A Framework to Enforce Access Control over Data Streams

BARBARA CARMINATI and ELENA FERRARI
DICOM, University of Insubria
and
JIANNENG CAO and KIAN LEE TAN
National University of Singapore

Although access control is currently 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 article, 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 the development of a general solution to access control enforcement more difficult. The framework we propose in this article 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. Furthermore, 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 article, 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

ACM Reference Format:
Carminati, B., Ferrari, E., Cao, J., and Tan, K. L. 2010. A framework to enforce access control over data streams. ACM Trans. Info. Syst. Sec. 13, 3, Article 28 (July 2010), 31 pages.
DOI = 10.1145/1805974.1805984 http://doi.acm.org/10.1145/1805974.1805984

J. Cao and K. L. Tan are partially supported from a research grant R-252-000-307-112 from NUS.
Authors' addresses: B. Carminati and E. Ferrari, DICOM, University of Insubria, Varese, Italy; email: {barbara.carminati, elena.ferrari}@uninsubria.it; J. Cao and K. L. Tan, National University of Singapore, Singapore; email: {caojiann, tankl}@comp.nus.edu.sg.

DOI 10.1145/1805974.1805984 http://doi.acm.org/10.1145/1805974.1805984

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, which allow access to data during a given time bound, and window constraints, which 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 goals 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 article, 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, 2003], Arasu et al. [2003], Cranor et al. [2003], and 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: preprocessing, postprocessing, and query rewriting. Preprocessing is a naïve way to enforce access control according to which 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, unlike ours, 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 preprocessing 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 a certain action and/or of those soldiers positioned in a given region. In contrast, postprocessing first executes the original user query, then it prunes from the result the unauthorized tuples before delivering the resulting stream to the user. Like preprocessing, 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 6.2, it may waste computation, since queries are evaluated even if they are denied by the specified access control policies.

For the previously mentioned reasons, we adopt the query rewriting approach. Thus, unlike conventional RDBMSs, our access control mechanism operates at query definition time, and hence avoids runtime 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 outputs 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 article, 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 [2008], StreamBase [2008], and Truviso [2008]. Unfortunately, at the time of this writing, it has not been possible to retrieve helpful documentation about Truviso and its 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 article 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 article substantially extends the work presented by us in Carminati et al. [2007a, 2007b]. Carminati et al. [2007a, 2007b] only presents the access control model underlying our framework, whereas Carminati et al. [2007a] presents an access control enforcement mechanism integrated into the Aurora data stream prototype [Abadi et al. 2003]. In this article, we build on what has been presented in Carminati et al. [2007a, 2007b], 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, a substantial redesign 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 article 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 article is organized as follows. Section 2 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 discusses the prototype we have developed and presents some experiments we have carried out to demonstrate the feasibility of our approach. Section 7 overviews the related work, and Section 8 concludes the article.

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, a standard query language for DSMSs has not yet emerged. In contrast, each DSMS adopts its own language, resulting in 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. 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.

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

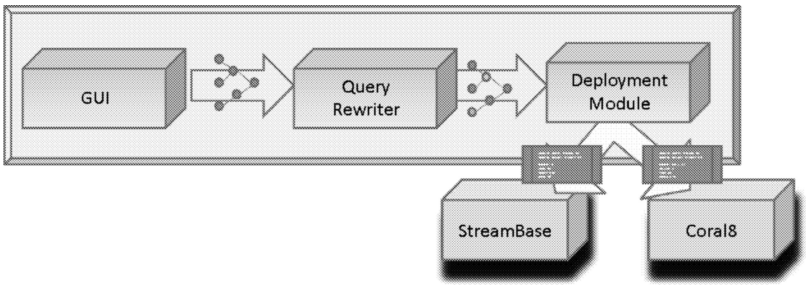


Fig. 1. Architecture of our framework.

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. Due to lack of space, we do not describe the Deployment Module in this article. However, we refer interested readers to Carminati et al. [2008] for a detailed discussion.

In the following text before going into the details of the Query Rewriter, 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, such as 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 $\mathtt{A}_1, \ldots, \mathtt{A}_n$, 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 `Att(S)` the set of attributes in `S`'s schema, and with $\mathtt{S.A}_j$, attribute $\mathtt{A}_j$ 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, that is, 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, that is, the IN and OUT operators.

In the following text, 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 (e.g., 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: π 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 $\{A_1,\ldots,A_n\} \in$ 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 π 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 π 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 $\{A_1,\ldots,A_n\} \in$ Att(S) is $\pi(A_1,\ldots,A_n)$(S).

Selection: σ operator. This operator selects specific tuples within a stream. More formally, given a stream S and a predicate P over attributes in 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 σ 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 σ parameter EXPs, and it is expressed through an SQL-like syntax. The corresponding expression is σ(P)(S).

Example 3.1. Throughout this article, 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,SID,Platoon,Pos), Health (ts,SID,Platoon,Heart,BPressure),

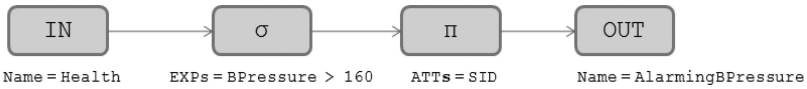


Fig. 2. An example of query graph.

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. Figure 2[1] 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, which 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 $A \in Att(S)$, 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, therefore, has 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, that is, min, max, count, avg, and sum. The algebraic expression corresponding to the Σ operator is $\Sigma(F, A, s, o)(S)$.

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, that is, S_1 and S_2, and one existing edge, that is, S'. The join predicate is contained into parameter EXPs. The algebraic expression corresponding to the Join operator is $Join(P, s_1, o_1, s_2, o_2)(S_1, S_2)$.

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

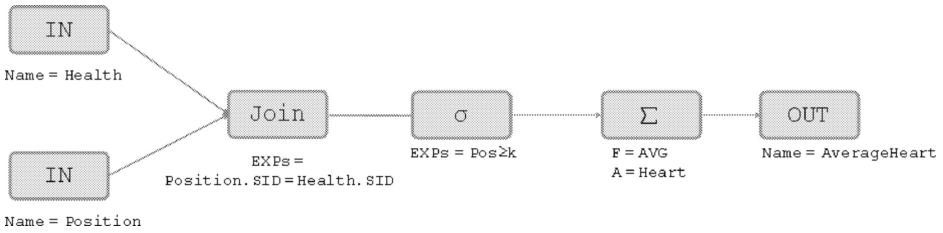


Fig. 3. An example of query graph.

Table I. Core Query Model Operators

Operator	Algebraic expression	Semantics
IN	-	Stream entering the query graph
OUT	-	Stream resulting from the query graph
π	$\pi(A_1,\ldots,A_n)(S)$	Projection of stream S over attributes $\{A_1,\ldots,A_n\} \in Att(S)$
σ	$\sigma(P)(S)$	Selection of stream S with respect to predicate P
Σ	$\Sigma(F, A, s, o)(S)$	Aggregation of attribute A of stream S according to function F over sliding windows generated with size s and offset o
Join	$Join(P,s_1,o_1,s_2,o_2)(S_1,S_2)$	Join with respect to predicate P over tuples of sliding windows of stream S_1 (i.e., S_2) generated with size s_1 (i.e., s_2) and offset o_1 (i.e., o_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 $Pos \geq k$), 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 heartbeat average requires to perform a join of the Position and Health streams, with predicate $Position.SID = Health.SID$, and then to select only those tuples with $Pos \geq k$. 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 $Pos \geq k$. Over the result of the selection, an aggregate operator Σ is evaluated.[2] 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

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

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 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: `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 the values of stream attributes. Additionally, the model allows the security administrator (SA) to restrict the exercise of the `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 in 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 in the following text.

Definition 4.1 (*Protection Object Specification [Carminati et al. 2007b]*). A *protection object specification* `p_obj` is a triple `(STRs,ATTs,EXPs)`, where:

—`STRs` is a set of names or identifiers of streams $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$;
—`ATTs` denotes a set of attributes $\mathtt{A}_1, \ldots, \mathtt{A}_l$, where $\mathtt{A}_j$, $j \in \{1, \ldots, l\}$, belongs to the schema of the stream resulting from the Cartesian product ($\mathtt{S}_1 \times \cdots \times \mathtt{S}_n$) 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 ($\mathtt{S}_1 \times \cdots \times \mathtt{S}_n$).
—`EXPs` is a boolean formula, built over predicates of the form: $\mathtt{A}_i \oplus \mathtt{value}_i$ or $\mathtt{A}_i \oplus \mathtt{A}_j$, where $\mathtt{A}_i$, $\mathtt{A}_j$ are attributes belonging to the schema of the Cartesian product ($\mathtt{S}_1 \times \cdots \times \mathtt{S}_n$), $\oplus$ is a comparison operator, and $\mathtt{value}_i$ is a value compatible with the domain of $\mathtt{A}_i$. If `EXPs` is omitted, it denotes all the tuples in ($\mathtt{S}_1 \times \cdots \times \mathtt{S}_n$).

The access control model also allows the SA to specify two different types of temporal constraints, that is, general and window-based constraints. General constraints 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 $\leq$ end, and end can assume the infinite value.[3] The begin and end values can be explicitly specified by the SA, or they can be returned by a predefined set of system functions $\mathcal{SF}$. 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 attribute, that is, the timestamp ts, a general time constraint *gtc* identifies all and only those tuples satisfying the predicate: ts $\geq$ begin $\wedge$ ts $\leq$ end. Window-based constraints are 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 constrain 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 in the following text.

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$\in$\{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).[4] The second access control policy authorizes doctors to compute the average of the positions of those soldiers not belonging

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

[4]We assume that each user has an associated profile, that is, a set of attributes modeling his/her

Table II. Examples of Access Control Policies for Data Streams

sbj	Protection Object			priv	gtc	wtc	
	Streams	Attributes	Expressions			**s**	**o**
Doctor	Position	Pos, SID	Position.Platoon=self.Platoon	read	-	-	-
Doctor	Position	Pos	Position.Platoon≠self.Platoon	avg	[start(a), end(a)]	1	1
Doctor	Health	*	Health.Platoon=self.Platoon	read	-	-	-
Doctor	Health, Position	BPressure, SID,Pos	Health.Platoon≠self.Platoon ∧ Position.SID=Health.SID ∧ Pos ≥ target(a)- δ ∧ Pos ≤ target(a)+ δ	read		-	-

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 position is at most δ distant from the target position of action a (Pos $\geq$ target(a) - $\delta \wedge$ Pos $\leq$ target(a) + δ).[5]

Finally, in the remainder of the article, we need to formally denote the tuples identified by the protection object specification of an access control policy acp. These are defined by function $\beta()$, 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) = $\pi(A_1, \ldots, A_n)(\sigma$(acp.obj.EXPs $\wedge$ ts $\geq$ acp.gtc.begin $\wedge$ ts $\leq$ acp.gtc.end)(acp.obj.STRs)), otherwise

—β(acp) = $\pi(A_1, \ldots, A_n)(\sigma$(acp.obj.EXPs $\wedge$ ts $\geq$ acp.gtc.begin $\wedge$ ts $\leq$ acp.gtc.end)(Cartesian($S_1, \ldots, S_n$))), $S_j \in$ acp.obj.STRs, $\forall j \in [1, n]$;

where Cartesian() takes as input a set of streams and returns their Cartesian product; whereas $\{A_1, \ldots, A_n\}$ = acp.obj.ATTs. If acp.obj.ATTs = $*$, then $\{A_1, \ldots, A_n\}$ 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 $\wedge$ Pos $\geq$ target(a)-δ $\wedge$ Pos $\leq$ target(a) + δ)(Cartesian(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

characteristics, such as the platoon one belongs to.

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

referring to soldiers whose position is close to the target position are considered (i.e., $\text{Pos} \geq \text{target(a)} - \delta \wedge \text{Pos} \leq \text{target(a)} + \delta$).

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 unauthorized 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 all unauthorized 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 retrieving policies 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 predefined views. However, this makes the retrieval of access control policies applicable 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 policy retrieval, we assume that all streams in a query graph (i.e., 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 from 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 π and Σ operators of the graph G are projected, and all predicates specified in the σ 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.

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, Sec_View of S with respect to policy acp is defined as follows:

$$\texttt{Sec\_View}(\texttt{S},\texttt{acp}) = \pi(\texttt{att})(\sigma(\texttt{P})(\texttt{S}))$$

where:

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

$$\texttt{P} = \texttt{exp} \wedge \texttt{window}$$

where:

$$\texttt{exp}^{6} = \begin{cases} (\texttt{acp.obj.EXPs}) & \text{if acp.obj.EXPs is not omitted} \\ \text{true} & \text{otherwise} \end{cases}$$

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

Based on the Sec_View operator, we next define the 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 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 algebraic expression. Given a user u, in the following, we denote with *Role*(u) the set of roles u is authorized to play.

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

$$\texttt{Sec\_Read}(\texttt{S},\texttt{u}) = \bigcup_{\texttt{acp}_\texttt{j} \in Pol(\texttt{S},\texttt{u})} \{\texttt{Sec\_View}(\texttt{S}, \texttt{acp}_\texttt{j})\}$$

Example 5.1. Let us consider the access control policies in Table II, and suppose that there exists a user, say Paul, belonging to platoon X and authorized to play the doctor role. Let us see the view resulting from the evaluation of Sec_Read(Position,Paul), denoted in what follows as *AuthView*. According to Definition 5.2, *AuthView* is defined as the union of the views

[6]True denotes a predicate that is always satisfied.

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 $\texttt{acp}_1$. Thus, *AuthView* consists of Sec_View(Position,$\texttt{acp}_1$). According to Definition 5.1, Sec_View(Position,$\texttt{acp}_1$) returns the following expression: π(Pos,SID)(σ(Position.Platoon=X)(Position)). Thus, the view of the Position stream on which Paul has the 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, 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 $Pol_{agg}(\texttt{S},\texttt{u})$ be the set of access control policies in SysAuth to be considered to evaluate u request to perform F over attribute A. More formally, $Pol_{agg}(\texttt{S},\texttt{u}) = \{\texttt{acp} \in \texttt{SysAuth} | \texttt{acp.obj.STRs} = \texttt{S.STRs}, \texttt{A} \in \texttt{acp.obj.ATTs}, \texttt{acp.sbj} \cap \mathit{Role}(\texttt{u}) \neq \emptyset, \texttt{acp.priv} = \texttt{F}, \forall \mathit{exp} \in \texttt{S.EXPs}, \exists \mathit{exp}' \in \texttt{acp.obj.EXPs}$, such that $\mathit{exp} \subseteq \mathit{exp}'$[7]$\}$. The secure aggregate operator, Sec_Aggr is defined as follows.

$$\texttt{Sec\_Aggr(S,F,A,s,o,u)} = \bigcup_{\texttt{acp}_j \in Pol_{agg}(\texttt{S},\texttt{u})} \{\Sigma(\texttt{F},\texttt{A},\texttt{max}_{size},\texttt{max}_{offset})(\pi(\texttt{A})(\sigma(\texttt{P})(\texttt{S})))\}$$

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

$$\texttt{P} = \texttt{exp} \wedge \texttt{window}$$

where:

$$\texttt{exp} = \begin{cases} (\texttt{acp}_j\texttt{.obj.EXPS}) & \text{if } \texttt{acp}_j\texttt{.obj.EXPs is not omitted} \\ \texttt{true} & \text{otherwise} \end{cases}$$

$$\texttt{window} = \begin{cases} (\texttt{ts} \geq \texttt{acp}_j\texttt{.gtc.begin} \wedge \texttt{ts} \leq \texttt{acp}_j\texttt{.gtc.end}) & \text{if } \texttt{acp}_j\texttt{.gtc is not null} \\ \texttt{true} & \text{otherwise} \end{cases}$$

Let us explain how policies are selected by function $Pol_{agg}()$. According to the proposed access control model, if there exists an access control policy acp granting user u the aggregate privilege F over the protection object acp.obj, this implies that F can be computed over all and only the tuples denoted by

[7]The $\subseteq$ operator verifies whether the expressions exp and exp' generate two streams S' and S, respectively, such that S is included in S'.

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 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 acp such that the stream S includes the stream denoted by the protection object specification of acp, that is, such that the tuples in β(acp) (see 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 acp.obj. Since S could be an internal stream generated by a graph G, verifying whether S includes the tuples in β(acp) requires to check a set of conditions. A first condition is that the names of the streams over which acp is specified (i.e., acp.obj.STRs) are equal to the names of the streams over which S is generated (i.e., S.STRs). Furthermore, in order to ensure that stream S includes the tuples in β(acp), it is required that, for each expression in *exp* S.EXPs, there exists a corresponding expression exp' in the EXPs component of the protection object specification of acp, such that tuples satisfying exp' are a subset of tuples satisfying *exp*. As a final condition, the attribute A over which the aggregate function has to be evaluated must be included among the attributes authorized by acp (i.e., acp.obj.ATTs).

Example 5.2. Suppose that Paul wishes to calculate the average of soldiers' position with windows of 5 hours and 5 as offset. Moreover, let us assume that action a, with starting time 105000, is currently taking place. Let us consider the view returned by the secure aggregate operator. In this case, Pol_{agg}(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, Sec_Aggr(Position,avg,Pos,Paul,5,5) is equal to Σ(avg,Pos,5,5) (π(Pos) (σ(Position.Platoon $\neq$ X $\wedge$ ts $\geq$ 105000 $\wedge$ ts $\leq$ ∞)(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 $\neq$ X, that is, tuples of soldiers not belonging to Paul's platoon, and such that: ts $\geq$ 105000 $\wedge$ ts $\leq$ ∞, 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 joining 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 Sec_Aggr, 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 S_1 and S_2 be two streams, P a join predicate over S_1 and S_2, and u be a user. Let $Pol_{join}(S_1,S_2,u)$ be the set of read access control policies in SysAuth applying to u specified for joins over S_1 and S_2. Let Att be the set of attributes over which predicates in S_1.EXPs and S_2.EXPs are defined. More formally, $Pol_{join}(S_1,S_2,u) = \{acp \in SysAuth | acp.obj.STRs = S_1.STRs \cup S_2.STRs, P \in acp.obj.EXPs, Att \in acp.obj.ATTs, acp.sbj \cap Role(u) \neq \emptyset, acp.priv = read\}$. The secure join operator, Sec_Join, is defined as follows.

$$Sec\_Join(S_1,S_2,P,u) = \bigcup_{acp_j \in Pol_{join}(S_1,S_2,u)} \{Sec\_View(Join(P)(S_1,S_2), acp_j)\}.$$

Similarly to the secure aggregate operator, $Pol_{join}(S_1,S_2,u)$ 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 S_1 and S_2 are generated. Moreover, it verifies whether the attributes over which predicates of streams S_1 and S_2 are defined are contained into the authorized attributes (i.e., acp.obj.ATTs). This check avoids possible inferences. Indeed, without this condition, a malicious user could insert predicates over unauthorized attributes (by means of σ operator) just before the join operator. In this way, even if the secure join operator returns only the authorized attributes, since the predicates over unauthorized 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 Pos $\geq k$). Since the position of a soldier is stored in the Position stream, whereas health information is in the Health stream, the first operation he needs to perform is the join of the Position and Health streams, with predicate Position.SID = Health.SID. Let us now see how the secure join operator evaluates over the requested query, that is, Sec_Join(Position,Health,Position.SID = Health.SID,Paul). According to Definition 5.4, the secure join operator first generates the joined stream, say S, resulting from Join(Position.SID = Health.SID)(Position, Health). Then, for each policy acp in Pol_{join}(Position, Health, Paul), it evaluates the secure view operator over S and acp. By Definition 5.4, Pol_{join}(Position, Health, Paul) contains only the fourth policy in Table II, say 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, target(a) returns null. Sec_Join returns the view generated by Sec_View(S,acp_4). By Definition 5.1, this view is given by the following expression: π(BPressure,SID,Pos)(σ(Position.SID=Health.SID $\wedge$ Pos $\geq$ null - $\delta \wedge$ Pos $\leq$ null + δ)(Position, Health)). Since the last predicate in the above σ operator evaluates to null, no tuples are selected, thus the authorized view is empty.

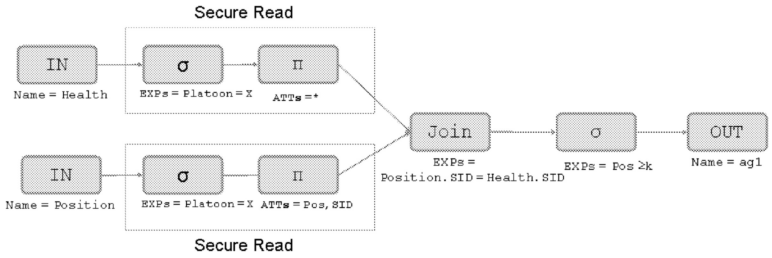


Fig. 4. Authorized graph ag_1.

5.2 Secure Query Rewriting

We recall that given a query graph G and a user u, the Query Rewriter rewrites G into a set of authorized graphs AG, on the basis of the access control policies applicable 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 $\mathtt{ag} \in \mathtt{AG}$ generates only tuples that: (a) answer the original query graph G and (b) there exists an access control policy authorizing $\mathtt{u}$ 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 G by simply preprocessing all entering tuples in order to filter out unauthorized tuples. Thus, the graph processes only authorized tuples and, as a consequence, it generates only authorized tuples. This preprocessing 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 S and a user u and returns only those tuples of S to which u has the read access. Thus, by simply inserting the secure read operator just after each `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 preprocessing 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 preprocessing 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 $\mathtt{Pos} \geq k$). Thus, he needs to perform the joining of `Health` and `Position` streams with predicate $\mathtt{Health.SID} = \mathtt{Position.SID}$, and then to select from the resulting stream those tuples with $\mathtt{Pos} \geq k$. To perform this query, he submits a query graph similar to the one in Figure 3, without the Σ operator. According to the preprocessing strategy, the authorized graph ag_1 (see Figure 4) is obtained by inserting the secure read operators just after the `IN` operators

in G (i.e., Sec_Read(Health,Paul) and 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, Sec_Read(Position,Paul) and 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 unauthorized 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 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 $\mathtt{ag}_1$.

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 $\mathtt{S}_1$, $\mathtt{S}_2$ if: (i) u has the read privilege over both the streams, or (ii) 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 (i) can be easily handled by preprocessing. In contrast, case (ii) can not be enforced by preprocessing, since the secure read operator could filter out from $\mathtt{S}_1$ and/or $\mathtt{S}_2$ 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: (i) he/she has a read privilege over the stream, or (ii) there exists an access control policy acp granting the aggregate privilege such that tuples in $\beta(\mathtt{acp})$ are included into the target stream. Case (i) is easily handled by preprocessing approach, that is, by inserting the secure read operator just after the IN operator of the target stream. However, similarly to what happens for the join operator, preprocessing does not work for case (ii). 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 preprocessing 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 $\mathtt{ag}_2$ by exploiting the secure join operator (see Figure 5). This is obtained by inserting into G Sec_Join(Health,Position,Health.SID = Position.SID,Paul) just

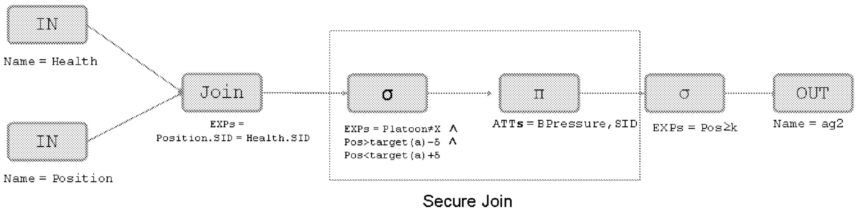


Fig. 5. Authorized graph ag_2.

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, the authorized graphs are obtained by recursively traversing G from the end of the graph, that is, the `OUT` operator, till the input streams, that is, the `IN` operators. Each time the algorithm encounters an operator `OP`$\neq$`IN`, it recursively calls itself by passing as input the subgraph $\overline{\mathtt{G}}$ generating the stream entering into `OP` and by performing over the authorized graphs returned from the recursion, denoted as `Ret_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 through 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 *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 *Insert()* function also inserts the `OUT` operator at the end of the resulting graph.

We now illustrate 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 $\overline{\mathtt{G}}$ generating the stream entering `OP`. Then, $\overline{\mathtt{G}}$ is recursively evaluated and, as result, the algorithm returns a set of authorized graphs `Ret_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

Algorithm 1: *Sec_Rewr*(G,u)

Let OP be the operator directly connected to the OUT operator;
Initialize AG, AG_j, AG_{sj}, AG_a, AG_{sa} to the empty set;
case OP = IN
    $AuthViews$ = Sec_Read(IN.Name,u);
    for *each* $av \in AuthViews$ **do**
        Let ag be a copy of G;
        $ag = Insert(ag, av)$;
        AG = AG∪ag;
    obj.STRs = IN.name, obj.ATTs= IN.ATTs;
    Return ⟨AG,obj⟩;
case OP = π *or* OP = σ
    Let $\overline{G}$ be a copy of G;
    Delete from $\overline{G}$ the OUT operator, and replace OP with OUT;
    ⟨Ret_AG,obj⟩ = *Sec_Rewr*($\overline{G}$,u);
    for *each* $ag \in$ Ret_AG **do**
        Replace OUT in ag with OP and add OUT at the end of ag;
        AG = AG∪ag;
    if OP = π **then**
        obj.ATTs = obj.ATTs∩ OP.ATTs;
    if OP = σ **then**
        obj.EXPs = obj.EXPs∪ OP.EXPs;
    Return ⟨AG,obj⟩;
case OP = join
    Let $\overline{G}_1$ and $\overline{G}_2$ be two copies of G;
    Delete OUT from $\overline{G}_1$ and $\overline{G}_2$;
    Delete OP and the whole subgraph generating the first operand of OP from $\overline{G}_1$, add OUT at the end of $\overline{G}_1$;
    Delete OP and the whole subgraph generating the second operand of OP from $\overline{G}_2$, add OUT at the end of $\overline{G}_2$;
    ⟨Ret_AG_1,obj_1⟩ = *Sec_Rewr*($\overline{G}_1$,u);
    ⟨Ret_AG_2,obj_2⟩ = *Sec_Rewr*($\overline{G}_2$,u);
    for *each* $ag_1 \in Ret\_AG_1$ **do**
        for *each* $ag_2 \in Ret\_AG_2$ **do**
            Delete OUT from ag_1 and ag_2;
            Create a new graph ag_j consisting of OP applied over ag_1 and ag_2, add OUT at the end of ag_j;
            $AG_j = AG_j \cup ag_j$;
    $AuthViews$ = Sec_Join(obj_1,obj_2,OP.EXPs,u);
    for *each* $av \in AuthViews$ **do**
        Let ag_{sj} be a copy of G;
        $ag_{sj} = Insert(ag_{sj}, av)$, $AG_{sj} = AG_{sj} \cup ag_{sj}$;
    AG= $AG_j \cup AG_{sj}$;
    obj.STRs = obj_1.STRs∪ obj_2.STRs; obj..ATTs = obj_1.ATTs∪ obj_2ATTs; obj.EXPs = obj_1.EXPs∪ obj_2.EXPs∪ OP.EXPs;
    Return ⟨AG,obj⟩;
case OP = Σ
    Let $\overline{G}$ be a copy of G;
    Delete OUT from $\overline{G}$ and replace OP with OUT in $\overline{G}$;
    ⟨Ret_AG,obj⟩ = *Sec_Rewr*($\overline{G}$,u);
    for *each* $ag \in$ Ret_AG **do**
        Replace OUT with OP in ag, Add OUT to ag;
        $AG_a = AG_a \cup ag$;
    $AuthViews$ = Sec_Aggr(S,OP.F,OP.A,OP.s,OP.o,u);
    for *each* $av \in AuthViews$ **do**
        Let ag be a copy of $\overline{G}$;
        Delete OP from ag, $ag = Insert(ag, av)$;
        $AG_{sa} = AG_{sa} \cup ag$;
    AG = $AG_a \cup AG_{sa}$, obj.ATTs = obj.ATTs∩ OP.A;
    Return ⟨AG,obj⟩;

the streams generated by graphs in Ret_AG. This implies that it has to insert the OP operator just at the end of the graphs in Ret_AG (lines 15 through 17).

In case OP = Join (lines 23 through 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, $\texttt{AG}_j$ and $\texttt{AG}_{sj}$, respectively. In particular, the steps performed to create graphs in $\texttt{AG}_j$ are similar to those of case $\texttt{OP} = \pi$ and $\texttt{OP} = \sigma$. Therefore, the algorithm recursively calls itself twice by passing the subgraphs $\overline{\texttt{G}_1}$ and $\overline{\texttt{G}_2}$, generating the first and the second stream entering in the Join operator, respectively. The results are collected into variables $\texttt{Ret\_AG}_1$ and $\texttt{Ret\_AG}_2$, respectively. Then, it applies the Join operator to each possible combination of graphs in $\texttt{Ret\_AG}_1$ and $\texttt{Ret\_AG}_2$ (lines 30 through 34). In contrast to manage case (ii), 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 the entire algorithm, we make use of variable 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 STRs component contains the input stream names (lines 9 and 40), whereas the EXPs component is set as the union of all predicates specified in the σ and Join operators (lines 21 and 40). In contrast, the 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 π or Σ operators (lines 19 and 54). Thus, the secure join operator is evaluated over variables $\texttt{obj}_1$ and $\texttt{obj}_2$, which are returned by the inner recursion (see line 35). Then, similarly to the case OP = IN, the expressions returned by the secure join operator are inserted into graph G by means of the *Insert()* function (lines 36 through 38), obtaining $\texttt{AG}_{sj}$.

The case $\texttt{OP} = \Sigma$ is very similar to the case OP = Join (lines 42 through 55). Indeed, also in this case, the algorithm has to consider two different kinds of access control policies, since a user is authorized to perform an aggregate operation if: (i) he/she is authorized to read the tuples over which the aggregation is performed, or (ii) 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, $\texttt{AG}_a$ and $\texttt{AG}_{sa}$, according to a procedure very similar to the one adopted in case (i) and (ii) when OP = Join.

Example 5.6. Suppose Paul submits the query graph in Figure 3. We now show 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, $\texttt{OP} = \Sigma$. Then, it calls recursively itself by passing $\overline{\texttt{G}}$, that is, the graph consisting of

the IN operators, the Join operator, the σ operator, and the OUT operator. As second recursion, Algorithm 1 calls itself again by passing $\overline{G}$ 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 $\overline{G}_1$ and $\overline{G}_2$, where $\overline{G}_1$ ($\overline{G}_2$, respectively) consists only of the IN operator modeling Position (resp. Health) (lines 28 and 29). When the algorithm evaluates $\overline{G_1}$, it performs the steps referring to OP = IN. Therefore, it evaluates Sec_Read(Position,Paul), and inserts into $\overline{G}_1$ 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 $\overline{G}_2$, 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: π(SID, Platoon, Heart, BPressure)(σ(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_j 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 through 34). This is similar to the authorized graph ag_1 obtained in Example 5.4 and represented in Figure 4, without the σ 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 σ operator.

Then, the algorithm evaluates the σ operator, by inserting into each authorized graphs in $\text{Ret\_AG} = \{\text{ag}_1,\text{ag}_2\}$ the σ operator. Finally, as last recursion, the algorithm evaluates the Σ 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 Σ operator over ag_1 and ag_2 (lines 46 through 48). In particular, the first authorized graph in AG_a returns the average of the heartbeats only of those soldiers belonging to platoon X. Whereas the second graph does not return any tuple in that the π 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 be 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.

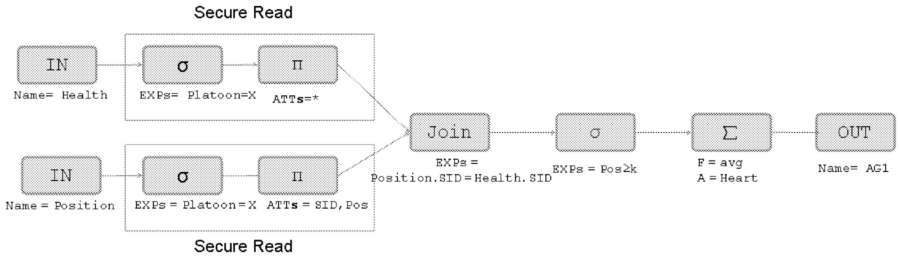


Fig. 6. First authorized graph returned by Algorithm 1.

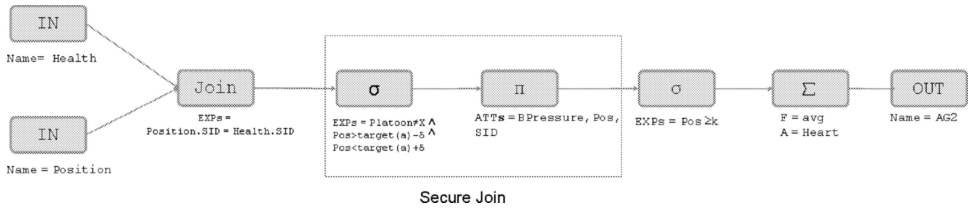


Fig. 7. Second authorized graph returned by Algorithm 1.

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∈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 Carminati et al. [2008].

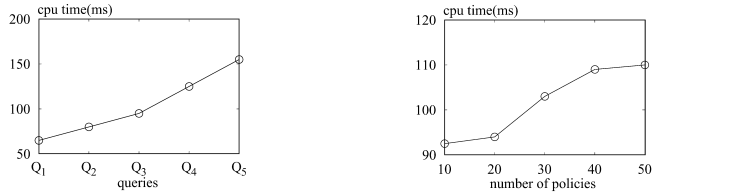
6. PROTOTYPE EVALUATION

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

Table III. Queries Complexity

Query	Streams	Operators
Q_1	2 IN, 1 OUT	5 operators: 2 π, 2 σ, 1 Join
Q_2	3 IN, 1 OUT	10 operators: 4 π, 3 σ, 2 Join, 1 Σ
Q_3	6 IN, 1 OUT	20 operators: 8 π, 7 σ, 4 Join, 1 Σ
Q_4	9 IN, 1 OUT	30 operators: 12 π, 9 σ, 7 Join, 2 Σ
Q_5	18 IN, 1 OUT	60 operators: 24 π, 18 σ, 14 Join, 4 Σ



(a) varying query complexity (b) varying the number of applied policies

Fig. 8. Secure query rewriting overhead.

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 postprocessing approach.

6.1 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 five queries that we used in the experiments—query Q_1 is the less complex with 5 operators, whereas query Q_5 is the most complex involving 60 operators. Figure 8(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 8(b) illustrates the required CPU time as we vary from 10 to 50 the number of policies applied to query Q_3 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 Q_3, the time required is less than 0.12 seconds. From the previously described 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.

6.2 Comparative Analysis

In this section, we compare the performance of our query rewriting algorithm with the postprocessing access control enforcement. In particular, we compare our approach with the postprocessing 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.

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 postprocessing. 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`, heartbeats and blood pressures are uniformly distributed, whereas `normHealth` is generated assuming that heartbeats 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 postprocessing 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.[8] Figure 9(a) reports the result. In Figure 9 the `rewrite-norm` line is the result when source stream `normHealth` and source stream `Position` are tested, whereas the *rewrite-uni* line is the result when `uniHealth` and `Position` streams are tested. The number of join evaluations under the postprocessing 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

[8]The selectivity of an access control policy is given by the percentage of tuples satisfying that policy. For example, selectivity equal to 0.1 means that only 10% of the tuples satisfy the access control policy.

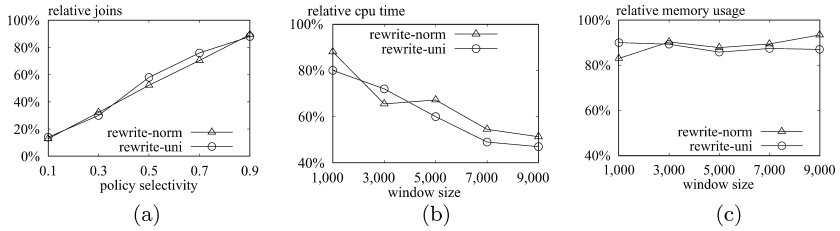


Fig. 9. Comparison with respect to relative joins, relative CPU time, and relative memory usage.

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 postprocessing method.

We also compare secure query rewriting and postprocessing with respect to the required CPU time. We define *relative CPU time* as the ratio between the CPU time required by secure query rewriting and that required by postprocessing. 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 CUP 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 9(b) 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 postprocessing. Finally, we compare secure query rewriting and postprocessing with respect to memory usage. The definition of *relative memory usage* is similar to that of relative CPU time. Figure 9(c) 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 postprocessing.

7. RELATED WORK

Data stream management systems have been the subject of intensive research in the context of different projects, 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 huge amount of work investigating a variety of data stream management issues [Babcock et al. 2002; Golab and Özsu 2003]. Some of them are, for example, related to data models and languages (see, e.g., Law et al. [2004] for a survey), continuous query processing problems, that is, load shedding, join problems, efficient window-based operators (see Babcock et al. [2004] and Bai and Zaniolo [2008]), data stream mining (see Gaber et al. [2005] for a survey), clustering and classification methods for data stream (e.g., 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] and 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 preprocessed before arrival, 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 preprocessing; ordered cash register, which implies that items arrive with a given order, but without any preprocessing; unordered aggregate, where, in no particular order, only one item per source arrives, whose value is computed according to a given preprocessing, that is, by aggregating different items from 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 preprocessing 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 also support operators to handle items order (e.g., the Bsort operator in Aurora). However, since the purpose of the adopted core model is to be suitable to as many 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), and 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 previous discussion shows how the proposed access control framework can be easily deployed into different data stream management systems, since the underlying core model fits the existing data stream models.

Other work related to our proposal are those on streaming data protection. This problem has not yet been 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, that is, 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] consider 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 enables clients 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 [2006] and by Nehme et al. [2008]. Lindner and Meier propose an 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 postprocessing 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 6). 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 the 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 DSMSs.

8. CONCLUSIONS

In this article, we have proposed a framework to enforce access control into different DSMSs. 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 article 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] and Rizvi et al. [2004]).

REFERENCES

ABADI, D., AHMAD, Y., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J., LINDNER, W., MASKEY, A., RASIN, A., ET AL. 2005. The design of the borealis stream processing engine. In *Proceedings of the Conference on Innovative Data System Research (CIDR'05)*. Online Proceedings, 277–289.

ABADI, D., CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK., S. 2003. Aurora: A new model and architecture for data stream management. *VLDB J. 12,* 2, 120–139.

AGGARWAL, C. C., HAN, J., WANG, J., AND YU, P. S. 2003. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*. Morgan Kaufmann, San Francisco, CA, 81–92.

AGGARWAL, C. C., HAN, J., WANG, J., AND YU, P. S. 2004. On demand classification of data streams. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04)*. ACM, New York, 503–508.

ALI, M., ELTABAKH, M., AND NITA-ROTARU, C. 2005. FT-RC4: A robust security mechanism for data stream systems. Tech. rep. TR-05-024, Purdue University.

ARASU, A., BABCOCK, B., BABU, S., DATAR, M., K. ITO, I. N., ROSENSTEIN, J., AND WIDOM., J. 2003. Stream: The stanford stream data manager. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, 665.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND THOMAS, D. 2004. Operator scheduling in data stream systems. *VLDB J. 13,* 4, 333–353.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'02)*. ACM, New York, 1–16.

BAI, Y. AND ZANIOLO, C. 2008. Minimizing latency and memory in dsms: a unified approach to quasi-optimal scheduling. In *Proceedings of the 2nd International Workshop on Scalable Stream Processing System (SSPS'08)*. ACM, New York, 58–67.

BISKUP, J. AND LOCHNER, J.-H. 2007. Enforcing confidentiality in relational databases by reducing inference control to access control. In *Proceedings of the 10th International Conference on Super Computing (ISC'07)*. ACM, New York, 407–422.

BRINKHOFF, T. 2002. A framework for generating network-based moving objects. *GeoInformatica 6,* 2, 153–180.

CAO, J., CARMINATI, B., FERRARI, E., AND TAN, K.-L. 2009. Acstream: Enforcing access control over data streams, demo. In *Proceedings of the International Conference on Data Engineering (ICDE'09)*. IEEE, Los Alamitos, CA.

CARMINATI, B., FERRARI, E., AND TAN, K. 2007a. Enforcing access control policies on data streams. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT'07)*. ACM, New York.

CARMINATI, B., FERRARI, E., AND TAN, K.-L. 2007b. Specifying access control policies on data streams. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA '07)*. Springer, Berlin, 410–421.

CARMINATI, B., FERRARI, E., TAN, K.-L., AND CAO, J. 2008. A framework to enforce access control over data streams. Tech. rep., University of Insubria.
http://www.dicom.uninsubria.it/˜barbara.carminati/TR/TR_Framework_AC_stream.pdf.

CHANDRASEKARAN, S., COOPER, O., A. DESHPANDE, M. F., HELLERSTEIN, J., W. HONG, S. K., MADDEN, S., V.RAMAN, REISS, F., AND SHAH., M. 2003. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proceedings of the Conference of Innovative Data System Research (CIDR'03)*. Online Proceedings.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. Niagaracq: a scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*. ACM, New York, 379–390.

CORAL8. 2008. Coral8 homepage. http://www.coral8.com/.

CRANOR, C., GAO, Y., JOHNSON, T., SHKAPENYUK, V., AND SPATSCHECK, O. 2003. Gigascope: A stream database for network applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. ACM, New York.

FARKAS, C. AND JAJODIA, S. 2002. The inference problem: A survey. *SIGKDD Expl. Newsl. 4,* 2, 6–11.

GABER, M. M., ZASLAVSKY, A., AND KRISHNASWAMY, S. 2005. Mining data streams: A review. *SIGMOD Record 34,* 2, 18–26.

GILBERT, A. C., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. 2001. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*. Morgan Kaufmann, San Francisco, CA, 79–88.

GOLAB, L. AND ÖZSU, M. T. 2003. Issues in data stream management. *SIGMOD Record 32,* 2, 5–14.

HAMMAD, M. A., FRANKLIN, M. J., AREF, W. G., AND ELMAGARMID, A. K. 2003. Scheduling for shared window joins over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03:)*. Morgan Kaufmann, San Francisco, CA, 297–308.

LAW, Y.-N., WANG, H., AND ZANIOLO, C. 2004. Query languages and data models for database sequences and data streams. In *Proceedings of the 30th international Conference on Very Large Data Bases (VLDB'04)*. Morgan Kaufmann, San Francisco, CA, 492–503.

LINDNER, W. AND MEIER, J. 2006. Securing the borealis data stream engine. In *Proceedings of the International Database Engineering and Application Symposium (IDEAS'06)*. IEEE, Los Alamitos, CA.

LIU, L., PU, C., AND TANG, W. 1999. Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowl. Data Eng. 11,* 4, 610–628.

MUTHUKRISHNAN, S. 2005. Data streams: algorithms and applications. *Found. Trends Theor. Comput. Sci. 1,* 2, 117–236.

NEHME, R. V., RUNDENSTEINER, E. A., AND BERTINO, E. 2008. A security punctuation framework for enforcing access control on streaming data. In *Proceedings of the 24th International Conference on Data Engineering (ICDE'08)*. IEEE, Los Alamitos, CA, 406–415.

PAPADOPOULOS, S., YANG, Y., AND PAPADIAS, D. 2007. Cads: continuous authentication on data streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*. Morgan Kaufmann, San Francisco, CA, 135–146.

RIZVI, S., MENDELZON, A., SUDARSHAN, S., AND ROY, P. 2004. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, New York, 551–562.

SCHREIER, U., PIRAHESH, H., AGRAWAL, R., AND MOHAN, C. 1991. Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB'91)*. Morgan Kaufmann, San Francisco, CA, 469–478.

STREAMBASE. 2008. StreamBase homepage. http://www.streambase.com/.

SULLIVAN, M. 1996. Tribeca: A stream database manager for network traffic analysis. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB'96)*. Morgan Kaufmann, San Francisco, CA, 594.

TERRY, D., GOLDBERG, D., NICHOLS, D., AND OKI, B. 1992. Continuous queries over append-only databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'92)*. ACM, New York, 321–330.

TRUVISO. 2008. Truviso homepage, http://www.truviso.com/.

ZHU, Y., RUNDENSTEINER, E. A., AND HEINEMAN, G. T. 2004. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, New York, 431–442.

Received January 2008; accepted February 2009