Enforcing Access Control Over Data Streams

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

Kian Lee Tan
National University of Singapore, Singapore
tankl@comp.nus.edu.sg

ABSTRACT

Access control is an important component of any computational system. However, it is only recently that mechanisms to guard against unauthorized access for streaming data have been proposed. In this paper, we study how to enforce the role-based access control model proposed by us in [5]. We design a set of novel secure operators, that basically filter out tuples/attributes from results of the corresponding (non-secure) operators that are not accessible according to the specified access control policies. We further develop an access control mechanism to enforce the access control policies based on these operators. We show that our method is secure according to the specified policies.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Data stream, Security, Access control

1. INTRODUCTION

Data stream management systems (DSMSs) have been increasingly used to support a wide range of real-time applications (e.g., battle field and network monitoring, telecommunications, financial monitoring, sensor network, and so on). In many of these applications, there is a need to protect sensitive streaming data from unauthorized accesses. For example, in battle field monitoring, the positions 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.

SACMAT'07, June 20-22, 2007, Sophia Antipolis, France.

Many access control models have been designed for traditional database systems [7]. However, these cannot be readily adapted to data stream applications. First, the long-running queries over unbounded data streams means that access control enforcement is data-driven (i.e., triggered whenever data arrive). This implies that it is computationally expensive to enforce access control. Second, as data are streaming, temporal constraints (e.g., sliding windows) become more critical. Third, operators are typically shared across multiple queries. Hence, access control can hardly be enforced at data stream or operator levels. In [5], we proposed a role-based access control model based on the Aurora model [2]. Objects to be protected are essentially views (or rather queries) over data streams. The model supports two types of privileges - Read privilege for operations such as Filter, Map, BSort, 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 over the data within a specified time window. The work in [5] largely presents the formal semantics of the model, and shows how access control policies can be specified. However, it does not consider access control enforcement.

In this paper, we build on the access control model in [5], and address how to enforce access control under the Aurora data stream prototype. Aurora query processing exploits a data-flow paradigm, modeling queries as a loop-free direct graph of operations (i.e., boxes). Users specify queries by adding boxes to the graph. In such a model, an access request is therefore a request to add a box in a graph. Thus, unlike conventional RDBMSs, our access control mechanism operates at query definition time, and hence avoids run-time overhead. Whenever a user tries to insert a box in a graph, the reference monitor checks the authorization catalogs to verify whether the access can be partially or totally granted, or should be denied. In case of partially authorized access request, the specified query is rewritten in such a way that it accesses only authorized data. To realize our access control mechanism, 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 (non-secure) operators those tuples/attributes that are not accessible according to the access control policies. We believe that this is a major extension to [5], since our secure operators together with the query rewriting mechanism are the essential components to integrate a reference monitor into a data stream engine.

Recently, Lindner and Meier also looked at the problem of securing data streams [8], and proposed a *Owner extended RBAC* (OxRBAC) security model to protect data stream from unautho-

rized accesses. Besides an object level security model that restricts accesses to objects (schemas, catalogs, queries, streams), there is a data level security model to protect data access. The basic idea of the data level security is to apply a newly designed operator, called SECFILTER, at the query output stream to filter out output tuples that do not conform to the access control rules. This "post-processing" approach means that many wasted computations are performed. Moreover, as noted in [8], it is possible for a user to remain "connected" to an output stream though he/she may not receive any output tuples (e.g., because his/her access rights have been revoked). This is not desirable. Finally, because the proposed framework is non-intrusive, SECFILTER cannot handle certain queries that involve output streams obtained from aggregating data over multiple streams.

The remainder of this paper is organized as follows. In the next section, we provide some background to this work. In Section 3, we present a set of secure operators derived from the Aurora algebra. Section 4 presents the access control mechanism to enforce access control in Aurora. Section 5 presents some formal results. Finally, we conclude the paper with directions for future work in Section 6.

2. BACKGROUND

In this section, we first review the Aurora data model, query model and algebra. Then, we review the access control model in [5].

2.1 Aurora query model and algebra

We adopt the Aurora data stream model and algebra [2]. We have cast our work in the Aurora framework as it is relatively mature (in comparison with the other stream processing systems [3, 6]). Moreover, Aurora has been successfully transferred to the commercial domain (i.e., the StreamBase engine [9]), and redesigned with distributed functionalities (i.e., Borealis [1]).

Aurora data model. A stream consists of an append-only sequence of tuples with the same schema. In addition to standard attributes $\mathtt{A}_1, \ldots, \mathtt{A}_n$ the stream schema contains an additional attribute, denoted as ts. Attribute ts stores the time of origin of the corresponding tuple, thus it can be exploited to monitor stream attributes values over time.

Aurora query model. The Aurora query processor exploits a dataflow paradigm - queries are modeled as a loop-free direct graph of operations (called boxes), where tuples flow through all the operations defined in the graph (called network). Figure 2(b) in Section 4.2 illustrates an example of Aurora network with two input streams, namely Health and Position, where Health is passed through two different boxes (i.e., Filter and Map) before being joined with Position.

Aurora algebra. The Aurora algebra consists of a set of operators to be applied on data streams. The Filter box, like a relational selection, applies several distinct selections at the same time on a stream, and output tuples that satisfy the predicates. The Filter syntax is: $\mathtt{Filter}(P_1, \ldots, P_n)(\mathtt{S})$, where $P_1, \ldots, P_n$ are predicates over stream S. The result consists of $n+1$ different streams $S^1 \ldots S^n$, where each stream S^j contains those tuples of S satisfying predicate P_j, $j \in \{1, \ldots, n\}$. Moreover, tuples that do not satisfy any predicate among $P_1, \ldots, P_n$ are returned in an additional stream S^{n+1}. In this paper, we do not consider the $(n+1)$th stream by simply assuming that these tuples are deleted. Moreover, if the predicates are omitted then all the tuples in the input stream are returned.

Another operator, the Map box, can be considered as a generalized projection operator. Instead of projecting the value of an attribute A_i, it projects the result of an arbitrary function applied on it. The syntax of Map is the following: $\mathtt{Map}(\mathtt{A}_i=\mathtt{F}_i, \ldots \mathtt{A}_j=\mathtt{F}_j)(\mathtt{S})$, where $A_i, \ldots, A_j$ are a subset of the attributes of S, and $F_i, \ldots F_j$ are arbitrary functions over the input stream.[1]

A further operator is Bsort, which sorts the tuples of a stream. Aurora also provides an aggregate operator, i.e., Aggregate box, that computes, according to a sliding window-based approach, an aggregate over data streams. The Aggregate operator receives as input both the size of the window and an integer, called hereafter step, specifying how to advance the window when it slides. The simplified syntax used throughout the paper is the following: Aggregate(Op,s,i)(S), where OP is a pair (F,A) such that F is either an SQL-style aggregate operation or a Postgres-style user-defined function, A is the attribute of S on which F has to be computed, whereas s and i are the size and step, respectively.

Aurora also supports a binary Join operator, where the join predicate P is specified as input. More precisely, the syntax of this operator is $\mathtt{Join(P)}(\mathtt{S}_1,\mathtt{S}_2)$. A further operator is the Resample box, which can be helpful to align pairs of streams. Finally, we also have the Union box, which is used to merge a set of streams, having a common schema, into a unique output stream.

2.2 An access control model for data streams

In this section we summarize the main characteristics of the access control model proposed in [5] and introduce the definitions we need throughout the paper. We refer the interested reader to [5] for a full description of the model and its semantics.

The access control model in [5] is a role-based model specifically tailored to the protection of data streams. Privileges supported by the model are of two different types that correspond to the two different classes of operations provided by the Aurora algebra (see Section 2.1): a read privilege, that authorizes a user to apply the Filter, Map and Bsort and Aggregate box on a data stream, that is, all operations that require to read tuples from the data stream. Additionally, it authorizes to apply the Union, Join and Resample box 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 aggregate functions allowed by Aurora and are provided to grant a user the authorization to perform an aggregate operations without accessing all the tuples over which the operation is computed. As introduced in Section 2.1, Aurora supports both SQL-style aggregate operations and Postgres-style user-defined functions. To be as system independent as possible, in the following we consider as aggregate functions only the standard SQL-style functions. Thus, the aggregate privileges are: 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 their attributes values. Additionally, the model allows the Security Administrator (SA) to restrict the exercise of the read privilege only to a subset of a joined stream. 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. We

[1] In the paper, for simplicity, we assume that F is the identity function.

subject	protection object			priv	gtc	wtc	
	streams	attributes	expressions			size	step
Captain	Position	Pos, SID	Position.Platoon=self.Platoon	read	-	-	-
Captain	Position	Pos, SID	Position.Platoon≠self.Platoon ∧ Pos≥Target(a)-δ ∧ Pos≤Target(a)+ δ	read	[TAction_start(a), TAction_end(a)]	-	-
Soldiers	Position	Pos	Pos≥Target(a)-δ ∧ Pos≤Target(a)+ δ	avg	[TAction_start(a), TAction_end(a)]	1	1
Doctor	Health	Heart, SID	Position.Platoon=self.Platoon	read	-	-	-
Doctor	Health, Position	Heart, SID	Position.SID= Health.SID ∧ Pos≥Target(a)-δ ∧ Pos≤Target(a)+ δ	read	[TAction_start(a), TAction_end(a)]	-	-

Table 1: Examples of access control policies for data streams

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 corresponds to the SELECT, FROM and WHERE clauses of an SQL statement.The formal definition of protection object specification is given below.

Definition 1. **(Protection object specification)** [5]. A *protection object specification* `p_obj` is a triple `(STRs,ATTs,EXPs)`, where:

- `STRs` is a set of names or identifiers of streams $\{S_1, \ldots, S_n\}$;
- `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 $(S_1 \times \cdots \times S_n)$ of the streams in `STRs`. If `ATTs` is equal to symbol '*', it denotes all the attributes belonging to the schema of the stream resulting from $(S_1 \times \cdots \times S_n)$.
- `EXPs` is a boolean formula, built over predicates of the form: $\texttt{A}_i \oplus$ `value` or $\texttt{A}_i \oplus \texttt{A}_j$, where $\texttt{A}_i$, $\texttt{A}_j$ are attributes belonging to the schema of the Cartesian product $(S_1 \times \cdots \times S_n)$, $\oplus$ is a comparison operator of the Aurora algebra, and `value` is a value compatible with the domain of $\texttt{A}_i$. If `EXPs` is empty, it denotes all the tuples in $(S_1 \times \cdots \times S_n)$. □

Given a protection object specification `p_obj`, we use the dot notation to refer to its components.

Note that, the format of protection object specifications does not allow the SA to specify the `read` privilege for a view consisting of the union of more data streams because we assume that the `Union` operation is authorized if the requesting user has the `read` privilege on the two operand streams.

The access control model in [5] also allows the SA to specify two different types of temporal constraints, that is, `general` and `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: `[begin, end]`, where `begin` and `end` are the lower and upper bounds of the interval, `begin` ≤ `end`, and `end` can assume the infinite value.[2] The `begin` and `end` values can be explicitly specified by the SA or can be returned by a predefined set of system functions $\mathcal{SF}$. For instance, we assume a function `TAction_start()`, which returns the time when a given action starts, and a function `TAction_end()` that 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`.

[2]We assume that `begin` and `end` values are specified by means of an SQL-like syntax.

The other class of constraints is related to window-based aggregate operators supported by the Aurora algebra. A window is specified by two information: the window's size and the advance step. A window time constraint *wtc* can therefore be defined by a pair: `[size,step]`, denoting the minimum size and step allowed in a window-based operation. The value 0 for size and/or step denotes that the corresponding aggregate operation can be performed with any size and/or step.

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

Definition 2. **(Access control policy for data streams)** [5]. 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 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` components, respectively. 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 2.1. *Here and in what follows we consider the military domain presented in [2] as running example. In this scenario, stream data are used to monitor positions and health conditions of platoon's soldiers. Hereafter, we consider two data streams, namely,* `Position` *and* `Health`*, with the following schemas:* `Position(ts,SID,Pos,Platoon)`, `Health(ts,SID, Platoon,Heart,BPressure)`*, where the* `SID` *and* `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. An example of* `SysAuth` *catalog referring to the* `Position` *and* `Health` *streams is given in Table 1.*

In particular, the first policy grants captains the `read` *privilege on the position and id of soldiers belonging to their platoons, where this condition is modeled as a predicate (i.e.,* `Position.Platoon=self.Platoon`*).*[3] *The second access control policy authorizes captains to read the id and position of soldiers not belonging to their platoons, but whose position is near to the target of action* `a`*, that is, whose positions distance at most* δ *from action* `a` *target's position (*`Pos`≥ `Target(a)`-δ∧`Pos`≤`Target(a)`+δ*).*[4]

[3]We 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.

[4]We assume a function `Target()` that returns the position of the target of a given action.

Additionally, this privilege is granted only during the time of action a. *By the third policy soldiers are allowed to compute the average of the positions of soldiers that are close to the target of action* a. *Moreover, this policy states that the average can be computed only during the action time and with windows of minimum 1 hour and with 1 as minimum step.*

Finally, the fourth access control policy states that a doctor is authorized to monitor the heart beats only of those soldiers belonging to his/her platoon, whereas, the fifth policy grants a doctor the possibility of monitoring the heart beats of all soldiers (not only those belonging to his/her platoons) but only if the soldier position is near to the target of action a *and only during the action time.* △

3. SECURE OPERATORS

In this section, we present a set of operators, derived from those of the Aurora algebra to keep into account the specified access control policies. The operators basically prune from the results of the corresponding Aurora operators those tuples/attributes that are not accessible according to the specified access control policies. These operators are the basis of the access control mechanism described in Section 4.2.

The first operator, called *Secure view*, takes as input a stream and an access control policy and 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. The view is represented by the corresponding expression in the Aurora algebra.

Definition 3. **(Secure view)**. Let S be a stream, and acp be an access control policy such that acp.obj.STRs = S. Let $Attr(S)$ be the set of attributes belonging to S's schema. The Secure view, Sec_View, of S wrt policy acp is defined as follows:

$$\texttt{Sec\_View}(S, acp) = \texttt{Map(att)(Filter(P)}(S))$$

where:

$$\texttt{att} = \begin{cases} Attr(S) \cap \texttt{acp.obj.ATTs} & \text{if acp.obj.ATTs} \neq * \\ Attr(S) & \text{otherwise} \end{cases}$$

$$\texttt{P} = \begin{cases} \texttt{acp.obj.EXPS} & \text{if acp.obj.EXPs is not empty} \\ \texttt{P} \wedge (\texttt{ts} \geq \texttt{acp.gtc.begin} \wedge \texttt{ts} \leq \texttt{acp.gtc.end}) & \text{if acp.gtc is not null} \\ \text{null} & \text{if acp.obj.EXPs is empty and acp.gtc is null} \end{cases}$$

□

Based on the Sec_View operator we can define the Sec_Read operator, which takes as input a user u and a data stream S and returns the view of S over which u can exercise the read privilege, according to the policies in 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 Sec_Read is actually a set of views, each of which denoted by the corresponding expression in the Aurora algebra.

Definition 4. **(Secure read)**. Let S be a stream and u be a user. Let *Role(u)* be the set of roles u is authorized to play, and let $Pol(S, u)$ be the set of read access control policies specified for S and which apply to u, that is, $Pol(S, u)$ ={$acp \in$ SysAuth| acp.obj.STRs = S, acp.sbj∩*Role(u)*≠ ∅, acp.priv = read}. The Secure read operator, Sec_Read, is defined as follows:

$$\texttt{Sec\_Read}(S, u) = \bigcup_{acp_j \in Pol(S,u)} \{\texttt{Sec\_View}(S, acp_j)\}.$$

□

EXAMPLE 3.1. *Let us consider the access control policies introduced in Table 1, assuming that there exists a user, say Paul, belonging to platoon X and authorized to play only the captain role. Moreover, we assume that action* a *is not currently taking place. In this case, only the first policy is applicable to Paul, which authorizes Paul to access position and id of soldiers belonging to his platoon. This is exactly the view, hereafter called $AuthView$, obtained by the evaluation of* Sec_Read(Position, *Paul). Indeed, according to Definition 4, $AuthView$ is defined as the union of the views returned by the Secure view operator, for each policy in Pol(Position,Paul). Pol(Position,Paul) consists of the first and the second access control policy of Table 1, say* $\texttt{acp}_1$ *and* $\texttt{acp}_2$. *Thus, $AuthView$ consists of the union of* Sec_View(Position,acp_1) *and* Sec_View(Position,acp_2). *According to Definition 3,* Sec_View(Position,acp_1) *returns the following Aurora expression:* Map(Pos,SID)(Filter(Position.Platoon=X)(Position)) *(denoted in what follows as $View_1$). In contrast,* Sec_View(Position,acp_2) *is given by:* Map(Pos,SID)(Filter(Position.Platoon≠X∧Pos≥Target(a)−δ∧Pos≤Target(a)+δ∧ts≥TAction_start(a)∧ts≤TAction_end(a))(Position)). *However, we assume that if an action is not currently taking place the evaluation of* Target(), TAction_start(), *and* TAction_end() *for that action returns null. Thus, all the predicates referring to these functions in the Filter operator above null, and therefore no tuples are selected by the* Filter *operator. Thus, the view of* Position *stream on which Paul has the* read *privilege consists only of $View_1$, that is, all the positions and ids 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 operation over a stream S and a user u, considers the policies specified over S for the requested aggregate operation that apply to u 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 Aurora algebra. Since in the case of aggregate operations both policies and operation requests may have some associated temporal constraints (i.e., the window size and the step), these may be considered when determining the result of Secure aggregate.

Definition 5. **(Secure aggregate)**. Let S be a stream, u be a user, Op be an aggregate operator consisting of an SQL-like aggregate function, $Op.F$, and an attribute of S, $Op.A$. Let s and i be two natural numbers, and let $Pol_{agg}(S, u)$ be the set of access control policies in SysAuth granting u the $Op.F$ privilege over attribute $Op.A$, that is, $Pol_{agg}(S, u)$ ={$acp \in$ SysAuth| acp.obj.STRs = S, $Op.A \in acp$.obj.ATTs, acp.sbj∩*Role(u)*≠ ∅, acp.priv = $Op.F$}. The Secure aggregate operator, Sec_Aggr is defined as follows:

$$\texttt{Sec\_Aggr}(S, Op, s, i, u) = \bigcup_{acp_j \in Pol_{agg}(S,u)} \{\texttt{Aggregate}(Op, \texttt{max}_{size}, \texttt{max}_{step})(\texttt{Map}(Op.A)(\texttt{Filter(P)}(S)))\}$$

where: $\texttt{max}_{size}$ = max($\texttt{acp}_j$.wtc.size,s), $\texttt{max}_{step}$ = max($\texttt{acp}_j$.wtc.step,i), and

$$\texttt{P} = \begin{cases} \texttt{acp}_j\texttt{.obj.EXPS} & \text{if acp}_j\text{.obj.EXPs is not empty} \\ \texttt{P} \wedge (\texttt{ts} \geq \texttt{acp}_j\texttt{.gtc.begin} \wedge \texttt{ts} \leq \texttt{acp}_j\texttt{.gtc.end}) & \text{if acp}_j\text{.gtc is not null} \\ \text{null} & \text{if acp}_j\text{.obj.EXPs is empty and acp}_j\text{.gtc is null} \end{cases}$$

EXAMPLE 3.2. *Suppose that a user authorized to play only the soldier role, say Alice, is interested in the average position of soldiers calculated with windows of 5 hours and 5 as step. Moreover, let us assume that action* a *is taking place, whose target has position 1000, whose starting time is 105000, whereas ending time is infinite, since the action has not ended yet.*

According to the access control policies in Table 1, since Alice's role is soldier, she is only authorized to perform avg operations on the Pos *attribute of those tuples referring to soldiers that are close to the target of action* a*. Moreover, the average can be performed with at minimum a window of size 1 hour and 1 as step. Let us consider the view returned by the Secure aggregate operator. In this case,* Pol_{agg}*(*Position*,Alice) consists only of the third access control policy. Thus,* Sec_Aggr*(*Position*,(*Avg, Pos*),Alice,5,5) is equal to* Aggregate*((*Avg,Pos*),5,5)(*Map(Pos)(Filter(Pos≥1000-δ∧Pos≤1000+δ∧ts≥105000 ∧ ts≤ ∞)(Position))*), since 5 is the maximum step (i.e, size) between the step (i.e, size) 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:* Pos≥1000-δ∧Pos≤1000+δ*, that is, tuples near to the target of* a*, and:* ts≥105000∧ts≤ ∞*, that is, tuples generated during action* a*. Then, only for those values, it calculates the average with windows of size 5 hours and with 5 as step.* △

The last operator we need to define, called *Secure join* is used to manage join operations. Indeed, according to our access control model, we can 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 are those that have more than one stream in the obj.STRs component. Similarly to Sec_Aggr, the Secure join operator returns the set of "views" over the joined stream corresponding to the authorized attributes and tuples.

Definition 6. **(Secure join)**. Let S_1 and S_2 be two streams, P a join predicate over S_1 and S_2, and u be a user. The Secure join operator, Sec_Join, is defined as follows:

Sec_Join(S_1, S_2, P, u) = Sec_Read(Join(P)(S_1, S_2),u). □

EXAMPLE 3.3. *Consider a user, say Rick, who is authorized to play only the role doctor and belongs to platoon X. Suppose that Rick is interested to know the position, id, and health information of those soldiers which are across some border k (modeled as* Pos≥k*). Since the position of a soldier is stored in the* Position *stream, whereas health information is in the* Health *stream, he needs to perform a join of the* Position *and* Health *streams, and selecting only those tuples which refer to soldiers whose position satisfies the condition above. According to the access control policies in Table 1, this operation is possible only during action* a *and only for those tuples referring to soldiers whose positions are near to the action's target. Let us assume that action* a *is not currently undergoing, and see how the Secure join operator evaluates over the requested query, that is,* Sec_Join *(*Position,Health,*(*Position.SID=Health.SID ∧ Pos ≥k*),Rick).*

According to Definition 6, the above expression first generates the joined stream, say S3, resulting from Join*(* Position.SID= Health.SID ∧Pos≥k*)(*Position, Health*). Then, it passes S3 to the Secure read operator, with the aim of generating the corresponding authorized view for Rick.* Sec_Read*(S3,Rick) considers the set of access control policies* $Pol(S3, Rick)$ *which consists only of the last access control policy of Table 1, say* acp_5*. By Definition 4, this implies that* Sec_Read *returns the view generated by* Sec_View*(S3,*acp_5*), that is, the tuples returned by the evaluation of the following Aurora expression:* Map(Heart,SID)(Filter(Position.SID=Health.SID ∧Pos≥k∧ts≥null∧ts≤null∧Pos≥null-δ∧Pos≤null +δ)(Position,Health)). *Since the last predicate in the above* Filter *operator evaluates null, no tuples are selected, thus the authorized view is empty.* △

Note that the Aurora algebra defines a further binary operator, that is, Resample. However this operator is defined as a semijoin, thus we omit the definition of Sec_Resample since it is very similar to Definition 6. We recall moreover that, in our access control model we assume that a policy cannot be specified on a view resulting from the union of two or more streams. This is due to the fact that we assume that a user can perform the union of two streams only if he/she has the read privilege over them. For this reason, we do not need to define the Secure union operator.

4. ACCESS CONTROL ENFORCEMENT ON A REAL DATA-STREAM PROTOTYPE: AURORA

In this section, we introduce an access control mechanism enforcing the access control policies presented in Section 2.2, by considering Aurora as the reference target. Enforcing access control having as target the Aurora query model (cfr. Section 2.1) implies to regulate whether a user can (or cannot) insert a box into a graph. As an example, consider the access control policy stating that a captain can read position of soldiers belonging to his/her platoons, whereas he/she cannot access the position of other platoon's soldiers. Whenever a captain tries to apply a Filter box to the Position stream, the access control mechanism must enforce the access control policy by limiting the Filter operation only to those tuples referring to soldiers belonging to captain's platoon.

According to the Aurora query model, access control enforcement can therefore be applied at *query definition time*. The access control mechanism starts up whenever a user requests to apply a box. If the user is not authorized to perform the operation denoted by the box, the access is denied. In case the user is partially authorized, the access control module rewrites the query in terms of the allowed operations, otherwise if the operation denoted by the box can be fully authorized, the reference monitor does not make any modification to the user query.

In the next sections we first introduce how we represent access requests, then we show how the secure operators introduced in Section 3 can be exploited for access control enforcement.

4.1 Access requests

According to the Aurora query model, an access request is equivalent to the request of a user to apply a box to one or more streams. Thus, an access request can be modeled in terms of three components: (i) the user submitting the access request; (ii) the stream(s) on which he/she requires to apply the box; (iii) the operator corresponding to the box to be inserted. We can therefore formalize an access request R as a triple (u,Objs,p), where u is the (identifier of the) requesting user, p is the Aurora operation that u requires to execute on the objects (i.e., streams) specified in Objs.

Before introducing the access control mechanism, we need to illustrate how streams are modeled in Objs. According to the Aurora query model a user is able to apply a box to one or more input

streams, as well as to one or more *internal streams*, that is, streams resulting after one or more input streams have flowed through the processing operations (i.e., boxes) defined in a graph. For what concerns access control, the first case is the simplest one. Indeed, in the first case, the reference monitor must verify whether there exists an access control policy `acp` applying on the input streams, that is, having `acp.obj.STRs` equal to one of the requested stream, and/or applying directly on the stream resulting from the requested operation (in case of join, resample and aggregate operations). In this case, the reference monitor has to decide whether the access can be fully or partially granted. Otherwise the access is denied.

The second case is more complicated and has some similarities to an access request on a view in a RDBMS, in that an internal stream is generated by applying one or more Aurora operators, according to the order specified by a given graph, on a set of input streams. For instance, with reference to Figure 2(b), if a user requires to apply a box on the stream resulting from the join box, it is not immediate to detect which access control policies should be considered to answer the access request.

To easily detect the access control policies that must be considered to evaluate access requests on internal streams, we have decided to model each internal stream *S*' on which the request to apply a box is performed with a protection object like representation, similar to the one given in Definition 1. Thus, we model an internal stream as a triple `(STRs,ATTs,EXPs)`. Note that this representation can be adopted even for input streams, by simply setting the `ATTs` component to * and omitting the `EXPs` component.

An internal stream can be defined in terms of (the portion of) the graph by which it results. Therefore, given an internal stream *S*' and the corresponding graph `G`, the protection object like representation of *S*' can be defined in terms of the boxes contained into `G`. More precisely, given a graph `G`, not containing an `Union` box, 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 `Map` boxes of the graph `G` are projected, and all predicates specified in the `Filter`, `Join` and `Resample` boxes are applied. Thus, a possible approach to convert a graph `G` into a protection object like representation `Objs` is to traverse the whole graph by collecting into the `Objs.ATTs` component all attributes specified in the encountered `Map` boxes, in the `Objs.EXPs` component the predicates specified in the encountered `Filter`, `Join` and `Resample` boxes, and by inserting into the `Objs.STRs` component all the input streams. If the graph contains an `Union` box, the function operates in the same way with the only difference that, when an `Union` box is encountered, instead of the Cartesian product of the input streams, only one of the two is considered in the protection object like representation. As it will be more clear from the explanation below, since, `Union` applies to two streams with the same schema and returns a stream with the same schema as the input streams, one of the two input streams can be indifferently selected in the protection object like representation.

Figure 1 presents a function to generate the protection object like representation of an internal stream. The protection object like representation is recursively obtained starting from the end of the graph, i.e., the last box applied over the internal streams, till reaching the input streams. In particular, each time the function encounters a box, say `box`, it verifies whether it is applied over a single or over more than one streams (step 4). In the first case, it further verifies if the stream, say `S_in`, over which `box` is applied is an input or an internal stream. Then, the function performs different steps on the basis of the `box` type. In the case `S_in` is an input stream and `box` denotes the `Map` or `Aggregate` operator, the function sets the protection object like representation being returned, called

FUNCTION 1. **Function proc_obj(G)**

INPUT: *A graph G representing an Aurora query.*
OUTPUT: *S_out=(STRs,ATTs,EXPs), that is, the protection object like representation of the stream resulting from G.*
1 `S_out.STRs` *and* `S_out.ATTs` *are initialized to be empty*
2 Let `box` *be the last Aurora box applied on the stream resulting from G*
3 Let `S_in` *be the input stream(s) of* `box`
4 **If** *(*|`S_in`| *= 1)*
  a **If** *(*`S_in` *is an input stream)*
    i `S_out.STRs:={S_in}`
    ii **If** *(*`box = Map` *OR* `box = Aggregate`*)*
      1 *Let* `att` *be the set of attributes specified in* `box`
      2 `S_out.ATTs:=att`, `S_out.EXPs` *is set empty*
      3 **Return** `S_out`
    iii **If** *(*`box = Filter`*)*
      1 *Let* `exp` *be the predicate specified in* `box`
      2 `S_out.ATTs:=*`, `S_out.EXPs:=exp`
      3 **Return** `S_out`
  Else
    iv Let G' be the subgraph of G generating `S_in`
    v `S_res:=proc_obj(G')`
    vi **If** *(*`box = Map` *OR* `box = Aggregate`*)*
      1 *Let* `att` *be the set of attributes specified in* `box`
      2 `S_out.STRs:=S_res.STRs`, `S_out.ATTs:=att`$\cap$ `S_res.ATTs`,`S_out.EXPs:=S_res.EXPs`
      3 **Return** `S_out`
    vii **If** *(*`box = Filter`*)*
      1 *Let* `exp` *be the predicate specified in* `box`
      2 `S_out.STRs:=S_res.STRs`, `S_out.ATTs:=S_res.ATTs`, `S_out.EXPs:=exp`$\vee$ `S_res.EXPs`
      3 **Return** `S_out`
  EndIf
Else
    b Let `S1_in` *and* `S2_in` *be the streams given in input to* `box`
    c **Case:** *(*`S1_in` *and* `S2_in` *are input streams)*
      i **If** *(* `box= Union`*)*
        1 `S_out.STRs:=S1_in`, `S_out.ATTs:=*`, `S_out.EXPs` *is set empty*
        2 **Return** `S_out`
      ii **If** *((*`box = Join`*) OR (*`box = Resample`*))*
        1 *Let* `exps` *be the join predicate in* `box`
        2 `S_out.STRs:={S1_in, S2_in}`, `S_out.ATTs:=*`, `S_out.EXPs:=exps`
        3 **Return** `S_out`
    d **Case:** *((*`S1_in` *is an input stream AND* `S2_in` *is an internal stream) OR (*`S1_in` *is an internal stream AND* `S2_in` *is an input stream))*
        i Let `S_int` *and* `S_inp` *be empty streams*
        ii **If** *(*`S2_in` *is an internal stream)*
          1 `S_int:=S2_in`, `S_inp:=S1_in`
        Else
          2 `S_int:=S1_in`, `S_inp:=S2_in`
        iii Let G' be the subgraph of G generating `S_int`
        iv `S_res:=proc_obj(G')`
        v **If** *(*`box = Union`*)*
          1 `S_out.STRs:=S1_in`, `S_out.ATTs:=S_res.ATTs`, `S_out.EXPs:=S_res.EXPs`
          2 **Return** `S_out`
        vi **If** *((*`box = Join`*) OR (*`box = Resample`*))*
          1 *Let* `exps` *be the join predicate specified in* `box`
          2 `S_out.STRs:={S_res}`$\cup$`S_inp`, `S_out.ATTs:=S_res.ATTs`, `S_out.EXPs:=S_res.EXPs` $\vee$ `exps`
          3 **Return** `S_out`
    e **Case** *(*`S1_in` *and* `S2_in` *are internal streams)*
        i Let G1 be the subgraph of G generating `S1_in`
        ii `S1_res:=proc_obj(G1)`
        iii Let G2 be the subgraph of G generating `S2_in`
        iv `S2_res:=proc_obj(G2)`
        v **If** *(*`box = Union`*)*
          1 `S_out.STRs:=S1_res.STRs`, `S_out.ATTs:=`, `S1_res.ATTs` `S_out.EXPs:=S1_res.EXPs` $\vee$ `S2_res:=EXPs`
          2 **Return** `S_out`
        vi **If** *((*`box = Join`*) OR (*`box = Resample`*))*
          1 *Let* `exps` *be the join predicate specified in* `box`
          2 `S_out.STRs:=S1_res.STRs`$\cup$`S1_res:STRs`, `S_out.ATTs:=S1_res.ATTs`$\cup$`S2_res.ATTs`, `S_out.EXPs:=S1_res.EXPs`$\vee$`S2_res.EXPs`$\vee$`exps`
          3 **Return** `S_out`
EndIf

Figure 1: A function for generating the protection object like representation of a data stream

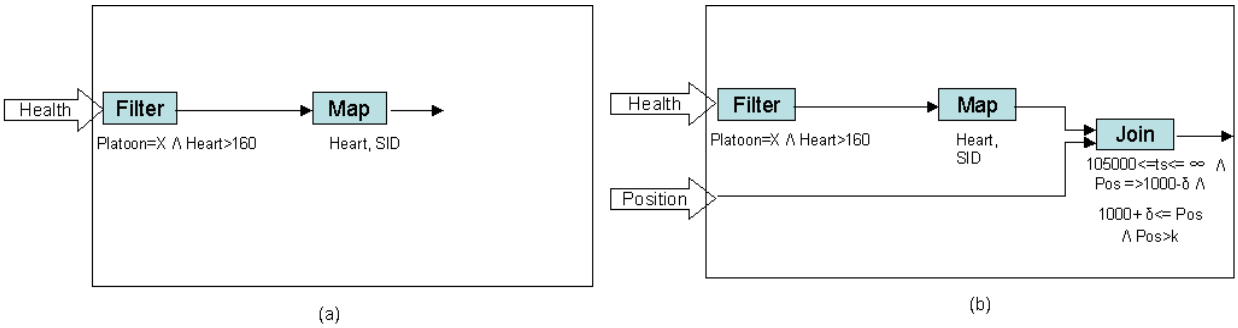

Figure 2: Aurora queries

S_out, as follows: the ATTs component contains the attributes specified in box, the STRs component is set to S_in, whereas the EXPs component is omitted (step $4.a.ii$). In contrast, in the case S_in is an input stream and box denotes the Filter operator,[5] the function sets S_out as follows: the EXPs component contains the predicate specified in box, whereas the STRs component is set to S_in, and the ATTs component is set equal to * (step $4.a.iii$).

If S_in is an internal stream, the function starts by recursively calling itself by passing the subgraph G' generating the stream S_in and storing the resulting protection object like representation in R_res (steps $4.a.iv$, $4.a.v$). Then, if box denotes the Map or Aggregate operator, it sets the ATTs component of S_out as the intersection of the set of attributes in S_res with the set of attributes specified in box, whereas the EXPs component is set equal to the value of EXPs component of S_res (steps $4.a.vi$.1-3). In contrast, if box denotes the Filter operator, the function sets the EXPs component of S_out as the disjunction of the predicate in R_res with the predicate specified in box (step $4.a.viii$). The STRs component of S_out is set equal to the corresponding one in S_res, in both cases.

If the box is applied on two streams, say S1_in and S2_in, the function first verifies whether both, one, or none of them are input streams (steps $4.c$, $4.d$, and $4.e$). Let us consider the first case. If box denotes the Union operator, the function inserts in S_out only a single stream, either S1_in or S2_in (step $4.c.i$.1).[6] Indeed, according to the definition of protection object inserting both the streams implies the Cartesian products S1_1× S2_in, which obviously does not respect the semantics of the Union operator. Then, the function considers the case when box denotes the Join or Resample operator (steps $4.c.ii$.1-3). In this case, the resulting protection object like representation is defined by inserting both S1_in and S2_in in the STRs component, and by setting the EXPs component equal to the join predicate specified in box. If both S1_in and S2_in are internal streams (step $4.e$), the function calls itself twice, by passing as input the subgraph G1 (i.e., G2) generating the stream S1_in (i.e., S2_in) and storing the resulting protection object like representation into S1_res (i.e., S2_res), respectively (steps $4.e.i$-iv). Then, if box denotes the Union operator, proc_obj() sets S_out components as follows: since the Union can be performed only if the streams' schema are identical, the function stores into the STRs and ATTs components the corresponding components of S1_res or S2_res, whereas the EXPs component is set as the disjunction of the predicates in the EXPs components of S1_res and S2_res (step $4.e.v$). By contrast, if box is the Join or Resample operator, proc_obj() sets the STRs and ATTs components of S_out as the union of S1_res and S2_res corresponding components, whereas to the EXPs component is added the join predicate in addition to the disjunction of the predicates contained into the EXPs components of S1_res and S2_res (steps $4.e.vi$.1-2). Then, the resulting protection object like representation S_out is returned. We omit the discussion of the case in which there only one between S1_in and S2_in is an input stream (step $4.d$), since it has many similarities with steps $4.c$ and $4.e$.

EXAMPLE 4.1. *Let us consider the graph G in Figure 2(a) and see which is the output of function* proc_obj*() when it is evaluated over G. The last Aurora operation applied on the stream resulting from G is the* Map *box. Moreover, since the stream over which this box is applied is an internal stream (i.e., the stream resulting from the* Filter *box), the function recursively calls itself with input G', that is, the subgraph of G generating this internal stream, and stores the result into* S_res*. When the function evaluates G', the last Aurora operation is the* Filter *box. This box is applied on an input stream (i.e.,* Health*), thus the function (step* $4.a.iii$*) initializes the protection object like representation to be returned, i.e.,* S_out*, by setting the* STRs *component to* Health*. Then, it sets the* EXPs *component of* S_out *with the predicate specified in the* Filter *box , that is,* Health.Platoon=X ∧ Heart>*160 (cfr. Figure 2(a)). The* ATTs *component is set equal to * (step* $4.a.iii$.2*). Thus,* proc_obj*(G') returns as a result the following protection object like representation: (*{Health}*,**,Health.Platoon=X ∧ Heart>*160*) (step* $4.a.iii$.3*) which is stored into variable* S_res*.*

Then, the computation of proc_obj*(G) continues (step* $4.a.v$*). Thus, since* box *is equal to* Map*,* proc_obj*(G) enters step* $4.a.vi$*, where* STRs*, and* EXPs *components of* S_out *are set to:* S_res.STRs*, i.e.,* Health*;* S_res.EXPs*, i.e.,* Health.Platoon=X ∧ Heart>*160, respectively. Moreover, the* ATTs *component is set equal to the set of attributes specified in the* Map *box (i.e.,* Heart *and* SID*) (step* $4.a.vi$.2*). Thus, the final returned protection object like representation of graph G has the following components:* {Health},{Heart,SID},*(*Health.Platoon=X ∧ Heart>*160).* △

Thus, when a user requires to apply a box to one or more streams, the reference monitor first generates the protection object like representation of these streams. After that, the Objs component of an access request is a set of triples of the form: (STRs,ATTs,EXPs), one for each input or internal stream over which the user requires to apply a box.

[5]Note that the function does not consider the case when box is the Bsort operator, since Bsort does not require to insert anything into ATTs and EXPs components of S_out.

[6]Since the Union operator can be performed only on two streams having the same schema, it does not matter which one is inserted.

4.2 Access control enforcement

The building block of the reference monitor is represented by the algorithm in Figure 3. The algorithm receives as input an access request R modelled as a triple (u,Objs,p), where the Objs component is generated by function proc_obj() (cfr. Section 4.1) by taking in input the graph representation G of the stream(s) on which the user requires to apply box p. Therefore, in case R is a request for a Map, Filter, Bsort or Aggregate box, Objs consists of a single triple, whereas if the request is for Join, Resample or Union, Objs consists of two tuples denoting the protection object like representation of the operand streams.[7]

Then, the algorithm exploits the secure operators introduced in Section 3 to generate a set of Aurora expressions, if any, whose evaluation over G generates the authorized views resulting from the requested operations, that is, the stream resulting from the application of the requested box and from which unauthorized tuples and/or attributes have been pruned. The algorithm returns access denied, if the user does not have the right to perform the requested operation.

Let us see in more details how Algorithm 1 works. The algorithm considers separately the case when R.Objs consists of one or two tuples. In the first case R.p is one among Map, Filter, Bsort or Aggregate. R.u is authorized to perform Map, Filter, or Bsort operations only if he/she has the read privilege over the stream denoted by R.Objs.STRs. In contrast, in case of Aggregate operator the algorithm has to consider two different kinds of access control policies. Indeed, a user is authorized to perform an aggregate operator if he/she is authorized to read the stream denoted by R.Objs.STRs or there exists one or more access control policies granting R.u the aggregate privilege specified in R.p on the stream denoted by R.Objs.STRs. In general, no matter which operator R.u requires, if the stream in R.Objs is an input stream, as a first step the algorithm evaluates the Secure read operator on it, by storing the returned Aurora expressions into Sec_Objs (step $2.a.i$). Note that, the Secure read operator is called only for input streams. If the stream(s) on which the request is performed is an internal one it is not necessary to evaluate on it the Secure read operator because each internal stream is generated by a graph defined starting from one or more input streams. Since the algorithm is called each time the user requires to insert a box in the graph, the non authorized portions of the input streams have already been pruned by the Secure read operator called the first time the stream is inserted into the graph. Thus, if the stream in R.Objs is an internal stream, the algorithm simply stores the Aurora expression corresponding to R.Objs directly into Sec_Objs (step $2.a.ii$). To this purpose, here and in the following we make use of a function *Cartesian*(), which takes in input a set of streams and returns their Cartesian product. Then, if R.p is different from the Aggregate operator, the algorithm simply generates the authorized views by applying over each Aurora expression in Sec_Objs the box corresponding to R.p (step $2.b.i$). In contrast, if R.p is the Aggregate operator, the algorithm generates a first set of authorized views, called *Auth_viewAGG*, by applying over each Aurora expression in Sec_Objs the Aggregate operator (step $2.b.vi.1$). Then, the algorithm applies the Sec_Aggr operator and merges the returned Aurora expressions with the expressions in *Auth_viewAGG*, obtaining the authorized views (step $2.b.vii$).

[7] Note that the limitation to two input streams is compliant to Aurora algebra where Join and Resample are defined as binary operators. The only operator that is not binary is Union, however it can be executed over more than two operand streams, by iteratively applying the binary union operator (i.e., Union(S_1,S_2,S_3) = Union(S_1, Union(S_2,S_3))).

ALGORITHM 1. **The Access control algorithm**

INPUT: *An access request R=(u,Objs,p), where u is the requesting user, p is the requested operation and Objs is the protection object like representation of the streams over which u requires p.*
OUTPUT: *Access denied or set of Aurora expressions generating the authorized streams.*

1 *Let Auth_views and Auth_viewsAGG be initialized to be empty*
2 *If (*|R.Objs| = *1)*
  *a If (*R.Objs.STRs *is an input stream)*
    i Sec_Objs:=Sec_Read(R.Objs.STRs,R.u)
  Else
    ii Sec_Objs:=Map(R.Objs.ATTs)*(Filter(*R.Objs.EXPs*)*
      (Cartesian(R.Objs.STRs)))
  Endif
  *b If (*R.p ≠ aggregate*)*
    i Foreach so∈ Sec_Objs *do*
      1 *If (*R.p = Map*)*
        a Let $atts$ *be the set of attributes specified in* R.p
        b Auth_views:=Auth_views∪Map($atts$)(so)
      2 *If (*R.p = Filter*)*
        a Let exp *be the predicate specified in* R.p
        b Auth_views:=Auth_views∪Filter(exp)(so)
      3 *If (*R.p = Bsort*)*
        a Auth_views:=Auth_views∪Bsort(so)
    EndFor
  Else
    ii Let s be the size specified in R.p
    iii Let i be the step specified in R.p
    iv Let F be the aggregate function specified in R.p
    v Let A be the attribute specified in R.p
    *vi If (*R.Objs.STRs *is an input stream)*
      1 *Foreach* so∈ Sec_Objs *do*
        a Auth_viewsAGG:=Auth_viewsAGG∪Aggregate((F, A),s,i)(so)
      EndFor
    EndIf
    vii Auth_views:=Auth_viewsAGG ∪Sec_Aggr(R.Objs.STRs,(F, A),s,i,R.u)
  EndIf
Else
  c Let Obj_1 *and* Obj_2 *be the elements in* R.Objs
  d If ($Obj_1.STRs$ *is an input stream)*
    i Sec_Objs$_1$:=Sec_Read($Obj_1.STRs$,R.u)
  Else
    ii Sec_Objs$_1$:=Map(Obj_1.ATTs)*(Filter(*Obj_1.EXPs*)*
      (Cartesian(Obj_1.STRs)))
  EndIf
  e If ($Obj_2.STRs$ *is an input stream)*
    i Sec_Objs$_2$:=Sec_Read($Obj_2.STRs$,R.u)
  Else
    ii Sec_Objs$_2$:=Map(Obj_2.ATTs)*(Filter(*Obj_2.EXPs*)*
      (Cartesian(Obj_2.STRs)))
  EndIf
  f Foreach so$_1$∈Sec_Objs$_1$ *do*
    i Foreach so$_2$∈Sec_Objs$_2$ *do*
      1 *If (*R.p = Union*)*
        a Auth_views:=Auth_views ∪Union(so_1,so_2)
      2 *If (*R.p= Join*)*
        a Let P *be the join predicate specified in* R.p
        b Auth_views:=Auth_views∪Join(P)(so_1,so_2)
      3 *If (*R.p= Resample*)*
        a Let P *be the join predicate specified in* R.p
        b Auth_views:=Auth_views∪Resample(P)(so_1,so_2)
    EndFor
  EndFor
  g Objs$_1$:=Map(Obj_1.ATTs)*(Filter(*Obj_1.EXPs*)*(Cartesian(*Obj_1.STRs*)))*
  h Objs$_2$:=Map(Obj_2.ATTs)*(Filter(*Obj_2.EXPs*)*(Cartesian(*Obj_2.STRs*)))*
  *i If (*R.p= Join*)*
    a Auth_views:=Auth_views∪Sec_Join(*Objs*$_1$, *Objs*$_2$,P,R.u)
  *l If (*R.p= Resample*)*
    a Auth_views := Auth_views ∪ Sec_Resample(*Objs*$_1$, *Objs*$_2$,P,R.u)
EndIf
If (Auth_views = ∅*)*
  Return *Access denied*
Else
  Return *Auth_views*

Figure 3: An algorithm for access control enforcement

Let us consider now the case that R.u requires to apply R.p over two streams, each one modeled through a protection object like representation say Obj_1 and Obj_2. In this case R.p is one among Union, Join and Resample. A user is authorized to apply one of such operator if he/she has the read privilege on both the streams denoted by the input protection object like representation. In this case, the user is allowed to perform R.p only over the authorized views of the streams, i.e., the views returned by Sec_Read. Additionally, if R.p is Join or Resample, the algorithm has to further verify whether there exists access control policies defined for the join itself. If this is the case, in addition to the join computed over the results of Sec_Read, the authorized view consists also of the views returned by Sec_Join evaluated over the requested streams. Therefore, the algorithm firstly applies the Secure read operator on Obj_1 (i.e., Obj_2), if Obj_1 (i.e., Obj_2) is an input stream. The returned Aurora expressions are stored into $\texttt{Sec\_Objs}_1$ and $\texttt{Sec\_Objs}_2$, respectively (steps $2.d$ and $2.e$). Then, the algorithm considers each possible combination of Aurora expressions in $\texttt{Sec\_Objs}_1$ and $\texttt{Sec\_Objs}_2$ and performs over them the required operation R.p (i.e., Union, Join and Resample) (steps, $2.f.i$.1-3). Moreover, if R.p is the Join or the Resample operator, the algorithm checks whether there exists some access control policies granting R.u the join (resample) privilege over the streams denoted by the Aurora expressions in $\texttt{Objs}_1$ and $\texttt{Objs}_2$. To this purpose, the algorithm makes use of the Sec_Join operator (Sec_Resample operator) over the requested stream. Thus, in this case the authorized views are given by the Aurora expressions obtained by applying the Sec_Join operator (Sec_Resample operator) over $\texttt{Objs}_1$ and $\texttt{Objs}_2$, plus the views generated by applying the Join (or Resample operator) over each possible combination of Aurora expressions in $\texttt{Sec\_Objs}_1$ and $\texttt{Sec\_Objs}_2$.

EXAMPLE 4.2. *Let us consider an example of the execution of Algorithm 1. In particular, let us assume that doctor Rick belonging to platoon X is interested to know information about soldiers having the number of heart beats greater than 160. Therefore, Rick tries to apply a* Filter *box over the* Health *stream, with predicate:* Heart>*160. This access request is modeled as follows: (Rick, ({Health},*, null), (Filter,*Heart>*160)). To process this access request, the algorithm first calls the Secure read operator, i.e.,* Sec_Read*(*Health*,Rick), which, according to the access control policies in Table 1 returns the following Aurora expression:* Map*(*Heart,SID*)(*Filter*(*Health.Platoon=X*)(*Health*)). Thus the authorized view given in output by Algorithm 1 is the one resulting from the evaluation of the following Aurora expression:* Filter*(*Heart>*160)(*Map*(*Heart,SID*)(*Filter*(*Health.Platoon=X*)(*Health*))). This Aurora expression is equivalent to the graph in Figure 2(a). Indeed, we assume that before inserting the corresponding box into the graph, the Aurora expressions resulting from Algorithm 1 are optimized, that is,* Filter/Map *boxes are collapsed together, if possible.*

Let us assume now that Rick is further interested to restrict the research only to those soldiers that have crossed a certain border k. Moreover, we assume that currently action a *is taking place, whose targets position is 1000, starting time is 105000, whereas ending time is infinite since the action has not ended. As pointed out in Example 3.3, since the position of a soldier is stored into the* Position *stream, in order to obtain such a restriction, Rick has to perform a join of the* Position *stream with the internal stream resulting by the previous query (i.e., the stream resulting by the graph of Figure 2(a)), by selecting only those tuples which refer to soldiers whose position satisfies the condition above. The access request is therefore transformed with the help of function proc_obj() (cfr. Figure 1) into the following triple: (Rick,{(*Position*,*,null), (*Health,{Heart,SID},Health.Heart>*160*∧Health.Platoon=X)},(Join,*Position.SID=Health.SID∧Pos≥*k)). In this case the algorithm verifies that the first object is in input stream, whereas the second is an internal one. Thus, in step $2.d.i$, the algorithm calls* Sec_Read*(*Position*,Rick). However, since no access control policies in Table 1 authorize the* read *privilege on* Position, $\texttt{Sec\_Objs}_1$ *is set empty. This implies that no iteration of the cycle in step $2.f$ is performed. However, since the requested operator is* Join*, the algorithm calls the Secure join operator. More precisely, the algorithm calls it by passing as input* $\texttt{Objs}_1$*=* Map*(*)(*Position*) and* $\texttt{Objs}_2$*=* Map*(*Heart,SID*) (*Filter*(*Heart>*160*∧Platoon=X*)(*Health*)). Similarly to Example 3.3, the operator returns as authorized view the one corresponding to the following Aurora expression:* Map*(*Heart,SID*)(*Filter*(*Position.SID= Health.SID∧ts≥105000 ∧ts≤ ∞∧Pos≥10000-δ∧Pos≤1000+δ∧Pos≥k*)(*Position,Health*)). This Aurora expression is equivalent to the graph in Figure 2(b).* △

5. FORMAL RESULTS

In this section we formally prove the correctness of Algorithm 1. In order to do that we have to prove that given an access request R=(u,Objs,p), for each tuple t contained in the streams AS generated by the Aurora expressions AE returned by Algorithm 1, there exists a set of access control policies in SysAuth that authorize u to access t.

This formulation can be better stated according to the following observations. The first is related to the minimum number of access control policies that must be checked to verify whether u can or cannot access t. Indeed, according to Algorithm 1, if the required operator is Filter, Bsort, and Map then a tuple t belongs to AS if there exists at least an access control policy granting the read privilege to u. Whereas, if the operator is an aggregate box, t is in AS if there exists at least an access control policy granting the read or the required operator (e.g., avg, sum, min, etc.) privilege to u. In contrast, if the required operator is Join, Resample, and Union, t belongs to AS if there exist at least two access control policies granting the read privilege to u over the operand streams, or in case p is Join or Resample, if there exists an access control policy granting the join or resample privilege over a view of the requested joined stream.

Further, we need to formally state which is the set of tuples over which a user can exercise a privilege according to an access control policy. Therefore, we exploit the semantics of the access control policies presented in [5]. In particular, in [5] we have defined the semantics of the protection object specification of a policy acp as the set of tuples over which a user to which the policy applies can exercise privilege acp.priv. We report the formal definition of the semantics of the protection object specification, which is exploited in formulation of the correctness theorem.

Definition 7. **(Protection object specification semantics)** [5]. Given an access control policy acp, the *protection object specification semantics* of acp is given by the β function defined as follows:

- if |acp.obj.STRs|=1, then β(acp)=Map($A_1,\ldots,A_n$)(Filter(acp.obj.EXPs∧ ts ≥ acp.gtc.begin ∧ ts≤ acp.gtc.end)(acp.obj.STRs)), otherwise
- β(acp)=Map($A_1,\ldots,A_n$)(Filter(acp.obj.EXPs∧ ts≥acp.gtc.begin∧ts≤acp.gtc.end)

(Cartesian($\{S_1, \ldots, S_k\}$))), $S_j \in$acp.obj.STRs $\forall j \in [1, k]$;

where $\{A_1, \ldots, A_n\}$ belongs to acp.obj.ATTs. If acp.obj.ATTs = *, then $\{A_1, \ldots, A_n\}$ are all the attributes of streams belonging to acp.obj.STRs.

We can now formally state the correctness of Algorithm 1.

THEOREM 5.1. *Let* R=(u,Objs,p) *be an access request and let* AE *be the set of Aurora expressions returned by Algorithm 1. Let* AS *be the set of streams resulting from the evaluation of the expressions in* AE*. For each tuple* t∈AS *then:*

- *if* R.p *is in* {Filter, Bsort, Map}*, there exists an access control policy* acp *having* acp.priv=read *such that* t∈ R.p*(*β*(*acp*));*[8]
- *if* R.p *is in* {Min, Max, Count, Avg, Sum}*, there exists an access control policy* acp *having* acp.priv={read∨R.p} *and such that* t∈ Aggregate*(*R.p*, s, i)(*β*(*acp*)), where s, i, are the window and step specified in* R.p*;*
- *if* R.p *is* Union*, there exist two access control policies* acp1 *and* acp2 *having* acp1.priv= acp2.priv=read *and such that* t∈Union*(*β*(*acp1*),*β*(*acp2*));*
- *if* p *is* Join *or* Resample*, one of the following condition holds:*
  - *there exist two access control policies* acp1 *and* acp2 *having* acp1.priv=acp2.priv=read *and such that* t∈ Join*(P)(*β*(*acp1*),* β*(*acp2*)) (i.e.,*Resample*(P)(*β*(*acp1*),* β*(*acp2*))), where P is the predicate specified in* R.p*;*
  - *there exists an access control policy* acp *having* acp.priv= join *and such that* t∈ β*(*acp*);*

The proof is reported in [4].

6. CONCLUSION

Today's data stream management systems do not offer adequate mechanisms to protect against unauthorized access of sensitive streaming data. In this paper, we investigated the problem of enforcing access control under the role-based access control model proposed in [5]. We designed a set of novel secure operators (namely, Secure Read, Secure View, Secure Join and Secure Aggregate). These operators essentially prune away tuples/attributes from results of the corresponding non-secure counterparts that are not accessible according to the specified access control policies. We also developed an access control mechanism to enforce the access control policies based on these operators. We showed that our method obeys the specified access control policies.

We plan to extend the work reported in this paper along several directions. First, we plan to investigate completeness issues related to the proposed enforcement mechanism. Then, we plan to implement and integrate the proposed mechanism into a stream processing engine. This requires to investigate very challenging issues. Indeed, the expressions returned by Algorithm 1 presented in this paper are just the basic building blocks of the access control mechanism we would like to develop. A crucial aspect is how the evaluation of these expressions can be optimized and integrated with the optimization techniques in place in the stream engine. Although we have casted our work in the Aurora framework because up to now is the most widely accepted and mature proposal, another issue to be investigated is how to adapt the proposed mechanism to other stream engine (e.g., the ones proposed in [3, 6]). The use of suitable indexing strategies for policies is also a topic we would like to investigate in the future. Moreover, once implemented we would like to perform an extensive evaluation of the performance of the proposed mechanism to evaluate the overhead implied by access control checks. Finally, we will extend the model (and hence the enforcement strategies) to deal with updates.

[8]For sake of simplicity, we use this simplified syntax R.p() instead of specifying the Aurora expression for each single cases, i.e., Filter, Bsort, and Map.

7. REFERENCES

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

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

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In Proceedings of *ACM SIGMOD'03*, page 665, San Diego, USA, 2003.

[4] B. Carminati, E. Ferrari, and K.L. Tan. Enforcing access control policies on data streams. Extended version of this paper. Available at:"http://www.dicom.uninsubria.it/~elena.ferrari/stream/TR0107.pd", 2006.

[5] B. Carminati, E. Ferrari, and K.L. Tan. Specifying access control policies on data streams. In Proceedings of the *Database System for Advanced Applications Conference (DASFAA 2007)*, Bangkok, Thailand, 2007.

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

[7] E. Ferrari and B. Thuraisingham. Secure Database Systems. In O. Diaz and M. Piattini editors, *Advanced Databases: Technology and Design*, Artech House, London, 2000.

[8] W. Lindner and Jorg Meier. Securing the borealis data stream engine. In Proceeding of the *International Database Engineering and Application Symposium (IDEAS'06)*, Dehli, India, 2006.

[9] StreamBase Home Page. http://www.streambase.com//.