FENCE: Continuous Access Control Enforcement in Dynamic Data Stream Environments

Rimma V. Nehme [#1], Hyo-Sang Lim [*2], Elisa Bertino [*3]

[#]*Microsoft Jim Gray Systems Lab*
Madison, WI 53703 USA
[1]rimman@microsoft.com

[*]*Purdue University*
West Lafayette, IN 47907 USA
[2]hslim@cs.purdue.edu
[3]bertino@cs.purdue.edu.edu

***Abstract*— In this paper, we present *FENCE* framework that addresses the problem of *continuous access control enforcement* in dynamic data stream environments. The distinguishing characteristics of *FENCE* include: (1) the *stream-centric* approach to security, (2) the *symmetric* modeling of security for both continuous queries and streaming data, and (3) *security-aware query processing* that considers both regular and security-related selectivities. In *FENCE*, both data and query security restrictions are modeled in the form of streaming security metadata, called "*security punctuations*", embedded inside data streams. We have implemented *FENCE* in a prototype DSMS and briefly summarize our performance observations.**

I. Introduction

A. Security in Dynamic Environments

Due to recent developments in pervasive and ubiquitous computing, many enterprises begin to provide high-quality services based on real-time data, e.g., patient monitoring, location-based services, etc. One of the biggest challenges in dynamic Data Stream Management Systems (DSMSs) is *access control enforcement* – the ability to permit or deny a request to perform an operation (e.g., a read operation). Given the *long-running* nature of queries, the content of the streaming data and along with it its "sensitivity" may change frequently over the lifetime of query execution. Furthermore, queries themselves may also experience changes in their access control privileges, while they are being executed, e.g., due to mobility of people receiving the results of the queries. Clearly, the users streaming their data to DSMS can be rightly concerned about a possible unauthorized access to their real-time information and a potential violation of their privacy.

B. Our Contributions: FENCE Framework

To address the above-mentioned challenges, we propose the *FENCE* framework (short for *Continuous Access Control Enforcement in Dynamic Data Stream Environments*) that supports online enforcement of access control in data stream environments. *FENCE* addresses the limitation of the *Security Punctuation* (*SP*) model in [1] that only focuses on security restrictions of data. To the best of our knowledge, this work is the first to address the problem of access control enforcement with *concurrent* changes in security of *both* data and queries. Our contributions can be summarized as follows:

- *FENCE* models *both* data-side and query-side dynamic security restrictions *symmetrically* using streaming "security punctuations" or *sp*s[1]. *FENCE* distinguishes between two types of *sp*s, namely, the *data security punctuations* (*dsp*s) and the *query security punctuations* (*qsp*s).
- *FENCE* supports security-aware query processing by combining *dsp*s and *qsp*s at runtime. *FENCE* is equipped with two adaptive methods, namely: (1) *Security Filter Approach* (*SFA*), and (2) *Query Rewrite Approach* (*QRA*).
- *FENCE* supports two types of security policy enforcement, namely the *deferred* and the *immediate* enforcements. In the former, the access control policies are enforced on only the data tuples that arrive *after* the policy. In the latter, a policy is enforced *instantly* including the tuples that have arrived *before* the new policy.
- We have implemented *FENCE* in a prototype DSMS [2]. Our experimental study shows that *FENCE* efficiently supports access control enforcement with frequent data and query security changes and its overhead is low.

II. Problem Formulation

To formulate the problem we address in this paper, we first give the definition of *Continuous Query Processing* (or CQP for short). In traditional CQP, queries are registered in DSMS, and only the data tuples that satisfy the predicates of the queries are produced as results. We call these predicates – *query predicates* – and formally define CQP as follows:

Definition 2.1: (**CQP**) *Suppose that a data element* d*=(*v_1*,* v_2*,...,*v_n*) from a data stream has* n *attributes and a query predicate* $\varphi_Q(attr_1, attr_2, ..., attr_n)$ *on* d *represents the condition of a given continuous query* Q*. Then, whenever* d *arrives, continuous query processing mechanism produces* d *as a result of* Q *if and only if* $\varphi_Q(v_1, v_2, ..., v_n)$ *is true.*

In *Security-Aware Continuous Query Processing* (SA-CQP), we introduce a new type of predicates called *security predicates*, which determine whether a query may access arriving data tuples based on the current security restrictions. We

[1]Similar to [1], in this paper, we chose to name the security metadata – "security punctuations", because by introducing *dsp*s and *qsp*s into data streams, we subdivide (i.e., punctuate) infinite data streams into finite partitions with associated security restrictions.

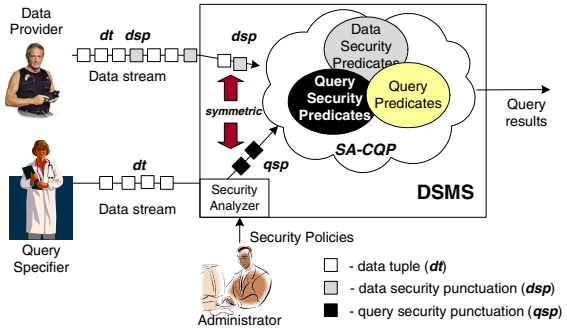

Fig. 1. Overview of *FENCE* architecture.

distinguish between two types of security predicates, namely: (1) the *data-side security predicates*, which represent the data provider's security policies on the streaming data and (2) the *query-side security predicates*, which describe the query specifier's current access authorizations. Queries, registered by a user (i.e., query specifier) implicitly acquire the access authorizations of that query specifier. Consequently, SA-CQP enforces access control on data streams by only producing the results that satisfy *both* the query predicates and the security predicates. SA-CQP can be formally described as follows:

Definition 2.2: (**SA-CQP**) *Suppose that a data element* $d = (v_1, v_2, ..., v_n)$ *from a data stream has* n *attributes, a query predicate* $\varphi_Q(attr_1, attr_2, ..., attr_n)$ *on* d *represents the condition of a given continuous query* Q*, and a security predicate* $\varphi_S(attr_1, attr_2, ..., attr_n)$ *on* d *represents a security policy* S*. Then, whenever* d *arrives, security-aware continuous query processing outputs* d *as a result of* Q *if and only if* $\varphi_Q(v_1, v_2, ..., v_n) \wedge \varphi_S(v_1, v_2, ..., v_n)$ *are both true.*

In *FENCE*, we provide an efficient SA-CQP solution for dynamic data stream environments.

III. Overview of FENCE Framework

System Architecture: Figure 1 shows a high level overview of *FENCE*. In a typical streaming environment, we distinguish between three types of users: (1) a data provider – a user continuously sending his or her streaming data with the interleaved *dsp*s, that describe the real-time security preferences on his or her streaming data; (2) a query specifier – a user who registers a continuous query on the server to be evaluated on the incoming streaming data. As can be seen from Figure 1, a query specifier also streams his or her real-time context (via a data stream), based on which *qsps* that describe the real-time access privileges of the continuous query are generated. (3) a DSMS administrator – a user responsible for specifying policies that guarantee that correct authorizations are given to the queries based on the context of the query specifiers.

Data and Query Model: We consider a centralized DSMS processing long-running select-project-join (SPJ) queries on a set of infinite data streams. A continuous data stream *S* is a potentially infinite sequence of tuples that arrive over time. The general schema of tuples in a data stream is described by: [*sid*, *tid*, *A*, *ts*], where *sid* is the stream identifier, *tid* is the tuple identifier, *A* is a set of attribute values in the tuple, and *ts* is the timestamp of the tuple. As commonly considered in other streaming systems, e.g., [3], [4], the timestamps of the stream elements are assumed to be ordered. For simplicity of presentation, we consider a single continuous query Q_i, registered by a query specifier in the DSMS. In DSMS, query Q_i is represented by a query execution plan composed of operators op_1,..., op_k, where each operator acquires the security restrictions associated with the query Q_i for which it processes data tuples.

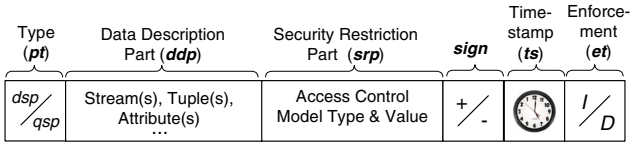

Fig. 2. General security punctuation schema.

Access Control: As an example of an access control model, we consider a *Role-Based Access Control* (*RBAC*) as one of the most well-known and widely used models in modern systems today [5] and show how it is implemented in *FENCE*.

IV. Modeling Dynamic Security

A. Security Punctuation Schema

We employ security punctuations to symmetrically model both data and query-side dynamic security restrictions. Figure 2 shows a general *sp* schema applicable to both *dsp*s and *qsp*s. We discuss each field in the schema next.

- *Punctuation Type* (*pt*): describes whether the punctuation is a data or a query security punctuation.
- *Data Description Part* (*ddp*): specifies which object(s) the access control policy applies to, e.g., which stream(s), tuple(s), or tuple attribute(s) [1]. For compactness of storage, we use *regular expressions* to describe objects and their policies inside *sp*s.
- *Security Restriction Part* (*srp*): denotes both the access control model type and the subjects authorized by the policy. Since we use RBAC in this work, the *srp* specifies RBAC as the model type and a set of role(s) that are authorized by the *sp*.
- *Sign*: indicates whether the authorization represented by the *sp* is positive or negative (see [6] for more details).
- *Timestamp* (*ts*): is the time when the *sp* was generated.
- *Enforcement* (*et*): indicates the security policy enforcement setting. We distinguish between two types of enforcement, namely the *Deferred* (*D*) enforcement and the *Immediate* (*I*) enforcement.

B. Examples of Security Punctuations

Consider the following streams: S_1 – a heart data stream, S_2 – a blood pressure data stream and S_3 – a respiration data stream. Let R = {D_1,D_2,D_3,D_4,D_5} be the set of roles in DSMS[2]. The following *dsp*s and *qsp*s may specified[3]:

dsp$_1$: <dsp|S_1,*,*|D_2|+|12:00:00PM|D>

only queries registered by a cardiologist (role D_2) can query stream S_1 (heart rate) after this punctuation arrives due to deferred semantics. This is an example of a *stream level policy*.

dsp$_2$: <dsp|*,[30,210],*|D_4|+|12:00:00PM|D>

only queries registered by a general physician (role D_4) can

[2]The roles can be as follows: D_1 = dermatologist, D_2 = cardiologist, D_3 = hospital employee, D_4 = general physician, and D_5 = nurse-on-duty.

[3]The different fields in an *sp* are separated by a vertical bar "|".

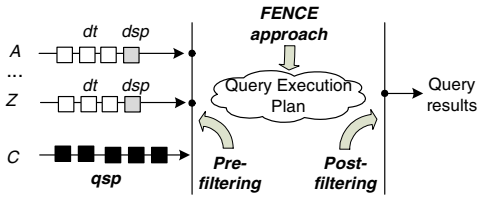

Fig. 3. Query processing with *sp*s.

access data tuples (from any data stream) of patients with ids between 30 and 210, after this punctuation arrives ($dsp_2.et$ = D). This is an example of a *tuple level policy*.

qsp$_1$: <qsp|null|D_1|+|12:00:00PM|D>

query acquires a role of a dermatologist (D_1) with deferred enforcement, i.e., the role applies to the query after the arrival of qsp_1 and will pertain to the data tuples with the timestamp greater than $qsp_1.ts$.

qsp$_2$: <qsp|S_1,*,*|D_4|+|12:00:00PM|D>

query acquires a general physician (D_4) role and the current authorization of role D_4 is the permission to only access stream S_1 (heart data stream). The enforcement is deferred.

To determine which data tuples, the query currently has access to, the *intersection* of the *dsp*s and the *qsp*s is evaluated [1]. Only if the intersection between the policies of *dsp*s and the authorizations of *qsp*s is non-empty, the access to the streaming data elements is granted.

C. Enforcement of Security Punctuations

To reflect various users' security preferences, we introduce two ways of policy enforcement, namely the *deferred* and the *immediate* enforcements. In the case of the former, a policy represented by an *sp* applies only to the data that arrives *after* the *sp*, i.e., the tuples whose timestamps are greater than that of the *sp*. This type of enforcement is the most frequent case, and is needed for applications that need to protect "future" data. For example, if a user carrying a cell phone device enters a casino, he or she may want to instantly prevent others from knowing his precise whereabouts. With the immediate enforcement, the new policy affects both the (near past) data that has arrived to the DSMS *before* the current *sp* as well as to the future data that follows *after* it. Hence, the policy here may apply to both the "historic" and the "future" data. This type of enforcement is needed for the applications that demand the immediate reflection of policy changes on the query results.

V. Query Processing in FENCE

A. Naive Approach

A naive method for query processing with dynamic security is to completely separate the access control processing from regular CQP. Such strategy evaluates security predicates at a designated point – either before or after query execution plan. The former and the latter strategies are also known as *pre-filtering* and *post-filtering*, respectively [1]. Figure 3 illustrates the naive approach along with *FENCE* approach. In both the pre- and the post-filtering methods, the fixed placement of the access control filters may considerably limit query performance. To overcome these limitations, we propose two efficient SA-CQP methods employed in *FENCE*. Both methods have a key advantage – they *integrate* security processing

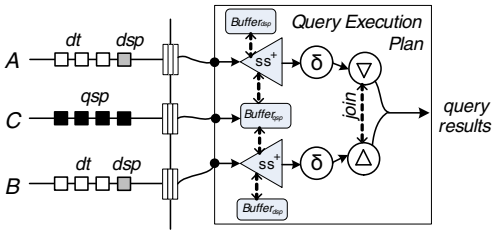

Fig. 4. *SFA*-based SA-CQP.

together with traditional query processing and can adapt to not only data-related but also security-related selectivities.

B. Security Filter Approach (SFA)

The main idea of *SFA* is to introduce a special physical operator into the query execution plan that performs access control-based filtering. We call this new operator a *Security Shield Plus* (SS^+) operator[4], and it is handled just like any other traditional query operator in query processing and query optimization. SS^+ operator can be viewed as a "select operator" that filters input data tuples based on the security predicates determined based on the arrived *dsp*s and *qsp*s. The filtering condition of SS^+ changes dynamically whenever a new *dsp* or a *qsp* arrives. Figure 4 shows how the *SFA*-based SA-CQP works with SS^+ operators. The triangle-shaped operators in the figure are the SS^+ operators filtering data based on the security predicates of the query. Just like for an ordinary select operator, the location of SS^+ in the query plan is determined by the query optimizer according to the selectivities of the security predicates.

By encapsulating all security processing inside SS^+ operators, *SFA*-based SA-CQP can interleave security predicates with traditional query predicates. *SFA*, however, may require substantial modifications to the codebases of the current DSMSs (see Section V-D). In the next section, we propose another SA-CQP approach that minimizes the need to change existing DSMSs and largely leverages them as they are.

C. Query Rewrite Approach (QRA)

The main idea behind the *QRA*-based SA-CQP comes from the observation that dynamic enforcement of security policies can be seen as a "dynamic rewriting of queries"[5]. According to the SA-CQP definition (in Section II), we consider a query registered in DSMS that consists of query predicates and security predicates, where security predicates are updated whenever a new *sp* arrives. A DSMS can support dynamic security changes in SA-CQP by creating a "new" query with the integrated in it latest security predicates and replacing with it the current query. Table I shows the example of the query rewriting method. Here, the original query predicate $(R.a = S.a) \wedge (0 < R.b < 100) \wedge (0 < S.c < 100)$ is rewritten into $(R.a = S.a) \wedge (0 < R.b < 50) \wedge (50 < S.c < 100)$ to reflect the access control policies described by the dsp_1 and the qsp_1.

[4] SS^+ is similar in spirit to the initially proposed *Security Shield* (*SS*) operator in [1], however, its semantics is more sophisticated, providing support for both *dsp*s and *qsp*s with richer semantics.

[5] Query rewriting is generally used to compose queries or manage views. In this paper, we exploit the query rewriting concept for the purpose of combining security and query predicates to adapt to dynamic changes in access control.

TABLE I

EXAMPLE OF QUERY REWRITING.

Original Predicates	Rewritten Predicates
$Q.p_1 \rightarrow (R.a = S.a)$	$Q.p_1$' $\rightarrow R.a = S.a$
$Q.p_2 \rightarrow (0< R.b <100)$	$Q.p_2$' $\rightarrow 0< R.b <50$ //$Q.p_2$+dsp_1
$Q.p_3 \rightarrow (0< S.c <100)$	$Q.p_3$' $\rightarrow 50< S.c <100$ //$Q.p_3$+qsp_1
$dsp_1 \rightarrow (0< R.b <50)$	
$qsp_1 \rightarrow (50<S.c <100)$	

Figure 5 gives an overview of the *QRA*-based SA-CQP. Compared to *SFA*, where SS^+ operators process *sp*s in the query plan, the *QRA* uses a centralized module, called the *Query Rewriting Module* (*QRM*), to process *sp*s. *QRM* consumes *dsp*s and *qsp*s immediately upon their arrival to the system and stores them in the global $Buffer_{dsp}$ and the $Buffer_{qsp}$, respectively. *QRM* also stores traditional query predicates in the $Buffer_{query}$. Whenever new *sp*s arrive, *QRM* rewrites the corresponding query using the information stored in these buffers. Regular data stream tuples are processed by the query processor in the same way as in traditional DSMSs. We note that *sp*s are not sent into the query execution plan, but rather consumed by the *QRM* module to generate a new query. In that regard, the query operators do not need to be "security punctuation-aware" as in the *SFA*.

D. Pros and Cons of QRA and SFA

The major difference between the *QRA* and the *SFA* is the abstraction level of the security predicates. In the *QRA*, security predicates are represented as logical conditions of a query. In contrast, in the *SFA*, security predicates are encapsulated in separate physical (SS^+) operators in the query execution plan. This difference contributes to both the pros and the cons of the approaches. The main advantage of the *QRA* is that the existing query processor infrastructure can be largely re-used as it is, since the *sp*s are not propagated into the query execution plan. Conceptually, the *QRA* treats the existing query optimizer as a "black box" and invokes it as a sub-routine, with a query specification that integrates both the query and the security predicates. Such approach is faithful to the goal of minimizing code changes in existing systems, but may result in a blow-up in optimization time by a factor equal to the number of sub-routine invocations. In the worst case, this may happen every time a new *dsp* or a *qsp* arrives to the system. Clearly, the main disadvantage here is that this approach is not very robust to dynamic changes in security. Potentially, every new *dsp* and *qsp* may lead to the query plan rewriting, the optimizer re-invocation and the physical query plan migration, thus consuming the precious resources from producing continuous query results.

The main advantages of the *SFA* include its high performance and robustness to dynamic changes in security. Whenever a new *qsp* or *dsp* arrives, only the SS^+ operators are affected to reflect the changes in security policy, and the rest of the query plan does not need to be modified. The *SFA* approach is also more amenable to shared query processing in the case of multiple queries. If queries have the same query predicates (even if their authorizations are different), the processing can be shared with the proper security filters installed before and after the shared sub-part in the execution plan. Introducing new operators into the query algebra and making the existing query operators security-aware, however, brings its disadvantages. The query optimizer must now become aware of these new operators, their semantics and must also adjust the cost model to reflect the streaming *sp*s statistics and the cost of their processing. In summary, the codebase of DSMS may need to undergo significant changes to accommodate the security-awareness inside the query processor.

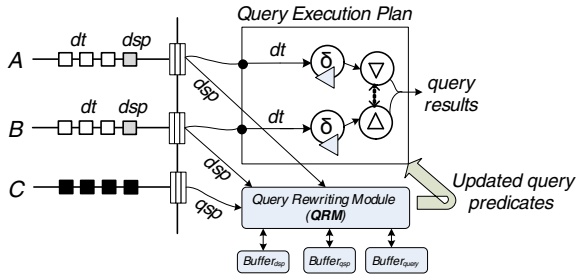

Fig. 5. *QRA*-based SA-CQP.

VI. CONCLUSIONS

In this paper, we address the problem of access control enforcement in streaming environments, where both data and query security restrictions may change while queries are being executed. We have proposed *FENCE* framework, where data and query access control policies are modeled symmetrically using *dsp*s and *qsp*s. We believe our work makes an important contribution to both fields of databases and security in that it is the first to propose and implement a practical approach for online continuous access control enforcement.

Our experimental observations can be summarized as follows:[6]

1) The symmetric model with *dsp*s and *qsp*s significantly outperforms the other alternatives, especially when more tuples share the same security policies.
2) The *SFA* can give up to 37% improvement over the naive approach and up to 22% over the *QRA* approach in the execution time and the output rate.
3) The runtime overhead of *continuous access control enforcement* relative to query execution is at most 21% for the *QRA*-based query processing and only 7% for the *SFA*-based query processing.
4) In general, the ability of query processor to adapt to not only data-related but also to security-related selectivities can significantly improve query performance.

REFERENCES

[1] R. Nehme et.al., "Security punctuation framework for enforcing access control on streaming data," in *ICDE*, 2008.

[2] E. Rundensteiner et al., "Cape: Continuous query engine with heterogeneous-grained adaptivity," in *VLDB*, 2004.

[3] A. Arasu et al., "STREAM: the stanford stream data manager," in *SIGMOD*, 2003, pp. 665–665.

[4] D. Abadi et al., "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[5] R. Sandhu et al., "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[6] E. Bertino et al., "An extended authorization model for relational databases," *TKDE*, vol. 9, no. 1, pp. 85–101, 1997.

[6]For complete experimental analysis, we refer the reader to our technical report.