CERIAS Tech Report 2006-29

SECURITY AND PRIVACY IN DATA STREAM MANAGEMENT SYSTEMS

by Rimma V. Nehme, Elke A. Rundensteiner, Elisa Beritno

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Security and Privacy in Data Stream Management Systems

Rimma V. Nehme[1] Elke A. Rundensteiner[2] Elisa Bertino[1]
[1]Department of Computer Science, Purdue University, West Lafayette, IN 47907 USA
[2]Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01608 USA
rnehme@cs.purdue.edu, *rundenst@wpi.edu*, *bertino@cerias.purdue.edu*

Abstract

*Privacy and security in the context of the streaming systems largely have been overlooked. We now tackle this important problem in this paper. Our work focuses on context-aware security and user-centric privacy preservation in data stream management systems (DSMS) by exploiting security constraints (called security punctuations) that are dynamically embedded into data streams. The novelty of our proposed approach is that access control policies are not persistently stored in the DSMS but rather streamed together with the data. We present novel query operators, termed Security Shield (*SS*) and Security-Compliant Join (*SCJoin*) that are designed to make queries comply with the security policies of the streaming data while still guaranteeing near real-time response. As a proof of feasibility, we have implemented the security punctuation framework within a real DSMS. Our experimental results show that our proposed solution incurs low overhead.*

1. Introduction

1.1. Security in Data Streaming Environments

The need for people to protect themselves and their assets is as old as humankind. The increasing use of electronic, sensor and GPS devices means that individuals today have an ever-growing range of electronic (data) assets that may potentially be at risk. When computing devices are integrated with people, various personal information is expressed in digital form. Devices can communicate this information over networks and users have no control over who and for what purpose may query their data. Some users, knowing that their personal information (e.g., location, health condition) is not safeguarded, may hesitate to use such (e.g., GPS or health monitoring sensors) devices because of the risk of data being inappropriately used, or the potential loss of control or power over their information assets.

Traditional access control schemes and privacy preservation mechanisms, which typically assume finite persistent datasets and system-centric access control policies, become largely inapplicable in this new stream paradigm. This inapplicability is due to the fact that stream environments tend to be highly dynamic. Data is continuously generated and may have different security sensitivities depending on the context, on personal preferences or on the stream content, all of which may frequently change. The users owning such devices should have the ability to control their exposure to the rest of the world. That is, they should be able to for example hide the knowledge about themselves whenever desired.

We envision that individual devices transmitting streaming data will be able to inject their respective security restrictions together with the data. Users will be able to either explicitly specify their restrictions at runtime, or the devices may come pre-set with customizable security rules that would emit and adjust different security settings at runtime based on the context information, such as location, time, physical condition, proximity of other users, etc.

In this paper, we assume that streaming data is transmitted securely from the data source to the streaming database[1]. We concentrate on access control mechanisms, wherein the streaming database engine examines streaming data tuples and continuously checks if a query conforms to the streaming security policies before permitting the query to access the data.

1.2. Motivating Examples

Example 1: Protection against context-aware spam. Users may want to block unwanted businesses from sending them advertisements based on their location or any other information. As a person is driving or walking, the device may adapt security constraints based on the proximity of the stores/businesses and the user preferences (possibly pre-set in advance) limiting to who would be allowed to "see" the user. This helps to impede focused marketing efforts and prevents from receiving the "context-aware spam" – services or information the users don't know of or agree to.

Example 2: Privacy protection of personal health data. A patient may be living at home with a health monitoring device attached to him which can detect early health abnormalities and transmit alert signals to relevant personnel. However, the patient may prefer only certain user groups, such as the closest hospital or his doctor to have access to his streaming data and prevent access for any third-parties (e.g., insurance company or other hospitals). Only if his vital signs

[1]That is, the possibility of the streaming data being intercepted and compromised on the network is beyond the scope of this paper.

go far above the norm and he is in imminent danger needing urgent care, would the closest hospital, ER or ambulance dispatch center gain access to his streaming data. By setting his privacy preferences, the patient can prevent unauthorized people from accessing the information or selectively choose who has access to which part of his data based on the real-time values streamed by his monitoring device.

Example 3: Privacy protection in the workplace. Many aspects of the modern workplace have introduced serious concerns about employee privacy. Company cars, cell phones, GPS devices, and laptops today provide employers with powerful capabilities to monitor the activities of their employees. During the course of a working day, an employee may go to both business-related and non-business-related places. Monitoring a personal trip, for example during a lunch break, might be an unreasonable intrusion on an employee's privacy. Sometimes a user may want to specify rules to "hide" or selectively limit which data the managers or co-workers can access from his data stream at either a certain time or a certain location.

1.3. Our Proposed Solution: SPs Framework

We propose to stream security constraints together with the actual data stream indicating security/privacy preferences on the current portion of the stream. Specifically, we propose to embed *security punctuations* (or short *sp*s) into data streams[2]. A security punctuation is a predicate that describes a subset of tuples and their access control policy, also called a *security policy*. It informs a stream processor of the access control privileges on a stream as a whole, or a certain substream of tuples, or on some attribute(s) of a tuple. A conceptual view of a stream with security punctuations is shown in Fig. 1. Data sources emit security punctuations (*sp*s) based on user specifications. In our work, we distinguish between two types of users: (1) users providing the streaming data, termed *data providers*, and (2) users querying the streaming data, termed *query specifiers*.

A streaming database has a security punctuation analyzer component which serves two purposes: first, to combine security punctuations with similar policies (to reduce memory overhead and save CPU) and second, to allow server-side specification of additional security policies. In the latter case, the server policies are translated into security punctuations and combined with the arriving data provider *sp*s. Such design allows organizations to enforce their own policies in addition to the ones specified by the data providers. We assume that server-specified policies may *not* override, but may further "refine" data provider policies, by putting in additional constraints. Tuples, preceded by their corresponding *sp*s, are streamed into a DSMS where continuous queries are evaluated subject to the tuples' security policies.

[2] We chose the name "security punctuations", because by introducing *sp*s into data streams, we subdivide i.e., *punctuate* infinite data streams into finite partitions with associated security policies.

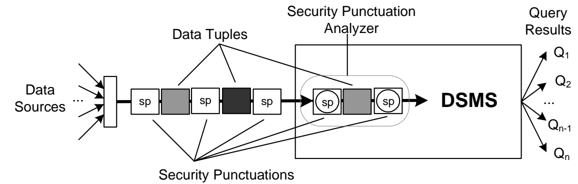


Figure 1. Data Stream With SPs.

Our Contributions:

- We propose a security punctuation model as a real-time, fine-grained access control mechanism to enforce security on streaming data (Section 4).
- We describe an extension to the CQL language [4] for the specification of security punctuations in data streams (Section 4.3).
- We present a novel *Security Shield* (SS) operator to enforce the compliance of continuous queries according to the policies expressed via streaming *sp*s. SS preserves the correctness of the security semantics even when policies may be missing or arrive out-of-sync. We also provide the proof of correctness of SS operation (Section 5.1).
- We propose a *Security-Compliant Join* (SCJoin) algorithm. SCJoin is designed to (i) join several data streams each streaming tuples with possibly distinct access control policies, and (ii) share the join processing among multiple queries specified on those same data streams but with different access privileges. The novelty of the SCJoin lies in the adaptive selection among several join strategies taking window and shared policies into account. (Section 5.4).
- We experimentally evaluate our approach in the DSMS system *CAPE* [15] against data streams that have no security policies embedded in them. Our experimental results show that our approach has low overhead (Section 6).

Roadmap: Section 2 reviews related work. Section 3 discusses the stream model and our assumptions. Section 4 introduces the concept of security punctuations. Section 5 discusses query processing framework with *sp*s, whereas Section 6 presents our experimental evaluation and Section 7 concludes the paper.

2. Related Work

Data Stream Management Systems. Streaming databases have been a hot topic in the past few years [5, 8, 13, 15]. *Punctuations* [17, 18] – dynamic annotations serving as substream delimiters inside data streams have been first presented in [17]. Further, *PJoin* [10] and *PWJoin* [11] apply punctuations to achieve join optimizations on streaming data. Fegaras et al. [12] use annotation in the streaming XML data to declare the data structure of the incoming data, and whether the data fragment following the annotation is a repeat or an update. We go beyond the simplistic notation of indicating future incoming values, by now employing a more sophisticated security-related semantics.

Security and Privacy Preservation. W3C developed the Platform for Privacy Preferences (P3P) specification [9] for

encoding web site privacy policies. P3P, however, is a cumbersome language for streaming environments. In P3P users cannot directly specify what is acceptable in a policy, only what is unacceptable. Simple policies often result in convoluted and verbose P3P specifications [2].

Agrawal et al. proposed the concept of Hippocratic databases [3] to incorporate privacy protection within RDBMS. The authors propose using privacy metadata. This work however addresses neither dynamic changes in policies, nor support for both user and system policy specification. Both those features we now address in our work.

Preserving privacy by ensuring limited disclosure of data in RDBMS was explored by Lefevre et al. [14]. The implementation is based on query modification techniques. The proposed approach has several limitations in the context of streaming systems. First, queries in DSMS are typically long-running, thus access policies may change many times during the execution of a query. Modifying it at runtime for privacy preservation would cause modifications in the query plan. Since sub-plans may be shared among multiple queries, this may cause a cascading effect on other query plans, potentially "stalling" the system and not producing any query results (while the query plan is migrated). Our technique solves this problem by using special stateful SS operators that filter data for the outstanding queries that they are not allowed to access. The SS operators' states depict current access privileges of the outstanding queries and can be simply updated whenever a query access privileges change, thus eliminating the need for a query modification.

In [1], the authors propose language constructs for fine grained access control (FGAC) in RDBMS. The work proposes an alternative to how relational databases currently support FGAC via traditional mechanisms based on views, triggers or special registers. The proposed method for FGAC works well only when the number of restrictions is small or the data is relatively static. Such conditions are rare for streaming data.

We advance the state-of-the-art by addressing the limitations of the current techniques and introduce a security-compliant data stream framework where access control policies are streamed together with the data instead of stored on the server.

3. Preliminaries

Subjects, Objects, Rights and Authorizations. The subject, object and right concepts are well known in access control. An $object$ is an entity that contains information. Access to an object implies right to use the information it contains. Examples of objects in streaming system are: streams, tuples, and tuple attributes. A $subject$ may invoke a request to access an object, e.g., a read request[3]. We use *flat*

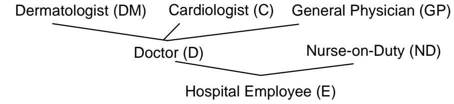


Figure 2. Roles Organized into a Hierarchy.

Role-Based Access Control (RBAC) model[4] as an example of an access control model and show how it can be implemented through security punctuations. We chose RBAC as it is one of the most widely used access control models. However, our framework is generic, that is any access control model (e.g., DAC, MAC) could be implemented using the *sp*s model. Query specifiers activate their roles when they sign into the streaming system. We require that each query specifier belongs to at least one role, and this assignment cannot be changed while executing the queries.

Rights are a set of privileges that a subject can hold and execute on an object. In this work, we consider a read right only[5]. A *security policy* (aka *access control policy*) is a set of rules indicating what query specifiers are allowed to access. *Authorization* is the granting of rights.

Streaming Model. We consider a centralized DSMS processing long-running queries on a set of data streams. A *continuous data stream s* is a potentially unbounded sequence of tuples that arrive over time. Tuples in the stream are of the form $t = [sid, tid, A, ts, ts.csn_{sp}]$, where sid is the stream identifier, tid is the tuple identifier[6], A is a set of attribute values of the tuple, ts is the timestamp of the tuple[7], and $ts.csn_{sp}$ is the timestamp and the *cumulative* sequence number of the corresponding security policy represented by *sp*s (Section 4 elaborates more on this parameter).

We consider a set of continuous queries $\{q_1,..\,.q_p\}$ executing over data streams, where each query q_i has associated security restriction(s) determined by the role (r) of the query specifier (denoted as q_i^r). Queries are comprised of a set of query operators $\{op_1, \ldots op_k\}$, where operators *inherit* the security restrictions of the queries for which they process the data (i.e., if $op_k \in q_p^r$, then $op_k = op_k^r$).

4. Security Punctuations (SPs)

Security punctuation (*sp*) is a meta-data introduced into a data stream to specify security restrictions on the tuples.

Applicability: Sps always precede the tuples for which they describe the access control policy (Fig. 3). An access control policy specified via *sp*(s) may apply to (1) a (sub)stream, (2) a tuple, or (3) an attribute of a tuple. Generally, we refer to them as $objects$. The tuples between two consecutive punctuations form an *s-punctuated segment*. The s-punctuated segment describes the applicability scope of the immediately

[3]We consider the subjects in our model to be a set of users who specify continuous queries in DSMS (i.e., query specifiers).

[4]Fig. 2 depicts a set of roles organized in a hierarchy. For more information on RBAC readers are referred to [16].

[5]Just about all stream systems are read-only right now, hence this is a natural focus. Our model can be extended to support other rights, e.g., update, append, etc.

[6]This may be similar to a primary key in relational tables, or it may be a unique attribute(s) that can be used to identify a particular data provider, e.g., patient_id.

[7]The timestamps of stream elements are assumed to have a global ordering based on the system's clock.

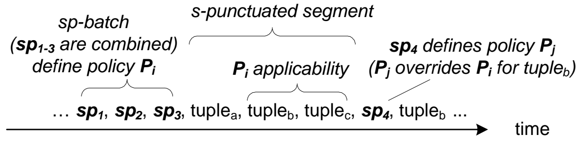


Figure 3. Security Punctuations' Applicability.

preceeding policy.

Any consecutive sps belong to the same *sp-batch* and are interpreted as a single access control policy P_i. A policy may consist of one or more *sp*s. A policy P_j applicable to an object o overrides a previous policy P_i that had arrived earlier and was applicable to o[8]. At any time, a policy may apply to zero or more tuples and any tuple may have either one or no policy.

*E*ach sp has a parameter $ts.sn_{sp}$, which contains the timestamp ts for *when* the policy was generated, and its sequence number (sn) in the policy[9]. Each tuple also carries a $ts.csn_{sp}$ parameter that contains the information about its policy. The csn here depicts the cumulative sequence number of the last *sp* that belongs to the policy. The stream system uses this parameter to detect the missing and out-of-order *sp*s and protects the data from an unauthorized access (Section 5.1). If an *sp* is missing, we then denote the policy as being *incomplete* and enforce *denial-by-default* (i.e., no access is allowed) to the objects with that policy.

Structure: Security punctuations are composed of four parts (Fig. 4): (1) *Data Description Part* (*DDP*), (2) *Security Restriction Part* (*SRP*), (3) a *Sign*, and (4) an *Immutable* field. *DDP* specifies to *which* object(s) the access control policy applies. *SRP* denotes both the access control model type and the value of the subjects that are authorized. As mentioned before, we use a role-based access control model as an example model in this work. Thus, the *SRP* part of the security punctuation specifies RBAC as model and a set of *role*(s) that are authorized by the *sp*[10]. However, our RBAC model usage is general, so in principle any access control model could be plugged into security punctuations' *SRP*. The *Sign* specifies if the authorization is *positive* or a *negative* [6]. Finally, the *Immutable* field indicates if the *sp* can be combined with other (e.g., server-specified) policies.

We use pattern expressions to describe objects and their restrictions. Patterns are suitable here, since many objects may share similar policies. Fig. 5 illustrates different kinds of patterns *sp*s match.

Let $eval(N, V, E)$ be a function that, given a set N of values, their type V and a pattern expression E, returns a subset $N_s \subseteq N$ (of type V) that matches E. We distinguish between four types of values in our model, that is, V can

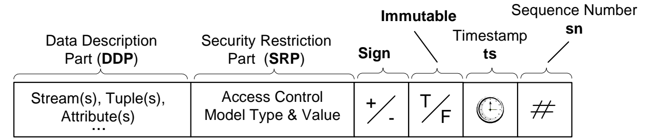


Figure 4. Structural Design of an SP.

Pattern	Description
∅	Empty (i.e., does not match any values)
*	Wildcard (i.e., matches any value)
c	Constant (i.e., matches only *c*)
(*c₁*,*c₂*)	Exclusive Range (i.e., matches values strictly in the range)
[*c₁*,*c₂*]	Inclusive Range (i.e., matches values in the range inclusively)
{*c₁*,*c₂*,*c₃*}	Set (i.e., matches values in the set)

Figure 5. Pattern Expressions for SPs.

be *Streams*, *Tuples*, *Attributes* or *Roles*. We now formally define security punctuations.

Definition 4.1. *(Security Punctuation) Let*
(1) $S = \{s_1 \dots s_m\}$, $m \geq 1$, *be the set of all streams,*
(2) $T = \{t_{i,1} \dots t_{i,n}\}$, $i = 1 \dots m$, $n \geq 1$, *be the set of tuples in a stream* $s_i \in S$,
(3) $A = \{a_{i,j,1} \dots a_{i,j,k}\}$, $i = 1 \dots m$, $j = 1 \dots n$, $k \geq 1$, *be a set of attributes in a tuple* $t_{i,j} \in T$ *in a stream* $s_i \in S$,
(4) $R = \{r_1 \dots r_l\}$, $l \geq 1$, *be the set of roles in a system.*
A security punctuation sp defines a security policy P *and has the form:* $<$ ***DDP*** | ***SRP*** | {+/-} | {***T/F***} | ***ts*** | ***sn***$>$ *where* $sp.DDP = (E_s, E_t, E_a)$ *and* $sp.SRP = E_r$. E_s, E_t, E_a, *and* E_r *are pattern expressions specified against* S, T, A *and* R, *respectively.* ts *is the timestamp of the policy* P *and* cs *is the sequence number of the* $sp \in P$. *Let* $O = \{o_s, o_t, o_a\}$ *be a set of objects where* o_s=$(\bar{s})$, o_t=$(\bar{s}, \bar{t})$, *and* o_a=$(\bar{s}, \bar{t}, \bar{a})$, *such that* $\bar{s} \in eval(S, Streams, E_s)$, $\bar{t} \in eval(T, Tuples, E_t)$ *and* $\bar{a} \in eval(A, Attributes, E_a)$. *The interpretation of the security punctuation is the following:*
• ***if Sign = '+':*** *a subject with role* $r \in eval(R, Roles, E_r)$ *may access any object* $o \in O$ *at any time* $ts_{access} \geq ts$.
• ***if Sign = '-':*** *a subject with role* $r \in eval(R, Roles, E_r)$ *is denied access to any object* $o \in O$ *at any time* $ts_{access} \geq ts$.
If Immutable = `False`*, sp may be combined with the server-specified policies applicable to the same objects. Otherwise, sp is immutable, and the server-side policies are ignored.*

A security punctuation can be specified at the level of a stream, a tuple, or an attribute. For simplicity of presentation, we assume positive and mutable *sp*s in the rest of our discussion[11].

4.1. Security Punctuation Examples

Here we give examples of *sp*s using three data streams (Fig. 6): *HeartRate*, *BodyTemperature* and *BreathingRate*. Let the set of roles *R* be the following, *R* = {*C,D,DM,E,GP,ND*} as in Fig. 2. The following *sp*s may be created (the ts and sn parameters are omitted):

Stream level restriction:
$<s_1$, *, * | *C* | + $>$ - Only queries registered by a cardiologist (C) can query the stream *HeartRate* (s_1).

Tuple level restriction:
$<$*, [120,133], * | *GP* | + $>$ - Only queries registered by a general physician (GP) can access data tuples (from any data stream) for patients with ids between 120 and 133.

[8] We plan to consider incremental policy change via *sp*s as a part of our future work.

[9] All *sp*s that belong to the same policy have the same timestamp ts, but a unique sequence number. The sequence number represents the order of the *sp* in the policy.

[10] We omit the access control model specification in the *sp*s, since all of them are assumed to use RBAC model.

[11] We omit the *Immutable* field. Unless noted otherwise, all *sp*s are assumed to be mutable.

s_1: HeartRate Stream	Patient_id | Beats_per_min | Timestamp 120 | 70 | Sep-12-05 9:17:00 ...
s_2: BodyTemperature Stream	Patient_id | Temperature | Timestamp 120 | 98.6 | Sep-12-05 9:21:00 ...
s_3: BreathingRate Stream	Patient_id | Frequency | Depth | Timestamp 120 | 8 | 38 | Sep-12-05 9:22:00 ...

Figure 6. Sample Data Streams.

Attribute level restriction:
$<\{s_1, s_2\}$, *, $\{Temperature,\ Beats_per_min\}$ | $\{D, ND\}$ | + > - Only a doctor (D) or a nurse-on-duty (ND) can query the temperature and the heart beats from s_1 and s_2 streams.

4.2. Combining Multiple SPs

For manipulating *sp*s in streams, we use three basic functions: *match*(), *union*() and *intersect*(). *match*() identifies what tuples are related to a security punctuation, after evaluating expressions in *sp.DDP*. If multiple *sp*s applicable to the same tuples have been defined (e.g., data provider and server-specified *sp*s), we consider two design choices for combining those *sp*s:

Union: This corresponds to the union of applicable security restrictions
• $match(t, (union(sp_1, sp_2))) \iff match(t, sp_1) \lor match(t, sp_2)$

Intersect: This corresponds to the intersection of applicable security restrictions
• $match(t, (intersect(sp_1, sp_2))) \iff match(t, sp_1) \land match(t, sp_2)$

With the intersect semantics, the access to data decreases as additional *sp*s are applied. Conversely, with the union semantics, access to data increases as additional *sp*s are applied. To dissallow server policies from increasing access, intersect semantics should be applied. Alternatively, a data provider can set *Immutable* field = `True` (`T`), thus preventing any modification to data provider policies on the server side.

4.3. CQL Extensions to Support SPs

We have extended CQL [4] syntax to support the specification of *sp*s on data streams (syntax is illustrated in Fig. 7[12]). Fig. 8a gives an example of using our proposed syntax and represents the following *sp*:

$<s3$, [120,200], Frequency | $R1^G$ | + | $T >$[13]

It is a positive and an immutable *sp* that allows role *R1* to access *breathing frequency* in stream *s3* for patients with ids *120* through *200*, and role *R1* can *delegate* access control at this granularity (i.e., Frequency attribute for this set of patients) to other roles.

4.4. Access Control Delegation Through SPs

In some circumstances, a data provider may want to delegate access rights to his/her streaming data. For example, a doctor may need to consult with a specialist regarding some patient data. He would thus need to make available the portion of the stream to the specialist with the permission of the patient. The patient controls access control delegation, by specifying which roles are permitted to act as delegates, and which objects they can give access to on his or her behalf.

[12] To keep the presentation concise, we have omitted the expressions for the *DDP* and the *SRP* syntax.

[13] We use superscripts ($^G/^{-G}$) to denote roles with/without delegation rights respectively.

```
INSERT {SECURITY PUNCTUATION | SP}
[[AS] sp_name ]
INTO STREAM [stream_name | stream_id] LET
[sp_name.]DDP = <ddp_expr>,
[sp_name.]SRP = <srp_expr>,
[[sp_name.]SIGN = { POSITIVE | NEGATIVE },]
[[sp_name.]IMMUTABLE = { TRUE | FALSE }]
```

Figure 7. Security Punctuation Syntax.

```
INSERT SP AS sp1
INTO STREAM s3
LET
sp1.DDP = TUPLES:[120, 200]
AND ATTRIBUTES:Frequency,
sp1.SRP = frbac:R1 WITH GRANT OPTION,
sp1.SIGN = POSITIVE,
sp1.IMMUTABLE = TRUE
```

(a) Creating a New *sp*

```
SELECT Patient_id,Beats_per_min,Timestamp
FROM S1
WHERE Beats_per_min > 80
ON RESULT(
INSERT SP AS sp_result INTO STREAM
LET
sp_result.DDP = TUPLES:ALL
sp_result.SRP = frbac:Cardiologist,
sp_result.SIGN = POSITIVE,
sp_result.IMMUTABLE = TRUE);
```

(b) Specifying *sp* on Streaming Results

Figure 8. CQL Examples of SPs.

To delegate access rights on stream data, a data producer would specify WITH GRANT OPTION when creating a security punctuation as illustrated in Fig. 8a.

Query specifiers with grant option may generate new security punctuations on the results of their queries. In such a case, a query specifier would append ON RESULT (*sp_expression*[,. . .n]) clause to the query, where *sp_expression* corresponds to the syntax for creating new sp (Fig. 8b).

In the implementation, we designed a *security punctuation generator* (SPGen) operator that is added as the root operator in the query plan. SPGen cleans the data stream by removing existing *sp*s and injects new *sp*s based on the grant authorizations and the ON RESULT clause of the query.

5. Security-Enhanced Query Processing

We now describe our proposed query processing mechanism that is aware and compliant with the security restrictions on the stream data. When a query specifier registers a continuous query, the role(s) of the query specifier are automatically detected by the DSMS and associated with that query. We must consider both the roles associated with the query and the streaming *sp*s in determining which data tuples are allowed "to be seen" by the query. For this purpose, we introduce two novel operators: *Security Shield* (SS) and Security-Compliant Join (SCJoin). Fig. 9 illustrates an example of a continuous query network with SS and SCJoin operators embedded inside the shared plan for three queries, Q_1, Q_2, and Q_3. We discuss the core functionality of the operators in detail next.

5.1. 'Security Shield' (SS) Operator

A "Security Shield" (SS) operator has multiple functionalities: (1) it filters tuples if queries do *not* have access privileges to them, (2) preserves correctness of the security semantics when *sp*s may be "missing" or arrive out of order, and (3) determines if the *sp*s can be discarded early in the query plan (to reduce memory requirements) without violating the security semantics. The SS operators are integrated with the rest of the query plan without affecting the functionalities of other continuous query operators.

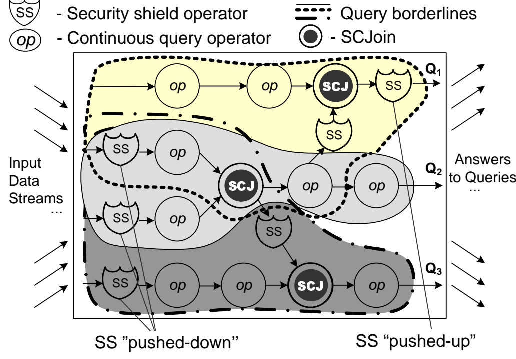


Figure 9. Security-Enhanced Query Network.

SS is a stateful operator, and its state consists of the following four components:

Associated Parent Operator(s) (APO): Set of *all* upstream operators in the query plan for which the SS "pre-filters" data tuples. This set is used to maintain the correspondence between the query operators and their access privileges. If a single SS is shared by several queries, and a query is removed or it changes its access rights, the DSMS determines its operators and updates the security predicates in the SS state elimintating the need for query modification.

Associated Processing Role(s) (APR): Set of security predicates (in the case of RBAC model, set of roles) for which the SS operator allows data tuples to "pass through". For example, tuples with policies where only a role $r \notin APR$ can access, are discarded.

Propagate *sp* Further Flag (PFF): Indicates whether the SS must allow the *sp*s to propagate further up the stream. This flag is used to determine if the *sp*s can be discarded early in the query plan. This feature helps to reduce memory requirements and improve the query execution performance.

Current Policy (CP): A data structure depicting the properties of the current policy for the tuples that are about to arrive. Current policy *P* is in the SS state until the arrival of another policy *P'* with a more recent timestamp.

Fig. 10 presents the SS algorithm. An input to the SS operation may be either a security punctuation *sp* or a tuple *t*.

When a new *sp* arrives, SS first checks if the timestamp of the current policy in the state equals to *0* or *L.CP.ts*, which indicates that the SS needs to either initialize a new policy or add the arrived *sp* to the current policy, respectively. If it is a new policy, the *sp*'s timestamp becomes the timestamp of the new current policy *L.CP.ts* (Line 1). If there is already a policy in the state, and the newly arrived *sp* has a more recent timestamp than the current policy, it becomes the new current policy (Line 3). Otherwise the *sp* is discarded, since it had arrived after the policy with a more recent timestamp.

When a new tuple *t* arrives (Line 4), SS checks if there is a policy in state for it (Line 6). If there are none, *t* is discarded (denial-by-default is enforced). Otherwise SS checks if this is the first tuple that begins the s-punctuated segment (Line 7). If true, we assume that the policy is complete, and from now on the tuples with this policy will be arriving. SS calculates the cumulative sequence number (*L.CP.csn*) based on the *sp*s received (Line 8). The *csn* indicates the largest cumulative sequence number of the *sp*s (with no gaps in their *sn*s). For example, if *csn* = 3, it means that three sequential *sp*s contributing to the same policy have been received. In Line 10, SS checks if the newly calculated *csn* is the same as the new tuple indicates. If they are not equal, some *sp*(s) are missing and it is an "incomplete" policy. Due to the uncertainty about policy, the SS discards both the policy and the tuple *t* (Line 16). If a policy is incomplete, we do not know the full set of restrictions on the tuple(s), and thus execute denial-by-default. If the policy's and the tuple's *csn*s are equal, the policy is complete, and the tuple can be processed further. SS procedes to check if the *t*'s access control policy matches the roles in the state's *L.APR*. If there is no overlap, the tuple is discarded. The interpretation is that the tuple's policy does not allow access to any query comprised of the operators in the *L.APO* set. If there is a match, the tuple is sent to the output stream (Line 14). The tuple's policy will precede the tuple, if *L.PFF* is true and the policy has not yet been sent by the SS operator (Line 13). When the subsequent tuples from the s-punctuated segment arrive, the processing is similar (Lines 17-18).

PROCEDURE SSOperation($sp \mid t$)
//L is the state of SS operator
01. **new security punctuation** sp **arrives**:
02. **if** (*L.CP.ts = 0*) **or** (*L.CP.ts = sp.ts*) **then** *L.CP.Add(sp)*; *L.CP.ts = sp.ts*;
03. **else if** (*L.CP.ts* $<$ *sp.ts*) **then** empty *L.CP*; *L.CP.Add(sp)*; set *L.CP.ts = sp.ts*;
    set *first_tuple_flag = true*;
04. **else if** (*L.CP.ts* $>$ *sp.ts*) **then** discard *sp*;
05. **new tuple** t **arrives**:
06. **if** *L.CP.ts = 0* **then** discard *t*; *//no policy in the state*
07. **else if** (*first_tuple_flag*) *//end of sp-batch*
08.   calculate *L.CP.csn* from all *sp*s that belong to *L.CP*
09.   *first_tuple_flag = false*
10.   **if** *L.CP.ts.csn = t.ts.csn*$_{sp}$
11.   **if** *t.R* $\subseteq$ *L.APR* **then**
12.   **if** *L.PFF = true* **and** *policy_sent* $\neq$ *true* **then**
13.   send *L.CP.sps* to output stream, set *policy_sent = true*
14.   send *t* to output stream
15.   **else** discard *t*
16.   **else** empty *L.CP*, discard *t*, set *L.CP.ts = 0*, set *first_tuple_flag = true*
17. **else** *//just another tuple from s-punctuated segment arrived*
18.   *//similar processing as in Lines 8-14*

Figure 10. SS Algorithm.

When generating a query plan, we try to push SS operators "down" in the query plan to filter out tuples as early as possible to minimize the traffic and improve query execution. However, in some cases it may be advantageous to push the SS operator "up", for example if the operators in the query plan have a low selectivity and queries have broad range of access privileges.

5.2. Correctness of SS

In this section, we provide a proof of correctness of the SS operation. We prove that SS will only process tuples with *complete* policies, otherwise it will enforce denial-by default.

Theorem 5.1. *For any tuple t, SS will process t only if its policy P is complete and in correct order. Otherwise SS will prevent access to t. Policy P will be propogated only once and only if it is complete.*

Proof. Let *SS* be a security shield operator in a query network. Let *L* be its state. Assume that $\exists$ sp_1,sp_2,$sp_3 \in P$ and *L.CP* = *P*, and $t \in T$ where *T* is a set of tuples. We identify the following three cases:

Case 1: policy *P* arrives prior to tuple *t*. In this case, a policy *P* can be a

(a) *P* is complete policy: If $t.ts.csn_{sp}$ = *P.ts.csn*, then the policy *P* is *complete*. The *SS* operator will then process the tuple *t*. If the flag *first_tuple* is set to *true* and $P \subseteq L.APR$, then both *P* and *t* will be propagated into the output stream (*P* preceeding *t*). If the flag *first_tuple* is set to *false* and $P \subseteq L.APR$, only *t* will be propagated; otherwise both *P* and *t* are discarded.

(b) *P* is an incomplete policy: If $t.ts.csn_{sp} \neq P.ts.csn$, the policy *P* is *incomplete*. The SS operator will then automatically discard both *P* and *t* (without processing *t*) to prevent an unauthorized access.

Case 2: tuple *t* arrives prior to its policy *P*. If $t.ts_{sp} > L.CP.ts$, (by definition 4.1) the policy *L.CP* is *not applicable* to *t*. The *SS* operator will then automatically discard both *L.CP* and *t*. *L.CP* is discarded because tuples with newer policies (as indicated by $t.ts_{sp}$) are now arriving, and *t* is discarded because there is no policy for it, and thus denial-by-default is enforced. When the *t*'s policy *P* arrives later to *SS*, it will be discarded by definition of SS, as it has arrived out of order.

Case 3: an unrelated policy *P'* follows immediately policy *P*. $P'.ts > L.CP.ts$. *SS* operator will discard the policy *P* (based on the more recent timestamp of *P'*) and initialize its state with *P'*, thus preserving the correct semantics for the upcoming tuples.

From the above cases, we conclude that only tuples with complete policies that outstanding queries have access privileges to will be output by SS operators. □

5.3. Select and Project Queries and SPs

Now we discuss if, and if so how, operators in stream query plans need to be modified due to the presence of *sp*s in the data streams. Select and project operators need to be made "security-punctuation-aware". These operators are henceforth assumed to pass through the streaming *sp*s. A select operator, while processing an *s-punctuated segment*, maintains a list of *sp*s representing the policy for that s-punctuated segment. If none of the tuples from the segment satisfy the select condition, their corresponding *sp*s are dropped as well.

A join operator, on the other hand, faces several challenges, including (i) joining streaming data with different access control policies and (ii) sharing join processing among queries with different access privileges[14]. We propose a novel *Security-Compliant Join* (SCJoin) algorithm that addresses these challenges.

5.4. Security-Compliant Join (SCJoin)

We present SCJoin as a sliding count-based window join algorithm, while time-based windows can be handled similarly. Equijoins are the most common for stream systems, so we focus on those – however, the proposed algorithm can be extended to other join conditions. The distinguishing characteristics of the SCJoin are: (1) it minimizes individual processing per tuple by sharing execution for the data with the same security policies, (2) it employs adaptive execution *inside* the join operator based on how the policies partition the data, and (3) it can utilize any other existing join algorithm from the literature for probing the tuples based on join predicate to further improve the execution.

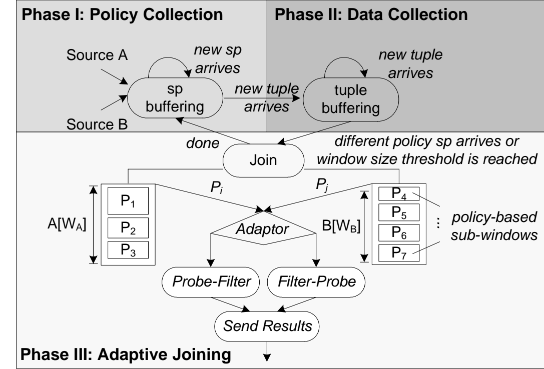


Figure 11. State Diagram of SCJoin.

The SCJoin has three phases: the policy collection, the data collection and the adaptive join phases. Fig. 11 provides the state diagram of SCJoin. SCJoin starts with the policy collection phase where input *sp*s from remote sources A and B are received. Incoming *sp*s from the same data stream are treated as a part of the same policy until a data arrives on that same data stream (which indicates the termination of the policy specification).

Once a data is received, the data collection phase begins. SCJoin takes a *lazy join* strategy and doesn't immediately join a tuple upon its arrival. Rather, it waits for the arrival of all tuples that have the same policy. The motivation behind this approach is that if several tuples share the same policy, and SCJoin can quickly determine if the policies have common restrictions (we denote such policies as *compatible*), we can reduce the number of probes needed to be performed, were these tuples processed individually.

While in the data collection phase, if an *sp* that belongs to a different policy arrives or a window size threshold is reached, SCJoin transfers control to the adaptive join (AJ) phase. In the AJ phase, tuples are already "partitioned" into sub-windows by their corresponding security policies.

When joining tuples with different policies, there are two approaches. First, we could join the tuples, and then for all join results filter out the results whose policies are not compatible. We term this approach - *probe-and-filter* (PF) method. The probe procedure can utilize any existing join algorithm from the literature[15]. Alternatively, we can first filter out the data based on compatibility of the policies and then perform the join on the resulting (after filtering) tuples. This approach is termed *filter-and-probe* (FP) method. SCJoin

[14] We plan to address other operators, such as union and aggregate operators as a part of our future work.

[15] For simplicity of presentation, we chose nested loop join.

```
PROCEDURE SCJoin()
01. while new security punctuations spa arrive from stream A:
02.    add spa to current policy Pa
03. store Pa in state A[WA]
04. while new tuples ai arrive from stream A and !maxWindowCount:
05.   Add tuple ai to sub-window A[Pa] ⊆ A[WA]
06.   Discard expired tuples in window A[WA]
07.   Discard policies if there are no more tuples in A[WA] they are applicable to
08. for each policy Pj ∈ B[WB]:
09.     (Plarger, Wlarger) = (|Pi|/|WA| > |Pj|/|WB| ? (Pi, WA) : (Pj, WB))
10.     if |Plarger| > 0.5*|WLarger| execute Filter-and-Probe(A[Pi], B[Pj])
11.     else if |Plarger| < 0.1*|WLarger| execute Probe-and-Filter(A[Pi], B[Pj])
12.     else execute either
 /* Similar processing is done for stream B*/

PROCEDURE Filter-and-Probe(A[Pi], B[Pj])
13. if Pi ∩ Pj ≠ ∅: //coarse filtering
14.   for each tuple a ∈ A[Pi]:
15.     for each tuple b ∈ B[Pj]:
          //let c denote a join attribute in both data streams
16.       if a.cp ∩ b.cp ≠ ∅: //join attribute policy-based filtering
17.         if ((result = a ⊳⊲ b) ≠ null)
18.           store join result in the buffer
19.   send join results preceded by their policies to the output stream
 /* Probe-and-Filter(...) executes in the reverse order */
```

Figure 12. SCJoin Algorithm.

takes an adaptive approach in selecting a join strategy. The adaptor component in the AJ phase takes two policies, one from each stream, and based on a heuristic(s) determines an execution strategy for the sub-windows of tuples that have these policies. The heuristic we chose was based on cardinalities of sub-windows of tuples with the same policy[16]:

Cardinality Heuristic: *For each* (P_i, P_j) *where* $P_i \in A[W_A]$ *and* $P_j \in B[W_B]$, *let* $(P_{larger}, W_{larger}) = (\frac{|P_i|}{|W_A|} > \frac{|P_j|}{|W_B|}\ ?\ (P_i, W_A) : (P_j, W_B))$.

- *if* $|P_{larger}| > 0.5*|W_{Larger}|$ *execute filter-and-probe*
- *else if* $|P_{larger}| < 0.2*|W_{Larger}|$ *execute probe-and-filter*
- *else, pick randomly either method.*

The heuristic chooses a filter-and-probe method for policies associated with a large set of tuples. The motivation is to try to quickly determine if a sub-window's policy is compatible with the policies from the opposite stream. If they are not, there is no need to probe them further. If the sub-windows are small, then a regular join is performed first, and the tuples that join are filtered further based on compatibility of their policies. Once the join is completed, the SCJoin returns to the policy collection phase.

Algorithm for the SCJoin is shown in Fig. 12. SCJoin processes tuples and security punctuations strictly in the order of their arrival. First a policy is collected (Lines 1-2). Once a new tuple arrives, the policy is stored in the operator state (Line 3) and the data collection phase begins (Lines 4-7). Arriving tuples are added to its stream window under its policy (Line 5) and are used to invalidate expired tuples from the window of its data stream (Step 6). If there are policies that no longer have any tuples in the window they are applicable to, they are discarded (Line 7). After the data collection phase is completed, adaptive join phase begins (Lines 8-12). We present the pseudocode for Filter-and-Probe procedure in Fig. 12[17].

First, Filter-and-Probe() begins with a coarse filtering and intersects the policies from the opposite streams (Line 13). If the result of the intersection is an empty set (∅), the tuples don't have intersecting policies and thus, there is no reason to further process them. Otherwise, the procedure continues further and processes each tuple from the opposite stream's sub-windows checking the policy compatibility on the join attribute (Line 16). If two tuples have policies that intersect on the join attribute, they are joined (Line 17), otherwise ignored. The join results are temporarily stored in the results buffer (Line 18). After all results have been produced, they are sent to the output stream preceded by their policies (Line 19).

6. Experimental Analysis

We have implemented our proposed security framework in a Java-based DSMS named CAPE [15]. We run CAPE on Intel Pentium IV CPU 2.4GHz with 1GB RAM running Windows XP and 1.5.0.06 Java SDK. We use the *Network-based Moving Objects Generator* [7] to generate 110K moving objects sending their location updates in the form of data streams. Given the generated data streams, we have injected *sp*s describing *tuple-granularity* access control policy on the location updates[18].

6.1. Overhead of Security Shield (SS) Operator

Varying Ratio of Security Punctuations: Fig. 13a gives the average processing cost per 1000 tuples of an SS operator with varying ratio of *sp*s versus tuples in the data stream. The ratio of *sp*s in the data stream was varied from 1 (every tuple has a unique policy) to 100 (where 100 tuples share a policy). The state of the SS operator was set to filter about 50% of the tuples. Fig. 13a shows that the overhead of SS operator is very low. Even when every tuple has its own policy, the average processing cost *per 1000 tuples* is about 4 ms[19]. As more tuples have common policies, the SS operator will have less overhead. The reason is that once a policy arrives, the SS quickly determines if the tuples that follow this policy should be *all* dropped or passed through. The more tuples share a policy, the less is the average processing cost per tuple.

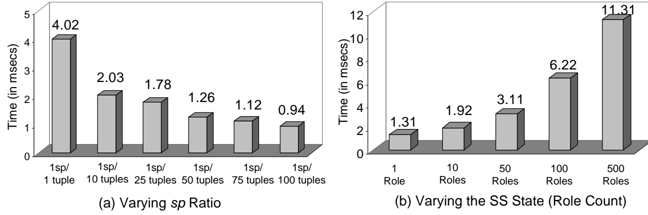
Varying SS State (Role Count): Fig. 13b gives the average overhead of an SS operator with varying *APR* set in the state. The role count was varied from 1 (typical when an SS serves one query) to 500 (where SS may be shared by multiple queries). The state of the SS was set to filter about 50% of the tuples. With the increased role count, the average overhead has increased from 1.31 ms (with 1 role) to 11.31

[16] However, any other heuristic can be used to adapt the execution.

[17] For brevity of presentation, we omit the pseudocode for Probe-and-Filter(...) procedure.

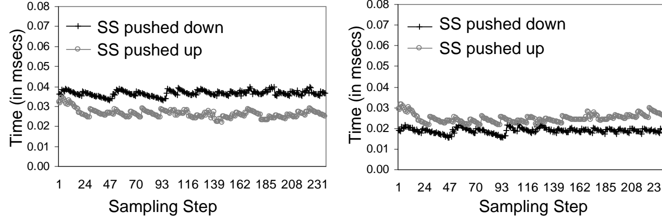
[18] We chose tuple level policy, because it is probably the most common granularity of security in such mobile environments.

[19] Note, this cost does not include the overhead of scheduling the operator, dequeing tuples from the input queue and sending tuples upstream.



(a) Varying *sp* Ratio (b) Varying the SS State (Role Count)

Figure 13. Overhead of SS Per 1,000 Tuples.



(a) Larger SS State (100 roles) (b) Smaller SS State (10 roles)

Figure 14. Average Processing Time.

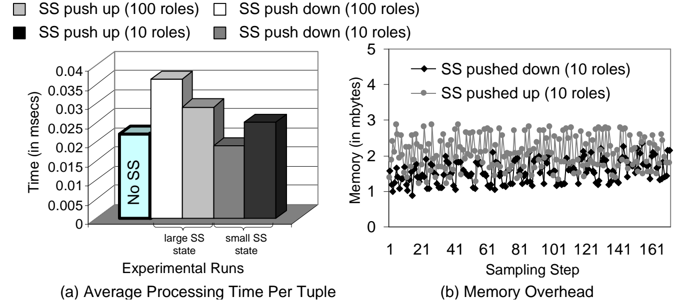
ms (with 500 roles). SS operator uses *short-circuit evaluation*. Whenever the first role that the *sp* matches is found, the evaluation of the *sp* against the rest of the roles is unnecessary. Thus the roles in SS state are scanned until either at least one role that the newly arrived *sp* matches is found, or the end of the *APR* set is reached. Overall, the overhead cost of SS is small. Thus integrating SS operators into the query plans should not pose a significant overhead. It would certainly be preferable over executing the queries first, and then checking on their access privileges.

6.2. SS Operator "Push Up" vs. "Push Down"

We conducted a set of experiments in which we tested the SS "push up" (SS is the root operator) vs. "push down" (SS is the first operator) in a select-project query plan. We evaluated the following query: *Continuously retrieve moving_object_id, speed and location (x-, y- coordinates) for those objects with speed greater than x*, where *x* was set to control the selectivity of the query to be approximately 50%.

In Fig. 14a we show the results regarding the average processing time *per tuple* in the query plan with the SS operator being pushed down versus pushed up. The SS state contains 100 roles. In Fig. 14b we illustrate the results of the same experiment where SS operator state is much smaller (it contains 10 roles). As can be seen from the figures, the larger state size (Fig. 14a) resulted in higher processing time for SS operator being "pushed down". If the selectivity of an operator (e.g., select) is low, and the SS operator has a large state, it is better to precede the SS by the low selectivity operator. This way more tuples would be filtered and only then probed against the SS state. On the opposite, if the selectivity of the query is large, i.e., many result tuples are produced and the SS has a small state, it is better performance-wise to push the SS down in the query plan.

SS Overhead: Figure 15a compares the average processing time with and without SS operator in the query plan. As can be observed, the overhead is low when compared with



(a) Average Processing Time Per Tuple (b) Memory Overhead

Figure 15. SS Operator Overhead.

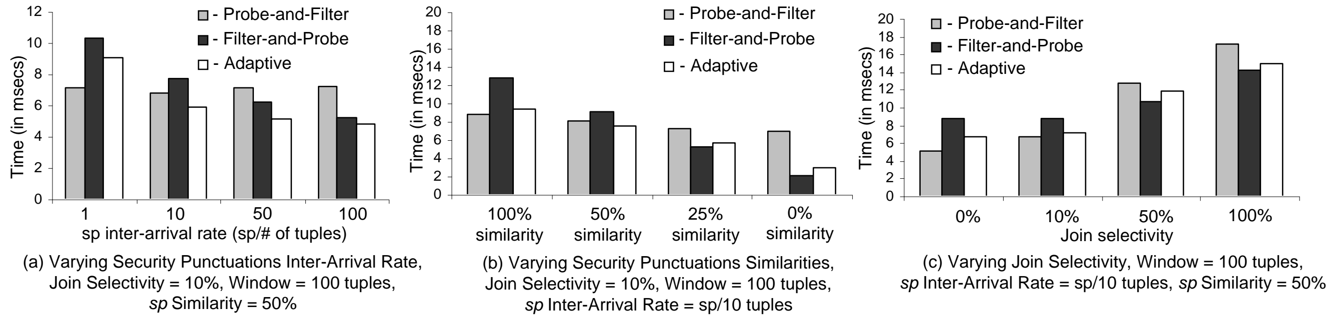
the execution run without any SS. To evaluate the memory overhead of running SS, we measured the memory consumption for executions with both the SS pushed down and up. A comparison of the memory usage for these executions is shown in Fig. 15b[20]. SS, pushed down, filters more data early, hence requiring fewer memory to store the tuples being processed compared to the SS being pushed up.

6.3. "Probe-and-Filter" vs. "Filter-and-Probe"

We performed three sets of experiments in evaluating *probe-and-filter* and *filter-and-probe* vs. *adaptive-PF-or-FP* execution strategies for SCJoin. In the case of the first two, the execution method of SCJoin is fixed, i.e., the operator always uses a probe-and-filter or a filter-and-probe strategy. Specifically, we wanted to measure the effectiveness of the adaptive approach inside the SCJoin. The adaptor used the heuristic introduced in Section 5.4. The window size was set to 100 tuples. We varied the *sp*s' ratio (Fig. 16a), the *sp*s' similarities in the opposite streams (Fig. 16b), and the join selectivity, i.e., the percentage of tuples that join per window (Fig. 16c). The join condition for SCJoin is the destination of the moving objects, the location they are currently moving towards [7].

Varying *sps*' Ratio: Fig. 16a compares the execution strategies when *sp*s' ratio is varied. The average processing time of probe-and-filter is *not* significantly affected by the *sp*s' ratio. This is due to the fact that in probe-and-filter method, join is performed first, and only if tuples produce a join result, their policies are checked to see if the result should be output. We fixed the join selectivity to be relatively small (10%). Thus few results were produced and the high *sp*s ratio did not significantly affect the average processing time. For the filter-and-probe method, on the other hand, the average processing time per tuple varied as *sp*s' ratio changed. Policies had to be frequently checked in order to either filter the tuples with similar access control policies first and then to join them. The adaptive strategy, similar to filter-and-probe, improved its performance when the sp-ratio increased. When fewer tuples shared a common *sp*, it randomly picked between the two strategies, and thus its performance did not suffer as much as the filter-and-probe method.

[20]For better visibility, we illustrate the memory overhead for only the SS with smaller state (10 roles).



(a) Varying Security Punctuations Inter-Arrival Rate, Join Selectivity = 10%, Window = 100 tuples, *sp* Similarity = 50%

(b) Varying Security Punctuations Similarities, Join Selectivity = 10%, Window = 100 tuples, *sp* Inter-Arrival Rate = sp/10 tuples

(c) Varying Join Selectivity, Window = 100 tuples, *sp* Inter-Arrival Rate = sp/10 tuples, *sp* Similarity = 50%

Figure 16. Average Processing Time Per Tuple by SCJoin Operator.

Varying *sp*s' Similarities: For this experiment, we varied the similarity of the *sp*s in the data streams. The similarity percentage of *sp*s in the opposite data streams determined how *compatible* the policies from the opposite data streams are. Fig. 16b illustrates that probe-and-filter on average exhibits a better performance when the similarity of the *sp*s is relatively high, ranging from 100% (the tuples in the two streams have identical policies for all tuples) to 50%. At 25% *sp* similarity, filter-and-probe method gives a better performance, because more tuples are filtered, and then a join is performed on the few resulting tuples. The adaptive strategy showed just as good performance as the probe-and-filter when *sp* similarity was high, and further improved its performance similar to filter-and-probe method when the similarity between the policies has decreased.

Varying Join Selectivity: In this experiment, we varied the join selectivity. We define join selectivity as the $\frac{\#input\ tuples}{\#output\ tuples}$ in a window. As Fig. 16c illustrates, with the join selectivity of 10% or below, the probe-and-filter method gives a better performance, because few results join. With the join selectivity of 50% and higher, the filter-and-probe requires less time in processing, because more tuples are filtered based on policies. The adaptive method gives a better average performance with varying conditions. This makes it favorable for long-running queries, when the conditions may change many times during the execution of a query.

7. Conclusions

We have proposed a novel approach to enforce security and preserve privacy in streaming environments using security punctuations. This work makes three important contributions to the field of data stream management systems: (1) a scheme for defining security semantics on streaming data; (2) a query processing approach compliant with the security restrictions; (3) an implementation and investigation of the security mechanism and its effect on stream system performance. Both our framework for streaming security restrictions (i.e., *sp*s) and query evaluation approach are scalable and flexible enough to be extended further, for example to support interoperability between different access control models depicted by streaming *sp*s. The significance of our experimental results is that we have shown that our security and privacy enforcement mechanism, integrated as a part of query processing itself, adds little overhead in a stream query system.

References

[1] R. Agrawal, P. Bird, and et. al. Extending relational database systems to automatically enforce privacy policies. In *ICDE*, pages 1013–1022, 2005.

[2] R. Agrawal and et. al. An xpath-based preference language for p3p. In *WWW*, pages 629–639, 2003.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, pages 143–154, 2002.

[4] A. Arasu and et. al. Cql: A language for continuous queries over streams and relations. In *DBPL*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.

[5] B. Babcock, S. Babu, and et. al. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[6] E. Bertino and et. al. An ext. authorization model for rdbms. *IEEE Trans. on Knowl. and Data Eng.*, 9(1), 1997.

[7] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.

[8] J. Chen and et. al. Niagaracq: A scalable cont. query system for internet databases. In *SIGMOD*, pages 379–390, 2000.

[9] L. Cranor, M. Langheinrich, and et. al. The platform for privacy preferences (p3p) 1.0 specification. W3C, 2002.

[10] L. Ding, N. Mehta, E. A. Rundensteiner, and et. al. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.

[11] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM*, pages 98–107, 2004.

[12] L. Fegaras, D. Levine, and et. al. Query processing of streamed xml data. In *CIKM*, pages 126–133, 2002.

[13] M. A. Hammad, M. F. Mokbel, and et. al. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.

[14] K. LeFevre, R. Agrawal, and et. al. Limiting disclosure in hippocratic databases. In *VLDB*, pages 447–452, 2004.

[15] E. A. Rundensteiner and et. al. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[16] R. S. Sandhu, E. J. Coyne, and et. al. Role-based access control models. *IEEE Computer*, 29(2), 1996.

[17] P. A. Tucker, D. Maier, and et. al. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Eng. Bull.*, 26(1), 2003.

[18] P. A. Tucker, D. Maier, and et. al. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3), 2003.