Securing the Borealis Data Stream Engine

Wolfgang Lindner
MIT,
Cambridge, MA, USA
wolfgang@csail.mit.edu

Jörg Meier
University Erlangen-Nuremberg,
Erlangen, Germany
joerg.meier@informatik.uni-erlangen.de

Abstract

As data stream management systems (DSMSs) become more and more popular, there is an increasing need to protect such systems from adversaries. In this paper we present an approach to secure DSMSs. We propose a general security framework and an access control model to secure DSMSs. We implement our framework into an existing data stream management system and show that our approach not only works, by protecting the system from major security threats, but also has little impact on the overall system performance.

1 Introduction

Data stream management systems (DSMSs) have been developed over the past several years. The focus of research was on query processing and optimization [1], distribution [9], and most recently integration of data sources [3]. Like data base management systems some 30 years ago, none of the current existing DSMSs provide any security functionality. In this paper, we develop a secure data stream architecture which addresses the major threats such systems face. We implemented a security framework with the Borealis data stream engine [2]. Our extended DSMS provides both, object level security and data level security. We focus on evaluating the system in terms of effectiveness and efficiency.

Compared to related information systems, such as data base management systems, there are unique properties of DSMSs that have to be adressed by security mechanisms. DSMSs process data flows by executing queries that continuously produce results.

As an example, consider a DSMS that processes stock prices. A company offers the running system as a platform for stream processing to its customers. Different information providers deliver stock prices as source data. The DSMS continuously receives changing price data as input feeds and executes queries of different customers over those streams. The company providing this system gets paid for delivering the results to its customers.

We identify the following special features of DSMSs, which require new capabilities to secure such systems: In contrast to discrete queries (as in database systems), users enter continuous queries to process streaming data in DSMSs. As a consequence of the streaming information, the requests (control flow) are handled asynchronous to the delivery of the results (data flow). The control and the data flow is separated. Further, the optimization process [2] continuously adjusts a given query network. This dynamic reconfiguration has to be considered because data and operations might be merged inside the system and we have to ensure that the results a user gets suit given rules. Another aspect is the different user abstraction level DSMSs provide. Either there is a SQL-like interface (analogous to DBMSs) [5] or users work with the system via a box-and-arrow-semantic specifying a data flow [2]. The security concept has to be implemented according to the used model because the system's changed behavior reflecting the security functionality affects directly the users' interactions with the system.

As discussed in [14], there are different adversary scenarios that illustrate how an unprotected data stream engine can be attacked. The scenarios can be clustered in three major threat categories, which are summarized in the following.

(C1) **Improper release of information** which can be further divided into:

- (C1a) **disclosure of data** (either inside the query network or while transferring it over the network) [Which source streams are available? What are the query results of others?] and
- (C1b) **disclosure of system internals** [What operations is someone running?].

(C2) **Improper modification of data** where we distinguish between:

10th International Database Engineering and Applications Symposium (IDEAS'06)
0-7695-2577-6/06 $20.00 © 2006 IEEE

(C2a) **changes outside the system** (before the input stream reaches the system or after the output stream leaves the system) [Faking incoming stock price feeds; Changing query results before user gets them] and

(C2b) **changes inside the query network** (either the streaming data or the query graph) [Modify data stream between certain operators; Changing someone's query].

(C3) **Denial of service attacks** [Overload system with expensive operations; Sending too much information].

We address all of these threats with our approach. First, we develop a general secure DSMS architecture and an access control model in Section 2. In Section 3, we present the implementation for Borealis in detail. The evaluation of our implementation is presented in Section 4. A scenario illustrates the protection of the system, followed by performance results which show the efficiency of the extended architecture. Finally, we state open research issues and ideas for future work in Section 5.

2 Security Framework

According to [10, 7, 17], there are three goals to secure an information system: Confidentiality, Integrity, and Availability. Confidentiality means ensuring that information is accessible only to those allowed to have access. Integrity refers to the validity of data, which means to avoid malicious and accidental alternation of data. Availability refers to the period in which a system is in a proper state to operate in the specified way. In accordance with [7], we use the following terms: A **subject** is a user or a program that runs on behalf of a user that accesses the system. Any entity in the system that contains data or allows operations to be executed, is called an **object**. **Access controls** are responsible for ensuring that all subjects access the system objects in accordance to certain security policies. We identify three tasks that need to be completed to reach the security goals and to face the introduced threat categories. These tasks are:

a) Associating identities with users and ensuring that to every request for the system, the corresponding user is known.

b) Deciding if, and in what way, access to certain objects is allowed and ensuring that a user only gets the information he is allowed to see.

c) Ensuring confidentiality and integrity of transferred requests and data.

We derived a general DSMS architecture from existing prototypes [19, 5, 13, 8], ignoring distribution [9] and high availability [11]. The general DSMS architecture was extended with security components fulfilling the mentioned tasks. The secure architecture is shown in Figure 1. We describe the new components (dark boxes) in the following.

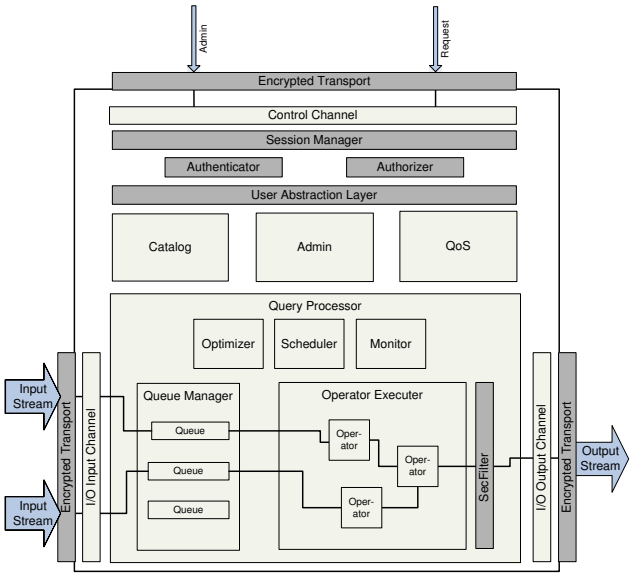

Figure 1. General Secure DSMS Architecture

Each of the components can be associated with one of the three stated tasks: a) SESSION MANAGER, AUTHENTICATOR, b) AUTHORIZER, USER ABSTRACTION LAYER, SECFILTER, and c) ENCRYPTED TRANSPORT.

The SESSION MANAGER assigns each request to a session which belongs to a subject. This assignment is the basis for further authentication and authorization. Before the first request is accepted by the system the user has to prove his identity via the AUTHENTICATOR.

The AUTHENTICATOR checks whether a user is the one he claims to be. The authenticated name is mapped to an internal user-ID which identifies the subject uniquely. This ID is the basis for the later authorization mechanism.

The security mechanisms we introduce are twofold: First, access control checks on *object level* ensure that every executed request suits the access control policy. However, for the processed data inside the QUERY PROCESSOR a second mechanism is needed that ensures that every emitted result tuple, which might be computed from multiple sources through different operators, reaches only destinations according to the system's control policy. We call this *data level security*.

The AUTHORIZER has to grant or deny any requested action. It implements an access control and security model illustrated in the following paragraph. In this way the system is able to decide whether or not a requested action on a certain object is allowed. This verification can be done before

0-7695-2577-6/06 $20.00

any other component is instructed to process the request. These checks belong to the *object level security* mechanisms.

To restrict access to the system, an access control model is needed. Based on RBAC [18], we propose a security model called *OxRBAC* (owner-extended RBAC) for DSMSs. It is illustrated in Figure 2. We distinguish between four entities: Users (subjects), roles, objects, and permissions. Because of continuous queries in DSMSs objects (e.g. a query-operator or an input-stream) in the system exists for a longer period of time. For this reason every object has an owner which is a user. Based on the owner-relationship we can grant special access rights, such as "only the owner is allowed to modify the object". Roles are associated with permissions on objects. Thus, roles summarize access rights necessary to perform a certain job function. Users "can play" certain roles and they activate one or more roles in a session when they log in to the system. Users get the permissions of all their activated roles. We point out details about the set of managed objects and permissions within our implementation in Section 3.

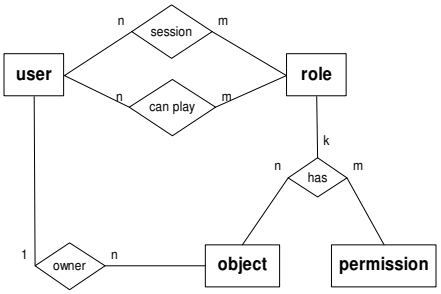

Figure 2. OxRBAC Security Model

To ensure that a subject only gets to see the objects it has permissions for, we provide individual views on the system. Such a view, which only includes objects and operations the subject is allowed to access, is provided by the USER ABSTRACTION LAYER. This component communicates with the AUTHORIZER to check access permissions on objects. The available interface to interact with a DSMS might be either a descriptive language (analogous to SQL, like CQL [5]) or a formal description of the desired data flow from source to destination including the transforming operators in between (like the boxes and arrows in [1]). The USER ABSTRACTION LAYER has to provide a user-specific view on the system corresponding to the used model.

The second module we introduce to avoid improper release of information is the SECFILTER at the end of the QP. It ensures that an output stream for a certain subject only contains data the subject is allowed to get. This mechanism implements *data level security*. Details about the SECFILTER are described in Section 3.

We propose to extend the DSMS architecture by certain security components to secure data transfers, both at the stream (input and output) and the request side of the system (ENCRYPTED TRANSPORT). By encrypting the data and the control channels, we ensure that data is transferred confidentially, i.e. only the authorized participants are able to access it. However, we do not consider the adoption of existing encryption algorithms for the special case of data stream processing. This is left for future work.

3 Securing Borealis

In this Section, we describe the implementation of the introduced secure DSMS architecture of Section 2 into Borealis. We focus on a single server node (no distribution) and we consider to encrypt data at the network layer, which is also out of scope of this paper.

The Borealis architecture is described in detail in [2] and is shown in Figure 3. We implemented the security framework based on the public Borealis version 0.2.

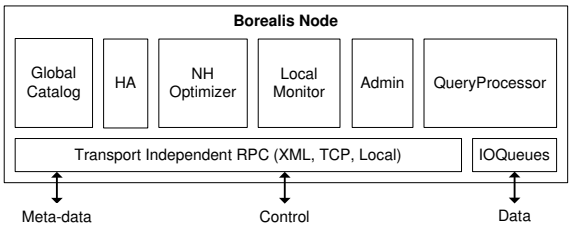

Figure 3. Borealis System Architecture

Our implementation extends the ADMIN and the QUERYPROCESSOR (QP) components. The ADMIN is responsible for handling client requests as well as communicating with other nodes. The QP processes incoming data streams and executes the inserted queries. Our approach reaches different design goals: We used existing interfaces and adopted the available control and data flow. We did not influence present components to stick to loose coupling and high cohesion of the architecture. The components are designed to be efficient in terms of memory and CPU usage. The classes of the implemented security framework are shown in Figure 4.

The core of the implementation is the SECADMIN component. It is an extension of the former ADMIN to provide a secure facade of the system to the clients. The SECADMIN is designed as a dedicated client interface, i.e. it manages all incoming requests and is not used for internal or node to node calls. Supported by SESSIONMANAGER and AUTHORIZER, it presents every user with an isolated view of the system. Through this view every subject is able to access objects only for which it has a corresponding access permission. The SECADMIN implements the conceptually

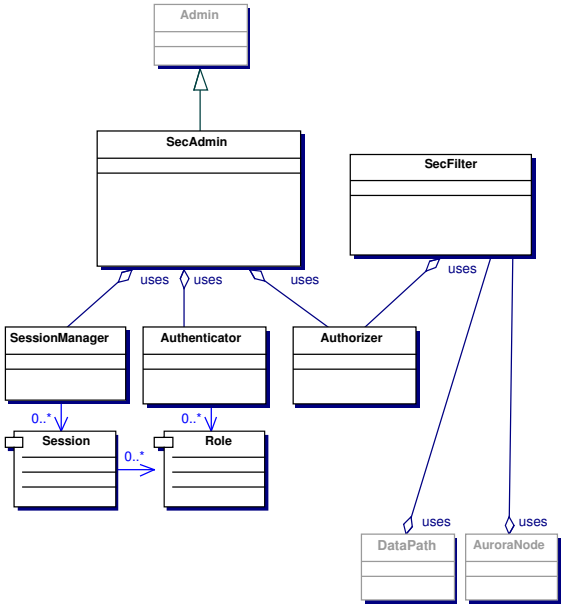

Figure 4. Security Classes for Borealis

described USER ABSTRACTION LAYER (UAL) of the general architecture.

The SESSIONMANAGER creates, maintains, and terminates the different SESSIONS. The AUTHENTICATOR ensures that only subjects who prove their identities' in a proper way (in our implementation by entering the correct password) are allowed to work with the system. Without a proper authentication, a user can not perform any operation. The AUTHORIZER implements the OxRBAC access control model. It handles all permissions and checks them when called by the SECADMIN. The permissions and objects the system manages are described in detail in Section 3.1.

The second important component of our approach is the SECFILTER. In addition to object level security of the SECADMIN, the SECFILTER provides data level security. It is used by the QUERYPROCESSOR of the node (via DATAPATH and AURORANODE). There are two major tasks the SECFILTER is responsible for: On the one hand it marks tuples that arrive at the QUERYPROCESSOR from outside. Therefore, an ID of the input stream, through which the tuple enters the system, is created. On the other hand it filters the tuples at the output of the QUERYPROCESSOR when they leave the system to be transferred to the clients. The SECFILTER works on arbitrary streams and uses the AUTHORIZER to check data access permissions when filtering tuples. The mechanism is described in detail in Section 3.2.

The access control model is applied on the object level when queries are first entered into the system. If at this time the user does not have the required permissions then the

```
<borealis>
    <input stream="input" schema="Packet"/>
    <output stream="output"/>
    <schema name="Packet">
        <field name="time" type="int"/>
        <field name="protocol" type="string" size="4"/>
    </schema>
    <query name="myQuery">
     <box name="myBox" type="aggregate">
       <input stream="input"/>
         <output stream="output"/>
         <parameter />
     </box>
    </query>
</borealis>
```

Figure 5. Query Example

query will be rejected. However, if the user permissions are sufficient when the query is entered but change during the course of time, e.g. by revoking the right to read a stream, applying the model at the object level falls short. This is also true if the system rearranges the query plan to accommodate queries from multiple users since not all users usually have the same permissions on all operators and streams. In this cases the access control model is applied on the data level by the SECFILTER. Therefore, both mechanisms ensure that delivered data always is in compliance with the user's access permissions.

3.1 Object Level Security

We introduce different system objects for managing access permission checks. Figure 5 shows an example of an XML schema definition of Borealis. We derived the following system objects for our implementation.

- **SCHEMA** Meta-data description about a stream.
- **STREAM** Data is written to an input stream and is read from an output stream.
- **QUERY** A Query is a collection of operators (boxes) and streams (input and output).
- **SYSTEM** That special object represents the system itself. Access permissions on it might include administrative actions, like shutdown, restart, change system parameters (e.g. switch SECFILTER on or off).

Based on the server interface and the operations someone can perform with the system, we introduce the following basic access permissions in our implementation.

- **VIEW CATALOG** The general right to access the catalog.
- **VIEW OBJECT** The right to see and use a concrete object in the catalog.
- **ADD** The right to add objects to the system (e.g. needed for inserting a query).

- **SET QUERY STATUS** The right to change the state of a query.
- **SUBSCRIBE** The right to subscribe to an output stream and receive data from it.
- **READ TUPLE** The right to read a single tuple (is used by SECFILTER).
- **CHANGE PERM** The right to change permissions on an object.
- **CHANGE SYSTEM** The right to change global system properties (e.g. switching filtering on or off).

Access permissions are recursively checked for every request. For instance, when a query is inserted, first the general right to add an object is checked. Second, the permission of every involved input stream of the query is checked and third the permissions for the newly created objects (boxes, intermediate streams, output streams) are set to default values (e.g. the owner of every object is set).

So far, we discussed the security mechanisms on object level. The described mechanisms work on requests that reach the node through the control channel. After passing these checks, the system performs the requested action. For instance a query gets started and produces results. At that point an application might subscribe to the output stream and tries to read data. Again, the subscription request to the SECADMIN is checked by the AUTHORIZER and, if granted, the QUERYPROCESSOR gets invoked to send results to the corresponding client. In the next paragraph we illustrate in detail how the data protection through the SECFILTER works.

3.2 Data Level Security

The SECFILTER component provides data level security in our implementation. As described in Section 2, it ensures that only authorized data reaches a client. Users who want to receive data from the system have to subscribe to a certain output stream first. This request which reaches the system through the SECADMIN is checked on object level by the AUTHORIZER. If the access to the output is granted, the subscription is passed to the QP. Every emerging tuple at this output stream is checked by the SECFILTER before being sent to the subscriber. Users need to have access permission on the input stream of a certain tuple in order to receive it at the end of the QP.

To be able to decide at an output stream from which source a tuple comes, we mark the packets when entering the node. With this strategy the QP and the OPTIMIZER work independently from the security mechanism. Every tuple in Borealis has a header with time stamps, type, and other information [2] (41 bytes altogether). We extend the header with a 4-byte identifier for the source streams the tuple originates from. This label is needed by the SECFILTER to check at the end of the query diagram whether or not a user gets a tuple.

Tuples can originate from more than one input source. Consider a situation as shown in Figure 6.

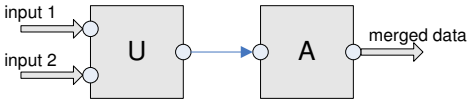

Figure 6. Merged Streams Example

The two input streams get merged with an union box and then aggregated. Thus, a result tuple of the aggregate might be computed from both input streams. The new tuple which contains values of merged data has to be labeled with an ID. We discuss solutions for this problem of combined result tuples. Different access control policies can be realized.

For instance, We can ensure that a user only gets an aggregated value, if he has **access permission on every participating input source**. That eliminates the possibility to infer confidential information from summarized values and from parts of the single input streams.

Another possible algorithm for dealing combined tuples is to introduce an **"aggregate-read right"**. That means a user is allowed to get summarized information, although he is not allowed to read the corresponding input sources separately. A simple example to demonstrate this approach might be a situation, where someone is allowed to read an average value of a certain information source, but is not allowed to read the single source values itself.

A similar problem arises with join operations. Consider a query like:

```
SELECT in1.*
FROM   from in1, in2
WHERE  in1.x = in2.x
```

The output stream only contains data from *in1*, although the tuples that are computed depend also on the input *in2*. Translated in a box-and-arrow semantic this query consists of a join `JOIN(in1.x = in2.x)` and a projection `PROJ(in1.*)` thereafter. The resulting stream is called *out1*. The question is what should the result of this query be if a user is allowed to access *in1* but not *in2*?

The answer to this question depends on the used access control semantic. Figure 7 summarizes what we will refer to as a *strong semantic*.

The users A, B, C, and D have different permissions on the input streams $in1$ and $in2$. Only if a user has permissions on all input streams, he is also granted permissions on the output stream out. Compared to a strong semantic, which basically reflects a logical `AND` operation, we refer to a semantic reflecting a logical `OR` operation, as a *weak*

	$p(in1)$	$p(in2)$	$p(out)$
A	✘	✘	✘
B	✘	✔	✘
C	✔	✘	✘
D	✔	✔	✔

Figure 7. Security Semantic

semantic. Please note, there are more semantics conceivable. However, to maximize the provided security we implemented the strong semantic into the system and therefore, in our implementation, the user would have no permissions to the query result *out1*.

Also, there are different strategies possible to label data packets that are combined of more than one input, either by an aggregate (as shown in the example above) or by a join box. We consider the following two possibilities.

One way is to use one bit out of the 32 available ones for one input stream and to create combination-IDs by a logical OR operation (ORing) of the bit vectors. We can address every possible combination of source streams, but with a fixed header size this algorithm is limited to a maximum amount of manageable input sources (here 32).

Another possibility to create these IDs is to dynamically add IDs when a certain combination occurs. We can number the source streams consecutively. When a combination occurs (e.g. at the output of an aggregate), we store the involved streams and create a new ID within the serial numbers. In this way only used combinations require the reservation of IDs and there are most likely much more possible input streams to be manageable.

It is possible for both strategies to dynamically extend the tuple header when more space is required for the IDs. Of course, the performance overhead has to be considered. Our implementation uses the fixed header bit-vector strategy for reasons of simplicity and performance.

With the SECFILTER we achieve different effects. The system does not only provide object level security (access control on output streams) at time of subscription, but data-level security based on access permissions on the sources of the data. A revocation of access permissions after the time of subscription immediately takes effect. Although a user might still be connected to an output stream, he will not get any further results, if his access right is revoked. Moreover, the optimization process can merge semantically equivalent parts of queries of different users and the system ensures that every client only gets results with proper permissions at the end.

4 Evaluation

In this Section we analyze our security prototype in two ways. First, we illustrate a detailed scenario, which shows the behavior of the extended system and how the protection mechanisms affect the users. Second, we prove that our ideas do not only work, but create a maintainable performance overhead. Finally, we discuss the reached goals with regard to the threat categories.

4.1 Scenario

With the introduced security functionality the behavior of Borealis changes in the following way: Users have to prove their identity through a login process. Then a session is created. Subjects are associated with predefined roles in the system to perform their tasks. Based on the OxRBAC model the access permissions are managed on roles. Every request a client sends is associated with a user-session and is checked for authorization. Users are isolated from each other in a logical way. The system provides everyone with an individual view of the system. Users only see objects in the catalog they are allowed to access. Data-level security is introduced by filtering tuples at the output streams, so everyone only gets permitted results, although multiple users and queries exist. The scenario is based on the example of Section 1: An organization uses Borealis to provide a data stream analysis platform for its customers. Certain partners provide source information (e.g. stock prices), which are available for querying.

We focus on the main mechanisms of our implementation approach and simplify some implementation details to make the scenario more clear. For simplicity we do not distinguish between different access rights in this example. The calls shown in the UML [16] sequence diagrams are abstracted from the real server interface and we only illustrate the important aspects to make it easier for the reader to follow.

In our example, there are four predefined roles in the Borealis system: ADMIN, CUSTOMER1, CUSTOMER2, INFO PROVIDER. Moreover, we introduce four users, each of them is associated with one role: Joe (ADMIN), Ed (INFO PROVIDER), Al (CUSTOMER1), and Bob (CUSTOMER2). Now imagine a situation as shown in Figure 8.

The tables show the internal state of the (simplified) permissions. At the beginning there is only one default object, “system”, which represents Borealis itself. The ADMIN role has access permission on it (e.g. for adjusting system parameters, switching filtering globally on or off). After logging in, Joe adds a schema definition *s* and two streams *in1* and *in2* to the system. The updated permission table is shown. Then he grants the access permission on these three objects to the roles CUSTOMER1 and INFO PROVIDER (again, the updated permission table is shown).

When Ed logs in and looks up the objects in the catalog, he sees the three objects (*s*, *in1*, and *in2*), he got permission for. Then Ed starts transferring input data (the source

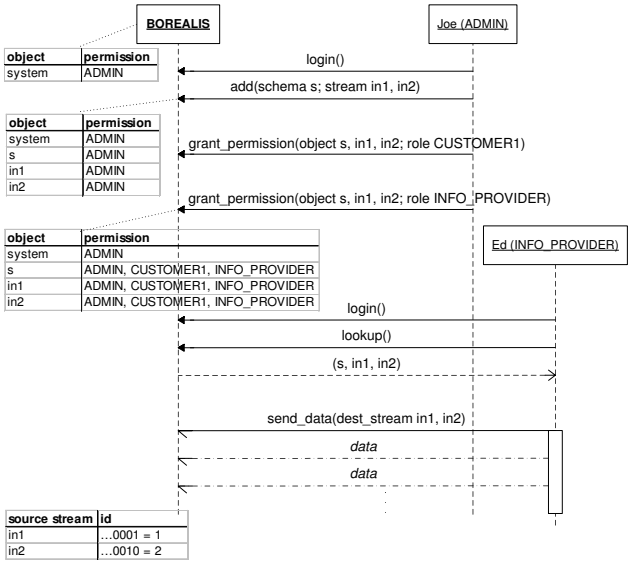

Figure 8. Scenario - Part 1

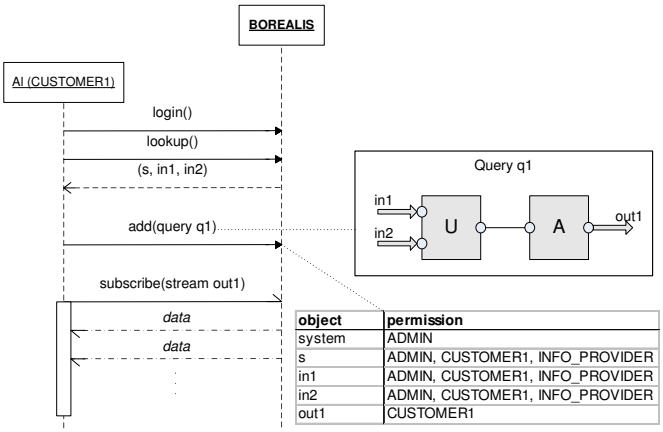

Figure 9. Scenario - Part 2

information) to the two streams *in1* and *in2*. For instance, this data might be stock prices from New York and from London. As the first tuple arrives, the system creates source stream IDs for the streams *in1* and *in2*. The IDs are also shown in the table at the bottom of Figure 8. In a next step Al connects to the system, see Figure 9.

After having successfully logged in, Al looks up the objects available. Because of the access permissions, the system returns *s*, *in1*, and *in2*. After that, he inserts a query *q1*, which is illustrated in Figure 9. With inserting this query, a new object for the output stream *out1* is created, for which the role CUSTOMER1 (as the owner of the object) has access permission (see updated permission table in Figure 9).

Then Al subscribes to the output of his query and receives the computed query results from Borealis. Every tuple Al receives goes through the SECFILTER component and is checked for authorization. As a result of the example query *q1*, two different possibilities exist: The data of the computed aggregate can originate either from a single source stream (*in1* or *in2*) or it can comprise of both. As described in Section 3, new IDs for tuples that are computed from more than one data source are created by combining the Bit vectors of the involved stream IDs with a logical OR operation. In this example the combination of *in1* (ID ...0001 = 1) and *in2* (ID ...0010 = 2) is ID ...0011 (equal to the integer value 3). The CUSTOMER1 role has permission on both input streams. For that reason Al gets results with the IDs 1, 2, and 3.

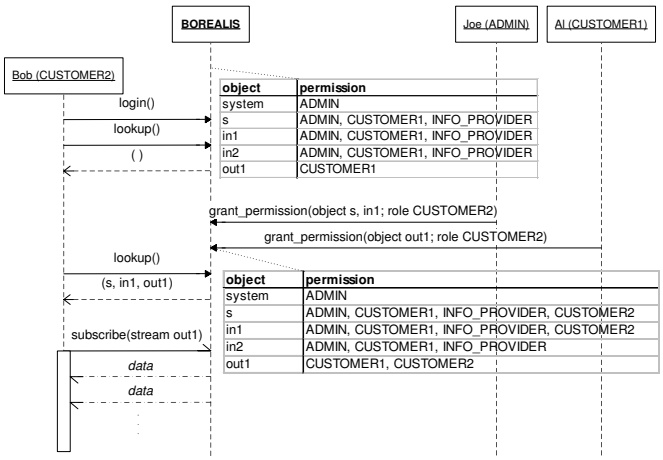

Figure 10. Scenario - Part 3

Now user Bob logs in Borealis as shown in Figure 10. Like Al, Bob calls lookup() to see the available objects. In contrast to Al, he gets an empty result, because at that time, he does not have any access permissions in the system (see the first table in Figure 10). Then Joe (ADMIN) grants access for Bob's role CUSTOMER2 on the schema *s* and the stream *in1*. After that, Al, the owner of the query *q1* and its stream *out1*, grants the access right of *out1* to CUSTOMER2. The updated permission table is shown. When Bob looks at the catalog again, he sees the objects *s*, *in1*, and *out1*, for which he got the permissions. Afterwards Bob subscribes to the stream *out1* and receives the output data. As described before, every tuple leaving Borealis is filtered. In this example Bob only has permission to read the input stream *in1*. Although he is connected to the output stream *out1*, where aggregated tuples of the stream *in2* or even of the combination of *in1* and *in2* might be created, Bob only receives aggregates that originate exclusively from source stream *in1*. These results have the source stream ID 1.

0-7695-2577-6/06 $20.00

4.2 Performance Results

With enabled security features the system has to perform some additional tasks at the time of setting up a query and at the time of processing tuples inside the query network. These are authentication, session management, authorization, permission management, marking tuples, and filtering tuples. These situations are analyzed in this Section. First, we discuss the initial time needed for setting up a query including security functionality. Second, we show the impact on tuple latency while a query is running.

Query Setup

For starting a query after a client's request has reached the system, a fixed set of operations has to be performed (login, session-establishment and -management). In addition, there is a dynamic part, which depends on the query (security checks on all involved objects). The influence of the query on the setup time only depends on the amount of used schema, stream and box elements, because the system handles all objects equally. For this reason, we will not distinguish between these object types. The type and the complexity of the boxes do not influence the setup time either, because this properties are not considered by the system at that point. Thus, we analyze different queries with increasing amount of objects. We refer to this query size as "query complexity" in the following. We chose the following six queries for the experiment:

Query 1: 4 objects (1 schema, 1 stream, 2 boxes)
Query 2: 8 objects (2 schema, 2 streams, 4 boxes)
Query 3: 22 objects (3 schema, 3 streams, 16 boxes)
Query 4: 40 objects (4 schema, 4 streams, 32 boxes)
Query 5: 74 objects (5 schema, 5 streams, 64 boxes)
Query 6: 140 objects (6 schema, 6 stream, 128 boxes)

Our test environment was a PC running Fedora Core 2 (kernel 2.6.5) with an AMD Athlon XP 1800 processor and 1 GB RAM. We measured the total time it takes to start the example queries. In Figure 11 the results of our experiments are shown for the six queries with increasing complexity. For every query the setup time was measured 10 times, first with an unprotected system and second with the secured Borealis engine. The standard deviation is smaller than 1%. The experiments produced constant values without greater variations.

As the increasing time from Query 1 to Query 6 show, more complex queries take in general more time for setup than simpler ones. The QP processes every object, creates schema definitions for intermediate streams, and the catalog has to be maintained. However, the secure system behaves analogous to the unprotected system. With increasing query complexity, the absolute overhead is only raising very little. Figure 11 demonstrates this fact: The average query setup time of an unprotected system is compared to the setup time of the secured Borealis system. In other words the cost for authentication in the setup phase is negligible.

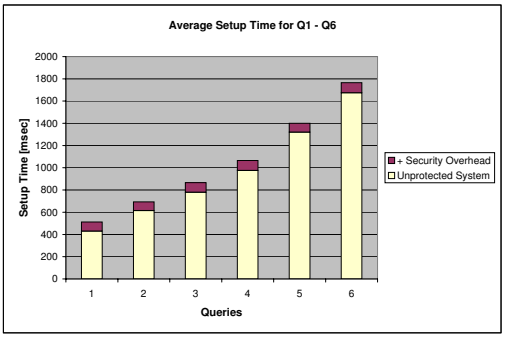

Figure 11. Average Setup Time for Q1 - Q6

The lower parts of the bars present the query setup time of a non-secured system. The time for computing the security operations is added to the bars at the top. With Figure 12, we illustrate the percentage of the query setup security overhead compared to the query setup time.

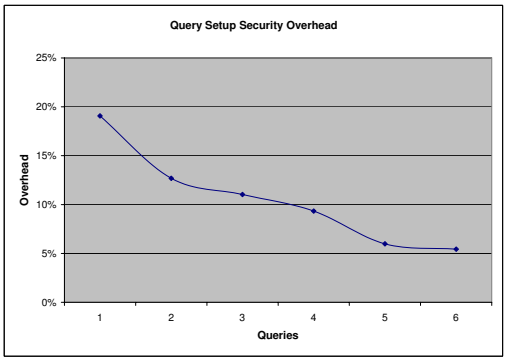

Figure 12. Initial Query Setup Overhead

The query setup security overhead decreases from around 18% with a very simple query (Query 1 has only 4 objects) to around 5% with Query 6. The additional processing time for the secure system is, with regards to the increasing setup time for more complex queries, comparably small. Due to the efficient object permission checks, the security overhead keeps almost constant with larger queries. As the total query setup time increases the percentage decreases.

Tuple Latency

Tuple latency refers to the total time it takes a tuple to pass the Borealis system. After entering the system a tuple is processed by one or more boxes and transferred to a client at the end. The SECFILTER component influences this tuple

latency, because the packet is labeled and the subscriber's authorization is checked before emitting the tuple. As introduced the second performance aspect we focus on is the impact on tuple latency by the SECFILTER component while queries are running. As stated before, a tuple gets labeled corresponding to its input source when entering the node and it is filtered according to access permissions before it is transferred to a client. The mechanism used by the SECFILTER component has two consequences when tuples are processed by the system. First, all operators have to process the source stream ID of a tuple. For aggregate or join boxes the computation of the ID is just an OR operation. All other boxes just copy the source ID (4 extra Bytes) to the emitted tuple. Second, the tuple header is 4 Bytes larger now (approx. 10%), which might impact memory consumption and tuple transfer time. However, we show in the following that the overall overhead is justifiable.

We focus on the absolute overhead the SECFILTER causes, which is the time it takes to perform the operations of the SECFILTER (labeling and filtering tuples). We refer to this overhead as *absolute* SECFILTER *overhead*. Therefore we analyze the impact of different parameters on this absolute SECFILTER overhead. The amount of in- or output streams has no impact on the absolute SECFILTER overhead, because the computation is not influenced by this number. The influence of the number of users and roles is comparably small on the performance, because the permission check is done once at the end of the QP by looking up an internal data structure with log(n)-complexity. However, there might be an impact by the number of tuples that have to be filtered in a certain time period. That is why we show in Figure 12 that the speed of the arriving tuples (tuple rate) does not significantly influence the filtering overhead either. We compared the time it takes for single tuples to be processed by an unprotected system with the processing time of the secured Borealis. To eliminate any influence, such as box processing times or enqueuing times inside the QP, we measured this tuple latency without any running query. The tuples were sent to the node, arrived at the QP on an input stream, and were read directly from this stream.

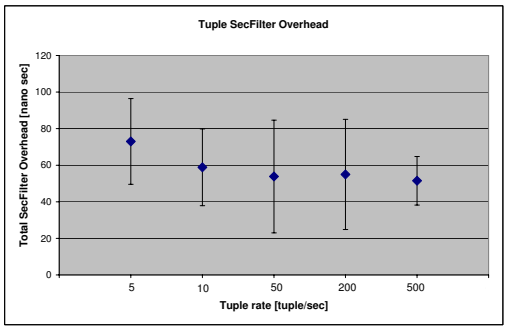

Figure 13. Tuple SecFilter Overhead

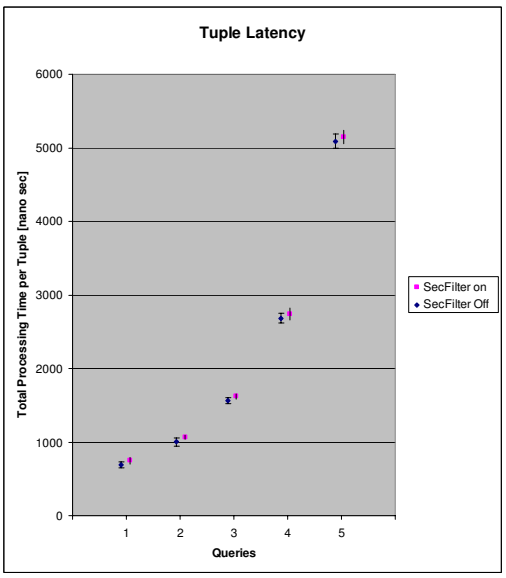

Figure 14. Tuple Latency

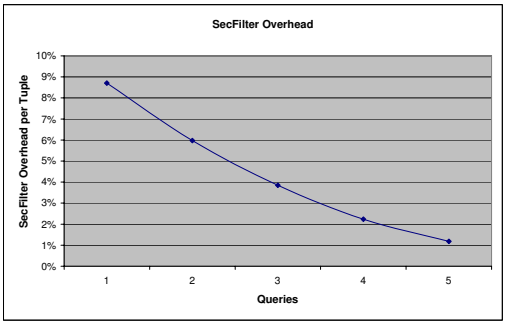

Figure 15. SecFilter Overhead

Figure 13 shows the absolute SECFILTER overhead while the tuple rate increases. In our test environment, the average processing time for the SECFILTER is 60 nanoseconds. The fluctuations in the diagram are caused by the Borealis scheduler and other non deterministic threads of the operating system (caused by system processes that can not be deactivated without crippling the whole system). Although there are variations in the measurement results, Figure 13 shows that the absolute tuple filtering overhead stays in a certain range around 60 nanoseconds.

Next, we compare the absolute SECFILTER overhead to tuple processing times while different queries are running. The experiment shows that the SECFILTER overhead is also constant with different queries running. We show that the SECFILTER overhead is negligible considering realistic query processing times.

We used five example queries to illustrate the proportion of the SECFILTER overhead to realistic query processing

times. All queries have one input stream, a sequential combination of filter and union boxes and one output stream at the end. For comparing the SECFILTER overhead to query processing times, we do not distinguish between different box types or different box arrangements in the query diagram, because the queries serve only as examples to get an impression about the processing time inside the QP. The queries vary in the amount of boxes they use: Query 1 uses 4 boxes, Query 2 uses 8 boxes, Query 3 uses 16 boxes, Query 4 uses 32 boxes, and Query 5 uses 64 boxes. The processing times for each query, with the SECFILTER enabled and disabled, are presented in Figure 14.

Although the measured times for each query vary largely, as the error bars in the diagram show, the overhead stays constant with increasing query complexity. With increasing query complexity, the total processing time for a tuple inside the QP increases.

Figure 15 shows the percentage of the SECFILTER overhead compared to the processing time for the given queries. The diagram shows that depending on the complexity of the query, the overhead for the data level security mechanisms is comparably small. It scales down from around 9% to around 1% for queries with increasing amount of boxes.

4.3 How secure is Borealis now?

Considering the threat categories of Section 1 the system provides protection against improper release of information (C1), both protection for system internals (through the USER ABSTRACTION LAYER) and for the data being processed inside the system. We did not consider network layer security. Inside the query diagram improper modification of data (C2) is also covered by our mechanisms. The implementation of operator-based permission checks, QoS-based security features, and auditing is left for future work, so the problem of denial of service (C3) is partly solved. In Section 5, we discuss consequences of additional access controls, such as operator-based checks. In a next step it has to be avoided that users are able to use resource consuming operators that load the system and influence other's QoS. We can ensure that direct attacks to system properties are avoided, as users are not able to perform unauthorized actions (such as actions only an administrator is allowed to do).

5 Future Work

We point out some ideas about developing a security language, integrating authorization for data senders, problems in a distributed environment and extending the access controls.

5.1 Security Language

Users, roles, objects, and access permissions have to be expressed in a certain way to instruct the system what to do. Therefore a security language is needed. For the future, we propose to develop a language that is able to express security features. The language should be integrated in the existing way of working with the data stream engine. Borealis uses an XML-based description of schema, streams, and queries. We propose to integrate an XML-based security language as well. There are standards, such as XACML for access control in distributed systems [15], which can be used as a template to define a security language for data stream engines.

5.2 Authorizing Data Senders

We propose to integrate security checks for applications which send data to the system. Such input access permission checks are not possible with the system architecture of version 0.2. It will be possible to integrate this functionality in upcoming versions.

5.3 Distribution

Although we use a single node system for our observations, we point out two challenges in a distributed data stream engine as far as security mechanisms are concerned.

First, it has to be ensured that changed access permissions get properly propagated to all involved nodes. That is necessary for being able to filter data at the output streams of the system, which might be on different nodes in the distributed environment.

Sencond, as described in Section 3, the SECFILTER uses unique IDs to mark and filter tuples. In a distributed system, such source stream IDs also have to be managed without conflicts.

5.4 Additional Access Controls

A next step is to introduce operator-based access control. An administrator could define who is able to use certain boxes within a query in the system. Expensive operators (such as the sort-box) could then be limited to special roles. Furthermore box-parameters could be checked, e.g. the window size users can specify (for instance for aggregate boxes) might also be restricted for performance reasons as large windows consume much memory. Operator-based access control is performed on object level. It would only influence the query setup time. As far as our experience shows, we expect that the additional processing time for operator permission checks is negligible.

6 Related Work

As far as we know none of the current DSMSs provides security. The following projects are examples for such data stream processing engines. **Borealis** [2], the prototype we used for our implementation, has been developed at Brandeis University, Brown University, and MIT. It is based on **Aurora** [1] and **Medusa** [19]. Aurora* is a distributed version of Aurora while Medusa is a federated distributed system. Many of the ideas in Borealis are developed in these two projects. These prototypes use an XML description for schemata and queries in a box-and-arrow semantic. **STREAM** [5] or the STanford stREam datA Manager is supposed to be a "general-purpose" DSMS and is a project of Stanford University. To express queries, a language called CQL (Continuous Query Language) is introduced. Declarative queries are compiled into a query plan. **PIPES** [13] is a project of the University of Marburg using a "hybrid multi-threaded scheduling" three layer architecture. **TelegraphCQ** [8] a general system for adaptive data flow processing with an extension to support shared continuous queries is a project developed at Berkeley University.

There are algorithms for integrating security constraints in real-time database systems without degrading real-time performance in terms of missed deadlines [4]. The authors present two concurrency algorithms to enforce security constraints and to be able to compromise with real-time constraints. Their approach is based on a mandatory access control model. The algorithms decide on a high level description of access levels whether a certain transaction is aborted, committed, or blocked.

Processing tuples in a data stream engine can be seen as routing data through a network of operators based on the tuple content (e.g. header information). There are different groups dealing with content based routing [12, 6]. To our knowledge security issues have not been discussed in this research community.

7 Conclusion

Like any information system, DSMSs face different threats, which can be summarized in three categories: Improper release and modification of information as well as denial of service attacks. Based on the conceptual work "Towards a secure Data Stream Management System" [14], we presented an implementation of a security framework for Borealis, a research data stream engine. We evaluated our implementation in terms of effectiveness and efficiency. The results show that we can properly protect Borealis against the two major threats improper release and modification without creating too much of a performance overhead. Denial of service attacks can be partly avoided by access controls, which limit actions users can perform. We stated open research issues for future work, as well as some ideas to extend our system.

Acknowledgments

We would like to thank Magda Balazinska (University of Washington) and Mike Stonebraker (MIT) for their constructive feedback and suggestions. In addition, we thank Elisa Bertino (Purdue University) and Kenneth Salem (University of Waterloo) for their comments.

References

[1] D. J. Abadi and et. al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 2003.

[2] D. J. Abadi and et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[3] D. J. Abadi, W. Lindner, S. Madden, and J. Schuler. An integration framework for sensor networks and data stream management systems. In *VLDB*, 2004.

[4] Q. N. Ahmed and S. V. Vrbsky. Maintaining security and timeliness in real-time database system. *J. Syst. Softw.*, 61(1):15–29, 2002.

[5] A. Arasu, B. Babcock, and et al. Stream: The stanford stream data manager. *IEEE Data Eng. Bulletin*, 26(1), 2003.

[6] P. Bizarro, S. Babu, and et al. Content-based routing: Different plans for different data. In *VLDB*, 2005.

[7] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison Wesley, 1994.

[8] S. Chandrasekaran, O. Cooper, and et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[10] D. K. Hsiao, D. S. Kerr, and et al. Privacy and security of data communications and data bases. In *VLDB*, 1978.

[11] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, 2005.

[12] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *EDBT*, 2004.

[13] J. Krämer and B. Seeger. Pipes - a public infrastructure for processing and exploring streams. In *SIGMOD*, 2004.

[14] W. Lindner and J. Meier. Towards a secure data stream management system. In *VLDB Workshop TEAA*, 2005.

[15] M. Lorch, S. Proctor, and et al. First experiences using XACML for access control in distributed systems. In *ACM workshop on XML security*, 2003.

[16] OMG. UML 2.0 superstructure FTF convenience document. http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-02.pdf.

[17] C. P. Pfleeger and S. L. Pfleeger. *Security in Computing*. Prentice Hall, 2003.

[18] R. S. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Commun. Mag.*, 32(9), 1994.

[19] S. Zdonik, M. Stonebraker, and et al. The Aurora and Medusa Projects. *IEEE Data Eng. Bulletin*, 26(1), 2003.