secure Read, Secure View, Secure Join and Secure Aggregate) that filter out from the results of the corresponding (not secure) operators those tuples/attributes that are not accessible according to the specified access control policies. In the paper, besides presenting the secure operators and the query rewriting algorithm, we formally prove the correctness of the algorithm and the completeness of its results with respect to the specified access control policies.

Since query rewriting is based on the defined core query model, it is independent from the target stream engine. The last step is therefore the translation of the rewritten query into the language of the target DSMS. In identifying the target DSMSs to be considered, we have focused on existing commercial DSMSs. To the best of our knowledge these are Coral8 [Coral8 2008], StreamBase [StreamBase 2008], and Truviso [Truviso 2008]. Unfortunately, at the time of this writing it has not been possible to retrieve helpful documentation about Truviso and the underlying query language. For this reason, we focus on Coral8 and StreamBase and we show how the rewritten query can be deployed in these systems.

To the best of our knowledge this is the first paper presenting a framework for access control over data streams, which supports a very expressive access control model and, at the same time, is as much as possible independent from the target DSMS.

The work reported in this paper substantially extends the work presented by us in [Carminati et al. 2007b] and [Carminati et al. 2007a]. [Carminati et al. 2007b] only presents the access control model underlying our framework, whereas [Carminati et al. 2007a] presents an access control enforcement mechanism casted into the Aurora data stream prototype [Abadi et al. 2003]. In this paper, we build on what has been presented in [Carminati et al. 2007b] and [Carminati et al. 2007a], and we present a framework able to deploy authorized queries into different stream engines. This is a substantial extension both from a technical and from a practical point of view, in that it greatly enhances the applicability of our system. Designing an access control framework mostly independent from the adopted stream engine has required the definition of the core query model and a substantial re-design of the secure operators and the secure rewriting algorithm defined in [Carminati et al. 2007a]. Additionally, differently from [Carminati et al. 2007a], the framework presented in this paper is equipped with a module able to deploy the rewritten query into different DSMSs. Finally, differently from [Carminati et al. 2007a], we have implemented a prototype and carried out a performance evaluation study.

The remainder of this paper is organized as follows. Next section presents the architecture of our framework, whereas the core query model is illustrated in Section 3. Section 4 presents the access control model, and Section 5 focuses on query rewriting. Section 6 shows how the rewritten queries can be deployed in both
StreamBase and Coral8. Section 7 discusses the prototype implementation we have developed and presents some experiments we have carried out to demonstrate the feasibility of our approach. Section 8 overviews the related work, and Section 9 concludes the paper. Finally the Appendix contains the proofs of the formal results presented in the paper.

2. OVERVIEW OF THE FRAMEWORK

The goal of the proposed framework is to provide a middleware able to enforce access control into several commercial data stream management systems. Achieving this goal requires to address several issues. The first issue arises from the fact that, differently from what has happened for RDBMSs, up to now no standard query language for DSMSs is still emerged. In contrast, each DSMS adopts its own language, with the result of several distinct languages (e.g., StreamSQL in StreamBase, CCL in Coral8). To enforce access control, we rewrite the submitted queries according to the specified access control policies. However, devising rewriting strategies suitable for all DSMSs without a standard query language is rather difficult. To overcome this problem, we have first identified an abstract query model, capturing operations common to most of the existing DSMS query languages. In particular, similarly to [Coral8 2008] and [StreamBase 2008], we model a query according to the data-flow paradigm. Therefore, rather than specifying a query according to the syntax of a specific language, we model a query as a loop-free directed graph. According to this representation, the nodes in the graph are the operations performed on the streams, whereas the edge connecting two nodes indicates the flow that tuples follow through the graph. In Section 3, we introduce the proposed core query model, by presenting the supported operators.

By exploiting the core query model and the access control model proposed in [Carminati et al. 2007b] (see Section 4), we develop a query rewriting mechanism (see Section 5) whose output can then be deployed into several DSMSs (see Section 6). In particular, our framework makes a user able to submit a query, formulated according to the core query model, and then it deploys the corresponding authorized query into several data stream management systems. By authorized query we mean the user query rewritten in such a way that the result contains all and only the tuples answering the original query and for which the user has the necessary authorizations according to the specified access control policies.

![Fig. 1. Architecture of our framework](image)

As depicted in Figure 1, our framework consists of three main components, namely, a GUI, the Query Rewriter and the Deployment Module. The first component provides a graphical environment by which users can define their queries, to be registered into the stream engines. The GUI supports all the operators of the core query model. The user query is then processed by the Query Rewriter component. More precisely, the Query Rewriter rewrites the query graph into a set of authorized graphs, that is, graphs giving in output all and only the authorized tuples satisfying the user query. To realize the Query Rewriter, we have designed a set of secure operators, inspired by those proposed in [Carminati et al. 2007a], and revisited according to the adopted core query model. Once the Query Rewriter has generated the authorized graphs, they have to be registered into the target stream engines, that is, StreamBase and Coral8. However, since the Query Rewriter produces authorized graphs independent from the DSMS selected for query execution, there is the need of an additional phase where the authorized graphs are translated according to the languages supported by StreamBase and Coral8, that is, StreamSQL and CCL, respectively. The component in charge of this task is the Deployment Module. For each authorized graph generated by the Query Rewriter, the Deployment Module provides a set of statements in the target query languages, such that their execution generates the same stream obtained by the authorized graph (see Section 6 for more details).

In the following, before going into the details of the Query Rewriter and Deployment Module, we illustrate the core query model and the supported access control model.

3. THE CORE QUERY MODEL

Although all DSMSs have their own query language, most of them are based on the SQL standard, which has been extended to support the inbound processing typical of DSMSs. Even if the languages adopted by the various DSMSs present some differences, it is still possible to identify several similarities. Indeed, all available systems support projection and selection over streams, as well as window-based operations, like join and aggregation. The aim of the core query model is to capture these similarities. However, before presenting the core query model, we need to introduce some preliminary notions on streams.

We model a stream as an append-only sequence of tuples with the same schema. In particular, in addition to standard attributes, denoted as $A_1, \ldots, A_n$, the stream schema contains a further attribute, denoted in the following as $ts$. Attribute $ts$ stores the time of origin of the corresponding tuple, thus it can be exploited to monitor attributes values over time. In the following, given a stream $S$, we denote with $\text{Att}(S)$ the set of attributes in $S$'s schema, and with $S.A_j$, attribute $A_j$ of stream $S$.

As introduced in Section 2, we adopt the data-flow paradigm to model queries. Therefore, we present the query model by introducing the set of supported operators, i.e., supported nodes in the query graph. In particular, the core query model supports the well-know relational operators, like selection and projection, plus additional operators defined to handle window-based operations. Moreover, it contains two further operators, useful for modeling a query as a graph, i.e., the $$ and $$.
and OUT operators.

In what follows, we provide a description of the supported operators. In particular, each operator has an associated set of parameters, conveying the information needed to evaluate the operator (for instance, the predicates to be evaluated in case of a selection operator). In addition to their descriptions, with some operators we also associate an algebraic expression.

**Input - IN operator.** The IN operator models the streams entering in the query. As such, the IN operator has no entering edges and only one exiting stream. The IN operator has two associated parameters: Name, containing the name of the input stream, and ATTs, storing the name of the attributes of the input stream.

**Output - OUT operator.** This operator generates the stream resulting from the evaluation of the query. It has only one entering edge and has no exiting edges, since it is assumed that all incoming tuples are passed directly to some external application, like in real DSMSs. The OUT operator has an associated parameter, called Name, containing the name of the resulting stream.

**Projection - \( \pi \) operator.** The projection operator performs the projection of streams according to selected attributes. We define the projection of a stream \( S \) over a set of attributes \( \{A_1, \ldots, A_n\} \in \text{Att}(S) \) as a stream \( S' \) consisting of all tuples of \( S \) from which attributes not belonging to \( \{A_1, \ldots, A_n\} \) have been pruned. In a query graph, the projection operator \( \pi \) has a unique entering edge, representing the stream over which the projection is performed, and generates a unique exiting edge, that is, the stream resulting from the projection. The \( \pi \) operator has a parameter ATTs, containing the set of attributes \( \{A_1, \ldots, A_n\} \) to be extracted. The expression corresponding to the projection of a stream \( S \) over a set of attributes \( \{A_1, \ldots, A_n\} \in \text{Att}(S) \) is \( \pi(A_1, \ldots, A_n)(S) \).

**Selection - \( \sigma \) operator.** This operator selects specific tuples within a stream. More formally, given a stream \( S \) and a predicate \( P \) over attributes in \( \text{Att}(S) \), we define the selection of \( S \) with respect to \( P \) as a stream \( S' \) consisting of all and only those tuples in \( S \) that satisfy predicate \( P \). Thus, the selection operator \( \sigma \) has a unique entering edge and a unique exiting edge, that is, the stream containing only the input tuples satisfying the selection predicate. The selection predicate is contained into the \( \sigma \) parameter EXPs, and it is expressed through an SQL-like syntax. The corresponding expression is \( \sigma(P)(S) \).

---

**Example 3.1.** Throughout the paper, we consider examples from the military domain. We assume that the streams are used to monitor positions and health conditions of platoon’s soldiers. Hereafter, we consider two data streams, Position and Health, with the following schemas: Position \((ts, \text{SID}, \text{Platoon}, \text{Pos})\), Health \((ts, \text{SID}, \text{Platoon}, \text{Heart}, \text{BPressure})\), where the \text{SID} and \text{Platoon} attributes store...
soldier’s and platoon’s identifiers, respectively, both in the Position and Health streams, the Pos attribute contains the soldier position, the Heart attribute stores the heart beats, whereas the BPressure attribute contains the soldier’s blood pressure value. Figure 2 represents a query graph returning the id of those soldiers whose blood pressure is greater than 160. The query graph consists of an IN operator modeling the input stream Health, connected with the σ operator, evaluating the condition BPressure > 160. The result of the σ operator enters the π operator, that projects attribute SID. This operator is connected to the OUT operator, which generates the resulting stream.

Aggregation - Σ operator. The core query model also provides an aggregate operator Σ, by which it is possible to apply aggregate functions over data streams. In DSMSs, the common strategy to implement aggregate operations as well as join over potentially infinite streams is to exploit sliding-windows. Sliding-windows are defined on the basis of two parameters: the size of the window, and the offset according to which the window is shifted. Therefore, the aggregate operator Σ is applied over a sequence of windows. Thus, given a stream S, an aggregate function F over an attribute A ∈ Att(S), and two natural numbers s and o, the aggregate operator Σ returns a stream containing a different aggregate value for each distinct sliding-window generated with size s and offset o. The Σ operator has a unique entering and exiting edge. The exiting edge contains the result of the aggregate operation over the defined sliding-windows. The aggregate operator has therefore the following parameters: F, A, s, and o, which model the aggregate function, the attribute over which the aggregate function is computed and the size and offset according to which the aggregate function is evaluated. We consider as aggregate functions only the standard SQL-style functions, i.e., min, max, count, avg, and sum. The algebraic expression corresponding to the Σ operator is Σ(F, A, s, o)(S).

Join - Join operator. This operator performs join over streams. In data stream engines, join is implemented by means of sliding-windows. More precisely, given two streams S_1 and S_2, the join is evaluated by performing the relational join between windows generated over S_1 and S_2. Thus, given two streams S_1 and S_2, the natural numbers s_1, o_1, and s_2, o_2, and a join predicate P expressed through an SQL-like syntax, the Join operator generates a stream S’ containing the tuples resulting by the relational join between the sliding-windows computed over S_1 and S_2, with s_1, o_1, s_2, o_2, as size and offset, respectively. The join operator, therefore, has two entering edges, i.e., S_1 and S_2, and one existing edge, i.e., S’. The join predicate is contained into parameter EXPs. The algebraic expression corresponding to the Join operator is Join(P, s_1, o_1, s_2, o_2)(S_1, S_2). Since s_1, o_1, s_2, o_2 are not relevant from an access control point of view, in the following, we omit them, by using the simplified syntax: Join(P)(S_1, S_2).

Example 3.2. A query graph generating the average of heart beats of those soldiers which are across some border k (modeled as Pos ≥ k) is represented in Figure 3. Since the position of a soldier is stored in the Position stream, whereas health information is stored in the Health stream, calculating the heart beat average requires

1For simplicity, here and in the following we omit the ATTs parameter of the IN operator.

to perform a join of the Position and Health streams, with predicate Position.SID = Health.SID, and then to select only those tuples with Pos ≥ k. Thus, the graph contains two IN operators, representing the Health and Position streams, which enter into the Join operator, whose predicate in EXPs is equal to Position.SID = Health.SID. The result of the Join operator enters into a σ operator having the predicate Pos ≥ k. Over the result of the selection, an aggregate operator Σ is evaluated.

The resulting tuples flow directly into the OUT operator.

A summary of the operators supported by the core query model is reported in Table I.

### 4. ACCESS CONTROL MODEL

In this section, we introduce the access control model on which our framework relies [Carminati et al. 2007b]. Our access control model is a role-based access control model specifically tailored to the protection of data streams. Privileges supported by the model are of two different types, which correspond to the two different classes of operations provided by the core query model: a read privilege that authorizes a user to apply the π, σ and Σ operators on a stream, that is, all operations that require to read tuples from a data stream. Additionally, it authorizes to apply the Join operator if the read privilege is granted on both the operand streams. The other class of privileges supported by our model, called aggregate privileges, corresponds to the aggregate functions allowed by the core query model. Such

---

**Table I. Core query model operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Algebraic expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>-</td>
<td>Stream entering the query graph</td>
</tr>
<tr>
<td>OUT</td>
<td>-</td>
<td>Stream resulting from the query graph</td>
</tr>
<tr>
<td>π</td>
<td>π(A₁,...,Aₙ)(S)</td>
<td>Projection of stream S over attributes {A₁,...,Aₙ} ∈ Att(S)</td>
</tr>
<tr>
<td>σ</td>
<td>σ(F)(S)</td>
<td>Selection of stream S with respect to predicate F</td>
</tr>
<tr>
<td>Σ</td>
<td>Σ(F,A,s,o)(S)</td>
<td>Aggregation of attribute A of stream S according to function F over sliding-windows generated with size s and offset o</td>
</tr>
<tr>
<td>Join</td>
<td>Join(P,s₁,o₁,s₂,o₂)(S₁,S₂)</td>
<td>Join with respect to predicate F over tuples of sliding-windows of stream S₁ (i.e., S₂) generated with size s₁ (i.e., s₂) and offset o₁ (i.e., o₂)</td>
</tr>
</tbody>
</table>

---

2Here and in the following, for simplicity, parameters related to sliding-windows are omitted from the query graph.

privileges are provided to grant a user the authorization to perform an aggregate operation, without having the right to access all the tuples over which the operation is performed. Thus, the aggregate privileges are: \texttt{min, max, count, avg, and sum}.

Privileges can be specified for whole streams, as well as for a subset of their attributes and/or tuples, where the set of authorized tuples is specified by defining a set of conditions on stream attributes values. Additionally, the model allows the Security Administrator (SA) to restrict the exercise of the \texttt{read} privilege only to a subset of a stream resulting from the join operator. This is a useful feature since sometimes a user should be allowed to access only selected attributes in a joined stream (as shown by Example 3.2). To model such a variety of granularity levels, we borrow some ideas from how access control is enforced in traditional RDBMSs, where different granularity levels are supported through views. The idea is quite simple: define a view satisfying the access control restrictions and grant the access on the view instead of on base relations. In a RDBMS, a view is defined by means of a CREATE VIEW statement, where the SELECT clause of the query defining the view specifies the authorized attributes, the FROM clause specifies a list of relations/views, and the WHERE clause states conditions on attributes’ values to be satisfied by the tuples contained into the view. We adopt the same idea to specify protection objects to which an access control policy applies. However, since a standard query language for data streams has not yet emerged, we give a language independent representation of protection objects. Basically, we model a protection object by means of three components, which correspond to the SELECT, FROM and WHERE clauses of an SQL query statement. The formal definition of protection object specification is given below.

\textbf{Definition 4.1. (Protection object specification) [Carminati et al. 2007b].} A protection object specification \(p_{\text{obj}}\) is a triple \((\text{STRs}, \text{ATTs}, \text{EXPs})\), where:

- \text{STRs} is a set of names or identifiers of streams \(\{S_1, \ldots, S_n\}\);
- \text{ATTs} denotes a set of attributes \(A_1, \ldots, A_l\), where \(A_j, j \in \{1, \ldots, l\}\), belongs to the schema of the stream resulting from the Cartesian product \((S_1 \times \ldots \times S_n)\) of the streams in \text{STRs}. If \text{ATTs} is equal to symbol '*\', it denotes all the attributes belonging to the schema of the stream resulting from \((S_1 \times \ldots \times S_n)\).
- \text{EXPs} is a boolean formula, built over predicates of the form: \(A_i \oplus \text{value}_i\) or \(A_i \oplus A_j\), where \(A_i, A_j\) are attributes belonging to the schema of the Cartesian product \((S_1 \times \ldots \times S_n)\), \(\oplus\) is a comparison operator, and \text{value}_i\) is a value compatible with the domain of \(A_i\). If \text{EXPs} is omitted, it denotes all the tuples in \((S_1 \times \ldots \times S_n)\).

The access control model also allows the SA to specify two different types of temporal constraints, that is, \texttt{general} and \texttt{window-based} constraints. Constraints of the first kind state limitations on the time during which users can exercise privileges on protection objects. They are expressed in the form: \([\text{begin}, \text{end}]\), where \text{begin} and \text{end} are the lower and upper bounds of the interval, \text{begin} \leq \text{end}, and \text{end} can assume the infinite value.\(^3\) The \text{begin} and \text{end} values can be explicitly specified by the SA, or they can be returned by a predefined set of system functions \(SF\).

\(^3\)We assume that \text{begin} and \text{end} values are specified by means of an SQL-like syntax.
Table II. Examples of access control policies for data streams

For instance, we assume a function `start()`, which receives as input an action and returns the time when the action starts, and a function `end()`, which receives as input an action and returns the time when a given action ends. Since, by definition, a stream always contains a temporal information, i.e., the timestamp `ts`, a general time constraint `gtc` identifies all and only those tuples satisfying the predicate: `ts ≥ begin ∧ ts ≤ end`. The other class of constraints is related to window-based aggregate operators supported by the core query model. In particular, these constraints are used to limit the sliding-windows over which an aggregate operator can be evaluated. This allows the SA to constraint the accuracy of the returned aggregated values on the basis of the confidentiality of raw data. A window time constraint `wtc` is therefore defined by a pair: `[s, o]`, denoting the minimum size `(s)` and offset `(o)` allowed in an aggregate operation. The value 0 for size and/or offset denotes that the corresponding aggregate operation can be performed with any size and/or offset.

The formal definition of access control policies for data streams is given below.

**Definition 4.2. (Access control policy for data streams)** [Carminati et al. 2007b]. An access control policy for data streams is a tuple: `(sbj, obj, priv, gtc, wtc)`, where: `sbj` is a role, `obj` is a protection object specification defined according to Definition 4.1, `priv` ∈ {read, min, max, count, avg, sum} is an access privilege, `gtc` is a general time constraint, and `wtc` is a window time constraint.

Given an access control policy `acp` we denote with `acp.sbj`, `acp.obj`, `acp.priv`, `acp.gtc` and `acp.wtc` the `sbj`, `obj`, `priv`, `gtc`, and `wtc` component, respectively. Moreover, given a protection object specification `acp.obj`, we use the dot notation to refer to its components. We assume that all the specified access control policies are stored into a unique authorization catalog, called `SysAuth`. `SysAuth` contains a different tuple for each access control policy, whose attributes store the access control policy components, as illustrated by the following example.

**Example 4.1.** Table II presents an example of `SysAuth` catalog containing four access control policies defined for the Doctor role and referring to the Position and Health streams introduced in Example 3.1. The first access control policy authorizes doctors to access the position and id of soldiers belonging to their platoons (this condition is modeled as: `Position.Platoon=self.Platoon`). The second access control policy authorizes doctors to compute the average of the positions of those

---

4We assume that each user has an associated profile, i.e., a set of attributes modeling his/her characteristics, like for instance the platoon one belongs to.
soldiers not belonging to their platoons. This privilege is granted only during the time of action a. Moreover, this policy states that the average can be computed only with windows of minimum 1 hour and with 1 as minimum offset. By the third policy, doctors are authorized to monitor the health conditions (i.e., all attributes of Health stream) only of those soldiers belonging to their platoons. Finally, the fourth access control policy allows doctors to monitor blood pressure, position, and id of those soldiers not belonging to their platoons, but whose position is near to the target of action a, that is, whose positions are distant at most δ from action a target’s position (Pos ≥ target(a) - δ ∧ Pos ≤ target(a) + δ).5

Finally, in the remainder of the paper we need to formally denote the tuples identified by the protection object specification of an access control policy acp. These are defined by function β(), presented next.

Definition 4.3. (Protection object specification semantics). Given an access control policy acp, the protection object specification semantics of acp is given by function β, defined as follows:

- if |acp.obj.STRs| = 1, then β(acp) = π(A1, ..., An)(σ(acp.obj.EXPs ∧ ts ≥ acp.gtc.begin ∧ ts ≤ acp.gtc.end)(acp.obj.STRs)), otherwise
- β(acp) = π(A1, ..., An)(σ(acp.obj.EXPs ∧ ts ≥ acp.gtc.begin ∧ ts ≤ acp.gtc.end)(Cartesian(S1, ..., Sn)), Sj ∈ acp.obj.STRs, ∀j ∈ [1, n]);

where Cartesian() takes as input a set of streams and returns their Cartesian product; whereas {A1, ..., An} = acp.obj.ATTs. If acp.obj.ATTs = *, then {A1, ..., An} are all the attributes of streams belonging to acp.obj.STRs.

Example 4.2. Let us consider the protection object specification of the fourth access control policy in Table II. According to Definition 4.3, this denotes the tuples returned by the algebraic expression π(BPressure, SID, Pos)(σ(Position.SID = Health.SID ∧ Pos ≥ target(a) - δ ∧ Pos ≤ target(a) + δ)(Cartesian(Position, Health, Position))), that is, those tuple resulting from the Cartesian product of Position and Health, where only BPressure, SID, and Pos attributes are projected. The condition expressed by the EXPs component ensures that only tuples having Position.SID = Health.SID (i.e., join predicate) and referring to soldiers whose position is close to the target position are considered (i.e., Pos ≥ target(a) - δ ∧ Pos ≤ target(a) + δ).

5. QUERY REWRITER

As introduced in Section 2, the Query Rewriter module enforces the access control policies, specified according to the access control model presented in Section 4, over query graphs, expressed according to the core query model presented in Section 3. In particular, given a query graph G and a user u, the Query Rewriter rewrites G such that the evaluation of the obtained graphs, called authorized graphs, generates only tuples answering the original query graph G and for which there exists an access control policy authorizing u the access. On support of the Query Rewriter we design a set of novel secure operators, which filter out from the result of the
corresponding (not secure) operators all not authorized tuples. In the following, we first introduce the secure operators. Then, we show how these operators are used for query rewriting.

5.1 Secure operators

Secure operators applied over a stream in a query graph filter out from it all not authorized tuples. To do this it is first necessary to identify the access control policies that apply to a stream in a query graph. Due to the flexibility of our access control model in defining protection objects, the task of retrieving the access control policies that apply to an internal stream, that is, a stream generated by a portion of a query graph, is more complicated than in conventional RDBMs. Usually, in a RDBMS users can submit queries over base relations or predefined views. Thus, in RDBMs the task of policies retrieval is very simple. It is only necessary to retrieve the access control policies that apply to the target relations/views. In contrast, to allow for more flexibility and to make easier query specification, we have decided to provide the user the ability to define its own graph, without the need of referring to pre-defined views. However, this makes the retrieving of access control policies applying to a stream more difficult. The difficulty relies on the fact that the protection object in an access control policy and the streams in a graph have different representations. Indeed, the first is specified according to Definition 4.1, whereas the second is modeled as a (portion of) query graph. To simplify policies retrieval, we assume that all streams in a query graph (that is, input, output and internal streams) are denoted by means of a specification similar to the one given by Definition 4.1. Thus, we denote each stream $S$ in a query graph by means of three components: $S_{.STRs}$, $S_{.ATTs}$ and $S_{.EXPs}$. According to this stream specification, an input stream $S$ can be denoted by simply setting the $ATTs$ component to $*$ and omitting the $EXPs$ component. In contrast, since an internal and output stream $S$ is defined in terms of (the portion of) the graph $G$ by which it results, its representation can be defined in terms of the operators contained into $G$. More precisely, given a graph $G$ defined over a set $S_{.in}$ of input streams, we can define the stream $S'$ resulting from graph $G$, as the Cartesian product of streams in $S_{.in}$, where all attributes specified in the $\pi$ and $\Sigma$ operators of the graph $G$ are projected, and all predicates specified in the $\sigma$ and $Join$ operators are applied (see Algorithm 1 for more details).

Denoting streams with a protection object like representation greatly helps in retrieving the access control policies applying to a stream. In the following, this task is performed by function $Pol()$, properly defined for each secure operator.

Let us now introduce the first operator, called Secure view. It takes as input an input stream and an access control policy and it returns the “view” of the stream that can be accessed according to the policy. This view may contain only selected attributes and/or tuples of the input stream, on the basis of the protection object specification contained into the access control policy. The view is represented by the corresponding algebraic expression.

\textit{Definition 5.1. (Secure view).} Let $S$ be an input stream, and $acp$ be an access control policy that applies to $S$, such that $acp.obj_{.STRs}=S_{.STRs}$. The secure view, $SecView$ of $S$ with respect to policy $acp$ is defined as follows:
A Framework to Enforce Access Control over Data Streams

\[
\text{Sec View}(S, acp) = \pi(\text{att})(\sigma(P)(S))
\]

where:

\[
\text{att} = \begin{cases} 
S.\text{ATTs} \cap acp.\text{obj}.\text{ATTs} & \text{if } acp.\text{obj}.\text{ATTs} \neq * \\
S.\text{ATTs} & \text{otherwise}
\end{cases}
\]

\[
P = \text{exp} \land \text{window}
\]

where:

\[
\text{exp} = \begin{cases} 
\{ (acp.\text{obj}.\text{EXPs}) & \text{if } acp.\text{obj}.\text{EXPs} \text{ is not omitted} \\
\text{true} & \text{otherwise}
\end{cases}
\]

\[
\text{window} = \begin{cases} 
(ts \geq acp.\text{gtc}.\text{begin} \land ts \leq acp.\text{gtc}.\text{end}) & \text{if } acp.\text{gtc} \text{ is not null} \\
\text{true} & \text{otherwise}
\end{cases}
\]

Based on the \text{Sec View} operator, we next define the \text{Sec Read} operator, which takes as input a user \( u \) and an input stream \( S \), and returns the view of \( S \) over which \( u \) can exercise the read privilege according to the policies in \text{SysAuth}. Note that, since more than one policy can apply to the same user on the same stream (referring for instance to different attributes and/or with different conditions over tuples) the result of \text{Sec Read} is actually a set of views, each of which denoted by the corresponding algebraic expression. Given a user \( u \), in the following we denote with \( \text{Role}(u) \) the set of roles \( u \) is authorized to play.

\textbf{Definition 5.2. (Secure read).} Let \( S \) be an input stream and \( u \) be a user. Let \( \text{Pol}(S, u) \) be the set of read access control policies in \text{SysAuth} specified for \( S \) and which apply to \( u \), that is, \( \text{Pol}(S, u) = \{acp \in \text{SysAuth} | acp.\text{obj}.\text{STRs} = S.\text{STRs}, acp.\text{obj} \cap \text{Role}(u) \neq \emptyset, acp.\text{priv} = \text{read} \} \). The secure read operator, \text{Sec Read}, is defined as follows:

\[
\text{Sec Read}(S, u) = \bigcup_{acp_j \in \text{Pol}(S, u)} \{\text{Sec View}(S, acp_j)\}
\]

\textbf{Example 5.1.} Let us consider the access control policies in Table II, and suppose that there exists a user, say \text{Paul}, belonging to platoon \text{X} and authorized to play the doctor role. Let us see the view resulting from the evaluation of \text{Sec Read}(\text{Position}, \text{Paul})\text{, denoted in what follows as AuthView. According to Definition 5.2, AuthView is defined as the union of the views returned by the secure view operator, for each policy in Pol(Position,Paul). Pol(Position,Paul) returns only the first access control policy of Table II, say \text{acp}. Thus, AuthView consists of Sec View(Position,acp). According to Definition 5.1, Sec View(Position,acp) returns the following expression: } \pi(\text{Pos, SID})(\sigma(\text{Position.Platoon=\text{X}})(\text{Position})). Thus, the view of the Position stream on which Paul has the \text{read} privilege consists of the position and id of soldiers belonging to his platoon.

Our access control model allows one to specify policies for aggregate privileges. We therefore need to define a further operator, called secure aggregate, which, given an aggregate operator over a stream \( S \) and a user \( u \), considers the policies applying to \( u \) and specified over \( S \) for the requested aggregate operation, and returns the result of the aggregate operation only over the “view” authorized by these policies. As for the previously defined operators, the view may actually be a set of views, each of which denoted by an expression of the adopted core query model. Since

\*True denotes a predicate that is always satisfied.
in the case of aggregate operations both policies and operations may have some associated temporal constraints (i.e., the window size and step), these must be considered when determining the result of secure aggregate.

**Definition 5.3. (Secure aggregate)** Let \( S \) be a stream, \( u \) be a user, \( F \) be an aggregate function, \( A \) be an attribute of \( S \). Let \( s \) and \( o \) be two natural numbers, representing the size and the offset, respectively, according to which the aggregate operation is required. Let \( Pol_{agg}(S,u) \) be the set of access control policies in SysAuth to be considered to evaluate \( u \) request to perform \( F \) over attribute \( A \). More formally, \( Pol_{agg}(S,u) = \{ \text{acp}\in\text{SysAuth}| \text{acp.obj.STRs}=S.\text{STRs}, A\in\text{acp.obj.ATTRs}, \text{acp.obj}\cap\text{Role}(u)\neq\emptyset, \text{acp.priv}=F \land \forall\text{exp}\in\text{S.EXPs}, \exists\text{exp}'\in\text{acp.obj.EXPs}, \text{Such that exp} \subseteq \text{exp}' \} \). The secure aggregate operator, \( \text{Sec}_\text{Aggr} \) is defined as follows:

\[
\text{Sec}_\text{Aggr}(S,F,A,s,o,u)=\bigcup_{\text{acp}\in\text{P}_{\text{Pol}_{\text{agg}}(S,u)}}\{\pi(S.|P|)\}
\]

where:

- \( \text{max}_{\text{size}} = \max(\text{acp}_j\text{.wtc.size},s) \),
- \( \text{max}_{\text{offset}} = \max(\text{acp}_j\text{.wtc.offset},o) \), and
- \( P = \text{exp} \land \text{window} \)

where:

- \( \text{exp} = \begin{cases} \text{true} & \text{if acp}_j\text{.obj.EXPs is not omitted} \\ \text{false} & \text{otherwise} \end{cases} \)
- \( \text{window} = \begin{cases} \text{true} & \text{if acp}_j\text{.gtc is not null} \\ \text{false} & \text{otherwise} \end{cases} \)

Let us explain how policies are selected by function \( P_{\text{Pol}_{\text{agg}}}() \). According to the proposed access control model, if there exists an access control policy \( \text{acp} \) granting user \( u \) the aggregate privilege \( F \) over the protection object \( \text{acp.obj} \), this implies that \( F \) can be computed over all and only the tuples denoted by \( \text{acp.obj} \). Indeed, since the aggregate operator returns statistical data, allowing a user to evaluate the aggregate function over a subset of the tuples denoted by \( \text{acp.obj} \) might return more precise statistical data, which could be potentially confidential. Thus, retrieving the access control policies specified over \( S \) for the aggregate function \( F \) to be considered by the secure aggregate operator requires to determine all the policies \( \text{acp} \) such that the stream \( S \) includes the stream denoted by the protection object specification of \( \text{acp} \), that is, such that the tuples in \( \beta(\text{acp}) \) (cfr. Definition 4.3) are a subset of the tuples produced by \( S \). Then, the selected access control policies are enforced by the secure aggregate operator, by pruning from stream \( S \) the not authorized tuples, thus ensuring that the aggregation is evaluated only on tuples denoted by \( \text{acp.obj} \). Since \( S \) could be an internal stream generated by a graph \( G \), verifying whether \( S \) includes the tuples in \( \beta(\text{acp}) \) requires to check a set of conditions. A first condition is that the names of the streams over which \( \text{acp} \) is specified (i.e., \( \text{acp.obj.STRs} \)) are equal to the names of the streams over which \( S \) is generated (i.e., \( S.\text{STRs} \)). Furthermore, in order to ensure that stream \( S \) includes the tuples in \( \beta(\text{acp}) \), it is required that, for each expression in \( \text{exp} S.\text{EXPs} \), there exists

---

The \( \subseteq \) operator verifies whether the expressions \( \text{exp} \) and \( \text{exp}' \) generate two streams \( S' \) and \( S \), respectively, such that \( S \) is included in \( S' \).
aggr(Position, avg, Pos, Paul, 5, 5)

view
join(S

asition with windows of 5 hours and 5 as offset. Moreover, let us assume that action

pol

retumed by the secure aggregate operator. In this case, Polagg(Position, Paul) consists only of the second access control policy of Table II. According to this access control policy, Paul is authorized to perform avg on the Pos attribute only during action a and for the soldiers not belonging to his platoon. Moreover, the average can be performed with at minimum a window of size 1 hour and 1 as offset. Thus, SecAggr(Position, avg, Pos, Paul, 5, 5)(π(Pos)(σ(Position.Platoon≠ X ∧ ts ≥ 105000 ∧ ts ≤ ∞)(Position))), since 5 is the maximum size (resp. offset) between the size (resp. offset) specified in the access control policy and the required one. Thus, the secure aggregate operator considers only the Pos attribute of those tuples in the Position stream satisfying predicate: Position.Platoon≠X, that is, tuples of soldiers not belonging to Paul’s platoon, and such that: ts≥105000∧ts≤∞, that is, tuples generated during action a. Then, for those values, it calculates the average with windows of size 5 hours and with 5 as offset.

The last operator we need to define, called secure join, is used to manage join operations. Indeed, according to our access control model, it is possible to specify policies that apply to the join of two or more streams, by authorizing the access only to selected attributes and/or tuples in the joined stream. These policies have more than one stream in the obj.STRs component. Similarly to SecAggr, the secure join operator returns the set of “views” over the joined stream corresponding to the authorized attributes and/or tuples.

Definition 5.4. (Secure join). Let S1 and S2 be two streams, P a join predicate over S1 and S2, and u be a user. Let Poljoin(S1, S2, u) be the set of read access control policies in SysAuth applying to u specified for joins over S1 and S2. Let Att be the set of attributes over which predicates in S1.EXP and S2.EXP are defined. More formally, Poljoin(S1, S2, u)={acp∈SysAuth| acp.obj.STRs = S1.STRs \cup S2.STRs, P ∈ acp.obj.EXP s, Att ∈ acp.obj.ATT s, acp.sbj ∩ Role(u) ≠ ∅, acp.priv = read}. The secure join operator, SecJoin, is defined as follows:

SecJoin(S1, S2, P, u) = \bigcup_{acp_j∈Poljoin(S1, S2, u)}{SecView(J oin(P)(S1, S2), acp_j)}.

Similarly to the secure aggregate operator, Poljoin(S1, S2, u) selects the access control policies that need to be evaluated for the requested join operation. In particular, it considers only those access control policies acp whose protection object specification includes the predicate P of the required join. Then, it checks that the streams’ names over which acp is specified (i.e., acp.obj.STRs) are equal to the streams’ names over which S1 and S2 are generated. Moreover, it verifies whether the attributes over which predicates of streams S1 and S2 are defined are contained

A Framework to Enforce Access Control over Data Streams · 15

into the authorized attributes (i.e., \texttt{acp.obj.ATTs}). This check avoids possible inferences. Indeed, without this condition, a malicious user could insert predicates over not authorized attribute (by means of $\sigma$ operator) just before the join operator. In this way, even if the secure join operator returns only the authorized attributes, since the predicates over not authorized attributes are evaluated before the secure join operator the user could infer sensitive data, for instance, values of not authorized attributes.

**Example 5.3.** Suppose now that Paul is interested to monitor the health conditions of those soldiers which are across some border $k$ (modeled as $\text{Pos} \geq k$). Since the position of a soldier is stored in the \texttt{Position} stream, whereas health information is in the \texttt{Health} stream, the first operation he needs to perform is the join of the \texttt{Position} and \texttt{Health} streams, with predicate $\text{Position.SID} = \text{Health.SID}$. Let us now see how the secure join operator evaluates over the requested query, that is, $\text{Sec}_\text{Join}(\text{Position}, \text{Health}, \text{Position.SID} = \text{Health.SID}, \text{Paul})$. According to Definition 5.4, the secure join operator first generates the joined stream, say $\mathcal{S}$, resulting from $\text{Join}(\text{Position.SID} = \text{Health.SID})(\text{Position}, \text{Health})$. Then, for each policy $\text{acp}$ in $\text{Pol}_\text{join}(\text{Position}, \text{Health}, \text{Paul})$ it evaluates the secure view operator over $\mathcal{S}$ and $\text{acp}$. By Definition 5.4, $\text{Pol}_\text{join}(\text{Position}, \text{Health}, \text{Paul})$ contains only the fourth policy in Table II, say $\text{acp}_4$. According to this access control policy, the required join is possible only for those tuples referring to soldiers whose positions are near to the target of action $a$. Let us assume that action $a$ is not currently undergoing, that is, $\text{target}(a)$ returns null. $\text{Sec}_\text{Join}$ returns the view generated by $\text{Sec}_\text{View}(\mathcal{S}, \text{acp}_4)$. By Definition 5.1, this view is given by the following expression: $\pi(\text{BPressure}, \text{SID}, \text{Pos})(\sigma(\text{Position.SID} = \text{Health.SID} \land \text{Pos} \geq \text{null} - \delta \land \text{Pos} \leq \text{null} + \delta)(\text{Position}, \text{Health}))$. Since the last predicate in the above $\sigma$ operator evaluates to null, no tuples are selected, thus the authorized view is empty.

**5.2 Secure query rewriting**

We recall that given a query graph $\mathcal{G}$ and a user $u$, the Query Rewriter rewrites $\mathcal{G}$ into a set of authorized graphs $\mathcal{AG}$, on the basis of the access control policies applying to $u$. In what follows, we refer to this task as **secure query rewriting**. In particular, secure query rewriting has to ensure that each authorized graph $ag \in \mathcal{AG}$ generates only tuples that: (a) answer the original query graph $\mathcal{G}$; and (b) there exists an access control policy authorizing $u$ the access on the information contained into the tuple. In this section, we illustrate our approach for secure query rewriting, which makes use of the secure operators introduced in Section 5.1. Moreover, we prove the correctness and completeness of the proposed secure query rewriting algorithm.

A naïve way to rewrite the input query is to rewrite $\mathcal{G}$ by simply pre-processing all entering tuples in order to filter out not authorized tuples. Thus, the graph processes only authorized tuples and, as a consequence, it generates only authorized tuples. This pre-processing enforcement can be easily implemented by means of the secure read operator introduced in Section 5.1. Indeed, the secure read operator takes as input an input stream $\mathcal{S}$ and a user $u$ and returns only those tuples of $\mathcal{S}$ for which $u$ has the read access. Thus, by simply inserting the secure read operator just after each $\text{IN}$ operator contained into the query graph, we are able to ensure...
that the graph is flowed only by authorized tuples.

However, as also discussed in the introduction, this simple pre-processing strategy is not enough, in that it is not able to correctly enforce all the access control policies supported by our access control model. The main problem is that pre-processing works well only for access control policies granting the read privilege on input streams. Whereas it can prevent authorized accesses for access control policies granting aggregate privileges or applying to a join result. The following example clarifies the point.

**Example 5.4.** Suppose once again that Paul is interested to monitor the health conditions of those soldiers which are across some border \( k \) (modeled as \( \text{Pos} \geq k \)). Thus, he needs to perform the join of Health and Position streams with predicate \( \text{Health.SID} = \text{Position.SID} \), and then to select from the resulting stream those tuples with \( \text{Pos} \geq k \). To perform this query he submits a query graph similar to the one in Figure 3, without the \( \Sigma \) operator. According to the pre-processing strategy, the authorized graph \( \text{ag}_1 \) (see Figure 4) is obtained by inserting the secure read operators just after the \( \text{IN} \) operators in \( G \) (i.e., \( \text{Sec.Read(Health,Paul)} \) and \( \text{Sec.Read(Position,Paul)} \)). According to Definition 5.2, they filter out all the tuples of Health and Position streams that Paul is not authorized to read. More precisely, \( \text{Sec.Read(Position,Paul)} \) and \( \text{Sec.Read(Health,Paul)} \) filter out all tuples referring to those soldiers not belonging to Paul’s platoon (see the first and the third access control policies in Table II). Moreover, they filter out not authorized attributes. As a consequence, the join is evaluated only over those tuples of Position and Health streams referring to soldiers belonging to Paul’s platoon. However, according to the fourth access control policy in Table II, under some conditions, Paul

![Figure 4. Authorized graph \( \text{ag}_1 \)](image)

![Figure 5. Authorized graph \( \text{ag}_2 \)](image)
is authorized to evaluate the join operation also over tuples referring to soldiers not belonging to his platoon. The resulting tuples are not contained by the authorized graph $ag_1$.

The problem pointed out by Example 5.4 is due to the fact that in our access control model, a join operation can be authorized by two different kinds of access control policies. Indeed, a user $u$ is authorized to apply a join over two streams $S_1, S_2$ if: (a) $u$ has the read privilege over both the streams, or (b) there exists an access control policy $acp$ granting the read privilege over a set of tuples that includes those contained into the stream resulting from the join operation. Case (a) can be easily handled by pre-processing. In contrast, case (b) can not be enforced by pre-processing, since the secure read operator could filter out from $S_1$ and/or $S_2$ some tuples/attributes authorized by $acp$.

Similar considerations are also applicable to aggregate operations. Indeed, according to the proposed access control model, a user is authorized to evaluate an aggregate operation on a stream if: (a) he/she has a read privilege over the stream, or (b) there exists an access control policy $acp$ granting the aggregate privilege such that tuples in $\beta(acp)$ are included into the target stream. Case (a) is easily handled by pre-processing approach, i.e., by inserting the secure read operator just after the $IN$ operator of the target stream. However, similarly to what happens for the join operator, pre-processing does not work for case (b). Therefore, to correctly enforce access control policies granting the right to perform join and aggregate operations, we make use of secure join and secure aggregate introduced in Section 5.1. Thus, in addition to the authorized graphs created according to the pre-processing strategy, we create further authorized graphs, obtained by complementing the join and the aggregate operators in the original graph $G$ with the corresponding secure operators. More precisely, to correctly enforce access control policies granting the right to perform a join, the authorized graph is obtained by inserting into the original graph $G$ the secure join operator just after the join operator. In contrast, to enforce access control policies granting the privilege to perform aggregate operations, the authorized graph is generated by replacing the aggregate operator with the corresponding secure version, which, by definition, evaluates the aggregation only on authorized tuples.

**Example 5.5.** Let us consider again Example 5.4. To correctly enforce the fourth policy in Table II it is necessary to define a further authorized graph $ag_2$ by exploiting the secure join operator (see Figure 5). This is obtained by inserting into $G$ the operator $Sec\_Join(Health,Position,Health.SID=Position.SID,Paul)$ just after the join operator. By Definition 5.4, the secure join filters from the result of the join of $Health$ and $Position$ with predicate $Health.SID=Position.SID$, the tuples for which $Paul$ does not have the read privilege, that is, tuples that do not satisfy the protection object specification of the fourth policy in Table II.

Algorithm 1 implements the proposed secure query rewriting strategy. The algorithm takes as input a query graph $G$ and the user $u$ submitting the query, and returns a set of authorized graphs $AG$ and the protection object like representation of graph $G$. As it will be clarified later on, the second output is needed during the recursive evaluation of graph containing join and aggregate operators. In general,
Algorithm 1: Sec\_Rewr(G,u)

1. Let OP be the operator directly connected to the OUT operator.
2. Initialize AG, AG\_1, AG\_2, AG\_3, AG\_4 to the empty set.
3. case OP = IN
4.  \  \ Authentication = Sec\_Read(IN.Name, u);
5.  \  \ for each av ∈ Authentication do
6.  \  \ \  \ Let ag be a copy of av;
7.  \  \ \  \ ag = Insert(ag, av);
8.  \  \ AG = AG ∪ ag;
9.  \  \ obj.STRs = IN.name, obj.ATTs = IN.ATTs;
10.  \  \ Return (AG.obj);
11. case OP = a or OP = σ
12.  \  \ Let G be a copy of G;
13.  \  \ Delete from G the OUT operator, and replace OP with OUT;
14.  \  \ (Ret\_AG, obj) = Sec\_Rewr(G, u);
15.  \  \ for each ag ∈ Ret\_AG do
16.  \  \ \  \ Replace OUT in ag with OP and add OUT at the end of ag;
17.  \  \ AG = AG ∪ ag;
18.  \  \ if OP = π then
19.  \  \ \  \ obj.ATTs = obj.ATTs \cap OP.ATTs;
20.  \  \ if OP = σ then
21.  \  \ \  \ obj.EXP = obj.EXP \cup OP.EXP;
22.  \  \ Return (AG.obj);
23. case OP = join
24.  \  \ Let G\_1 and G\_2 be two copies of G;
25.  \  \ Delete OUT from G\_1 and G\_2;
26.  \  \ Delete OP and the whole subgraph generating the first operand of OP from G\_1, add OUT at the end of G\_1;
27.  \  \ Delete OP and the whole subgraph generating the second operand of OP from G\_2, add OUT at the end of G\_2;
28.  \  \ (Ret\_AG\_1, obj) = Sec\_Rewr(G\_1, u);
29.  \  \ (Ret\_AG\_2, obj) = Sec\_Rewr(G\_2, u);
30.  \  \ for each ag\_1 ∈ Ret\_AG\_1 do
31.  \  \ \  \ for each ag\_2 ∈ Ret\_AG\_2 do
32.  \  \ \  \ Delete OUT from ag\_1 and ag\_2;
33.  \  \ \  \ Create a new graph ag\_j consisting of OP applied over ag\_1 and ag\_2, add OUT at the end of ag\_j;
34.  \  \ \  \ \  \ AG = AG ∪ ag\_j;
35.  \  \ Authentication = Sec\_Join(obj\_1, obj\_2, OP.EXP, u);
36.  \  \ for each av ∈ Authentication do
37.  \  \ \  \ Let ag\_j be a copy of G;
38.  \  \ \  \ ag\_j = Insert(ag\_j, av), AG\_j = AG\_j ∪ ag\_j;
39.  \  \ \  \ AG = AG ∪ AG\_j;
40.  \  \ \  \ obj.STRs = obj\_1.STRs ∪ obj\_2.STRs; obj.ATTs = obj\_1.ATTs ∪ obj\_2.ATTs; obj.EXP = obj\_1.EXP ∪ obj\_2.EXP;
41.  \  \ \  \ Return (AG.obj);
42. case OP = Σ
43.  \  \ Let G be a copy of G;
44.  \  \ Delete OUT from G and replace OP with OUT in G;
45.  \  \ (Ret\_AG, obj) = Sec\_Rewr(G, u);
46.  \  \ for each ag ∈ Ret\_AG do
47.  \  \ \  \ Replace OUT with OP in ag, Add OUT to ag;
48.  \  \ \  \ AG = AG ∪ ag;
49.  \  \ Authentication = Sec\_Aggre(G, OP, P, OP, A, OP, S, OP, o, u);
50.  \  \ for each av ∈ Authentication do
51.  \  \ \  \ Let ag be a copy of G;
52.  \  \ \  \ Delete OP from ag, ag = Insert(ag, av);
53.  \  \ \  \ AG = AG ∪ AG\_ag;
54.  \  \ \  \ \  \ AG = AG ∪ AG\_ag, obj.ATTs = obj.ATTs ∩ OP.A;
55.  \  \ \  \ Return (AG.obj);

the authorized graphs are obtained by recursively traversing $G$, from the end of the graph, i.e., the OUT operator, till the input streams, i.e., the IN operators. Each time the algorithm encounters an operator $OP \neq IN$, it recursively calls itself by passing as input the subgraph $\mathcal{F}$ generating the stream entering into $OP$ and by performing over the authorized graphs returned from the recursion, denoted as $\text{Ret}_{\mathcal{AG}}$, different operations, on the basis of $OP$. In contrast, when Algorithm 1 encounters the IN operator, it makes use of the secure read operator (see lines 3-10) to filter out unauthorized tuples. More precisely, it evaluates the secure read operator on the stream represented by the IN operator (line 4). The operator returns a set of expressions denoting the authorized views, that is, the tuples for which $u$ has the read privilege. Note that, since the secure read operator may return more authorized views on the basis of the specified access control policies, the algorithm generates a different authorized graph $ag$ for each returned authorized view (line 5). To insert the authorized view returned by the secure read operator into the authorized graph $ag$, the algorithm exploits function $\text{Insert}()$ (line 7). This function takes as input a graph $ag$ and the expression representing the authorized view $av$ returned by the secure read operator, and generates a new graph by replacing into $ag$ the OUT operator with a subgraph whose operators encode $av$. The $\text{Insert}()$ function also inserts the OUT operator at the end of the resulting graph.

Let us now illustrate which are the steps performed by Algorithm 1 when it encounters an operator $OP \neq IN$. In case $OP = \pi$ or $OP = \sigma$, the algorithm recursively calls itself by passing the subgraph $\mathcal{F}$ generating the stream entering $OP$. Then, $\mathcal{F}$ is recursively evaluated and, as result, the algorithm returns a set of authorized graphs $\text{Ret}_{\mathcal{AG}}$. Since these graphs are defined in such a way that they generate only tuples for which $u$ has the read privilege, it is no more necessary to apply the secure operators. Thus, Algorithm 1 has to simply evaluate $OP$ directly over the streams generated by graphs in $\text{Ret}_{\mathcal{AG}}$. This implies that it has to insert the $OP$ operator just at the end of the graphs in $\text{Ret}_{\mathcal{AG}}$ (lines 15-17).

In case $OP = \text{Join}$ (line 23-41), the algorithm has to consider two different kinds of access control policies. Indeed, a user is authorized to perform a join operation if: (a) he/she is authorized to read the tuples over which the join is performed, or (b) there exists one or more access control policies granting the user the right to perform the required join. To handle both these cases, the algorithm generates two distinct set of authorized graphs, namely, $\mathcal{AG}_j$ and $\mathcal{AG}_{sj}$, respectively. In particular, the steps performed to create graphs in $\mathcal{AG}_j$ are similar to those of case $OP = \pi$ and $OP = \sigma$. Therefore, the algorithm recursively calls itself twice by passing the subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$, generating the first and the second stream entering in the Join operator, respectively. The results are collected into variables $\text{Ret}_{\mathcal{AG}_1}$ and $\text{Ret}_{\mathcal{AG}_2}$, respectively. Then, it applies the Join operator to each possible combination of graphs in $\text{Ret}_{\mathcal{AG}_1}$ and $\text{Ret}_{\mathcal{AG}_2}$ (lines 30-34). In contrast, to manage case (b), Algorithm 1 makes use of the secure join operator, by applying it over the streams entering the Join operator (line 35). Note that, by definition, the secure join receives as input the protection object like representation of the operand streams (see Section 5.1). To obtain this representation, through all the algorithm we make use of variable $\text{obj}$, which contains the protection object like representation of the stream resulting by graph $G$, generated during the recursive traversal of the graph.
In particular, the \textbf{STRs} component contains the input stream names (lines 9 and 40), whereas the \textbf{EXPs} component is set as the union of all predicates specified in the $\sigma$, and \textbf{Join} operators (lines 21 and 40). In contrast, the \textbf{ATTs} component is given by collecting all attributes of the input streams (lines 9 and 40) and recursively intersecting them with attributes specified in the $\pi$ or $\Sigma$ operators (lines 19 and 54). Thus, the secure join operator is evaluated over variables $\text{obj}_1$, and $\text{obj}_2$, which are returned by the inner recursion (see line 35). Then, similarly to the case $\text{OP}=\text{IN}$, the expressions returned by the secure join operator are inserted into graph $G$ by means of the $\text{Insert()}$ function (lines 36-38), obtaining $AG_{\text{sj}}$.

The case $\text{OP}=\Sigma$ is very similar to the case $\text{OP}=\text{Join}$ (lines 42-55). Indeed, also in this case the algorithm has to consider two different kinds of access control policy, since a user is authorized to perform an aggregate operation if: (a) he/she is authorized to read the tuples over which the aggregation is performed, or (b) there exists one or more access control policies granting the user the required aggregate privilege on the target stream. Thus, Algorithm 1 generates two distinct set of authorized graphs, namely, $AG_a$ and $AG_{sa}$, according to a procedure very similar to the one adopted in case (a) and (b) when $\text{OP}=\text{Join}$.

**Example 5.6.** Suppose Paul submits the query graph in Figure 3. Let us show which are the authorized graphs returned by Algorithm 1, assuming that Paul is a doctor and that the only specified access control policies are those in Table II. We recall that the algorithm generates the authorized graphs by recursively calling itself. The algorithm starts to evaluate the last operator, that is, $\text{OP}=\Sigma$. Then, it calls recursively itself by passing $G$, that is, the graph consisting of the $\text{IN}$ operators, the \textbf{Join} operator, the $\sigma$ operator and the $\text{OUT}$ operator. As second recursion, Algorithm

1 calls itself again by passing $G$ consisting of the IN operators and the Join operator. Thus, during the third recursion, the algorithm elaborates the case $OP=Join$. In this case, it doubly calls itself by passing $G_1$ and $G_2$, where $G_1$ (resp. $G_2$) consists only of the IN operator modeling Position (resp. Health) (lines 28 and 29). When the algorithm evaluates $G_1$, it performs the steps referring to $OP=IN$. Therefore, it evaluates $Sec_{Read}(Position, Paul)$, and inserts into $G_1$ the operators encoding the unique authorized view returned by the secure read operator (see Example 5.1). Then, the recursion halts by returning an authorized graph $Ret_{AG_1}$ consisting of the IN operator modeling the Position stream and a set of operators encoding the expressions returned by the secure read operator. Similarly, during the evaluation of $G_2$, Algorithm 1 performs the steps referring to $OP=IN$. Thus, it evaluates $Sec_{Read}(Health, Paul)$. This operator enforces the unique access control policy granting the read privilege to doctors over the Health stream, that is, the third policy in Table II. Thus, it returns the following expression: $\pi(SID, Platoon, Heart, BPressure)(\sigma(Platoon=X))(Health)$. Therefore, this recursion returns an authorized graph $Ret_{AG_2}$ consisting of the IN operator modeling the Health stream and a set of operators encoding the above expression.

Then, Algorithm 1 evaluates the Join operator. The authorized graphs to be returned are given by the union of $AG_1$ and $AG_{sj}$ (line 39). In particular, $AG_j$ contains a unique graph consisting of the Join operator applied to graphs $Ret_{AG_1}$ and $Ret_{AG_2}$ (see lines 30-34). This is similar to the authorized graph $ag_1$ obtained in Example 5.4 and represented in Figure 4, without the $\sigma$ operator. Then, Algorithm 1 evaluates $Sec_{Join}(Position, Health, Position.SID=Health.SID, Paul)$. This returns a unique expression (see Example 5.3). The authorized graph $AG_{sj}$ obtained by inserting the operators encoding this expression is equal to the authorized graph $ag_2$ obtained in Example 5.5 and represented in Figure 5, without the $\sigma$ operator.

Then, the algorithm evaluates the $\sigma$ operator, by inserting into each authorized graphs in $AG_{a} = \{ag_1, ag_2\}$ the $\sigma$ operator. Finally, as last recursion, the algorithm evaluates the $\Sigma$ operator on graphs in $Ret_{AG}$. Algorithm 1 generates the authorized graphs as the union of $AG_a$ and $AG_{sa}$ (line 54). $AG_a$ consists of two authorized graphs obtained by applying the $\Sigma$ operator over $ag_1$ and $ag_2$ (lines 46-48). In particular, the first authorized graph in $AG_a$ returns the average of the heart beats only of those soldiers belonging to platoon $X$. Whereas the second graph does not return any tuple in that the $\pi$ operator applied after the secure join operator projects only the $BPressure$ attribute. Note that, even if the algorithm returns this graph as a result, it will be not deployed in the data stream engine, since its corresponding query is syntactically wrong. To generate the authorized graphs in $AG_{sa}$, Algorithm 1 evaluates the secure aggregate operator. However, there does not exist an access control policy granting doctors the $avg$ privilege over the stream resulting from the join of Health and Position streams. Thus, the secure aggregate operator does not return any expression, which implies that $AG_{sa}$ is empty. Thus, the authorized graphs returned by Algorithm 1 are those reported in Figures 6 and 7.

Finally, the following theorems prove the correctness and completeness of our secure query rewriting algorithm.

**Theorem 5.5.** (Correctness property). Let $G$ be a query graph submitted by a user $u$, and $AG$ be the set of authorized graphs returned by Algorithm 1. Let $AS$ be...
the set of streams generated by graphs in $AG$. The correctness property ensures that for each tuple $t \in AS$, there exists an access control policy in $SysAuth$ authorizing $u$ to access $t$.

**Theorem 5.6.** (Completeness property). Let $G$ be a query graph submitted by a user $u$, and $OS$ be the stream generated by $G$. Let $AG$ be the set of authorized graphs returned by Algorithm 1. The completeness property ensures that for each tuple $t \in OS$ such that there exists an access control policy authorizing $u$ to access $t$, $t$ is included into one of the streams generated by $AG$.

Formal proofs are reported in appendix.

6. DEPLOYMENT

In this section we show how, from the authorized graphs returned by the Query Rewriter, the Deployment Module generates queries executable into StreamBase and Coral8. Given an authorized graph, generating the corresponding query implies to translate and encode the graph according to the SQL-based language adopted in the specific stream engine, i.e., StreamSQL (adopted in StreamBase) and CCL (adopted in Coral8). In particular, for each of the authorized graph $ag$ generated by the Query Rewriter, the Deployment Module generates a set of SQL-based statements $AQ$, such that the execution of $AQ$ generates the same stream obtained by $ag$. This translation is carried out by encoding each operator in $ag$ into a corresponding SQL-based statement. Additionally, it must be ensured that the translated operators are executed in $AQ$ according to the flow defined by $ag$. In what follows, we show how this translation takes place in our system, by focusing on each of the supported language.

6.1 StreamSQL

In order to present the strategy to translate authorized graphs into StreamSQL statements, we first need to briefly introduce the StreamSQL syntax. In particular, we present only the syntax of those StreamSQL statements exploited during the translation, whereas we refer the interested reader to [StreamBase 2008] for a complete overview of StreamSQL.

The StreamSQL statement to specify queries over streams is the `SELECT` statement, whose syntax and semantics is similar to the ones of SQL but with the extensions needed to support evaluation on data streams. The syntax is the following:

```sql
SELECT target_list
FROM tuple_source [..]
[WHERE predicate]
INTO resulting_stream_name;
```

where:

---

- `target_list` consists of one or more entries, separated by commas, of one of the following forms: (a) `attribute_name`, representing the name of an attribute; (b) `expression [AS] alias`, where `expression` can be a simple or an aggregate function over stream attribute(s);
—tuple_source consists of one or more entries of the form: stream_name [SIZE size ADVANCE offset]. If [SIZE size ADVANCE offset] is present, the query target is a sliding-window stream generated over stream_name by using the size and the offset as parameters;
—predicate contains the conditions to select the tuples to be given in output;
—INTO resulting_stream_name specifies the name (resulting_stream_name) of the stream where the tuples resulting by the evaluation of the SELECT statement have to be pushed.

The Deployment Module exploits the SELECT statement to translate the operators of our core query model. For instance, the \( \pi \) operator is translated into a SELECT statement having as target list the attributes in \( \pi \).atts. Whereas, the \( \sigma \) operator is encoded by a SELECT statement whose WHERE clause contains the predicates in \( \sigma \).exp. A similar encoding is performed for the \( \Sigma \) and Join operators. In these cases, the window specification in the FROM clause is given by using the parameters \( \Sigma.s, \Sigma.o \), or \( \text{Join}_1.s, \text{Join}_1.o, \text{Join}_2.s, \text{Join}_2.o \).

Other StreamSQL statements relevant for the translation are those for stream creation. StreamSQL provides three different statements that generate three different types of streams:

1. CREATE INPUT STREAM stream_name (attribute_name attribute_type[,]);
2. CREATE OUTPUT STREAM stream_name (attribute_name attribute_type[,]);
3. CREATE STREAM stream_name (attribute_name attribute_type[,]);

Each of the above reported StreamSQL statements creates a stream with name stream_name and schema consisting of the attributes specified into the parenthesis. The first kind of streams, generated by statement (1) and called input streams, represents the streams over which a query is executed. In contrast, the second kind of streams, called output streams and generated by statement (2), is those streams into which the results of queries are pushed. These two kinds of streams are used to represent the streams modeled by the IN and OUT operators of our core query model. In contrast, by means of statement (3) it is possible to define a stream available only to queries belonging to the same module. We refer to these streams as internal streams. Internal streams cover a key role in the translation of an authorized graph \( \text{ag} \), since they make it possible to constraint the execution order of the StreamSQL statements encoding the operators belonging to \( \text{ag} \). Indeed, in order to have a correct translation we have to ensure that the StreamSQL statements are evaluated according to the flow defined by the edges in \( \text{ag} \). We recall that an edge between two operators \( \text{OP}_1 \) and \( \text{OP}_2 \) implies that the stream resulting from \( \text{OP}_1 \) enters directly into the \( \text{OP}_2 \) operator. This flow can be easily modeled by creating an internal stream, called \( \text{OUT}_\text{OP}_1 \), and including in the StreamSQL statement encoding \( \text{OP}_1 \), the clause ‘INTO \( \text{OUT}_\text{OP}_1 \)’. This forces the stream resulting from the StreamSQL statement corresponding to \( \text{OP}_1 \) into the internal stream \( \text{OUT}_\text{OP}_1 \). Then, to ensure the correct execution order it is necessary to put \( \text{OUT}_\text{OP}_1 \) into the FROM clause of the StreamSQL statement encoding \( \text{OP}_2 \). Clearly, when creating the internal stream, its schema has to be appropriately defined. For instance, if the internal stream represents the output of a \( \pi \) operator, its schema must contain all and only the attributes in \( \pi \).atts. In contrast, if the internal stream is created to contain the output of a \( \sigma \)
CREATE INPUT STREAM Health (ts timestamp, SID int, Platoon int, Heart int, BPressure int);
CREATE INPUT STREAM Position (ts timestamp, SID int, Platoon int, Pos int);

--The following statement translates the π and σ operators evaluated over Health stream
CREATE INTERNAL STREAM OUT_πσ₁ (ts timestamp, SID int, Platoon int, Heart int, BPressure int);
SELECT * FROM Health WHERE Platoon=X INTO OUT_πσ₁;

--The following statement translates the π and σ operators evaluated over Position stream
CREATE INTERNAL STREAM OUT_πσ₂ (SID int, Pos int);
SELECT SID, Pos FROM Health WHERE Platoon=X INTO OUT_πσ₂;

--The following statement translates the Join operator
CREATE INTERNAL STREAM OUT_Join (ts timestamp, SID int, Platoon int, Heart int, BPressure int, Pos int);
SELECT * FROM OUT_πσ₁, OUT_πσ₂ WHERE Position.SID=Health.SID INTO OUT_Join;

--The following statement translates the σ operator
CREATE INTERNAL STREAM OUT_σ (ts timestamp, SID int, Platoon int, Heart int, BPressure int, Pos int);
SELECT * FROM OUT_Join WHERE Pos ≥ k INTO OUT_σ;

--The following statement translates the Σ operator
CREATE OUTPUT STREAM AG1 (AVGHeart int);
SELECT AVG(Heart) FROM OUT_Join INTO AG1;

Fig. 8. StreamSQL statements corresponding to the authorized graph in Figure 6

operator, its schema is defined equal to the schema of the stream entering into the σ operator. Similar considerations hold for the Σ and Join operators.

Figure 8 reports, as an example of translation, the StreamSQL statements corresponding to the authorized graph in Figure 6.

6.2 CCL

The strategy to translate an authorized graph ag into CCL statements is very similar to the one presented in the previous section. Indeed, similar to StreamSQL, CCL offers the SELECT statement to specify queries over streams, as well as the CREATE statement to create input, output, and internal streams (called local streams in CCL). In the following, we report the syntax of the CCL statements exploited during the translation, whereas we refer the interested reader to [Coral8 2008] for a complete overview of the CCL language.
SELECT statement:

```
INSERT INTO resulting_stream_name
SELECT target_list
FROM tuple_source [...] 
[WHERE predicate];
```

CREATE statements:

(1) CREATE INPUT STREAM stream_name (attribute_name attribute_type[,
(2) CREATE OUTPUT STREAM stream_name (attribute_name attribute_type[,
(3) CREATE LOCAL STREAM stream_name (attribute_name attribute_type[,

Similarly to the strategy for translating authorized graphs into StreamSQL, given an authorized graph \(ag\) it is possible to encode each operator belonging to \(ag\), by using the CCL SELECT statement. Then, by defining proper internal streams, through the CREATE LOCAL STREAM statement, it is possible to constraint the execution order of the CCL statements according to the flow defined by \(ag\).

7. PROTOTYPE EVALUATION

In this section, we present some performance results of the prototype system we have developed, implementing our framework [Cao et al. 2009]. Currently, the prototype supports the most relevant modules of the architecture illustrated in Figure 1, that is, the Query Rewriter and the Deployment Module.

To overcome the current lack of the GUI, users submit queries by means of a textual interface. In particular, we use the XML query encoding adopted by StreamBase to represent query graphs. The operators that can be used when defining the query graphs are restricted to those supported by our core query model. Thus, each query graph is stored into an XML document.

When the Query Rewriter receives the XML document encoding the user query, it parses the document and obtains the corresponding query graph. The query graph is then rewritten according to the rewriting strategy defined in Algorithm 1. The resultant authorized graphs are then converted into distinct XML documents, and passed to the Deployment Module. The current version of the Deployment Module translates the received query graphs into StreamSQL only. Translation into CCL is currently under development.

The current prototype is implemented in Java and the experiments were run on a Core 2 Duo 2.33GHz CPU machine, with 4G RAM, running windows XP. We have carried out two main kinds of experiments. The first aims to evaluate the overhead of secure query rewriting, whereas the second class of experiments compares the proposed access control enforcement against the post-processing approach.

7.1 Overhead of secure query rewriting

To evaluate the overhead of secure query rewriting, we have performed a set of experiments to measure the time required by the Query Rewriter. We first measure the CPU time required by secure query rewriting by varying the query complexity (in terms of number of operators). Table III shows the 5 queries that we used in the
A Framework to Enforce Access Control over Data Streams

<table>
<thead>
<tr>
<th>Query</th>
<th>Streams</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q₁</td>
<td>2 IN, 1 OUT</td>
<td>5 operators: 2 π, 2 σ, 1 Join</td>
</tr>
<tr>
<td>Q₂</td>
<td>3 IN, 1 OUT</td>
<td>10 operators: 4 π, 3 σ, 2 Join, 1 Σ</td>
</tr>
<tr>
<td>Q₃</td>
<td>6 IN, 1 OUT</td>
<td>20 operators: 8 π, 7 σ, 4 Join, 1 Σ</td>
</tr>
<tr>
<td>Q₄</td>
<td>9 IN, 1 OUT</td>
<td>30 operators: 12 π, 9 σ, 7 Join, 2 Σ</td>
</tr>
<tr>
<td>Q₅</td>
<td>18 IN, 1 OUT</td>
<td>60 operators: 24 π, 18 σ, 14 Join, 4 Σ</td>
</tr>
</tbody>
</table>

Table III. Queries complexity

Fig. 9. Secure query rewriting overhead

experiments - query Q₁ is the less complex with 5 operators, whereas query Q₅ is the most complex involving 60 operators. Figure 9(a) shows the results. As expected, when the query complexity increases, rewriting takes more time. However, in all the considered cases, the time required is less than 0.2 seconds. Note that this overhead is negligible compared to the lifespan of the query - as DSMSs queries are continuous and long running, query rewriting is performed once before the long running query is registered into the system and continuously executed on the target streams.

In the next experiments, we evaluate the effect of the number of policies that are simultaneously applied to a query. Figure 9(b) illustrates the required CPU time as we vary from 10 to 50 the number of policies applied to query Q₃ of Table III. When the number of policies increases, more CPU time is needed to rewrite the query. However, even with 50 access control policies being simultaneously applied to Q₃, the time required is less than 0.12 seconds.

From the above results, it is clear that the proposed framework is quite scalable. The overhead of query rewriting is small even for complex queries and large number of policies.

7.2 Comparative analysis

In this section, we compare the performance of our query rewriting algorithm with the post-processing access control enforcement. In particular, we compare our approach with the post-processing strategy proposed in [Lindner and Meier 2006]. The approach described in [Lindner and Meier 2006] does not rewrite a query on the basis of the specified access control policies. Rather, the output of a query graph is given as input to a SecFilter operator, which prunes unauthorized tuples from the result. In particular, before a tuple enters the query graph, SecFilter marks it with a label, which indicates from which source stream this tuple comes.
from. Then, SecFilter checks each output tuple of the query graph, and verifies, by exploiting its label, whether or not the tuple can be delivered to the user, that is, whether or not the user has the read privilege over it.

![Relative Joins Graph](image1)

**Fig. 10.** Comparison with respect to relative joins

![Relative CPU Time and Memory Usage](image2)

**Fig. 11.** Comparison with respect to relative CPU time and relative memory usage

In order to empirically evaluate the benefits of secure query rewriting, given a query $Q$ and a set of access control policies $ACP$, we compare the computational cost of evaluating the corresponding rewritten query $RQ$ (generated by Algorithm 1), against that of evaluating $Q$ with post-processing. To do that, we generate synthetic streaming data to simulate a military drill. In particular, we create the soldiers' position stream by the generator of moving objects [Brinkhoff 2002]. Moreover, to simulate the stream of soldiers' health conditions, we create two Health streams, namely, $uniHealth$ and $normHealth$. In $uniHealth$, heart beats and blood pressures are uniformly distributed; whereas $normHealth$ is generated assuming that heart beats and blood pressures are normally distributed. For simplicity, we consider a query $Q$ containing only one join operator. In this case, the computational cost is due to the cost of evaluating the join predicate over each pair of tuples entering the join operator. We introduce the relative joins measure. In particular, let $num_{p}(join)$ be the number of times that the join predicate is evaluated by the post-processing scheme, and $num_{r}(join)$ be the number of times that the join predicate is evaluated using the query rewriting approach, the relative joins is defined as:

$$\frac{num_{r}(join)}{num_{p}(join)} \times 100.$$
We calculate the relative joins by varying the selectivity of access control policies. Figure 10 reports the result. In the figure, the \textit{rewrite-norm} line is the result when source stream \texttt{normHealth} and source stream \texttt{Position} are tested, whereas the \textit{rewrite-uni} line is the result when \texttt{uniHealth} and \texttt{Position} streams are tested. The number of join evaluations under the post-processing scheme does not change with the variation of selectivity since all the pruning work is done after the join operator. The number of joins required by the query rewriting scheme increases when selectivity increases because the number of pruned tuples is smaller with higher selectivity. Therefore, as expected when the selectivity increases, the relative joins value increases. However, the relative joins value is always less than 100%, which shows that the rewritten query always requires a lower number of joins than the post-processing method.

We also compare secure query rewriting and post-processing with respect to the required CPU time. We define \textit{relative CPU time} as the ratio between the CPU time required by secure query rewriting and that required by post-processing. In the experiments, we set the policy selectivity to 0.5, and vary the join window size. Secure query rewriting prunes some tuples before they enter the join window, so it can save some CPU time. When the window size increases, each arriving tuple of one input stream is compared with more tuples of the other input stream. Thus, the saving of CPU time is more effective when the window size increases. Figure 11(a) illustrates the result. As expected, when the window size increases, the relative CPU time reduces, that is, secure query rewriting saves more CPU time than post-processing. Finally, we compare secure query rewriting and post-processing with respect to memory usage. The definition of \textit{relative memory usage} is similar to that of \textit{relative CPU time}. Figure 11(b) reports the relative memory usage when the window size varies. It is always less than 100%, so secure query rewriting always uses less memory than post-processing.

8. \textbf{RELATED WORK}

Data stream management systems have been the subject of intensive research in the context of different projects, like, for example: Tapestry [Terry et al. 1992], Alert [Schreier et al. 1991], Tribeca [Sullivan 1996], OpenCQ [Liu et al. 1999], NiagaraCQ [Chen et al. 2000], Telegraph [Chandrasekaran et al. 2003], Aurora [Abadi et al. 2003], STREAM [Arasu et al. 2003], Nile [Hammad et al. 2003], and CAPE [Zhu et al. 2004]. As a consequence, literature offers a quite huge amount of work investigating a variety of data stream management issues [Babcock et al. 2002; Golab and Ozsu 2003]. Some of them are, for example, related to data models and languages (see, for example, [Law et al. 2004] for a survey), continuous query processing problems, i.e., load shedding, join problems, efficient window-based operators (see [Babcock et al. 2004; Bai and Zaniolo 2008]), data stream mining (see [Gaber et al. 2005] for a survey), clustering and classification methods for data stream (as example, [Aggarwal et al. 2003; 2004]). Among these issues the ones that are most related and/or most affect our work are those on data models.
and the operators defined on that. Indeed, since our access control framework relies on its own core query model, it is interesting to discuss how this is related to existing models and operators.

As pointed out in [Golab and Özsu 2003; Muthukrishnan 2005], a common model to represent a data stream is as a sequence of data items arriving from several sources. In particular, based on the arrival order and on whether items have been pre-processed before arriving, four different types of data stream models can be identified [Gilbert et al. 2001; Muthukrishnan 2005]: unordered cash register, where items of different sources arrive without a particular order and pre-processing; ordered cash register, which implies that items arrive with a given order, but without any pre-processing; unordered aggregate, where, in no particular order, only one item per source arrives, whose value is computed according to a given pre-processing, i.e., by aggregating different items of the same source; ordered aggregate, which is similar to the previous one but items arrive with some given order. With respect to these models, we have to note that the proposed core model represent streams as append-only sequence of tuples with the same schema, which contains the additional attribute $ts$ storing the time of origin of the corresponding tuple. Thus, any possible arrival order and pre-processing can be implemented by the core model, making it able to support all of the four types of data models.

Regarding the operators, all the considered data stream management systems support the basic relational operators (i.e., selection, projection, aggregate, join), plus additional operators to handle windows (see, [Golab and Özsu 2003] for more details). Note that some of them support also operators to handle items order (like for example, the Bsort operator in Aurora). However, since the purpose of the adopted core model is to be suitable to as more data stream management systems as possible, it has been defined to handle only relational and window-based operations. More precisely, it supports all relational operators, thus it is able to represent all relational queries specified by data stream management systems. Regarding the type of windows, these can be classified according to three main criteria [Golab and Özsu 2003]. The first is the direction of movements of the window endpoints, which gives rise to the following window types: two fixed endpoints (fixed windows), two sliding endpoints (sliding endpoints); only one sliding endpoint (i.e., landmark window). The second criteria is about the size of the window, that is, whether it is specified in terms of time intervals (time-based windows) or in terms of the number of tuples (count-based windows). Another criteria is the frequency of updating a window (i.e., update upon each tuple arrival or by means of a batch process). Among these criteria, the ones relevant for our query model are the first two, since the last one is more related to execution, that is, to the data stream engine. In particular, the core query model is able to specify both fixed windows and sliding windows, by properly setting the size and offset value (in the fixed window the offset is set null). The core query model can also be easily extended to support landmark windows. Also time-based and count-based windows can be specified by simply specifying the number of tuples or time interval in the size parameter. Therefore, the core model is flexible enough to support existing window-based query operators.

The above discussion shows how the proposed access control framework can be easily deployed into different data stream management systems, since the underly-
Other work related to our proposal are those on streaming data protection. This problem has not been yet investigated so deeply as the other DSMS issues mentioned above. Thus, the literature offers few proposals. We can classify them in two main categories: those aiming to ensure authenticity, integrity and confidentiality of data streams during transmission [Papadopoulos et al. 2007; Ali et al. 2005], and those related to access control [Lindner and Meier 2006; Nehme et al. 2008]. An example of the first category is the work by Ali, ElTabakh and Nita-Rotaru [Ali et al. 2005], which proposes an extension of the RC4 algorithm, i.e., a stream cipher encryption scheme, to overcome possible decryption fails due to desynchronization problems. The proposed encryption scheme has been developed in the Nile [Hammad et al. 2003] stream engine. Another example of these proposals is [Papadopoulos et al. 2007]. Here, authors address the authenticity problem of outsourced data streams. More precisely, [Papadopoulos et al. 2007] considers a scenario where a data owner constantly outsources its data streams, complemented with additional authentication information, to a service provider. Then, instead of querying the data owner, clients register continuous range queries directly to the service provider. The proposal makes clients able to verify the authenticity and the completeness of the results received from the service provider, by using the authentication information provided by the data owner.

Recently the problem of access control for data streams has been investigated by Lindner and Meier [Lindner and Meier 2006] and by Nehme, Rundensteiner and Bertino [Nehme et al. 2008]. Lindner and Meier propose a Owner extended RBAC (OxRBAC) model to protect data streams from unauthorized accesses [Lindner and Meier 2006]. The basic idea is to apply a newly designed operator, called SecFilter, at the stream resulting from the evaluation of a query to filter out output tuples that do not conform to the access control rules. As mentioned in Section 1, this post-processing approach has the drawback of wasting computation time, when unauthorized queries are performed. Indeed, as noted in [Lindner and Meier 2006], it is possible for a user to remain “connected” to an output stream though he/she may not receive any output tuple (e.g., because his/her access rights have been revoked). This is not desirable (see experiments in Section 7). Finally, because the proposed framework is not intrusive, SecFilter cannot handle certain access control policies on views on data from multiple streams.

Access control for data streams has also been investigated in [Nehme et al. 2008]. Here, authors consider access control from a different point of view with respect to our proposal. Indeed, in our scenario, we assume that access control policies are specified by the SA, whereas in [Nehme et al. 2008] policies on a data stream are stated by the user owning the device producing the data stream itself. This makes user able to specify how the DSMS has to access his/her personal information (e.g., location, health conditions). As such, their approach is more related to privacy protection, whereas our focus is on access control. Moreover, in [Nehme et al. 2008], access control policies are not stored in the DSMS, rather they are encoded via security constraints (called security punctuations) and embedded directly into data streams. A set of operators is also defined, able to enforce security punctuations, and implement them into the CAPE engine [Zhu et al. 2004]. In contrast, we
propose a framework able to work on top of different DSMs.

9. CONCLUSIONS

In this paper, we have proposed a framework to enforce access control into different DSMs. The framework exploits an expressive role-based access control model, and a set of novel secure operators (namely, Secure Read, Secure View, Secure Join, and Secure Aggregate), defined on support of secure query rewriting. Preliminary performance evaluations showed the effectiveness of the proposed techniques.

We plan to extend the work reported in this paper along several directions. First, we plan to develop a complete prototype and to carry out a more extensive performance study. Additionally, we plan to investigate how queries can be further optimized on the basis of the optimization techniques in place in the target stream engines. We will also extend the model (and hence the enforcement strategies) to deal with updates. The support for sharing of queries among multiple users is also a topic we would like to investigate in the future.

Finally, it is important to remark that our access control model is a discretionary access control model, as most of the models adopted by current commercial data management systems. As such, it prevents explicit accesses to data, but leaving the responsibility to the SA of correctly assigning access rights in such a way that inference of unauthorized information is prevented [Farkas and Jajodia 2002]. An interesting direction we plan to investigate is how our system can be complemented with inference control techniques (e.g., [Biskup and Lochner 2007], [Rizvi et al. 2004]).

REFERENCES


A Framework to Enforce Access Control over Data Streams


Before proving Theorems 5.5 and 5.6, we need to introduce some lemmas stating the correctness of the secure operators introduced in Section 5.1.

**Lemma 10.1.** *(Secure view correctness).* Let $S$ be an input stream, and $acp$ be an access control policy such that $acp.STRs = S.STRs$. Let $AE$ be the expressions returned by $Sec.View(S, acp)$. Let $AS$ be the set of streams resulting from the evaluation of expressions in $AE$. For each tuple $t \in AS$, $t$ is authorized by access control policy $acp$, that is, $t \in \beta(acp)$.

**Proof.** We prove the lemma by proving that if there exists a tuple $\bar{t} \in AS$ such that $\bar{t}$ does not belong to $\beta(acp)$, then a contradiction arises. According to Definition 5.1, if $\bar{t} \in AS$, then $\bar{t}$'s attribute are all those in the set $\{Att(S) \cap acp.obj.ATTs\}$, if $acp.obj.ATTs \neq *$, they are equal to attributes in $Att(S)$, otherwise. Moreover, $\bar{t}$ satisfies the following predicate: $(acp.obj.EXPs \land ts \geq acp.gtc.begin \land ts \leq acp.gtc.end)$.\(^9\)

Suppose now that $\bar{t}$ does not belong to $\beta(acp)$. By Definition 4.3, $\beta(acp) = \pi(A_1, \ldots, A_n)(\sigma(acp.obj.EXPs \land ts \geq acp.gtc.begin \land ts \leq acp.gtc.end)(acp.obj.STRs))$, where $\{A_1, \ldots, A_n\} = acp.obj.ATTs$. If $\bar{t}$ does not belong to $\beta(acp)$, this implies that $\bar{t}$ does not satisfy the expression specified in the protection object specification of $acp$ and/or the general time constraints stated in $acp$. However, by definition of $Sec.View$, $\bar{t}$ satisfies both of them. Thus, a contradiction arises, which proves the thesis.

**Lemma 10.2.** *(Secure read correctness).* Let $S$ be a stream and $u$ be a user. Let $AE$ be the set of expressions returned by $Sec.Read(S, u)$. Let $AS$ be the set of streams resulting from the evaluation of expressions in $AE$. For each tuple $t \in AS$, there exists an access control policy $acp \in SysAuth$ authorizing $u$ to access $t$, that is, $t \in \beta(acp)$.

**Proof.** We have to prove that if $\bar{t} \in AS$, then there exists an access control policy $acp \in Pol(S, u)$, such that $\bar{t} \in \beta(acp)$. By construction, for each $acp_1 \in Pol(S, u)$ the $Sec.Read$ operator returns the expressions generated by $Sec.View(S, acp_1)$. Since $AS$ is computed by evaluating these expressions, it implies that if $\bar{t} \in AS$, then there exists an access control policy $acp$, such that $Sec.View(S, acp)$ returns an expression denoting a set of tuple containing $\bar{t}$. By Lemma 10.1, $\bar{t} \in \beta(acp)$, which proves the thesis.

**Lemma 10.3.** *(Secure aggregate correctness).* Let $S$ be a stream, $u$ be a user, $F$ be an aggregate function, $A$ be an attribute of $S$, and $s$ and $o$ be two natural numbers. Let $AE$ be the set of expressions returned by $Sec.Aggr(S, F, A, s, o, u)$. Let $AS$ be the set of streams resulting from the evaluation of expressions in $AE$.

---

\(^9\)We recall that $Att(S)$ denotes the set of attributes belonging to $S$’s schema.

\(^10\)According to Definition 5.1, the expression can also be null or it can be a subset of the above one, however we prove the lemma for the more general case, being the others very similar.
For each tuple $t \in \text{AS}$, there exists an access control policy authorizing $u$ to access $t$. More formally, for each tuple $t \in \text{AS}$, $\exists \text{acp}\in \text{SysAuth}$, such that $\text{acp}$.priv = F and $t \in \Sigma(\text{F}, \text{A}, \text{max}_\text{size}, \text{max}_\text{offset})(\beta(\text{acp}))$, where $\text{max}_\text{size} = \max(\text{acp}.\text{wtc}.\text{size}, \text{u})$, and $\text{max}_\text{offset} = \max(\text{acp}.\text{wtc}.\text{offset}, \text{o})$.

**Proof.** We prove this lemma by proving that if there exists $t \in \text{AS}$ such that for each access control policy $\text{acp}$ in $\text{SysAuth}$ the condition $t \notin \Sigma(\text{F}, \text{A}, \text{max}_\text{size}, \text{max}_\text{offset})(\beta(\text{acp}))$ holds, then a contradiction arises.

According to Definition 5.3, $\exists t \in \text{AS}$ implies that there exists an access control policy $\text{acp} \in \text{Pol}_{\text{agg}}(\text{S}, \text{u})$ such that $\text{AS}$ contains a stream generated by the following expression (returned by the secure aggregate operator): $\Sigma(\text{F}, \text{A}, \text{max}_\text{size}, \text{max}_\text{offset})(\pi(\text{A})(\sigma(\text{P})(\text{S})))$, where $\text{P}$ is equal to $\text{acp}.\text{obj}.\text{EXPs} \land \text{ts} \leq \text{acp}.\text{gtc}.\text{begin} \land \text{ts} \leq \text{acp}.\text{gtc}.\text{end}$. By Definition 4.3, the expression can be rewritten as $\Sigma(\text{F}, \text{A}, \text{max}_\text{size}, \text{max}_\text{offset})(\beta(\text{acp}))$, which proves that a contradiction arises.

**Lemma 10.4.** (Secure join correctness). Let $\text{S}_1$ and $\text{S}_2$ be two streams, $\text{P}$ a join predicate over $\text{S}_1$ and $\text{S}_2$, and $\text{u}$ a user. Let $\text{AE}$ be the set of expressions returned by $\text{Sec}_\text{Join}(\text{S}_1, \text{S}_2, \text{P}, \text{u})$. Let $\text{AS}$ be the set of streams resulting from the evaluation of expressions in $\text{AE}$. For each tuple $t \in \text{AS}$, there exists an access control policy authorizing $u$ to access $t$, that is, $\exists \text{acp}\in \text{SysAuth}$, such that $t \in \beta(\text{acp})$.

**Proof.** We prove the Lemma by proving that if there exists $t \in \text{AS}$ and for each access control policy $\text{acp}$ in $\text{SysAuth}$ the condition $t \in \beta(\text{acp})$ does not hold, then a contradiction arises.

According to Definition 5.4, for each $\text{acp} \in \text{Pol}_\text{join}(\text{S}_1, \text{S}_2, \text{u})$ the $\text{Sec}_\text{Join}$ operator returns the expressions generated by $\text{Sec}_\text{View}(\text{Join}(\text{P})(\text{S}_1, \text{S}_2), \text{acp})$, that is, the expressions generated by evaluating the secure view operator over the stream obtained by joining $\text{S}_1$ and $\text{S}_2$ according to predicate $\text{P}$. Since $\text{AS}$ is computed by evaluating these expressions, it implies that if $t \notin \text{AS}$, then there exists an access control policy $\text{acp}$, such that $\text{Sec}_\text{View}(\text{Join}(\text{P})(\text{S}_1, \text{S}_2), \text{acp})$ returns an expression denoting a set of tuple containing $t$. By Lemma 10.1, $t \in \beta(\text{acp})$, which proves the thesis.

**Proof.** of Theorem 5.5

We prove the theorem by induction on the *dimension* of the graph $G$, where we define the dimension of a graph as the maximum length among the length of all the paths connecting the IN operators to the OUT operator. Thus, proving the theorem by induction implies to prove that the correctness property holds for graphs with minimal dimension (Basis $l=3$). Then, we assume that the correctness property holds for graphs of dimension $n$ ($n > 3$), and we prove that it holds also for graphs of dimension $n + 1$ (Induction).

**Basis $l=3$.** We consider as graph with minimal dimension the one having the maximum length of its paths equal to three. Note that paths of this length consist

---

11The length of a path is given by the number of operators in the path.

Algorithm 1 is recursively evaluated two times. The first evaluation of $Sec\_Rewr()$ is on the subgraph of $G$ consisting of the IN and OP operators, whereas the second evaluation considers the subgraph containing only the IN operator. During the second recursion, the algorithm performs the $Sec\_Read$ operator over the input stream represented by the IN operator (line 4). Then, during the first recursion the algorithm performs different operations on the basis of OP, i.e., $OP=\pi$ or $OP=\sigma$, $OP=\Sigma$ and $OP=Join$. In all the cases, we have to prove that for each tuple $t \in AS$ there exists an access control policy $acp \in SysAuth$ authorizing $u$ the access on $t$. In the following, we consider each possible case.

**Case $OP=\pi$ or $OP=\sigma$.** According to Algorithm 1, the authorized graphs $AG$ generating streams in $AS$ are obtained by applying the OP operator directly on the graphs returned by the first recursion of $Sec\_Rewr()$, i.e., $Ret\_AG$ (see lines 15-17). By construction, each graph $rag \in Ret\_AG$ contains the operators encoding the expressions returned by the secure read operator (see lines 5-8). Lemma 10.2 ensures that for each tuple $t$ belonging to the streams generated by the expressions returned by the secure read operator there exists $acp \in SysAuth$, such that $t \in \beta(acp)$. Thus, if $t \in AS$, it implies that $t$ belongs to the stream generated by $rag$, which implies the thesis.

**Case $OP=\Sigma$.** In this case the authorized graphs $AG$ are defined as the union of $AG_a$ and $AG_{sa}$ (line 54). Let $AS_a$ and $AS_{sa}$ be the streams generated by graphs $AG_a$ and $AG_{sa}$, respectively. We have to show that, for each tuple $t \in AS_a$ and $t' \in AS_{sa}$, there exists an access control policy $acp$ authorizing $u$ to access them.

Let us start to consider tuples in $AG_a$. This set of authorized graphs is computed with a procedure similar to the one adopted in case $OP=\pi$ or $OP=\sigma$. Indeed, graphs in $AG_a$ are generated by appending the $\Sigma$ operator to each graph $rag \in Ret\_AG$ returned by the first recursion of $Sec\_Rewr()$ (lines 45-48). Thus, each tuple in $AS_a$ is computed by aggregating only tuples of one of the streams generated by graphs in $Ret\_AG$. By construction, each graph $rag \in Ret\_AG$ contains the operators encoding the expressions returned by the secure read operator (see lines 5-8). By Lemma 10.2, for each tuple belonging to a stream generated by graphs in $Ret\_AG$, there exists an access control policy granting $u$ the access to it. Thus, each tuple in $AS_a$ is computed by aggregating only authorized tuples. We have to recall that according to the semantics of the proposed access control policies, a user is authorized to evaluate an aggregate operation on a stream if: (a) he/she has a read privilege over the stream, or (b) there exists an access control policy $acp$ directly granting the aggregate privilege on the stream. Since, by construction each tuple in $t \in AS_a$ is computed by aggregating only authorized tuples, we ensure that $u$ is authorized to access $t$.

Let us now prove that the correctness property holds also for tuples belonging to $AS_{sa}$. The authorized graphs in $AG_{sa}$ are computed by replacing the $\Sigma$ operator with the operators encoding the expressions $\varphi \in AuthViews$ returned by the $Sec\_Aggr$ operator (lines 49-53). Lemma 10.3 ensures the correctness of the secure aggregate operator, that is, it ensures that for each tuple $t$ belonging to one of the streams generated by the expressions returned by the secure aggregate operator $\exists acp \in SysAuth$, such that $t \in \Sigma(\pi,F,\Sigma.A,\Sigma.s,\Sigma.o)(\beta(acp))$, which implies the thesis.
Case $\text{OP}=\text{Join}$. In case of Join operator, the authorized graphs are the union of $AG_j$ and $AG_{sj}$ (line 39). Let $AS_j$ and $AS_{sj}$ be the streams generated by graphs in $AG_j$ and $AG_{sj}$, respectively. We have to show that for each tuple belonging to the streams in $AS_j$ or $AS_{sj}$, there exists an access control policy $acp$ authorizing $u$ to access it. The authorized graphs in $AG_j$ are generated by applying the Join operator over each pair of graphs $rag_1 \in \text{Ret}_j AG_1$ and $rag_2 \in \text{Ret}_j AG_2$ returned by the first recursion of $\text{SecRewr}()$ (lines 28-34). By construction, each graph $rag_1 \in \text{Ret}_j AG_1$ (resp. $rag_2 \in \text{Ret}_j AG_2$) contains the operators encoding the expressions returned by the secure read operator, i.e. streams generated by $rag_1$ and $rag_2$, there exists $acp\in\text{SysAuth}$, such that $t \in \beta(acp)$. Thus, for each tuple in $AS_j$ there exists an access control policy granting $u$ the read access on it, which proves the thesis.

In contrast, the authorized graphs in $AG_{sj}$ are computed by replacing into the original graph $G$ the join operator with the operators encoding the expressions $av \in \text{AuthViews}$ returned by $\text{SecJoin}$ (lines 35-38). Lemma 10.4 ensures the correctness of the secure join operator, that is, it ensures that for each tuple $t$ belonging to the streams generated by expressions returned by the secure read operator, i.e. streams generated by $rag_1$ and $rag_2$, there exists $acp\in\text{SysAuth}$, such that $t \in \beta(acp)$. Thus, for each tuple in a stream belonging to $AS_{sj}$, there exists an access control policy authorizing $u$ the read access on it, which proves the thesis.

Induction. Let us now assume that thesis holds for a graph $G$ of dimension $n$, $n \geq 3$, (denoted in the following as $G^n$). We prove the thesis for a graph $G^{n+1}$ of dimension $n + 1$. A graph $G^{n+1}$ is given by a graph $G^n$ plus an additional operator $\text{OP}$ just before the $\text{OUT}$ operator. Algorithm 1 processes graph $G^{n+1}$ by recursively calling itself to process graph $G^n$, and then performing different steps on the basis of $\text{OP}$. We will show that the correctness property also holds for graph $G^{n+1}$, by considering each possible $\text{OP}$ value.

Case $\text{OP}=\pi$ or $\text{OP}=\sigma$. To elaborate $G^{n+1}$, Algorithm 1 recursively calls itself by passing as input $G^n$ (line 14), and by applying, then, the $\text{OP}$ operator directly over each of the returned authorized graphs, that is, the graphs in $\text{Ret}_u AG$ (lines 15-17). By induction, $G^n$ satisfies the correctness property. This means that, for each tuple in the stream generated by $G^n$, denoted as $AS^n$, there is an access control policy authorizing $u$ to access it. The $\text{OP}$ operator is evaluated only on tuples in $AS^n$. This implies that $G^{n+1}$ generates a stream $AS^{n+1}$, such that for each tuple $t \in AS^{n+1}$ there exists an access control policy authorizing $u$ to access it, which proves the thesis.

Case $\text{OP}=\Sigma$. The algorithm generates the authorized graphs $AG$ as the union of $AG_a$ and $AG_{sa}$ (line 54). Let $AS_a$ and $AS_{sa}$ be the streams generated by graphs $AG_a$ and $AG_{sa}$, respectively. We will show that, for each tuple in $AS_a$ or $AS_{sa}$, there exists an access control policy $acp$ authorizing $u$ to access it. In particular, the authorized graphs in $AG_a$ are computed by applying the $\Sigma$ operator over each graph $rag \in \text{Ret}_u AG$ returned by the recursion over graph $G^n$ (see lines 46-48). This procedure is very similar to case $\text{OP}=\pi$ or $\text{OP}=\sigma$, discussed above and as a consequence, the proof is similar too. For this reason we omit it.

We now focus on the authorized graphs $AG_{sa}$. The authorized graphs $AG_{sa}$ are
computed by inserting the operators encoding the expressions \( \alpha \in \text{AuthViews} \) returned by the \text{SecAggr} operator into a copy of the graph \( G^{n+1} \), where the \( \Sigma \) operator has been removed (lines 49-53). Lemma 10.3 ensures the correctness of the secure aggregate operator, that is, it ensures that for each tuple \( t \) in the streams resulting from the evaluation of the expressions returned by the secure aggregate operator, there exists \( \text{acp} \in \text{SysAuth} \), such that \( t \in \Sigma(\Sigma.F.\Sigma.A.\Sigma.a.\Sigma.o)(\beta(\text{acp})) \). Thus, for each tuple \( t \) in \( \text{AS}_{\text{as}} \), there exists an access control policy authorizing \( u \) the access to \( t \), which proves the thesis.

**Case** \( \text{OP}=\text{Join} \). The algorithm generates the authorized graphs \( \text{AG} \) as the union of \( \text{AG}_j \) and \( \text{AG}_{\text{as}} \) (see line 39). Let \( \text{AS}_j \) and \( \text{AS}_{\text{as}} \) be the streams generated by graphs \( \text{AG}_j \) and \( \text{AG}_{\text{as}} \), respectively. We will show that the correctness property holds for tuples generated by both \( \text{AS}_j \) and \( \text{AS}_{\text{as}} \). In particular, the authorized graphs in \( \text{AG}_j \) are generated by applying the \text{Join} operator over each pair of authorized graphs \( \text{ra}_j \in \text{Ret}_j \text{AG}_i \) and \( \text{ra}_2 \in \text{Ret}_2 \text{AG}_2 \) returned by the recursion of Algorithm 1 over \( G_1^n \) and \( G_2^n \), respectively (lines 28-34). \( G_1^n \) (resp. \( G_2^n \)) is the graph obtained from \( G^{n+1}_1 \) by deleting the \text{OP} operator and the whole subgraph generating the first operand stream (resp. the second operand stream). This procedure is similar to the case \( \text{OP}=\pi \) or \( \text{OP}=\sigma \) discussed above and, as a consequence, the proof is similar too. For this reason we omit it.

Let us consider the authorized graphs in \( \text{AG}_{\text{as}} \). The authorized graphs in \( \text{AG}_{\text{as}} \) are computed by replacing into graph \( G^{n+1} \) \text{OP} with the operators encoding the expressions \( \alpha \in \text{AuthViews} \) returned by the \text{SecJoin} operator (lines 35-38). Lemma 10.4 ensures the correctness of the secure join operator, that is, it ensures that for each tuple \( t \) belonging to the stream resulting from evaluation of expressions \( \alpha \), \( \exists \text{acp} \in \text{SysAuth} \), such that \( t \in \beta(\text{acp}) \). Thus, for each tuple \( t \) in a stream belonging to \( \text{AS}_{\text{as}} \), there exists an access control policy authorizing \( u \) the read access to \( t \), which proves the thesis.

\[\square\]

**Proof. of Theorem 5.6**

We have to prove that for each tuple \( t \in \text{OS} \), such that there exists an access control policy authorizing \( u \) to access \( t \), \( t \) is included into one of the streams generated by graphs in \( \text{AG} \). More formally, we have to prove that there does not exist a tuple \( t \) such that: (1) \( t \in \text{OS} \); (2) \( \exists \text{acp} \in \text{SysAuth} \) authorizing \( u \) the access on \( t \); (3) \( \forall \text{ag} \in \text{AG} \), \( t \notin \text{as} \), where \( \text{as} \) is the stream generated by \( \text{ag} \).

We prove the theorem by induction on the **dimension** of the graph \( G \). Proving the theorem by induction implies to prove that the completeness property holds for graphs with minimal dimension (**Basis** \( l=3 \)). Then, we assume that the completeness property holds for graphs of dimension \( n \), \( n > 3 \), and we prove that it holds also for graphs of dimension \( n + 1 \). (**Induction**).

**Basis** \( l=3 \). We consider as graph with minimal dimension the one having the length of its longest path equal to three. Paths of this length consist of a single operator \( \text{OP} \), plus the default \text{IN} and \text{OUT} operators. To handle this graph, Algorithm 1 is recursively evaluated two times. The first evaluation of \text{SecRewr()} is on the subgraph of \( G \) consisting of the \text{IN} and \text{OP} operators, whereas the second evaluation considers the subgraph containing only the \text{IN} operator. During the second recursion, the
algorithm performs the $\text{Sec\_Read}$ operator over the input stream represented by the $\text{IN}$ operator (line 4). Then, during the first recursion the algorithm performs different steps on the basis of $\text{OP}$, i.e., $\text{OP}=\pi$ or $\text{OP}=\sigma$, $\text{OP}=\Sigma$ and $\text{OP}=\text{Join}$. In all the cases, we have to prove that it does not exist a tuple $t$ such that: (1) $t \in \text{OS}$; (2) $\exists \text{acp} \in \text{SysAuth}$ authorizing $u$ the access on $t$; (3) $\forall \text{ag} \in \text{AG}, t \not\in \text{as}$, where $\text{as}$ is the stream generated by $\text{ag}$, and $\text{AG}$ is the set of authorized graphs returned by Algorithm 1. In the following, we consider each possible case.

**Case $\text{OP}=\pi$ or $\text{OP}=\sigma$.** We prove the thesis showing that for each tuple $\bar{t}$ such that (1) and (2) hold, then (3) does not hold. Let us consider the first condition, i.e., $\bar{t} \in \text{OS}$. If (1) holds, it means that $\bar{t}$ has been given in output by graph $G$, that is, $\bar{t}$ has passed through the $\text{OP}$ operator and satisfies the predicates in $\text{OP}$. $\text{EXPs}$, if $\text{OP}$ is $\sigma$, or it contains only attributes in $\text{OP}. \text{ATTs}$, if $\text{OP}$ is $\pi$. Moreover, if (2) holds, it means that there exists an access control policy $\text{acp} \in \text{SysAuth}$ such that $\bar{t} \in \beta(\text{acp})$. Let us see why according to Algorithm 1, each tuple $\bar{t}$ satisfying conditions (1) and (2) does not satisfy condition (3). According to the algorithm, $\text{OP}$ is applied directly on the graphs returned by the first recursion, i.e., $\text{Ret\_AG}$. More precisely, the authorized graphs in $\text{AG}$ are generated by applying over each $\text{rag} \in \text{Ret\_AG}$ the $\text{OP}$ operator (lines 15-17). By construction, each graph $\text{rag}$ in $\text{Ret\_AG}$ contains the operators encoding the expressions returned by the secure read operator (see lines 4-8). Lemma 10.2 ensures the correctness of the secure read operator, that is, it guarantees that for each tuple $t$ belonging to one of the streams resulting from the evaluation of the expressions returned by the secure read operator, there exists $\text{acp} \in \text{SysAuth}$ such that $t \in \beta(\text{acp})$. Thus, since condition (2) holds, we infer that $\bar{t}$ is enclosed into a stream generated by one of the graphs in $\text{Ret\_AG}$, say $\text{ggg}$. Thus, notwithstanding the operator (i.e., $\pi$ or $\sigma$), $\text{OP}$ is applied also over $\bar{t}$. By condition (1), we know that $\bar{t}$ satisfies $\text{OP}$, thus $\bar{t}$ is enclosed into one of the authorized streams generated by a graph in $\text{AG}$, which implies that condition (3) does not hold. Thus, the thesis holds.

**Case $\text{OP}=\Sigma$.** We prove the thesis by showing that for each tuple $\bar{t}$ such that (1) and (2) hold, (3) does not hold. Let us consider condition (1), i.e., $\bar{t} \in \text{OS}$; if it holds, it implies that $\bar{t}$ is the result of the evaluation of the $\Sigma$ operator, and it contains the aggregate value $\Sigma.F$ computed over the attribute in $\Sigma. A$. Regarding condition (2), we have to consider two possible cases. Indeed, according to the semantics of our access control policies, a user is authorized to access an aggregate value if: (a) he/she has a read privilege over the set of tuples over which the aggregate value has been computed, or (b) there exists an access control policy $\text{acp}$ directly granting the aggregate privilege to $u$ on the target stream. Thus, if condition (2) holds, one of the following is true: (a) the $\Sigma$ operator is computed only on tuples $t$ such that there exists an access control policy granting the read privilege to $u$ on $t$; (b) there exists an access control policy directly granting the aggregate privilege on tuples entering into the $\Sigma$ operator. Let us see why according to Algorithm 1, each tuple $\bar{t}$ satisfying conditions (1) and (2) does not satisfy condition (3).

The algorithm generates the authorized graphs in $\text{AG}$ as the union of $\text{AG}_a$ and $\text{AG}_sa$ (see line 54). In particular, the authorized graphs in $\text{AG}_a$ are generated by applying the $\Sigma$ operator over each graph $\text{rag} \in \text{Ret\_AG}$ returned by the first recursion of $\text{Sec\_Recur()}$ (lines 46-48). By construction, each graph $\text{rag}$ in $\text{Ret\_AG}$ contains the
operators encoding the expressions returned by the secure read operator (lines 4-8).
Lemma 10.2 ensures the correctness of the secure read operator, that is, it ensures
that, for each tuple \( t \) belonging to one of the stream resulting from the evaluation of the expressions returned by the secure read operator, there exists a read access control policy in \( \text{SysAuth} \), say \( \text{acp}_r \), such that \( t \in \beta(\text{acp}_r) \). Thus, each tuple in a stream generated by a graph in \( \text{AG}_o \) is computed by aggregating only authorized tuples.

In contrast, the authorized graphs in \( \text{AG}_sa \) are computed by replacing the \( \Sigma \) operator with the operators encoding the expressions \( av \in \text{AuthViews} \), returned by the \( \text{SecAggr} \) operator (lines 49-53). Lemma 10.3 ensures the correctness of the secure aggregate operator, that is, it ensures that, for each tuple \( t \) belonging to the stream resulting from the evaluation of the expressions returned by the secure aggregate operator, there exists \( \text{acp}_a \in \text{SysAuth} \), say \( t \in \Sigma(\Sigma.F, \Sigma.A, \Sigma.o, \Sigma.s)(\beta(\text{acp}_a)) \).

Thus, since condition (2) holds, one of the following is true: there exists a set of read access control policies \( \text{acp}_r \), evaluated by the secure read operator such that \( \overline{t} \) is the result of computing \( \Sigma \) only on tuples belonging to \( \beta(\text{acp}_r) \) or there exists an access control policy \( \text{acp}_a \), evaluated by the secure aggregate operator such that \( \overline{t} \in \Sigma(\Sigma.F, \Sigma.A, \Sigma.o, \Sigma.s)(\beta(\text{acp}_a)) \). Thus, notwithstanding of the access control policies (i.e., \( \text{acp}_r \) or \( \text{acp}_a \)), the \( \Sigma \) operator is evaluated only over authorized tuples.

By condition (1), we know that \( \overline{t} \) resulted by the evaluation of the \( \Sigma \) operator, thus \( \overline{t} \) is enclosed into one of the authorized stream generated by graphs in \( \text{AG}_o \) or \( \text{AG}_sa \).

In any case, \( \overline{t} \) is enclosed into a stream generated by an authorized graph in \( \text{AG} \), which implies that condition (3) does not hold.

**Case \( \text{GP} = \text{Join} \).** In this case the graph \( G \) contains two \( \text{IN} \) operators, representing the input streams over which the \( \text{Join} \) operator is evaluated. We prove the thesis proving that each tuple \( \overline{t} \) satisfying conditions (1) and (2) does not satisfy condition (3). The first condition, i.e. \( \overline{t} \in \text{OS} \), implies that \( \overline{t} \) is contained in the result of the evaluation of the \( \text{Join} \) operator over the input streams modelled by the \( \text{IN} \) operators. That is, \( \overline{t} \) satisfies the predicates contained into \( \text{Join.EXP} \). Condition (2) implies that \( \overline{t} \) is authorized. According to our access control model, a user is authorized to access a tuple belonging to a joined stream if: (a) he/she has the read privilege over the set of tuples over which the join has been computed; (b) there exists an access control policy \( \text{acp}_j \) directly granting the join privilege to \( u \) on the joined stream. Thus, if condition (2) holds one of the following cases is true: (a) there exist two access control policies \( \text{acp}_{j1} \) and \( \text{acp}_{j2} \), and the join is performed over the tuples resulting from \( \beta(\text{acp}_{j1}) \) and \( \beta(\text{acp}_{j2}) \), thus \( \overline{t} \in \text{JoinResult} \subseteq \beta(\text{acp}_{j1}) \times \beta(\text{acp}_{j2}) \), where \( \text{JoinResult} \) is the stream resulting from the \( \text{Join} \) operator; (b) there exists an access control policy \( \text{acp}_j \), such that \( \overline{t} \in \beta(\text{acp}_j) \). Let us see why according to Algorithm 1, each tuple \( \overline{t} \) satisfying conditions (1) and (2) does not satisfy condition (3).

According to Algorithm 1, the authorized graphs in \( \text{AG} \) are defined as the union of \( \text{AG}_j \) and \( \text{AG}_{sj} \) (line 39). The authorized graphs in \( \text{AG}_j \) are generated by applying the \( \text{Join} \) operator over each pair of graphs \( \text{rag}_1 \in \text{RetAG}_1 \) and \( \text{rag}_2 \in \text{RetAG}_2 \) returned by the first recursion (lines 28-34). By construction, each graph \( \text{rag}_1 \in \text{RetAG}_1 \) (resp. \( \text{rag}_2 \in \text{RetAG}_2 \)) contains the operators encoding the expressions returned by the secure read operator (see line 4-8). Lemma 10.2 ensures the correctness of the
secure read operator, that is, it ensures that for each tuple \( t \) belonging to streams resulting from the evaluation of expressions returned by the secure read operator, there exists \( \text{acp}\in\text{SysAuth} \), such that \( t \in \beta(\text{acp}) \). The authorized graphs \( \text{AG}_{a_j} \) are computed by replacing into the graph \( G \) the \( \text{Join} \) operator with the operators encoding the expressions \( av \in \text{AuthViews} \) returned by the \( \text{SecJoin} \) operator (see lines 35-38). Lemma 10.4 ensures the correctness of the secure join operator, that is, it ensures that for each tuple belonging to the streams resulting from evaluation of expressions in \( \text{AuthViews} \), \( \exists \text{acp}\in\text{SysAuth} \), such that it belongs to \( \beta(\text{acp}) \).

Thus, since condition (2) holds, one of the following cases is true: there exist two access control policies \( \text{acp}_1 \) and \( \text{acp}_2 \) evaluated by the secure read operator such that \( \mathcal{T} \in \text{JoinResult} \subseteq \beta(\text{acp}_1)\times\beta(\text{acp}_2) \); or there exists an access control policy \( \text{acp}_j \) evaluated by the secure join operator such that \( \mathcal{T} \in \beta(\text{acp}_j) \). Thus, regardless of the access control policies (i.e., \( \text{acp}_1 \), \( \text{acp}_2 \) or \( \text{acp}_j \)), the \( \text{Join} \) operator is evaluated only over authorized tuples. Then, by condition (1), we know that \( \mathcal{T} \) resulted by the evaluation of the \( \text{Join} \) operator, thus \( \mathcal{T} \) is enclosed into one of the authorized streams generated by graphs in \( \text{AG}_j \), or it is enclosed into one of the authorized streams generated by graphs in \( \text{AG}_{a_j} \). In any case, \( \mathcal{T} \) is enclosed into a stream generated by an authorized graph in \( \text{AG} \), which implies that condition (3) does not hold.

**Induction.** Let us now assume that thesis holds for a graph \( G \) of dimension \( n \) (denoted in the following as \( G^n \)). We prove the thesis for a graph \( G^{n+1} \) of dimension \( n + 1 \). Graph \( G^{n+1} \) is given by graph \( G^n \) plus an additional operator \( \text{OP} \) before the \( \text{OUT} \) operator. Algorithm 1 evaluates graph \( G^{n+1} \) by recursively calling itself to evaluate graph \( G^n \), and then performing different steps on the basis of \( \text{OP} \). We will show that the completeness property is ensured also for graph \( G^{n+1} \), by considering each possible \( \text{OP} \).

**Case \( \text{OP}=\pi \) or \( \text{OP}=\sigma \).** We prove that the completeness property holds for \( G^{n+1} \), showing that it can not exist a tuple \( t \) such that (1) \( t \in \text{OS}^{n+1} \), where with \( \text{OS}^{n+1} \) we denote the stream generated by \( G^{n+1} \); (2) \( t \) is authorized, and (3) \( \forall a_g \in \text{AG}, t \not\in a_g \), where \( a_g \) is the stream generated by \( a_g \) and \( \text{AG} \) is the set of the authorized graphs returned by Algorithm 1 after \( G^{n+1} \) evaluation. In the following, we will use \( \text{AS}^{n+1} \) to denote the set of authorized streams generated by \( \text{AG} \). More precisely, we show that each tuple \( \mathcal{T} \) that satisfies condition (1), i.e., \( \mathcal{T} \in \text{OS}^{n+1} \), and condition (2), that is, \( \mathcal{T} \) is authorized, never satisfies condition (3), that is, \( \mathcal{T} \not\in \text{AS}^{n+1} \).

We recall that graph \( G^{n+1} \) is given by graph \( G^n \) plus the additional operator \( \text{OP} \) before the \( \text{OUT} \) operator. Thus, to elaborate \( G^{n+1} \) Algorithm 1 recursively calls itself by passing as input \( G^n \) (line 14). Then, it applies the \( \text{OP} \) operator directly over each of the returned authorized graphs, that is, those in \( \text{Ret}_{\text{AG}} \) (lines 15-17). By assumption, \( G^n \) satisfies the completeness property, that is, each tuple in \( \text{OS}^n \), which is authorized to \( u \), is included in the streams belonging to \( \text{AS}^n \). Thus, the \( \text{OP} \) operator is only entered by authorized tuples, which implies that the \( \text{OP} \) operator gives in output only authorized tuples, thus condition (2) always holds. Moreover, if \( t \in \text{OS}^{n+1} \), it means that \( \mathcal{T} \) has been output by \( G^{n+1} \), that is, \( \mathcal{T} \) has passed through the \( \text{OP} \) operator by satisfying the predicates in \( \text{OP} \). \( \text{EXP}s \), if \( \text{OP}=\sigma \) or containing only attributes in \( \text{OP}.\text{ATTs} \), if \( \text{OP}=\pi \). Thus, notwithstanding of the operator (i.e., \( \pi \) or \( \sigma \)), by condition (1), we know that \( \mathcal{T} \) satisfies \( \text{OP} \), thus \( \mathcal{T} \) is enclosed into one of the
authorized streams as generated by $AG$, which implies that condition (3) does not hold.

Case $OP=\Sigma$. The algorithm generates the authorized graphs in $AG$ as the union of $AG_a$ and $AG_{sa}$ (line 54). We will show that the completeness property holds for both authorized streams generated by graphs in $AG_a$ and $AG_{sa}$, denoted as $AS_a$ and $AS_{sa}$ respectively. In particular, the authorized graphs in $AG_a$ are computed by applying the $\Sigma$ operator over each graph $rag \in Ret_{AG}$ returned by the recursion over graph $G'$ (lines 46-48). This procedure is very similar to case $OP=\pi$ or $OP=\sigma$, and, as a consequence, the proof is similar too. Therefore, we omit it. We now focus on the authorized graphs in $AG_{sa}$. We prove the thesis showing that for each tuple $t$ such that (1) and (2) hold, then (3) does not hold. Let us consider the first condition, i.e., $t \in OS_{n+1}^+$, it means that $t$ has passed through the $\Sigma$ operator and it contains the aggregate value computed over attribute $A$. With respect to condition (2), since we are considering tuples generated by graphs in $AG_{sa}$, we have to look only for policies evaluated by the secure aggregate operator. Thus, if condition (2) is satisfied, then there exists an access control policy $acp \in SysAuth$, with $acp.priv = \Sigma,F$, such that $t \in (\Sigma,F,\Sigma.A,\Sigma.o,\Sigma.a)(\beta(acp))$. Let us see why according to Algorithm 1, each tuple $t$ satisfying conditions (1) and (2) does not satisfy condition (3). The authorized graphs in $AG_{sa}$ are computed by replacing the $\Sigma$ operator with the operators encoding the expressions $av \in AuthViews$ returned by the SecAggr operator (lines 49-53). Lemma 10.3 ensures the correctness of the secure aggregate operator, that is, it ensures that for each tuple belonging to the stream resulting from the evaluation of $av$, there exists $acp \in SysAuth$, such that the tuple belongs to $\Sigma(\Sigma,F,\Sigma.A,\Sigma.o,\Sigma.a)(\beta(acp))$. Thus, since condition (2) holds, there exists an access control policy $acp_a$ evaluated by the secure aggregate operator such that $t \in (\Sigma,F,\Sigma.A,\Sigma.o,\Sigma.a)(\beta(acp_a))$. Thus, the $\Sigma$ operator is evaluated over only authorized tuples, that is, tuples in $\beta(acp_a)$. Then, by condition (1), we know that $t$ satisfies the $\Sigma$ operator, thus the result of the aggregation over $t$ is enclosed into one of the authorized streams generated by graphs in $AG_{sa}$. Therefore, a contradiction arises, which proves the thesis.

Case $OP=Join$. Similarly to the previous case, the algorithm generates the authorized graphs in $AG$ as the union of $AG_j$ and $AG_{sj}$ (see line 39). We will show that the completeness property holds both for authorized streams generated by graphs in $AG_j$ and $AG_{sj}$. In particular, the authorized graphs in $AG_j$ are generated by applying the Join operator over each pair of authorized graph $rag_1 \in Ret_{AG_1}$ and $rag_2 \in Ret_{AG_2}$ returned by the recursion of Algorithm 1 over $G_1$ and $G_2^\sigma$, respectively (lines 29-34). Where $G_1$ (resp. $G_2$) is the graph obtained by $G^\sigma$ by deleting the $OP$ operator and the whole subgraph generating the first operand stream (resp. the second operand stream). This case is similar to case $OP=\pi$ or $OP=\sigma$, and, as a consequence, the proof is similar too. For this reason, we omit it.

Let us consider the authorized graphs in $AG_{sj}$. We prove the thesis showing that it can not exist a tuple $t$ such that conditions (1) and (2) hold, whereas condition (3) does not. The first condition, i.e., $t \in OS_{n+1}^+$, implies that $t$ satisfies the predicates contained into $Join.EXPs$. Condition (2) implies that $t$ is authorized. Since we are considering tuples in $AG_{sj}$, $t$ is authorized if there exists an access control policy $acp_j$ such that $t \in \beta(acp_j)$. In Algorithm 1, the authorized graphs in $AG_{sj}$ are computed.
by replacing into the graph $G^{n+1}$ OP with the operators encoding the expression $av \in AuthViews$ returned by the SecJoin operator (lines 35-38). Lemma 10.3 ensures the correctness of the secure join operator, that is, it ensures that for each tuple $t$ belonging to the stream resulting from the evaluation of expression $av$, $\exists acp \in SysAuth$, such that $t \in \beta(acp)$. Thus, since condition (2) holds, there exists an access control policy $acp_j$ evaluated by the secure join operator such that $t \in \beta(acp_j)$. By condition (1), we know that $\bar{t}$ satisfies the Join operator, thus $t$ is enclosed into one of the authorized streams generated by graphs in $AG_{aj}$, which implies that condition (3) does not hold.