SparkXS: Efficient Access Control for Intelligent and Large-Scale Streaming Data Applications

Davy Preuveneers and Wouter Joosen
iMinds-DistriNet-KU Leuven
Leuven, Belgium
{davy.preuveneers, wouter.joosen}@cs.kuleuven.be

***Abstract*—The exponential data growth in intelligent environments fueled by the Internet of Things is not only a major push behind distributed programming frameworks for big data, it also magnifies security and privacy concerns about unauthorized access to data. The huge diversity and the streaming nature of data raises the demand for new enabling technologies for scalable access control that can deal with the growing velocity, volume and variety of volatile data. This paper presents SparkXS, an attribute-based access control solution with the ability to define access control policies on streaming latent data, i.e. hidden information made explicit through data analytics, such as aggregation, transformation and filtering. Experimental results show that SparkXS can enforce access control in a horizontally scalable way with minimal performance overheads.**

***Keywords*-access control, streaming data, intelligent applications, Internet of Things, privacy**

I. Introduction

Intelligent environments are evolving to open ended large scale and dynamic network infrastructures fueled by low cost, connected and wirelessly communicating devices that collect data, relay information to one another, process the information collaboratively, and take actions in an autonomic way. Processing usually takes place at aggregation points in the network or in the cloud. However, as the classical *store-and-analyze-later* approach does not scale well with the exponential data growth of these connected devices, being able to make sense of large *volumes* of data with uncertain *veracity* and *value* from a *variety* of sources in real-time becomes a key differentiator.

Orthogonal to these non-functional concerns are a multitude of security and privacy challenges caused by unrestricted access to high volumes of sensitive data. In distributed programming frameworks, such as Hadoop MapReduce [1], Yahoo's S4 [2], Twitter's Storm project [3], Mahout [4], Dremel [5], a cluster of distributed and parallel processes manipulate, map and reduce huge scattered data sets to useful information. Enforcing who has access to this volatile but possibly sensitive data is an increasingly daunting concern. Encryption at the network level, but also in the storage tier (also known as *at-rest* encryption) can help ensure that data is not disclosed to or modified by unauthorized people. Cloudera and Intel are collaborating under the umbrella of Project Rhino[1] on such protection capabilities at the data cell level for Apache Hadoop. However, in a streaming context where the lifetime of data is limited, managing the visibility of volatile data with encryption is obviously not without a non-negligible computational overhead. Therefore, our focus in this work is on policy-based access control mechanisms for data processed by distributed data processing frameworks similar to the aforementioned ones.

The main contribution in this work is SparkXS, an attribute-based access control solution with the ability to define access control policies not only on regular batch data, but also on streaming and especially latent data, i.e. hidden information made explicit through aggregation, transformation and filtering, with support for lazy policy evaluation. Our software solution is built on top of the Apache Spark [6] framework and its extensions for stream processing. We evaluate the performance of SparkXS in the frame of an urban computing use case. Experimental results show that SparkXS can enforce access control in a horizontally scalable way with minimal performance overheads.

After reviewing related work in section II, we discuss the basic foundations of SparkXS in section III and its access control extensions in section IV. In section V we analyze the feasibility and performance of SparkXS in a distributed deployment. We conclude in section VI summarizing the main insights and identifying possible topics for future work.

II. Related work

This section reviews various related works in the domain of access control for distributed systems and state-of-practice distributed data processing frameworks. With this overview of related work, our goal is to clarify the gap that SparkXS aims to bridge.

The data explosion of the Internet of Things is often linked with the *Big Data* paradigm. MapReduce [7] – and its Hadoop [1] implementation – is a software framework and programming model that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of computers. The shortcomings and drawbacks of batch-oriented data processing have been widely recognized as many applications are in need of real-time [8], [9] and in-stream processing capabilities [10]. This concept got a lot of traction with various distributed event stream processing (ESP) engines emerging. Yahoo's S4 [2] and Twitter's Storm project [3] were among the first to attract a lot of attention. Also Google acknowledged the limitations

[1]https://github.com/intel-hadoop/project-rhino/

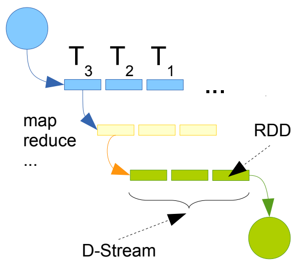

Fig. 1. Conceptual decomposition of streaming big data with Spark

of MapReduce, with its MillWheel [11] framework and programming model dedicated to fault-tolerant stream processing at Internet scale. Spark [6] is another state-of-practice software solution for large-scale data processing.

Access control is an important information protection mechanism, with the most common, oldest, and most well-known identity-based access control models being Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC) [12], [13], [14]. Recently, there has been growing interest in Attribute Based Access Control (ABAC) [15] to overcome the limitations of the aforementioned access control models. The ABAC model makes decisions on permitting or denying access by relying on attributes of subjects, resources, actions, and the environment. It allows for resource owners to grant access to unanticipated users as long as they have attributes that meet certain criteria. The latest trend in access control models is Risk-Adaptive Access Control (RAdAC) (RAdAC) [16], [17] where access decisions depend on dynamic risk assessments. In policy-based access control, such as Ponder [18], Rei and KAoS [19], and the eXtensible Access Control Markup Language (XACML) specification [20], regulation of access to protected resources is expressed external to the applications as high-level rules that define who has access to what resources under what conditions.

Apache Hadoop is without a doubt one of the most popular big data platforms, but it was initially conceived for a trusted environment and not designed with security in mind. Various open source solutions, such as the Apache Knox Gateway[2], Apache Sentry[3] and the Apache Accumulo[4] framework, are now emerging trying to address this void. Access control in streaming systems was explored in [21], [22], but there does not seem to be any equivalent solution for access control that are stream processing oriented supporting latent data.

III. Motivating use case and foundations of SparkXS

In this section we will present a motivating use case in the area of urban computing, and discuss the access control primitives used in SparkXS. The policy-based access control is inspired by XACML, though the architectural design is adapted to deal with the streaming nature of the data.

A. Motivating use case

Urban computing [23] is a paradigm that fits well with the smart city vertical domain of the Internet of Things. This research domain focuses on solutions for city-scale problems, such as traffic congestion and air pollution, by leveraging streaming big data produced by such smart city environments to make better informed decisions. In our motivating use case, we will leverage traffic flow information of taxis using the T-Drive trajectory data set [24]. This data set contains about 15 million data points of 10357 taxis in a period of one week, with for each taxi an individual file with GPS trajectories in the following format:

```
#taxi id, date time, longitude, latitude
1,2008-02-02 15:36:08,116.51172,39.92123
1,2008-02-02 15:46:08,116.51135,39.93883
1,2008-02-02 15:46:08,116.51135,39.93883
1,2008-02-02 15:56:08,116.51627,39.91034
1,2008-02-02 16:06:08,116.47186,39.91248
```

This use case merely serves as a proof-of-concept as our objective is to enable policy-based access control to streaming data for a variety of stakeholders and circumstances. Assume these 10357 taxis belong to different organizations, and that all these data streams are processed by SparkXS. We want to enforce the following diverse set of policies:

1) The *admin* of SparkXS has super-privileges to access all streaming data of all taxis of all organizations.
2) Each *taxi driver* has full access to his own data stream, but not to those of other taxi drivers.
3) Only *taxi drivers* of registered organizations are allowed to upload their location data.
4) The *call center* has access between 7:00 AM and 6:00 PM to location data of taxis that are waiting for passengers (have not moved for 3 minutes).
5) The *manager* has access to taxi locations of his organization if his taxis cross the speed limit 50 km/h.
6) The *call center* has access to traffic and congestion info, based on different taxis speeds on busy streets.

In the following subsections, we will elaborate how these policies will be implemented and evaluated with attribute based access control in SparkXS.

B. Foundations of Spark Streaming

Our framework is built on top of Apache Spark and its Spark Streaming [5] extension. Spark has the advantage that it provides one API for both batch and streaming computation. A *Resilient Distributed Dataset* (RDD) is the storage abstraction of Spark. As functions and computations on an RDD must be idempotent and side-effect free, Spark can rebuild lost data without replication by tracking the operations needed to recompute the lost data.

[2]http://knox.apache.org/
[3]http://sentry.incubator.apache.org/
[4]http://accumulo.apache.org/

[5]https://spark.apache.org/streaming/

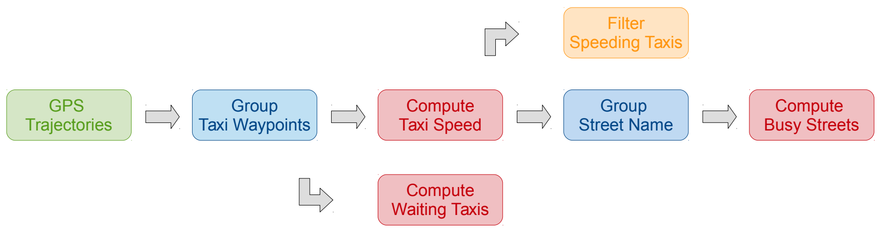

Fig. 2. The GPS trajectories of 10357 taxis and derived data streams

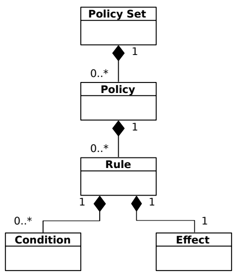

Fig. 3. Policy language

Spark Streaming processes streaming data as a series of deterministic batch computations on small time intervals. The basic abstraction of Spark Streaming is the *Discretized Stream* (D-Stream), which is a continuous sequence of RDDs. Spark Streaming receives live input data streams. If the time interval ends, the corresponding batch of input data is stored in a Resilient Distributed Dataset (RDD). The RDD is then processed via Spark's deterministic parallel operations, such as *filter()*, *map()*, *reduce()* and *foreach()*, to produce a new batch of processed data as an intermediate D-Stream of the program, as depicted in Figure 1. Note, however, that no computation is carried out on a D-Stream at runtime, unless a *output operator* is called. For D-Streams, *foreachRDD(func)* is the fundamental output operator that applies a function *func* to each RDD in the stream. Output operators are the only ones that can have side effects to print each RDD, save them to files, or transfer each RDD over the network to subscribers.

In the following sections, we will elaborate on how we implement access control with an additional layer on top of D-Streams to avoid unauthorized access to both the original and intermediate data streams (see Figure 2).

C. Key concepts of ABAC and XACML

Attribute-based access control (ABAC) is a paradigm that grants access rights to users through the use of policies or rules which combine various types of attributes. XACML is such a policy-based and attribute-based access control standard. A *policy set* in XACML represents a container of other policy sets and policies (see Figure 3). A *policy* consists of zero or more rules. Each *rule* can specify certain *conditions* as additional constraints to determine whether a rule applies and the desired *effect* (i.e. *permit* or *deny*). A *policy set*, *policy* and *rule* can define a *target* which lists the conditions that determine whether a policy applies to a particular request, based on a set of attributes of:

- A *resource* defines the data, system component or service to be accessed.
- The *subject* is the actor who makes a request to access a certain resource.
- The *action* declares the operation (e.g. read, write, update, delete) on the resource for which permission is requested.
- The *environment* is a set of attributes (independent of a particular subject, resource or action) that are relevant to an authorization decision.

Additionally, a policy set declares a policy combination algorithm (e.g. *permit overrides* and *deny overrides*) and a policy declares similar rule combination algorithm.

IV. Access control in SparkXS

This section will elaborate how we used D-Streams to implement access control to the different data streams in Figure 2. Listing 1 illustrates how these D-Streams were implemented using the Spark Streaming framework using lambda expressions and map-reduce data processing methods.

A. Authentication of the subjects

Consider again the motivating use case in section 3.1. In the policies, we identified 4 types of *subjects* with the following usernames: *admin, manager, taxi<id>, callcenter*.

Before requesting access to a data stream, each user must authenticate himself. Our implementation makes use of ForgeRock's OpenAM identity and access management (IAM) platform[6]. As part of the authentication step, SparkXS can fetch and cache additional attributes, such as the *role* or the *organization* of the *manager* and the *taxi<id>* users.

B. Authorizing access to data streams

Revisiting the 6 policies defined in section 3.1, access is enforced based on 4 categories of attributes:

- **Subject**: The admin, the manager, the callcenter as well as all the taxi drivers.
- **Resource**: The different D-Stream data streams as depicted in Figure 2.
- **Action**: Only the *read* and *write* operations in the policy examples.
- **Environment**: The current time and working hours at the call center.

These attributes are used in the *request* to decide on granting access based on predefined set of policies.

In the following subsections, we will define the attribute-based access control policies and rules that govern authorization to access streaming data.

[6]http://openam.forgerock.org/

Listing 1. Definition of D-Streams for GPS trajectories and derived data streams

```
// Initialize the TaxiXS streaming application that processes data in batches of 10 second intervals.
SparkConf sparkConf = new SparkConf().setAppName("TaxiXS");
JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, new Duration(10000));

// Process the taxi log files (sorted by time) with a custom data stream receiver.
JavaReceiverInputDStream<String> lines = ssc.receiverStream(new TaxiXS("datastream.txt"));

// Parse 'comma separated value' text line and convert it into a data object.
JavaDStream<Trajectory> gpslocations = lines.map(e −> parse(e));

// Group the GPS location entries in each batch per taxi_id as waypoints for each 10 second time interval.
JavaPairDStream<String,Trajectory> waypoints = gpslocations.mapToPair(e −> new Tuple2<String,Trajectory>(e.taxi_id, e));

// Compute the maximum speed for each 10 second time interval through pairwise reduction with a lambda function using the associative
// and commutative maxspeed() function, and select the second attribute of the resulting tuples.
JavaDStream<Trajectory> speed = waypoints.reduceByKey((a, b) −> maxspeed(a, b)).map(t −> t._2);

// Filter taxis that drive too fast by selecting those location elements with a speed value that crosses the given threshold.
JavaDStream<Trajectory> speedingtaxi = speed.filter(e −> e.speed > 50);

// Compute for each 60 second time interval the maximum distance travelled during the past 180 seconds. The distance is again computed
// through pairwise reduction with the maxdistance() function. Filter those taxis that drove less than 10 meters during these 3 minutes.
JavaDStream<Trajectory> waitingtaxi = waypoints.reduceByKeyAndWindow((a, b) −> maxdistance(a, b), new Duration(180000),
    new Duration(60000)).map(t −> t._2).filter(e −> e.distance < 10);

// Group the taxi locations and speeds in each batch per street name.
JavaPairDStream<String,Trajectory> streets = speed.map(e−>addstreet(e)).mapToPair(e−>new Tuple2<String,Trajectory>(e.street, e));

// Filter busy streets where taxis drive at less than 10 km/h.
JavaDStream<Trajectory> busystreets = streets.reduceByKey((a, b) −> maxspeed(a, b)).map(t −> t._2).filter(e −> e.speed < 10);
```

1) Policy 1: The *admin* of SparkXS has super-privileges to access all streaming data of all taxis of all organizations. We will use the following simplified syntax to express the policy:

```
if (subject.id == 'admin') return Permit;
```

The decision to grant access only depends on the *subject*'s attributes, and not on the content of the data stream (i.e. the *resource*), allowing for efficient policy evaluation.

2) Policy 2: Each *taxi driver* has full access to his own data stream, but not to those of other taxi drivers.

```
if ((action.type == 'read') && (subject.role == 'taxidriver'))
   return ConditionalPermit(e −> e.taxi_id == subject.id);
```

As some streams contain data from different taxi drivers, this policy returns a conditional permit that requires the Policy Enforcement Point to execute additional filtering before streaming out the data stream. In XACML terminology, this additional filtering would be called an *obligation*.

3) Policy 3: Only *taxi drivers* of registered organizations are allowed to upload their location data to SparkXS and backend services.

```
if ((action.type == 'write') && (subject.role == 'taxidriver')
   && ([ 'speedytaxi', 'yellowcab', 'airporttaxi', 'limotaxis' ].
        contains(subject.organization)))
   return Permit;
```

The *organization* attribute of the subject is assigned after authentication of the user and fetched from the Policy Information Point.

4) Policy 4: The *call center* has access between 7:00 AM and 6:00 PM to location data of taxis that are waiting for passengers (have not moved for 3 minutes).

```
if ((action.type == 'read') && (subject.id == 'callcenter')) {
    if (environment.currtime < 7.00)
        return Deny;

    if (environment.currtime > 18.00)
        return Deny;

    if (resource.id = 'waitingtaxi')
        return Permit;
}
```

Rather than writing one long rule, we can decompose the policy in individual smaller rules. The *currtime* attribute of environment is computed automatically.

5) Policy 5: The *manager* has access to taxi locations of his organization if his taxis cross the speed limit 50 km/h.

```
if ((action.type == 'read') && (subject.role == 'manager')) {
    if (resource.id == 'speedingtaxi')
        return ConditionalPermit(e −> subject.organization ==
            user(e.taxi_id).organization);
}
```

This policy declares a conditional permit with additional stream processing to filter speeding taxis that belong to a particular organization. Note that this policy uses a function *user()* to retrieve an individual's cached attributes from the Policy Information Point.

6) Policy 6: The *call center* has access to traffic and congestion info, based on different taxis speeds on busy streets.

```
if ((action.type == 'read') && (subject.id == 'callcenter')) {
    if (resource.id == 'busystreets')
        return Permit;
}
```

A more effective declaration of this policy would be to combine it with Policy 4.

7) Default deny policy: A policy that we did not explicitly listed before is the *default deny policy* that refuses unauthorized access:

```
return Deny;
```

This policy is enforced when no other policy applies, i.e. under unspecified conditions.

C. *Lazy processing of streaming data*

Our proof-of-concept exposes the D-Streams as RESTful interfaces with CRUD mappings. For example, to request *read* access to the *busystreets* data stream, the user would initiate the following HTTP GET request:

```
Read: http://host.com/taxixs/busystreets?token=AIC5wNT31

GET /taxixs/busystreets?token=AIC5wNT31 HTTP/1.1
Host: host.com
Accept: */*
```

The *AIC5wNT31* parameter represents the random authentication token that binds the subject for subsequent operations that require authentication and authorization. In practice, these authentication tokens are much longer.

To upload GPS trajectories, the taxi driver would initiate a HTTP POST request like the following:

```
Write: http://host.com/taxixs?token=AIC5wNT31

POST /taxixs?token=AIC5wNT31 HTTP/1.1
Host: host.com
Accept: */*
Content-type: application/text
Content-Length: 40

1,2008-02-02 15:36:08,116.51172,39.92123
```

The *update* and *delete* actions are not considered in the previous examples, but follow the same mapping on HTTP PUT and HTTP DELETE respectively.

After authentication, access to the different data streams is enforced by the *Policy Enforcement Point* based on the outcome of the *Policy Decision Point*. If *read* access is granted, the *foreachRDD()* output operator is invoked on the requested data stream.

```
// Print each RDD in the 'busystreets' data stream
// foreachRDD() is output operator with side effects, no results
busystreets.foreachRDD(rdd −> {
    System.out.println("Busy Streets RDD: " + rdd.id() + " = "
        + rdd.collect());
    return null;
});
```

In the above example, we invoke *System.out.println()* to produce the stream's side effect. In our proof-of-concept, however, we stream out the data in each RDD using HTTP Server-Sent Events (SSE).

If there is no output operator like *foreachRDD()* applied on the *busystreets* D-Stream, the preceding D-Streams (e.g. the *speed* data stream) will not be computed. As such, the data streams are processed in a lazy way, i.e. whenever access to the data stream is requested and/or granted.

D. *Parallel and distributed evaluation of access control policies with D-Streams*

In the previous section we defined the policies using a simplified syntax. In practice, the attribute-based access control policies are converted into Java 8 lambda expressions and map-reduce invocations for enhanced parallel evaluation on distributed multi-core systems[7].

Listing 2 gives a (partial) overview of the policy evaluation. It illustrates a policy set *policySet1* with 3 policies *policy1, policy2* and *defaultDeny* (lines 14-17), and how this policy set is applied on a request *request1* (line 21) with a *permitOverrides* combining algorithm (line 22).

The example above illustrates how we evaluate a single request. To deal with a high-volume of access requests, we combine and evaluate them as D-Streams of 0.5 second batch intervals, i.e. they are grouped per 0.5 second intervals and streamed as batches of RDDs that are then processed as outlined earlier.

V. Performance and Scalability Evaluation

We evaluated the streaming data motivating use case of section 3 with the policy-based access control using the policies of section 4.

A. *Experimental setup*

SparkXS makes use of Spark 1.0.1 and is built on top of the Spark Streaming extension. SparkXS is run with JDK 8 so that streaming computations and policy evaluations can make use of efficient lambda expressions. We use an experimental setup of 20 machines, each equipped with an Intel Core 2 Duo 3.00 GHz CPU and 4GB of memory. All machines are linked to a 1 Gigabit network.

- 10 machines are used to host SparkXS in a cluster configuration involving one master node for coordination and 9 worker nodes.
- We use an additional 5 machines to simulate the GPS trajectories of the taxi drivers using the T-Drive trajectory data set [24].
- We use another 5 machines to initiate various data access requests for the different data streams processed by the SparkXS cluster.

The last 10 machines in fact simulate the load on the SparkXS cluster setup. Each taxi driver uploads his current location and requests write permission to do so. The other stakeholders (callcenter, admin and manager) request read permission to access certain D-Streams.

B. *Methodology*

To systematically compare the performance and scalability results of different configurations under different loads (i.e. growing number of taxi drivers), we benchmark 2 different types of data streams, (a) the GPS location updates and derived streams (see Figure 2), and (b) streams with access requests to the former streams. A cluster of worker nodes will process these records in RDD batches of 1 second. We increase the load, i.e. the number of records per second, until the RDD has too many elements to be processed in real-time. If processing

[7]http://openjdk.java.net/projects/lambda/

Listing 2. Parallel and distributed policy evaluation with Java 8 lambda expressions and map-reduce

```
enum Decision { Permit, Deny, NotApplicable, Indeterminate; }

@FunctionalInterface
interface Policy { Decision evaluate(Request r); }

public static Decision permit(boolean condition) { return condition ? Decision.Permit : Decision.NotApplicable; }
public static Decision deny(boolean condition) { return condition ? Decision.Deny : Decision.NotApplicable; }
public static Decision evaluate(Request request, List<Policy> policySet, BinaryOperator<Decision> combine) {
    return policySet.parallelStream().map(p −> p.evaluate(request)).reduce(Decision.NotApplicable, combine); }

BinaryOperator<Decision> permitOverrides = (d1, d2) −> {
    return (d1 == Decision.Permit || d2 == Decision.Permit ? Decision.Permit : (d1 == Decision.NotApplicable ? d2 : d1)); };

Policy policy1 = (req −> permit(req.subject.id.equals("admin")));
Policy policy2 = (req −> conditionalPermit(req.action.type.equals("read") && req.subject.role.equals("taxidriver"),
    (e −> e.taxi_id.equals(req.subject.id))));
Policy defaultDeny = (req −> deny(true)));

List<Policy> policySet1 = new ArrayList<Policy>() {{ add(policy1); add(policy2); add(defaultDeny); }};

Request request1 = Request.parse("{ 'subject': { 'id': 'admin' }}");
Decision decision1 = evaluate(request1, policySet1, permitOverrides);
```

Location updates/sec	**Minimum**	25^{th} **%ile**	**Median**	75^{th} **%ile**	**Maximum**
1000	263 msec	292 msec	*304 msec*	322 msec	376 msec
2000	356 msec	376 msec	*388 msec*	399 msec	427 msec
4000	510 msec	527 msec	*538 msec*	544 msec	571 msec
6000	666 msec	677 msec	*683 msec*	692 msec	721 msec
8000	720 msec	746 msec	*752 msec*	757 msec	775 msec
10000	842 msec	852 msec	*868 msec*	895 msec	942 msec
12000	919 msec	934 msec	*951 msec*	966 msec	1014 msec

TABLE I
PROCESSING TIME BY 1 WORKER NODE FOR INCREASING NUMBER OF LOCATION UPDATES PER SECOND

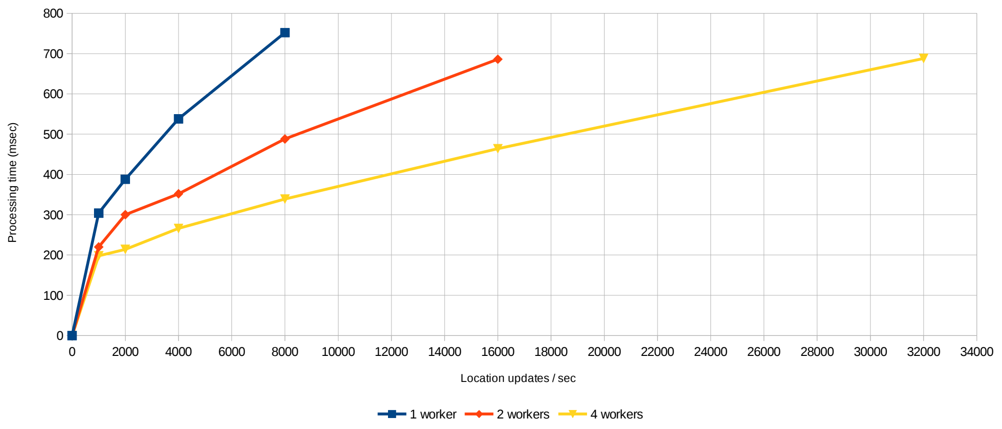

Fig. 4. Processing time (median) in msec for location updates with 1, 2 and 4 workers

an RDD takes more than 1 second, then processing the next batches will be delayed and this delay will keep on increasing.

C. Benchmarking the location streams

In a first experiment, we benchmark the scalability of Spark Streaming as a baseline by measuring its performance for processing the various data streams (see Figure 2) for a growing number of taxi driver location updates. As a worst case scenario, we enforce the explicit computation of the intermediate data streams by calling the *foreachRDD()* output operator on each of them. Table I shows a.o. the minimum, median and maximum performance values for processing one RDD on a single worker node. As input, it uses the log files of the taxi drivers, merged and sorted by time. SparkXS processes from 1000 up to 12000 lines of GPS locations per second. For 14000 location updates per second, we found the processing could no longer be executed in real-time.

Figure 4 depicts the median processing times for the same experiment with a growing number of worker nodes. Whereas the processing time at 8000 updates per second was 752 msec

Access requests/sec	**Minimum**	25^{th} **%ile**	**Median**	75^{th} **%ile**	**Maximum**
10000	85 msec	101 msec	*104 msec*	107 msec	129 msec
20000	82 msec	123 msec	*127 msec*	130 msec	183 msec
40000	156 msec	206 msec	*211 msec*	216 msec	247 msec
60000	218 msec	316 msec	*331 msec*	338 msec	405 msec
80000	318 msec	415 msec	*466 msec*	479 msec	500 msec
160000	768 msec	876 msec	*989 msec*	1086 msec	1352 msec

TABLE II
PROCESSING TIME BY 1 WORKER NODE FOR INCREASING NUMBER OF ACCESS REQUESTS PER SECOND

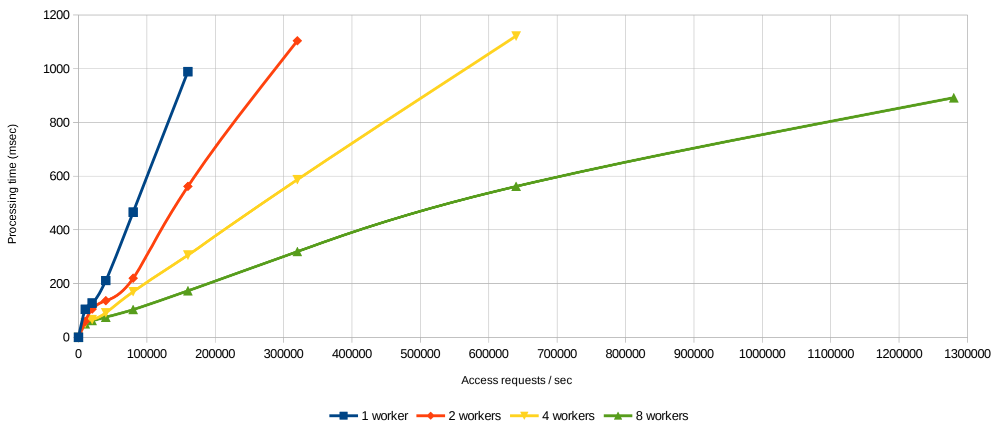

Fig. 5. Processing time (median) in msec for access requests with 1, 2, 4 and 8 workers

for 1 worker node, the performance was 488 msec and 339 msec for 2 and 4 worker nodes respectively. The figure also shows how SparkXS can easily process 32000 location updates per second on 4 worker nodes in real-time at around 688 msec per RDD.

D. Benchmarking the access request streams

The previous experiments illustrated the performance results for processing the GPS locations and intermediate data streams. In the following experiments, we will measure the impact of evaluating access control policies to these data streams. We do so by issuing a mixture of access requests, and have SparkXS evaluate them with the 6 policies mentioned in section 4.2. Note that the performance results will be different when either more or more complex access policies are used.

Table II lists the processing times for evaluation with access requests on a single worker node. These performance numbers reflect only the policy evaluation, and not the computation of the intermediate data streams of Figure 2. A single node can process up to 160000 access requests in real-time.

Figure 5 presents the results of similar experiments with up to 8 worker nodes. This graph illustrates how SparkXS can linearly scale with increasing number of access requests per second. In the given example, 8 worker nodes can process 1280000 requests in real-time at around 892 msec per RDD.

E. Discussion

Comparing both tables for single worker node deployments, we can observe that for this taxi driver use case, the capacity results for the policy evaluation are more than a magnitude higher than those for the actual location data streams. The same observations can be made for multi-node worker deployments.

In contrast to the XACML reference architecture, SparkXS merges the functionality of the Policy Decision Point and Policy Enforcement Point to enforce access control to streaming data in an efficient way. We can even further improve upon these results by caching policy evaluation results, such that e.g. *write* access requests for taxi drivers to upload their current location are not continuously evaluated. However, we also expect the processing time for access requests will increase with a growing number of access control policies. The number of policies can increase as long as they can be processed in realtime, as otherwise the incoming events would need to be buffered, which will fail when the system runs out of memory.

Figure 6 compares the amount of records (location updates vs. access requests) can be processed by 1, 2 and 4 worker nodes in 500 msec. This graph clearly shows the performance overhead for processing access requests is minimal compared to the location updates baseline.

In summary, the experimental results show that SparkXS can enforce access control in a horizontally scalable way in real-time and with minimal performance overheads. These unique features make our SparkXS solution very well suited for access control in an Internet of Things ecosystem. We have looked for similar access control systems for streaming applications, but beyond some works that do query rewriting – a fundamentally different and less practical approach – we

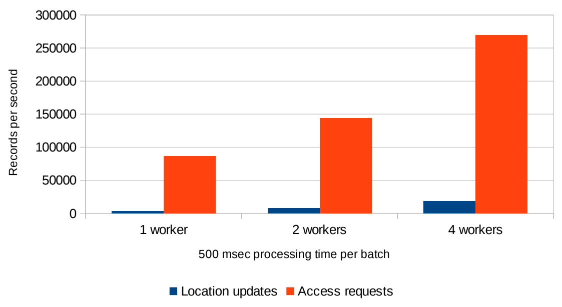

Fig. 6. Capacity comparison with multiple worker nodes given a 500 msec processing time constraint

found no alternative solutions with which we could compare the performance of SparkXS.

VI. Conclusion

Security and privacy are key challenges in the Internet of Things, and there is a growing need for new enabling technologies for access control that can handle the exponential explosion and streaming nature of data in the Internet of Things. In this paper, we presented SparkXS, an attribute-based access control solution built on top of the Apache Spark Streaming framework to enforce access control policies on incoming and intermediate data streams. An additional benefit is when access is not granted, the intermediate data stream is not computed.

Experimental results in a worst case scenario with a motivating use case in the area of urban computing show that SparkXS can enforce access control in a horizontally scalable way in real-time and with minimal performance overheads. The required capacity is less than 10% of the capacity needed for the baseline computation for the intermediate data streams.

Future work will evaluate SparkXS on use cases with more extensive access policies, and measure the performance impact of different serialization mechanisms and replication strategies. Due to the heterogeneous nature of IoT ecosystems, we will explore trade-offs between scale-up (vertical scalability) and scale-out (horizontal scalability). Additionally, we will improve tool support such that policies defined as lambda expressions can be easily modified and dynamically and efficiently loaded at runtime through reflection.

Acknowledgment

This research is partially funded by the Research Fund KU Leuven.

References

[1] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[2] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ser. ICDMW '10, Washington, DC, USA, 2010, pp. 170–177.

[3] J. Leibiusky, G. Eisbruch, and D. Simonassi, *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly, 2012.

[4] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in Action*. Greenwich, CT, USA: Manning Publications Co., 2011.

[5] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Commun. ACM*, vol. 54, no. 6, pp. 114–123, 2011.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010.

[7] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[8] M. Stonebraker, U. Cetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, Dec. 2005.

[9] M. Barlow, "Real-Time Big Data Analytics: Emerging Architecture," O'Reilly, Tech. Rep., Jun. 2013. [Online]. Available: http://www.oreilly.com/data/free/big-data-analytics-emerging-architecture.csp

[10] G. Cugola and A. Margara, "Processing Flows of Information: From Data Stream to Complex Event Processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.

[11] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," in *Very Large Data Bases*, 2013, pp. 734–746.

[12] R. S. Sandhu, "Lattice-Based Access Control Models," *Computer*, vol. 26, no. 11, pp. 9–19, Nov. 1993.

[13] R. Sandhu and P. Samarati, "Access control: principle and practice," *Communications Magazine, IEEE*, vol. 32, no. 9, pp. 40–48, Sept 1994.

[14] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[15] X. Jin, R. Krishnan, and R. Sandhu, "A Unified Attribute-based Access Control Model Covering DAC, MAC and RBAC," in *Proceedings of the 26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy*, ser. DBSec'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 41–55.

[16] S. Kandala, R. Sandhu, and V. Bhamidipati, "An Attribute Based Framework for Risk-Adaptive Access Control Models," in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, Aug 2011, pp. 236–241.

[17] Q. Ni, E. Bertino, and J. Lobo, "Risk-based Access Control Systems Built on Fuzzy Inferences," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '10. New York, NY, USA: ACM, 2010, pp. 250–260.

[18] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, ser. POLICY '01. London, UK, UK: Springer-Verlag, 2001, pp. 18–38.

[19] G. Tonti, J. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok, "Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder," in *International Semantic Web Conference*, ser. Lecture Notes in Computer Science, D. Fensel, K. P. Sycara, and J. Mylopoulos, Eds., vol. 2870. Springer, 2003, pp. 419–437.

[20] XACML-V3.0, "eXtensible Access Control Markup Language (XACML) Version 3.0. Candidate OASIS Standard 01," http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cos01-en.html, September 2012. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cos01-en.html

[21] B. Carminati, E. Ferrari, J. Cao, and K.-L. Tan, "A framework to enforce access control over data streams." *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 3, 2010.

[22] R. V. Nehme, H.-S. Lim, and E. Bertino, "Fence: Continuous access control enforcement in dynamic data stream environments," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 243–254.

[23] Y. Zheng, L. Capra, O. Wolfson, and H. Yang, "Urban Computing: Concepts, Methodologies, and Applications," *ACM Transaction on Intelligent Systems and Technology*, 2014.

[24] Y. Zheng, "T-drive trajectory data sample," http://research.microsoft.com/apps/pubs/default.aspx?id=152883, August 2011.