A Privacy Preserving Framework for Managing Vehicle Data in Road Pricing Systems

Huayu Wu #, Wee Siong Ng #, Kian-Lee Tan *, Wei Wu #,
Shili Xiang #, Mingqiang Xue #
Institute for Infocomm Research, A*STAR, Singapore
{huwu, wsng, wwu, sxiang, xuem}@i2r.a-star.edu.sg
* School of Computing, National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

The Electronic Road Pricing (ERP) system was implemented by the Land Transport Authority of Singapore to control traffic by road pricing since 1998. To better understand the traffic condition and improve the pricing scheme, the government initiated the next generation ERP (ERP 2) project, which aims to use the Global Navigation Satellite System (GNSS) collecting positional data from vehicles for analysis. However, most drivers fear of being monitored once the government installs the devices in their vehicles to collect GPS data. The existing data stream management systems (DSMS) centralize both data management and privacy control at server site. This framework assumes DSMS server is secure and trustable, and protects providers' data from illegal access by data users. In ERP 2, the DSMS server is maintained by the government, i.e., data user. Thus, the existing framework is not adoptable. We propose a novel framework in which privacy protection is pushed to data provider site. By doing this, the system could be safer and more efficient. Our framework can be used for the situations such as ERP 2, i.e., data providers would like to control their own privacy policies and/or the workload of DSMS server needs to be reduced.

Categories and Subject Descriptors

C.2.4 [**Computer-communication Networks**]: Distributed Systems—*Distributed applications*; H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; K.4.1 [**Computers and Society**]: Public Policy Issues—*privacy*

Keywords

Road pricing; Decentralized framework; Privacy Preservation; Hippocratic Data Stream Systems

1. INTRODUCTION

1.1 Context

The increase of number of vehicles causes many urban problems such as traffic congestion in many cities. To relief congestion in busy roads in Singapore, the Land Transport Authority (LTA) designed and implemented the Electronic Road Pricing (ERP) systems since 1998. The ERP is an electronic toll collection scheme that charges drivers based on the usage of heavily traveled roads. On most roads connecting the city center to suburban areas in Singapore, ERP gantries was set up to detect the vehicles passing through them. Every vehicle in Singapore is equipped with an In-vehicle Unit (IU) device, in which a CashCard with sufficient value must be inserted when the vehicle passes through an ERP gantry in chargeable peak hours. Then the sensor installed on the ERP gantry will communicate with the IU device in the vehicle to deduct the charge amount from the CashCard. Drivers without inserting CashCard or inserting CashCard with insufficient values when they pass through ERP gantries in chargeable hours will be fined.

This ERP scheme to discourage the usage of busy roads has been effectively carried out for more than 10 years to relief road congestion in Singapore. However, two main drawbacks of the current ERP systems make the government re-consider the scheme. First, the existing ERP gantries charge all vehicles (of the same type) the same amount, even though some vehicles only use the road for a short distance. Second, the data collected by the current ERP gantries are not sufficient for further analysis to help urban planning. Consequently, in 2011 the government initiated the next generation ERP (ERP 2) project, in which the Global Navigation Satellite System (GNSS) will be used to replace gantries to track the usage of busy roads by each vehicle. Ideally, the GPS data collected by the LTA through the GNSS can reflect the exact usage of each busy road by each vehicle. This will greatly help the government design more reasonable pricing scheme and analyze behaviors of drivers for further urban planning.

The ERP 2 system is still under development and testing by the government-appointed companies. One obvious challenge for the ERP 2 project is to collect and manage streaming data from around one million vehicles in Singapore. On one hand, the storage and processing of such big amount of streaming data requires implementation, adaption and testing of R&D works in streaming data management and big data analytics; on the other hand, the government needs to consider the reluctance from vehicle owners once their location data are monitored. In the ERP 2 scheme, the location data of a vehicle must be disclosed to the government when the vehicle is on a busy road at chargeable time. However, the driver should be able to control whether or not to disclose his/her data to the government when the vehicle is not on a chargeable road or not at a chargeable time. This requires the data stream management system (DSMS) that manages vehicle data to be privacy preservation enabled.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
KDD'13, August 11–14, 2013, Chicago, Illinois, USA.
Copyright 2013 ACM 978-1-4503-2174-7/13/08 ...$15.00.

1.2 Hippocratic Data Stream System

Agrawal et al. [1] proposed the concept of Hippocratic database to manage data with concern of privacy protection. They discussed ten principles for Hippocratic database systems. Later, Ali et al. [2] extended these principles to data stream systems (DSMS). Among these principles, *limited disclosure* and *limited collection* are at the core of Hippocratic data stream system design[1].

Limited disclosure allows or disallows disclosing streaming data to different user queries, according to certain privacy policies specif ed by data providers. From the system's view, it is actually *access control* based on data provider specif ed policies. *Limited collection* protects providers' data from being excessive collected by the stream systems. It requires the data collected by a stream system to be the minimum amount to serve queries.

Limited disclosure in DSMS has been extensively investigated in research works. Basically, the existing approaches follow the same paradigm as shown in Fig. 1. In this framework, a Policy Manager module is added into a normal DSMS to make it a Hippocratic DSMS. In particular, this module manages the access control policies specif ed by the streaming data providers and/or the system administrator. Also, for each incoming query, the Policy Manager checks with the policies to allow, rewrite or disallow the query. Then the authorized/rewritten queries are issued to the query processor. Some detailed research work and implementations are reviewed in Section 6.

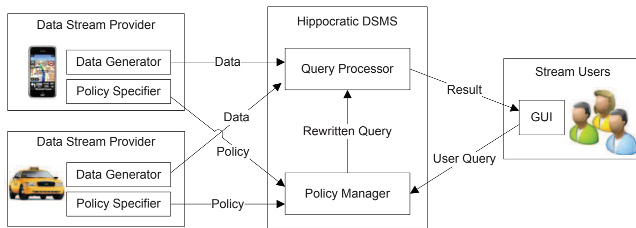


Figure 1: A general design of Hippocratic DSMS

Limited collection did not attract much research interest. In fact in a streaming environment, data are not "collected" (pulled) from data providers by the system. Instead, they are initiatively pushed by the providers. Thus in the current framework, limited collection actually cannot be achieved, as we discuss in detail below.

1.3 Motivation

As illustrated in Fig. 1, in the existing privacy protection framework for Hippocratic DSMS, all privacy enforcement operations take place at the DSMS server. Such a design suffers from the following drawbacks for the applications such as ERP 2.

Privacy Protection. In the current framework, DSMS server is assumed to be secure, trustable and independent to both stream providers and stream users. However, in ERP 2, the government plays both server and user roles. Even if vehicle drivers specify policies to control the access to their data, those policies need to be enforced against all data at the server site. In other words, drivers will only be told by the government that their data are used according to their policies, but will not know whether the government misuse the data as the government actually collects all data.

Performance. Since policy enforcement takes place at the server site, the DSMS server for ERP 2 will not only manage data from a large amount of vehicles, but also need to maintain and enforce different policies on each data stream. This puts extra workload to the server. Moreover, combining policy enforcement and analytical query processing makes many multi-query optimization techniques, e.g., query plan sharing [5], diff cult to be directly applied, because queries may be frequently changed as policies updated.

Energy. Communication with server is energy costly for most sensors and devices. Sending all data to DSMS server for policy enforcement will increase the energy cost in vehicles.

Because of these problems, the existing Hippocratic DSMS framework and its implementations cannot be used for managing vehicle data in ERP 2 systems.

1.4 Contribution

In this paper, we focus on a novel framework design for Hippocratic DSMS, which is visioned in [1, 2]. Our framework is applicable to the ERP 2 project and future projects that requires citizens providing data to help urban planning by the Singapore government. Different from the existing framework that centralizes privacy policy management and enforcement at the DSMS server, our framework leverages the resource at the data provider site for privacy control. In particular, the DSMS server remains the functionality of continuous and analytical query optimization and processing and applies a part of access control to meet system-level requirement, while data providers will f lter their data tuples based on their own privacy requirement and sends only the useful and qualif ed tuples to the server. In this framework, each user (e.g., vehicle) is free to control the disclosure of their data, meanwhile, he/she also takes the responsibility for not disclosing data when necessary (e.g., on a busy road at peak hours).

To tackle the most challenging problem in our framework, i.e., synchronizing server and client for policy enforcement result, we design a bit-vector wrapping and routing technique. Bit-vector is lightweight and eff cient in passing between data provider and server. Furthermore it is very eff cient to check and update so that policy enforcement result can be easily interpreted at a server site.

We demonstrate that our framework perfectly implements the limited disclosure and limited collection principles, and is potentially safer and more time and energy eff cient in query processing than the existing framework. We also conduct comprehensive experiments on our prototype to validate our framework in privacy protection and performance.

1.5 Organization

The rest of the paper is organized as follows. We describe our policy model and def ne the privacy considered in Section 2. Our novel framework is introduced in Section 3, and the analysis of our framework is discussed in Section 4. The experimental evaluation is presented in Section 5. We revisit related work in Section 6. Finally we conclude this paper in Section 7.

2. POLICY MODEL AND PRIVACY DEFINITION

2.1 Policy Model

Our framework adapts the policy model used in Role-based Access Control (RBAC). We follow the same idea to classify all potential users into different roles, each of which may have a different privilege to access streams.

DEFINITION 2.1. *A privacy policy used in our framework is in form of quintuplet:*

<Role, Attr, Purpose, SysCond, ContCond>

where Role *specif es a role name,* Attr *is a list of attributes of the local stream and* Purpose *is the query purpose. These three f elds are compulsory for a policy.* SysCond *is the system condition of a*

[1]The details of other principles can be found in Ali's paper.

policy that involves the constraints on system parameters such as time, number of tuples, etc., and ContCond *is the content condition involving constraints on the stream attributes. These two fields are optional. The semantics of a policy in this form is that* Role *is allowed to access* Attr *of the stream under* Purpose*, with conditions of* SysCond *and* ContCond*.*

We further constrain that given a user role and a query purpose, for each attribute there is at most one policy qualifying the access to this attribute. If we want to qualify a role and a purpose to access an attribute under different system/content conditions, we can express the conditions in conjunction or disjunction form in one policy.

For example, a vehicle driver can specify his/her policy as

<LTA, All, ERP, peak hours, in busy road>

meaning that the driver only allows LTA users to access his data with ERP pricing purpose when the time is in peak hours and the vehicle is in busy road. For other time and vehicle location, LTA will not see his/her data. Drivers can also specify access policies for other government agencies or application users who need to use his/her data.

The compulsory fields of privacy policies are used for stream-level and attribute-level control, and the optional fields are used for tuple-level control. We separate *SysCond* and *ContCond* for performance concern. For the above policy, in peak hours (i.e., *SysCond* is satisfied), each vehicle data tuple will be checked locally and only those satisfying *ContCond* (i.e., location is in busy road) are sent to the server. However, in off-peak hours (i.e., when *SysCond* is not satisfied), no data will be sent, which means the *ContCond* will not be evaluated at all.

2.2 Policy Definition

Based on our policy model, we define the privacy we considered, i.e., limited disclosure (LD) and limited collection (LC). We consider the granularity of stream-level, attribute-level and tuple-level for both LD and LC.

Stream-level LD states that a stream s can be disclosed to a query q which is trying to access s if and only if there exists a policy p such that q's issuer, query attributes, and purpose satisfy the role, the attributes, and the purpose of p and p's system condition is evaluated as true (i.e., p is active under the current system settings). Attribute-level LD is similar to Stream-level LD, except that in attribute-level LD particular attributes will be verified, rather than a stream. Tuple-level LD aims to control the access to each tuple of a stream. As a result, in tuple-level LD we not only require the stream/attribute-level visibility, but also need to verify the content condition in the relevant policy, to qualify a tuple. Tuple-level LD states that a tuple t of a stream s can be disclosed to a query q if and only if there exists a policy p such that q satisfies the stream/attribute-level requirement by p and p's system condition and content condition (on t) are both evaluated as true.

Stream-level LC restricts which streams should be collected by the system and which should not. A stream being collected does not mean all the stream tuples are collected. Instead, it mean the data from this stream will be collected, and thus this stream must be considered for query processing. Whether all the tuples or some tuples are collected is controlled by tuple-level LC. For a stream, if there is no query issued on it, or there is no tuple or attribute from it will be collected, we consider that this stream is not collected. Attribute-level LC states that an attribute a of a stream s should be collected if and only if there exists a query q asking for a and a can be disclosed to q. Similarly, tuple-level LC states that a tuple t of a stream s should be collected if and only if there exists a query q issued to s, and t can be disclosed to q.

THEOREM 2.1. *The existing framework with privacy policies enforced at the server site cannot satisfy the attribute-level limited collection principle.*

THEOREM 2.2. *The existing framework with privacy policies enforced at the server site cannot satisfy the tuple-level limited collection principle.*

In the existing framework, as long as an attribute a appears in the content condition of a policy that qualifies a query q, a has to be collected by the server for policy enforcement purpose, no matter whether a is required by any query. Similarly, since policy enforcement happens at the server site, all tuples will be sent to the server for policy checking, before being decided whether should to be disclosed to user queries.

3. OUR FRAMEWORK

3.1 Overview

Our proposed framework for privacy enforcement to achieve LD and LC in Hippocratic DSMS is depicted in Fig. 2. We can see that in our framework one important highlight is to let the data provider control his/her policy. A data provider will evaluate his/her privacy policies against every data tuple, and only send a tuple to the server when it is asked by some queries whose issuers are qualified to view this tuple. On the other hand, the server is not aware of any provider's privacy policies, and thus, it does not need to spend computational recourse in enforcing such privacy policies. We will describe the framework in detail in the following sections.

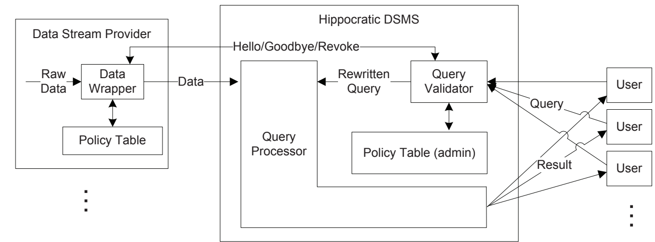


Figure 2: Our proposed framework

To shift privacy policy enforcement to data provider site, we need to resolve a major challenge of synchronization between data provider and server.

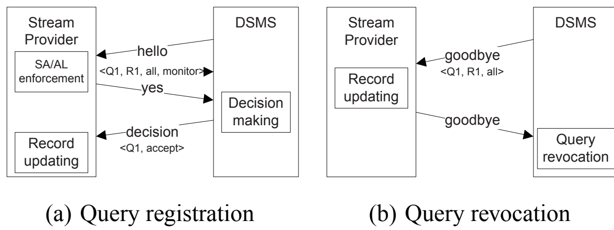
- A data provider needs to keep updated about the queries and the corresponding users who are currently requesting their tuples, in order to enforce relevant policies.
- A DSMS server needs to know the result of the policy enforcement at provider site for each incoming tuple, so that it can determine which queries requesting this tuple are qualified to receive it.

We resolve the synchronization problem by designing the Query Update Protocol for provider-server communication and introducing bit-vector header to pass tuple-level policy enforcement result.

3.2 Query Update Protocol

The purpose of the Query Update Protocol (QUP for short) is to make every stream provider aware of up-to-date queries and roles who are using his/her streams. In privacy view, with QUP a data provider is able to (1) check whether a tuple is needed by the server, to meet the limited collection principle, and (2) know what queries are qualified for accessing a tuple, to meet the limited disclosure principle.

The Query Update Protocol comprises two parts, handling query registration and query revocation respectively. The protocol is illustrated in Fig. 3.



(a) Query registration (b) Query revocation

Figure 3: Query Update Protocol

When a new query is registered into the DSMS system, the Query Validator (refer to Fig. 2) identifies the streams involved in the query. Then it sends a *hello* message to every relevant stream provider. A hello message is a quartet $<$*QID, Role, Attr, Purpose*$>$, where *QID* is the identifier of the query, *Role* is the role of the query issuer, *Attr* is a set of stream attributes to be accessed by the query, and *Purpose* is the query purpose. After that, the Query Validator waits for the response, i.e., a yes/no message from each stream provider, to see whether the query satisfies the stream-level and attribute-level privacy policy in each stream provider. Only if the query passes the policies in all relevant stream providers, the DSMS server will proceed with query processing. Otherwise, it rejects the query. Then in the last step, the server will send a decision message $<$*QID, (accept/reject)*$>$ to the streams to inform them whether the query QID is successfully registered with the server. Once the query is accepted, each relevant stream provider will update its record, as discussed later.

When an existing query is revoked, the Query Validator sends a *goodbye* message to each relevant stream provider, with $<$*QID, Role, Attr*$>$. Then each stream provider will update its record, and acknowledge the server.

The responses from stream providers are compulsory for query registration and revocation in QUP. Thus the timeout and re-sending scheme will be applied for both hello and goodbye messages. The detailed handlers for the hello message and goodbye message at the stream provider site will be discussed in Section 3.5.

3.3 Bit-vector header

As illustrated in QUP in Fig. 3, stream-level and attribute-level policy enforcement is on one-time basis. However, tuple-level policy enforcement is more complicated. Consider the example in Section 1, i.e., a taxi driver does not want his/her data to be viewed by anyone when he/she is not working. In this example, the driver can simply decide whether or not to send a tuple to the DSMS server based on the speed value. However, if the privacy policy is changed such that the driver only restricts Role 2 (e.g., users of taxi booking service) users from seeing his/her anchored record and allows Role 1 users (e.g., the traffic authority) to see all his/her data, there will be a problem. Since both Role 1 queries and Role 2 queries exist, the driver has to send all data to the server. However, the server is not aware of any data provider's policy, thus it cannot decide whether all queries on this taxi stream should get a result.

In fact, the stream provider must send a tuple to the server, as long as there is a query requesting this tuple. Meanwhile, the provider also needs to specify the details about what queries should not access this tuple. Only by doing this, the provider's privacy policies are indeed enforced.

To facilitate tuple-level policy enforcement in our framework, we design a bit-vector header to wrap each data tuple to be sent to the server. In such a header, each bit corresponds to a role. In the header of a tuple, a bit is set to 1 if the corresponding role with queries is qualified to view this tuple. Otherwise, the corresponding bit is set to 0.

The bit-vector is attached to a tuple as an additional attribute. In most practical RBAC schemes, the number of roles is quite limited, compared to the potentially large number of users. Thus the length of a bit-vector header is short. Also the bit-vector itself is lightweight. Furthermore, in many stream systems for complex data with multiple attributes, e.g., [6], each tuple is transformed as an object in the input buffer, and the size for each object is preserved. As a result, in most of such DSMS, our lightweight header brings in little additional transmission overhead. The details of how bit-vector is used are demonstrated later.

3.4 Design of Hippocratic DSMS

In this section, we discuss in detail how a Hippocratic DSMS under our privacy enforcement framework is designed. There are two main components in the DSMS server, Query Validator and Query Processor, as shown in Fig. 2.

The Query Validator is used to validate a new query. The validation process has two parts. In the first part, as described in Section 3.2, the Query Validator sends the query information to all relevant stream providers, to see whether the query satisfies the stream-level and attribute-level privacy policy of every stream it is trying to access. While waiting for the response from stream providers, the Query Validator proceeds with the second part of validation. In this part, it checks with the policy table in the DSMS server. Especially to be mentioned is the difference between policies in DSMS server and policies in stream provider sites. The policies stored in each stream provider site are privacy preserving policies. They are specified by the corresponding stream provider. However, the policies in the DSMS server are system-level access control policies. They are specified by the system administrator to control the access to the system resources. Our design lets each stream provider control and enforce his/her privacy policies, while the access control policies can only be enforced by the system. If the new query satisfies both privacy policies and system policies, it will be registered into the Query Processor. Otherwise, it will be rejected. If the system policies managed by the DSMS server allow limited access (tuple-level control) to some streams, the Query Validator will rewrite the query by appending the additional constraints, and then register it to the Query Processor. This step is the same as tuple-level access control enforcement in other framework, e.g., [4].

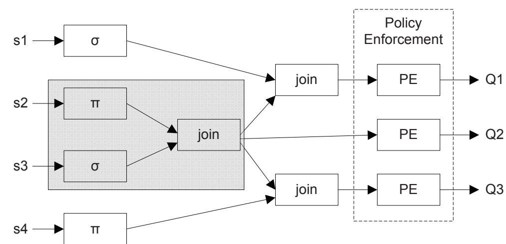


Figure 4: Post-filtering for policy enforcement

Any continuous query processor is adaptable as the Query Processor in our framework. The highlight is that in our framework, queries (or query plans) are not rewritten based on frequently updated privacy policies. Thus, once a query is registered into the Query Processor, it will not be updated until it is revoked. Under this situation, the multiple queries can be optimized based on common operators. In many existing centralized frameworks, using query rewriting to enforce tuple-level policies makes multiple query optimization difficult to be applied [4], because rewritten queries are frequently updated as policies being updated. The only

way to share common operators in the existing framework is to separate policy enforcement from query operators. To do this, the first approach is to treat policy enforcement as post-filtering, as illustrated in Fig. 4 where the gray area is the shared operators of the three queries. In this approach, the normal query processor is used, and before a tuple is returned to a particular query, policies will be checked to validate the tuple. The problem of this approach is that many useless intermediate results that satisfy the query but not qualified by the policies will be produced. In the second approach, policy enforcement is conducted during tuple routing in the query plan [10]. The major drawback of this approach is repeated policy enforcement. For the same example in Fig. 4, since the streams s2 and s3 are commonly used by the three queries, at the very beginning, policies for the three queries will be enforced, such that as long as a tuple from s2 or s3 satisfies one of the queries, it will be sent to the query plan. Later, after the shared operators are executed, the satisfied tuples will be routed to three directions. For each direction, the privacy policies will be executed again w.r.t. the corresponding query. The overhead of policy enforcement includes the policy table scan, tuple checking and filtering. To enforce the same policy several times may seriously affect the performance.

In our framework, there is no policy enforcement in the query processor. All privacy policies are enforced at the stream provider site, and the result in bit-vector will be sent with tuples and verified by the Query Processor during tuple routing. The pseudo code of tuple routing in query plan is presented in Algorithm 1.

Algorithm 1 Query processing

Input: a DAG query plan G, in which each directed edge from an operator u to an operator v is marked with a bit-vector $v.bv$ to indicate the roles of queries requiring v; an input tuple t with bit-vector header $t.bv$
Output: a set of queries taking t as a result
1: initiate the result set S
2: u := the first operator t encounters in G
3: *routing(u, t, S)*
4: return S

Procedure 2 routing(operator u, tuple t, result S)

1: let *bits* := *t.bv*
2: **for all** tuple *t'* such that *t'* is also an input of u and will be executed with t by u **do**
3: *bits* := AND(*bits*, *t'.bv*)
4: t := execute(u, t)
5: **if** *bits* is not all-0 bit vector && t != null **then**
6: **if** u is the last operator for query Q **then**
7: $S := S \cup \{Q\}$
8: **else**
9: **for all** operator v such that *(u, v)* is a directed edge in G **do**
10: let *check_bv* := AND(*t.bv*, *v.bv*)
11: **if** *check_bv* is not all-0 bit vector **then**
12: *route(v, t, S)*

In Procedure 2, at the beginning, a new bit vector is produced by *anding* the headers of all the input tuples to the operator node u during one execution. This is only applicable to the operators with multiple inputs, e.g., join. In line 4, the operator u executes the tuple t, and produces a new tuple to replace t. Then the new tuple will be routed to the next level. Line 6-7 means the current operator u is the last operator of a query Q. In this case, Q will be one of satisfied queries, if the tuple is not null (line 5). If u is not the last operator, t will be routed to all next-level operators which are used by queries satisfying the roles specified in t's header (line 10-11). The generation of bit-vectors will be discussed later.

EXAMPLE 3.1. *Fig. 5 illustrates the process of tuple routing with bit-vector header. Suppose three queries, Q1, Q2 and Q3 are issued by users from different roles R1, R2 and R3. Tuples t1 and t2 are sent by the providers of streams s2 and s3. Assume the stream providers have enforced the privacy policies, and the header of t1 is 110, i.e., only Role 1 and Role 2 can access t1 and the header of t2 is 101, i.e., only Role 1 and Role 3 can access t2. After selection and projection, the tuple headers will not change. When t1 and t2 are joined and become one tuple t4, the header of t4 becomes the AND of the headers of t1 and t2, i.e., 100. It specifies that only Role 1 can access both of the tuples, and thus the join result t4. In the next round of routing, t4 is only sent to the operators related to Role 1 queries. Q2 and Q3 will not receive result from t1 and t2, though these tuples satisfy their query constraints.*

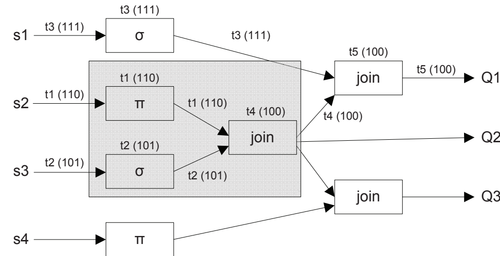


Figure 5: Query Processor in our framework

Compared with the approach in the existing Hippocratic DSMS framework, for each tuple routing, we do not need to enforce any policy. Instead, we only need to check a bit-vector, to know whether the tuple should be forwarded to a next-level operator.

3.5 Design of stream provider site

In our framework, privacy policy enforcement happens at the stream data provider site. Thus the data provider site needs to maintain a local policy manager, which is in charge of adding, deleting and updating its own privacy policies. Besides policy management, another main component at the data provider site is the Data Wrapper. The Data Wrapper enforces privacy policies and then wraps the data tuples with policy enforcement result, in bit-vectors.

3.5.1 Stream-level/attribute-level enforcement

Each provider site maintains a role-query list for every role of user, which is used to enforce stream-level and attribute-level privacy policies. For a role R, if the corresponding role-query list is empty, it means no query from R is qualified to access this stream. Then the corresponding bit value for R is 0 for every tuple. Furthermore, if all role-query lists are empty, the stream will not send any tuple to the server.

For limiting server collection, stream provider site also maintains an attribute-query list for each attribute. For any attribute, if its attribute-query list is empty, which means there is no query requesting this attribute, then the value of this attribute is hidden in every tuple sent to the server.

Recall the Query Update Protocol (QUP). When a new query is registered into the DSMS system, the server will send a hello message <*QID, Role, Attr, Purpose*> to the relevant stream providers. Once a stream provider receives such a message, it will verify with the local policies whether *Role* is qualified to access *Attr* under *Purpose*. In our policy model, as long as there is a policy specifying *Role*, *Attr* and *Purpose* (possibly with other conditions), the query *QID* is said to be qualified. If the query is not permitted by policies,

the provider will send a *no* message to the server. Otherwise, the stream provider will send a *yes* message to the server. After that, the provider will wait for the decision message from the server, to see whether the query *QID* is accepted by the server. If the query *QID* is accepted, i.e., satisfies the policies in all relevant streams, every relevant stream provider will put the *QID* to the role-query list for *Role* and to every attribute-query list for the attributes in *Attr*. If any affected attribute-query list was empty before, that attribute will not be hidden any more.

If an existing query is revoked, the DSMS server will send a goodbye message <*QID, Role, Attr*> to the provider. In this case, the provider will remove the corresponding *QID* from the role-query list of *Role* and attribute-query list of attributes in *Attr*. Meanwhile, an empty checking operation is activated, to check whether the affected role-query list and attribute-query lists are empty or not. If the role-query list becomes empty, the corresponding bit value in the following tuples is set to 0. If any attribute-query list becomes empty, the corresponding attribute will be hidden in future tuples. Then the provider acknowledges the server. Algorithm 3 illustrates the whole process.

Algorithm 3 Stream/attribute-level policy enforcement

Input: a stream *S*, a hello/goodbye message *m*, a local policy table *T*, the role-query lists for all roles *RL*, and the attribute-query lists *AL* for all attributes of S
Output: a response message with updated RL and AL
1: **if** *m* is a hello message with $< Q_i, R_j, Attr, Purpose >$ **then**
2:   **if** there exist a policy in T satisfying $R_j, Attr, Purpose$ **then**
3:     insert Q_i into $RL.R_j$
4:     **for all** attribute A_k in *Attr* **do**
5:       insert Q_i into $AL.A_k$
6:     send a *yes* message to the server
7:   **else**
8:     send a *no* message to the server
9: **else if** *m* is a goodbye message with $< Q_i, R_j, Attr >$ **then**
10:   delete Q_i from $RL.R_j$
11:   **for all** attribute A_k in *Attr* **do**
12:     delete Q_i from $AL.A_k$
13:   send *goodbye* message to the server

A stream provider also records the hello messages of qualified queries, which is useful for policy updating as described later.

EXAMPLE 3.2. *Suppose there is a query Q1 from LTA querying a vehicle data stream for ERP pricing purpose. Q1 requests the location of the vehicle and also the vehicle ID and type. Another query Q2 from URA (Urban Redevelopment Authority) querying the same stream but only need the location information. Suppose the two queries are both qualified to receive data from the stream. When the server sends hello messages, the vehicle (i.e., data provider) stores Q1 and Q2 in the role-query lists for LTA and URA respectively. Also, the vehicle stores Q1 and Q2 in the attribute-query list for location, and stores only Q1 in the attribute-query lists for ID and type. The vehicle sends all tuples to the server by keeping the location, ID and type values and hiding other attribute values. When Q1 is revoked, a goodbye message is sent from the server. The vehicle removes Q1 from LTA's role-query list and the attribute-query lists for location, ID and type. Now the attribute-query lists for ID and type are empty. Then for all following tuples sent to the server, ID and type values are also hidden. When Q2 is revoked, all lists are empty, so no tuples will be sent to the server.*

3.5.2 Tuple-level enforcement

Recall that in our policy model, each policy is specified by the role of user, the permitted attributes, the query purpose and optional conditions. The first three components are used for stream-level and attribute-level privacy control, while the last component is used for tuple-level privacy control.

If a query is qualified and there is no condition specified in the policy qualifying this query, then all tuples should be sent to the server to evaluate this query. In this case, there is no tuple-level enforcement. On the other hand, the stream provider will enforce tuple-level privacy control.

The operation to enforce tuple-level policies in our framework is quite similar to that in the existing framework, i.e., for each tuple, check whether the policy conditions are satisfied. If yes, the tuple will be qualified; if no, the tuple will not be shown to the user. The only difference in our framework is that the tuple-level enforcement happens at the provider site, rather than the DSMS server site. Then in the provider site, if a tuple satisfies the tuple-level condition for a role, the corresponding bit value in the tuple's header is set to 1 before the tuple is sent to the server. Otherwise, the bit value for this role is set to 0. If all bit values in a tuple's header are 0, the tuple will not be sent. The pseudo-code for tuple-level policy enforcement and tuple wrapping is shown in Algorithm 4.

Algorithm 4 Tuple-level policy enforcement

Input: a generated data tuple *t*, the role-query lists for all roles *RL*, and the attribute-query lists for all attributes *AL*
Output: a wrapped tuple *t'* for *t*, which is sent to the server
1: initiate a bit-vector *bv*
2: append *bv* to *t*, to make it *t'*
3: **for all** role R_i **do**
4:   **if** $RL.R_i$!= null **then**
5:     *bv*.set(*i*)
6: **if** *bv* is not all-0 **then**
7:   **for all** attribute A_j **do**
8:     **if** $AL.A_j$ == null **then**
9:       hide the value of A_j in *t'*
10: send *t'* to the server

EXAMPLE 3.3. *Continue with the example queries in Example 3.2. Suppose LTA and URA correspond to roles R1 and R2. URA can view all the data, but LTA is only permitted to view data when the vehicle is on a busy road and at a peak hour. When the vehicle sends a tuple in a busy road at a peak hour, the header of the tuple is 11 (assuming there are only two roles). Later, when the vehicle is not on the chargeable road, by enforcing the local policy, the vehicle will send data with header of 01. By checking the header, the server knows that the first tuple can be routed to both Q1 and Q2, while the second tuple should only be routed to Q2.*

3.5.3 Update of policy

We discuss the actions taken when privacy policies are updated at provider site. If a new privacy policy is added, there will be no change for the existing status, because in our policy model, the addition of new policy will not affect the existing policies that qualify streams and tuples. For future queries the new policy will be in use.

The deletion of an existing policy is more complicated. Once a policy is removed, the provider site will check the record of all hello messages for the qualified queries, which are sent from the server, to see which queries are qualified by the policy to be deleted. Then the policy is removed from the policy table, the relevant hello messages are deleted, and the relevant queries in the query lists are removed as well. Again, if there is no more queries in the query list

for a certain role, the bit value for that role in all further tuples is 0. Furthermore, since those relevant queries are no longer qualif ed, the stream provider will send a revoke message to the server, with all these queries. The server then revokes those queries.

4. ANALYSIS

4.1 Limited collection

THEOREM 4.1. *For a data stream, all its attribute values to be sent to the server will be used to process certain queries.*

Based on the QUP protocol, when a query registers to the server, the corresponding stream provider stores the query ID in the attribute-query lists, for all relevant attributes. Before the provider sends a tuple to the server, it will check with the attribute-query lists. All attributes with empty list will be hidden. Then the attributes with values must be the ones requested by some qualif ed queries. Thus Theorem 4.1 holds.

THEOREM 4.2. *For a data stream, all the tuples sent to the server will be used to process certain queries.*

Recall that a stream tuple is sent to the server if and only if there exist some 1s in its header, which means there are queries qualif ed to evaluate the tuple. If there is no query qualif ed for the stream, or there are queries qualif ed for the stream but none of them is satisf ed by the *SysCond* or *ContCond* of the tuple, the tuple will not be sent to the server. Thus Theorem 4.2 holds.

4.2 Limited Disclosure

THEOREM 4.3. *For a data stream provider, each of his/her tuples will only be received by the qualif ed user queries.*

For each tuple, the provider will f gure out the users that can access it, and attach the information as the header of the tuple. Tuple routing in the server Query Processor strictly follows the header of each tuple. Thus, each tuple only arrives at the qualif ed queries.

Since in our framework, the enforcement of LD is totally at the stream provider site. The server is only aware of the enforcement result. Event if the server is compromised, the attacker cannot view or change the privacy policies specif ed by each stream provider.

4.3 Performance and Energy Consumption

In our framework, the query plan in the Query Processor is not modif ed, compared to the normal (without privacy protection) stream systems. Thus the complexity of continuous query processing is unchanged. In our framework, the tuples that do not satisfy any user roles will not be sent to the server. Then the server may process fewer tuples than the server in the existing framework, and the transmission energy cost is reduced. Furthermore, even for the tuples sent to the server, our framework only checks the bit-vector, which is more eff cient than policy enforcement over those tuples.

5. EXPERIMENTS

5.1 Experimental Settings

All algorithms in our experiments were implemented in Java. We used a machine with 3.07GHz quad-core CPU (Intel Core i7) and 8G RAM. We obtained a real-life taxi data set from a local taxi company. The data set contains the status tuples streaming from all taxis under the company. Each tuple contains the reporting time, the taxi ID, the current location (in longitude-latitude pair), the current speed and the availability.

All the continuous queries were semi-randomly designed. For each testing purpose, we may control/vary the selectivity and the complexity of queries. Within each selectivity and complexity scale, the detailed query predicate (selection and join condition) were randomly designed. We focus on two core query operators, selection and join. For another core operator, projection, it does not impact the performance of tuple-level policy enforcement. Thus we do not test it.

5.2 Implementation

We implement our prototype with multi-threading, which fully utilizes the multi-core CPU resource in the test machine. In particular, in a query plan (with shared part and non-shared part), each operator is ran by a separate thread. There are piped input/output streams between each pair of adjacent operators.

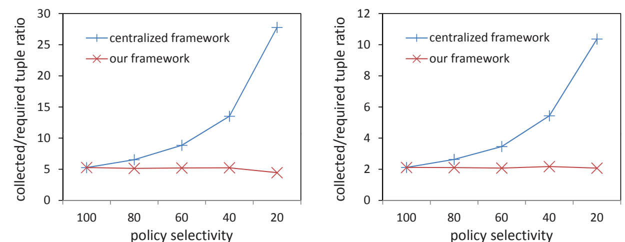
With a preliminary experimental study, our multi-threading based prototype much better utilizes the quad-core CPU resource, and 2-4 times faster than the single-threading prototype. This paper does not focus on the implementation and performance of continuous query processing with multi-threading approach, though it can be a promising direction for our future research. We only focus on examining the effectiveness and eff ciency of our decentralized policy enforcement framework. Thus we implement both our framework and the compared framework under the multi-threading architecture, and do not compare with other single-threading systems for query processing issues.

5.3 Privacy Preservation

Limited disclosure can hardly be tested in experiment. Normally, unless the system is compromised, all limited disclosure policies will be guaranteed in any Hippocratic DSMS. In this section, we only focus on limited collection test.

The intuition behind limited collection is that a Hippocratic DSMS should only collect as few as possible streaming tuples that are suf f cient to answer all queries. Thus in our test, we take the metric of the ratio of the number of collected tuples over the number of resulting tuples to measure the effectiveness of limited collection control. The higher this ratio is, the more useless tuples collected.

We f rst test the selection operation for both centralized framework and our proposed decentralized framework. We use 1000 tuples reported from working taxis, with speed value uniformly distributed. We set the selectivity of the selection to be 20% . The evaluation result is reported in Fig. 6(a). We then vary the selectivity to be 50% and 80%, and get the results with similar shapes to Fig. 6(a) with different scales.



(a) Selection with σ=20% (b) Join with win-size=50

Figure 6: Privacy preservation test

The result proves that for the centralized framework, no matter the selection predicate leads a high selectivity or low, the ratio between collected tuples and required tuples always increases rapidly as the policy selectivity increases. That means if the selectivity of a privacy policy is high, the system will actually collect a large number of tuples that are not returned to the users. Obviously data providers will feel uncomfortable to disclose too many query-irrelevant data to the system. In our framework, since the policies are enforced at the provider site, only satisf ed tuples are sent to

the server. Then for a particular selection operation, this ratio is constant[2] to the policy selectivity.

We also test the join operator. We adopt two taxi streams and perform an equijoin between them. We vary the window size for join among 50, 100 and 200 (number of tuples). We show the experiment result for the window size of 50 in Fig. 6(b). Again, when the window size is varied, the figure shape does not change. We can see the result for join is similar to the result for selection. The same reason explains the result.

Note that the figures above only show the result for a single role. If there are multiple roles, each of which has its own policy with selectivity of σ_i, in average case that the policies are independent to each other, the overall selectivity is

$$1-\prod_{i \in Roles}(1-\sigma_i)$$

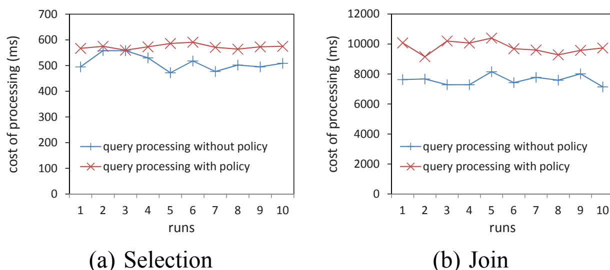
As the number of roles increases, the overall selectivity is close to 1. Moreover, in worst case, if two policies are predicated on the same attribute with contradictory conditions, e.g., one is location within a region R and the other one is location outside R, the overall selectivity will be 1 (i.e., all tuples need to stream into the system). For these cases, actually our framework also has to send nearly all tuples to the server. However, the experimental result in this section shows that our framework will minimize the number of tuples required by the server.

5.4 Performance evaluation

5.4.1 Overhead of policy enforcement

In this section, we test the overhead introduced by policy enforcement in our proposed framework. Note that in our framework, all privacy policies are enforced at the data provider sites and the enforcement result will be passed to the query processor by bit-vector. Furthermore, we do not introduce any new operator to the query plan generated by the query processor. Thus, the overhead is only caused by the bit-vector checking.

We evaluate the overhead brought by the bit-vector checking for selection and join separately. In the first test for selection, we make use of 100,000 taxi tuples as an input stream[3] without delay between adjacent tuples. We set a dummy predicate for the selection operator which filters nothing. The reason is that if a tuple is not selected, it will be dropped immediately and its bit vector will not be checked. To measure the overhead of policy enforcement, we can only monitor those selected tuples. We made 10 runs of experiments, to reduce occasionality. The result is shown in Fig. 8(a). We can see that the overhead of policy enforcement in our framework is around 10% of the query processing time. More exactly, the average policy enforcement time is 6.2e-4 millisecond per input tuple. This result is acceptable compared to the policy checking overhead introduced by other framework [4].



(a) Selection (b) Join

Figure 8: Overhead on policy enforcement

[2] The slight difference is because the tuples used are different in every run.

[3] The processing time for a single tuple is too fast to measure.

We further test the join operation, as shown in Fig. 8(b). We fix the join window to be 50 tuples, and the statistics shows that each incoming tuple in one stream produces 2.5 joined tuple on average as output. From the figure we can see that the join operator is much more costly than selection. The overhead introduced is 8.4e-3 millisecond per input tuple, which is higher than the overhead for selection. This is because every input tuple actually corresponds multiple policy checkings.

5.4.2 Scalability of overhead

In this part, we test how the overhead increases as the number of roles increases. We use the same data, query and policy settings as that in the previous experiments, and we take the pure overhead which is the running time difference between queries with and without policy. We vary the number of roles from 10 to 100. The result in Fig. 9 shows that the overhead scales well as the number of roles increases. The overhead on policy checking in other frameworks is also linear to the number of roles [10]. In fact, in practice the number of stream users may be huge, but they are normally classified into a limited number of roles.

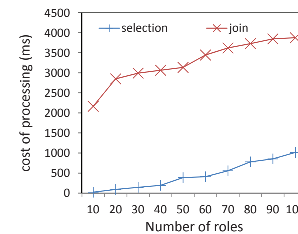
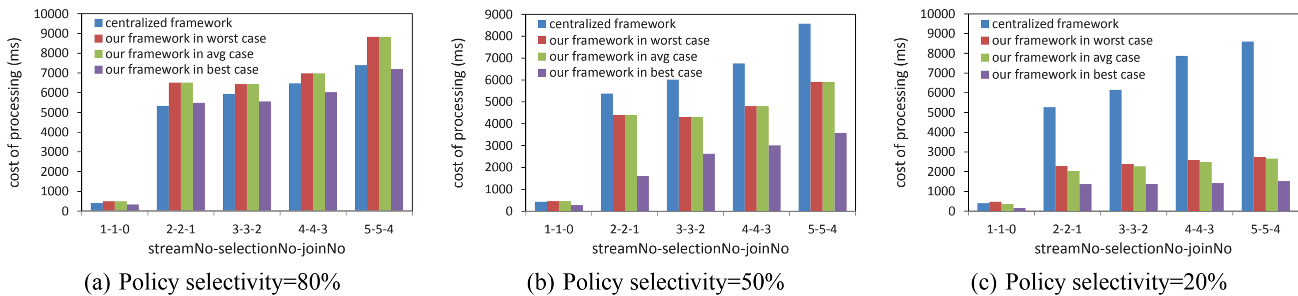


Figure 9: Scalability of overhead as number of roles increases

5.4.3 Performance comparison

In this section, we compare the overall performance at the server site between our framework and the centralized framework. We use the same taxi streaming data with 100,000 tuples as input of each stream. We randomly compose five query plans, with selectivity of 50% for each selection and window size of 50 for each join. We consider 10 roles of users/queries and each tuple-level policy is randomly specified with the average selectivity of 80% (low), 50% (medium) and 20% (high). In the centralized framework, we assume the queries share the same plan and then the relevant policy is enforce before each tuple is returned to the user. For our framework, we consider three cases. In worst case, we assume different policies have contradictory condition. That means all input tuples are sent to the server. In average case, we assume different policy conditions are independent to each other. Then the overall selectivity of policies is as specified in Section 5.3. In the best case, we assume the conditions are totally overlap with each other. Then the number of tuples sent to the server is proportional to the selectivity of each policy. Fig. 7 shows the result. Note that the x-axis streamNO-selectionNo-joinNo indicates the property of each query plan, i.e., the number of input streams, the number of selections and the number of joins.

From the figure we can see that the performance of the centralized framework does not change much as the policy selectivity varies. It is easy to understand, because no matter what kind of selectivity, the centralized framework will do the evaluation over every tuple. The performance of our framework tightly depends on the policy selectivity. When the policy selectivity is rather low, i.e., 80%, our framework does not outperform the centralized framework for the worst case and average case. But the difference is not significant. However, as the policy selectivity goes strict, the benefit of our framework becomes obvious, as shown in Fig. 7(b) and 7(c). In practice, actually policy selectivity is normally high.



(a) Policy selectivity=80% (b) Policy selectivity=50% (c) Policy selectivity=20%

Figure 7: Performance comparison

5.5 Energy Consumption

Last, we evaluate the energy consumption at a stream data provider site. We simulate a data provider using a HTC smart phone with Android OS. The smart phone continuously sends 1 million tuples to a server, at a frequency of 200 tuples per second, via a Wif connection. For the existing centralized framework, each tuple is directly sent out, while in our framework, each tuple is evaluated against the local policies before being sent. We vary the policy selectivity as 50% and 20%, and measure the battery consumption under each framework (battery is fully charged before each run). The result is shown in Fig. 10. We can see that under our framework, the data provider consumes less energy when the policy selectivity is higher. This is because the transmission cost is far more expensive than the cost for local policy enforcement. Although the battery is also consumed by other applications in the phone, for the same time period, the energy consumption difference between different frameworks is suff cient to validate our analysis.

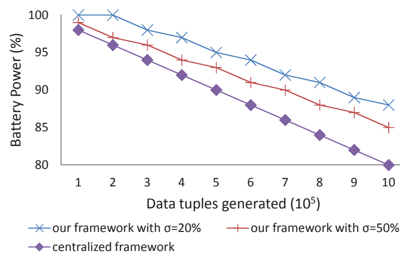


Figure 10: Energy consumption test

6. RELATED WORK

Limiting data disclosure is essential to a Hippocratic DSMS. Normally, the data stream providers (owners) specify access control policies for the system to comply, so that their data will only be disclosed to authorized users. Thus enforcing access control in Hippocratic DSMS attracting most research interest.

All the related works are based on the same centralized framework, in which both query processing and privacy protection take place at the DSMS server site. Lindner and Meier [7] design a f l-tering operator and apply it to the query processing results to f lter the output based on relevant privacy policies. Nehme et al. [10] embed security policies into data stream, by security punctuations (SPs). Their query processor analyzes the SPs in each data stream, and enforces the policies during query processing. However, they may redundantly enforce a same policy multiple times. This framework is further improved by supporting dynamic access authorization of query issuers [8][9]. Carminati et al. [4] propose another framework to enforce access control policies for stream query processing. They model continuous queries as graph of algebraic operators, and focus on query rewriting to incorporate policies into query graphs. Finally, the rewritten query graph can be translated into different query languages according to different stream query processors [3]. As discussed earlier, this attempt makes multiple query optimization hard to apply.

7. CONCLUSION

In this paper, we point out that the existing framework for Hippocratic DSMS cannot fully support limited collection, and suffers from security and performance problems for limited disclosure. Thus they cannot be used in ERP 2 systems initiated by Singapore government. Motivated by this, we propose a novel decentralized framework, which enforces privacy policies at the data provider site and leaves the server only responsible for query processing. We conduct experiments to validate our framework in privacy preserving, performance and energy consumption.

In future work, we will collaborate with the government-appointed testing companies to integrate our prototype with ERP 2 system prototype and evaluate the performance. We will also try to integrate the data stream applications from other government agencies into our platform.

8. ACKNOWLEDGMENT

This work was supported by the A*STAR Grant No. 1021580037.

9. REFERENCES

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

[2] M. H. Ali, M. Y. ElTabakh, and E. Bertino. Hippocratic data streams - concepts, architectures and issues. Technical Report 05-025, Purdue University, 2005.

[3] J. Cao, B. Carminati, E. Ferrari, and K.-L. Tan. ACStream: Enforcing access control over data streams. In *ICDE*, pages 1495–1498, 2009.

[4] B. Carminati, E. Ferrari, J. Cao, and K. L. Tan. A framework to enforce access control over data streams. *ACM Trans. Inf. Syst. Secur.*, 13:28:1–28:31, 2010.

[5] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *SIGMOD*, pages 379–390, 2000.

[6] EsperTech. http://esper.codehaus.org/. 2004.

[7] W. Lindner and J. Meier. Securing the borealis data stream engine. In *IDEAS*, pages 137–147, 2006.

[8] R. V. Nehme, H.-S. Lim, and E. Bertino. FENCE: Continuous access control enforcement in dynamic data stream environments. In *ICDE*, pages 940–943, 2010.

[9] R. V. Nehme, H.-S. Lim, E. Bertino, and E. A. Rundensteiner. StreamShield: a stream-centric approach towards security and privacy in data stream environments. In *SIGMOD Conference*, pages 1027–1030, 2009.

[10] R. V. Nehme, E. A. Rundensteiner, and E. Bertino. A security punctuation framework for enforcing access control on streaming data. In *ICDE*, pages 406–415, 2008.