future internet

MDPI

Article

Access Control with Delegated Authorization Policy Evaluation for Data-Driven Microservice Workflows

Davy Preuveneers * and Wouter Joosen

imec-DistriNet-KU Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium; wouter.joosen@cs.kuleuven.be
* Correspondence: davy.preuveneers@cs.kuleuven.be; Tel.: +32-16-327853

Received: 31 August 2017; Accepted: 25 September 2017; Published: 30 September 2017

Abstract: Microservices offer a compelling competitive advantage for building data flow systems as a choreography of self-contained data endpoints that each implement a specific data processing functionality. Such a 'single responsibility principle' design makes them well suited for constructing scalable and flexible data integration and real-time data flow applications. In this paper, we investigate microservice based data processing workflows from a security point of view, i.e., (1) how to constrain data processing workflows with respect to dynamic authorization policies granting or denying access to certain microservice results depending on the flow of the data; (2) how to let multiple microservices contribute to a collective data-driven authorization decision and (3) how to put adequate measures in place such that the data within each individual microservice is protected against illegitimate access from unauthorized users or other microservices. Due to this multifold objective, enforcing access control on the data endpoints to prevent information leakage or preserve one's privacy becomes far more challenging, as authorization policies can have dependencies and decision outcomes cross-cutting data in multiple microservices. To address this challenge, we present and evaluate a workflow-oriented authorization framework that enforces authorization policies in a decentralized manner and where the delegated policy evaluation leverages feature toggles that are managed at runtime by software circuit breakers to secure the distributed data processing workflows. The benefit of our solution is that, on the one hand, authorization policies restrict access to the data endpoints of the microservices, and on the other hand, microservices can safely rely on other data endpoints to collectively evaluate cross-cutting access control decisions without having to rely on a shared storage backend holding all the necessary information for the policy evaluation.

Keywords: authorization; policy-based access control; microservices; data processing pipelines; workflows; feature toggles; circuit breakers

1. Introduction

With the advent of trends like the Internet of Things [1], distributed software systems increasingly obtain more useful information about users and their environment. Context-aware computing [2,3] is therefore becoming a game changer for businesses, as a timely procurement (selection, processing and delivery) of complex context information enables them to deliver smarter systems and applications that can accommodate their users' varying needs. Our batch and streaming context analytics architecture SAMURAI [4] was developed with these objectives in mind. Due to the above paradigm shifts, enterprises are exploring big data [5] and analytics technology [6] to (1) identify useful data and (2) transform that data to identify useful patterns and deviations from patterns in real time. Any contemporary data analytics architecture must be able to analyze in real time heterogeneous static silos of data and dynamic data streams. Distributed stream processing systems like Storm [7] and Heron [8] are frequently used for real-time analysis, online machine learning and continuous computing. As also identified by researchers at Facebook [9], the ease of use, performance, fault

tolerance, scalability and correctness are five important design decisions for real-time data stream processing systems. Two architectural styles for real-time data processing are the Lambda architecture (http://lambda-architecture.net, by Nathan Marz, creator of Storm) [10] and the Kappa architecture (http://kappa-architecture.com, by Jay Kreps, from LinkedIn). They both deliver fault-tolerant and scalable data processing with support for incremental data updates. However, the Lambda architecture has received some fair criticism [11] on the coding overhead due to the necessity of maintaining two code bases for the batch and speed layers. The Kappa architecture aims to avoid this pitfall by handling both the real-time data processing and the reprocessing of (historic) data using a single stream processing engine. Beyond this observation, many data processing platforms that follow a Lambda or Kappa architectural style also often assume a single albeit distributed and scalable storage system for processing large amounts of data.

However, the design trade-offs of monolithic Lambda and Kappa architectures may jeopardize the flexibility to quickly change the data processing workflows to evolving needs and requirements. That is why enterprises such as Netflix are adopting microservice architectures [12,13] to build data-intensive applications, mainly to address managerial concerns with the development and deployment life cycle of monolithic service architectures. Proposed by James Lewis and Martin Fowler, the basic principles behind microservices are that (1) a microservice only deals with a restricted and clearly-defined set of functions, (2) each microservice is developed independently of the most appropriate programming language, (3) each microservice is stateless and manages its own database and (4) each microservice runs autonomously in its own process. With microservices, applications are developed as compositions of smart data processing endpoints and dumb pipes, as depicted in Figure 1. The low coupling and high cohesion of such distributed choreographies of independent microservices simplify the introduction of new functionalities in the data processing workflow, and they allow for each microservice to be developed, deployed, modified and scaled out independently, a clear advantage for dynamic cross-enterprise collaborations [14].

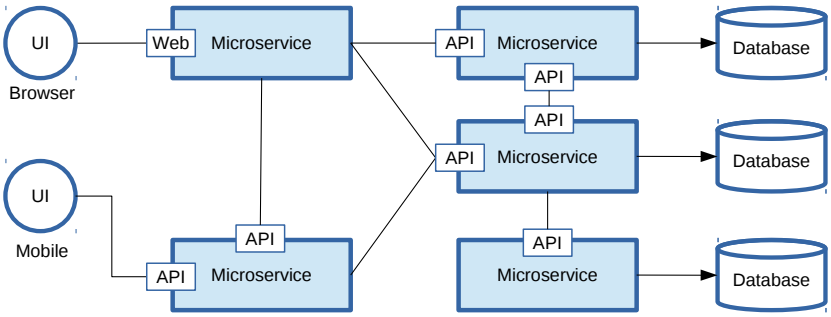

Figure 1. High-level decomposition of a microservice based application with various user interfaces (UI) and application programming interfaces (API).

In this paper, we research the feasibility of decentralized microservice architectures to develop flexible and scalable security analytics applications. More specifically, we investigate to what extent microservices can be leveraged to realize a policy-based access control architecture where the access policies are defined in terms of data-intensive authorization decisions. For example, multiple microservices may process different streams of data and contribute their risk assessments to influence the final and joint decision whether a subject is granted or denied access to an online resource. In such a deployment scenario, each microservice carries out its own risk assessment—using a rule engine, complex event processing, statistical analysis, machine learning algorithms, etc.—based on the parameters it receives or monitors and the historic information previously processed and stored in its own database. These individual assessment outcomes are then combined in an authorization policy that evaluates a set of authorization rules to come to the final access control decision.

The advantages of pursuing a microservice-based design are that (1) each individual data processing service can be deployed close to the system, application or user being monitored and that (2) it can be scaled in terms of the amount of information it needs to process; and this irrespective of the other microservices. Additionally, (3) the decentralized design spreads the risk of data breaches and reduces the ability of a data broker to mine for sensitive information, as a single microservice only provides access to a subset of the data. Indeed, monolithic applications prefer a single database for persistent data, whereas microservices manage their own database on top of possibly different database technologies. A consequence of the decentralized governance and data management, however, is that enforcing access control on the data endpoints to prevent information leakage or preserve one's privacy becomes far more challenging. Indeed, on the one hand, authorization policies should restrict access to the data endpoints of the microservices itself, but on the other hand, they rely on the same data endpoints to evaluate their access control decisions, as we will later demonstrate.

The contribution of this paper is a workflow-oriented authorization framework that mutually authorizes microservices to grant access to each others' data endpoints based on the state of the data processing workflow; and this while protecting against illegitimate access from unauthorized subjects and microservices. Our framework also enables the evaluation of authorization policies in a decentralized manner where the delegated policy evaluation leverages feature toggles that are managed at runtime by software circuit breakers to decouple the execution of the data processing workflow from the policy-based authorization logic.

The remained of this paper is structured as follows. Section 2 provides background information on the microservice building blocks that are used in our framework. Section 3 presents a motivating use case in a healthcare context. The core and implementation details of our workflow-based authorization framework are presented in Section 4 and evaluated in Section 5. We end the paper by summarizing the main contributions of our work and highlighting opportunities for future research in Section 6.

2. Background

As our policy-based authorization framework leverages microservices, this section discusses feature toggles and circuit breakers, two concepts that are not a contribution of this research, but that we use to decouple the data processing workflow from the policy-based authorization logic.

2.1. Feature Toggles

Feature toggles [15] had been initially proposed in the frame of continuous software delivery to incrementally integrate new features into existing applications or to test bug fixes. Feature toggles enable rapid software releases while reducing the probability of integration conflicts. That is why a major company like Google is using feature toggles for the release of different branches of its Chrome browser. Even after the final software release, these feature toggles can be used to only enable certain features for specific users or to quickly disable a feature that is misbehaving.

As depicted in Figure 2, a feature toggle is typically a variable that is used in a conditional statement to enable or disable a piece of code for testing or release purposes. In the past, such feature toggles were used at compile time to exclude certain features from the application binary. Modern feature toggles, however, allow for switching them at runtime without recompilation of the software. Martin Fowler (https://martinfowler.com/articles/feature-toggles.html) describes several types of feature toggles that mainly differ in dynamism and longevity. These feature toggles include 'release toggles' to disable incomplete or untested code paths in production systems, 'experiment toggles' for multivariate user testing of different code paths and 'ops toggles' to control operational aspects of a system's behavior.

To support workflow-oriented authorization, we enable and disable features or code blocks using ops toggles to modify the runtime configuration of the system. Each microservice that is feature toggle enabled provides a configuration file, which declares all the feature toggle definitions and their default state. Feature toggles require a mechanism to switch their state. For traditional applications, this can

be a program startup parameter or a runtime setting in the user interface that can be manually changed by the product manager or the end-user. For our access control framework where data processing workflows adapt at runtime, it is not a human being that manipulates these feature toggles. Instead, our microservices typically offer a RESTful API to enable or disable certain features and rely on the dynamic authorization policies to switch these feature toggles.

```
// Feature toggle
boolean enableCacheSupport = true;

int getRiskScore() {
  if (enableCacheSupport) {
    getCachedRiskScore();
  } else {
    computeRiskScore();
  }
}

int computeRiskScore() {
  /* Compute and return a fresh result */
}

int getCachedRiskScore() {
  /* If available return recent cached result, else delegate to computeRiskScore() */
}
```

Figure 2. Feature toggles for returning cached results of a data-intensive process to improve performance.

2.2. Circuit Breakers

Software-based circuit breakers [16] work in a similar fashion to the electrical ones that toggle a magnetic switch to protect appliances from excess current on the power lines. In such failure scenarios, the circuit breaker trips from the closed into the open state to break the flow of the current to the appliance. This way, it protects the appliance from breaking down completely.

We use a software variant of circuit breakers (1) to protect distributed data processing workflows from unauthorized access by other parties or (2) to disable the execution of the data procession functionality of the microservice. We group sets of authorization rules per feature toggle into an authorization policy and encapsulate the evaluation of the policy in a circuit breaker. The state transitions of a circuit breaker are depicted in Figure 3. Under normal operations, the circuit breaker is closed. If authorization is denied, the circuit breaker trips and switches from the closed into the open state. As a result, it disables external access to the data processing functionality of the microservice, as well as the database it encapsulates and the corresponding feature toggle from being switched.

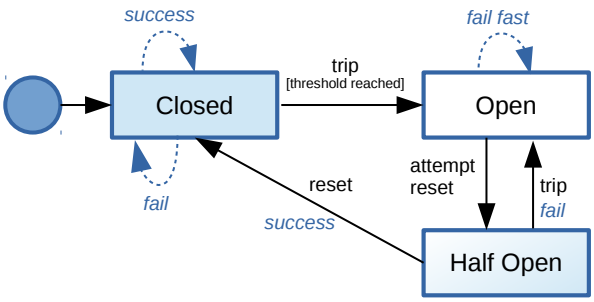

Figure 3. State transitions of a basic circuit breaker.

The software circuit breaker can also cache authorization decisions for a period of time (i.e., a back-off period) during which the previous decision is automatically returned without re-evaluating the authorization policies. To do so, after the back-off period has passed, the circuit breaker automatically shifts from the open into the half-open state, and if the next evaluation of the authorization policy succeeds, the circuit breaker resets to the closed state. If access is denied again, then the circuit breaker reverts back from the half-open to the open state until the next timeout of the back-off period occurs. Figure 4 provides a high-level Java implementation for dynamic access control evaluation at runtime. It provides the two functionalities a typical circuit breaker has to offer, i.e., the trip() and reset() methods, with a back-off period of 10,000 ms to attempt a reset whenever the circuit breaker changes to the State.OPEN state.

```
private enum State { OPEN, HALF_OPEN, CLOSED }
private int backoffPeriod = 10,000;
private State state = State.CLOSED;

public synchronized void reset() {
  state = State.CLOSED;
}

public synchronized void trip() {
  state = State.OPEN;
  new Timer().schedule(new TimerTask() {
    public void run() {
      state = State.HALF_OPEN;
    }
  }, backoffPeriod);
}

public synchronized void execute() {
  if (state == State.OPEN)
    return;

  try {
   // Evaluate the authorization rules and policies
   evaluateAuthorizationPolicies();
   // Access is granted, so reset circuit breaker
   reset();
  }
  catch (Exception e) {
   // Timeout error or access is denied, so trip circuit breaker
   trip();
  }
}
```

Figure 4. Java-based implementation of a circuit breaker.

The caching feature of circuit breakers is useful when the evaluation of authorization policies is computationally expensive. This is, for example, the case when a risk-based access control decision relies on processing large amounts of data to quantify the risk and where the continuous evaluation of such authorization policies would jeopardize the overall performance of the data processing workflow.

3. A Motivating Use Case for Healthcare

In healthcare systems, authorization is put in place to manage access control to sensitive data, such as a patient's medical records or data collected by mobile health trackers. However, medical processes are often represented as workflows involving multiple stakeholders. As a result, authorization too involves complex procedures (e.g., patient consent, approval delegation of authority, break-the-glass, audit access logs, need-to-know), and these procedures are not easily implemented as a set of access rules. For example, rather than just evaluating a rule to verify whether a patient has given consent in the past, the authorization system may explicitly ask the patient for consent. In this case, patient consent can be implemented as a task in an authorization workflow. However, in emergency situations, patient consent may be implicit as a result of a break-the-glass procedure, again a task where a healthcare professional (i.e., the subject) can override access to read (i.e., the action) the patient's medical data (i.e., the resource) under the condition that he/she motivates why he/she triggered the break-the-glass procedure. Last but not least, the need-to-know principle states that even if a healthcare professional has the necessary approvals (such as patient consent), the subject should not be given access to medical data unless there is a specific need (so not only who and what, but also why) in order to avoid healthcare data exposure due to insider attacks. If a patient is enrolled in a consultation, he/she is part of a workflow, and only the medical actors involved in the same workflow may be granted access to (a subset of) that patient's data. As a result, the execution of the main medical workflow may be subject to the execution of a co-existing authorization workflow. Furthermore, certain subjects (e.g., trainees) may be prohibited from executing a break-the-glass authorization procedure. All these examples illustrate that externalizing access control in workflow-oriented applications is challenging.

Figure 5 provides a simplified overview, identifying the stakeholders, the data being collected and processed and the processes in place. Note that these services can be hosted by different online service providers, each with their own access control solutions. For example, the Mobile Health Tracker platform that collects and processes fitness and health data, may be hosted in the cloud by the company that sells the wearables. Similarly, the Patient Appointment scheduling system may be offered online as a multi-tenant cloud service with multiple doctor's offices as clients. The data being managed by the general practitioners themselves are the medical records and the user enrollment of the members of the doctor's office (e.g., physicians, trainees, dietists, secretaries, etc.).

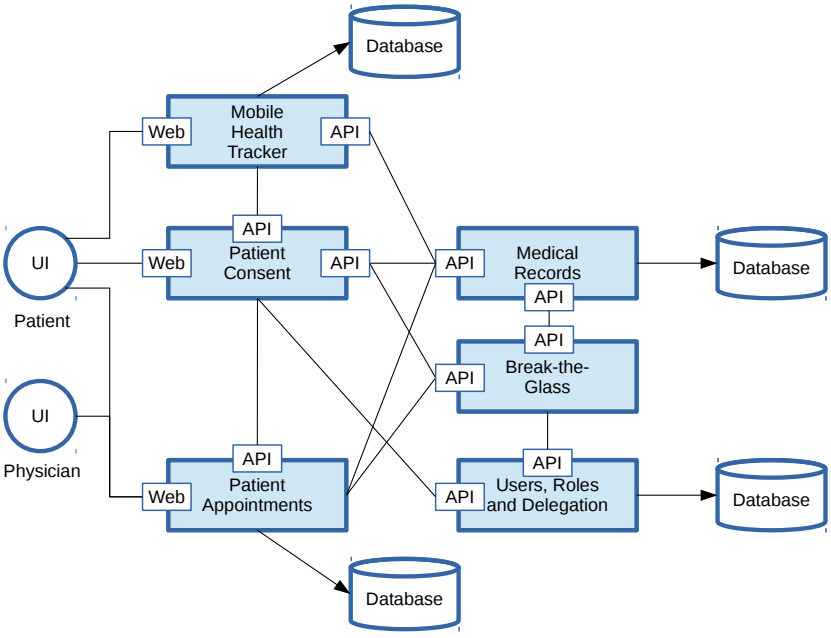

Figure 5. Authorization as a workflow in a healthcare context.

Here are some non-trivial access control use cases that cross-cut the different services. We will use them later on to illustrate our workflow-oriented authorization of microservices with feature toggles and circuit breakers:

1. After patient consent, the Medical Records system receives summaries of aggregated healthcare results of the Mobile Health Tracker platform, as well as detailed information (time, location, health parameters) when certain (aggregated) health parameters are above or below critical thresholds (e.g., heart rates, blood glucose values, etc.) defined by the physician. The latter should not be able to retrieve all data simply by changing the thresholds.
2. A physician should have a specific need to access sensitive data, even if he/she has the necessary consent and permissions. This need-to-know concern is reflected by the presence of a future patient appointment. However, the physician should not be able to override access simply by creating a patient appointment. The appointment should either be scheduled by the patient himself/herself or otherwise be approved by the patient. This is a typical example of a 'Separation of Duty' (SoD) [17] policy to prevent a single user from executing all critical tasks in a workflow.
3. Rather than having an authorization policy deny access, authorization workflows with human involvement may be triggered to grant access. For example, a workflow may request consent from the patient if missing. Another workflow may approve the delegation of authority for subordinates. When the break-the-glass service grants access, it will enforce an obligation workflow for healthcare professionals to motivate why access was overridden.

These healthcare scenarios demonstrate the need for authorization policies with dependencies and decision outcomes cross-cutting data in multiple microservices. They illustrate the necessity to dynamically grant or deny access to certain microservices depending on the flow of the data, with multiple microservices collectively making authorization decisions, while having adequate measures in place to prevent illegitimate access or privacy breaches.

4. Workflow-Based Access Control with Authorization Policies

Policy-based access control externalizes authorization from an application by evaluating a set of access rules to determine whether a subject is authorized to execute a particular action on a resource. In certain application domains, authorization resembles more the enforcement of a set of procedures rather than a set of rules that are evaluated atomically. This section describes the overall framework for enabling authorization in the form of workflows implementing these procedures as tasks, delegating policy evaluation and enforcing dynamic separation of duty in complex workflows [18].

4.1. Access Control Language

Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC) ([19–21]) are several well-known identity-based access control models. Recently, there has been growing interest in Attribute-Based Access Control (ABAC) ([22]), as it offers more expressivity and flexibility compared to the aforementioned access control models. The ABAC model grants access rights to subjects through the use of policies or rules that combine various types of attributes to facilitate user access to the right resources under the right conditions:

- 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 is relevant to an authorization decision.

In policy-based access control (with the eXtensible Access Control Markup Language (XACML) 3.0 specification ([23]) being the industry standard), 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. In XACML, a policy set represents a container of other policy sets and policies (see Figure 6). 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. Additionally, a policy set declares a policy combination algorithm (e.g., permit overrides and deny overrides), and a policy declares a similar rule combination algorithm.

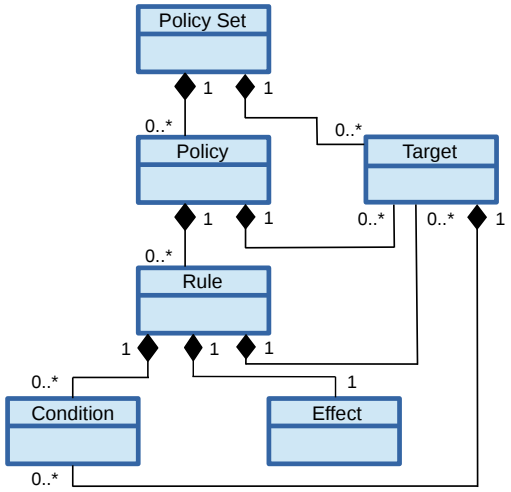

Figure 6. Metamodel of authorization policy language.

An XACML 3.0 policy is defined in XML and is therefore fairly complex to parse and modify. Additionally, for workflow-based authorization with delegated policy evaluation (each microservice evaluates part of the policy based on its own information), our framework must be able to frequently and dynamically rewrite authorization policies to reduce policy complexity after partial evaluation. As XACML is less suited for such purposes, we defined a more lightweight access control policy language based on the JavaScript Object Notation (JSON) specification leveraging the most important concepts as XACML, but with a few simplifications. Figure 7 illustrates our policy language by means of a small example. Each authorization policy has a unique identifier and a version number with an optional description that expresses the corresponding permissions in a human-understandable language. Resources are identified with a reference to the relevant microservice (in the example patientappointments) and path, and the four actions (i.e., read, write, update and delete) correspond to the typical RESTful Hypertext Transfer Protocol (HTTP) requests (i.e., GET, POST, PUT and DELETE).

Our policy language allows for variable binding in resource identifiers, as illustrated for the policyid variable. Based on the resource description of the access request, the variable will be instantiated so that it can be used in the remainder of the policy evaluation and, more particularly, in the evaluation of the conditions. Note how the resources attribute refers to a single entity. Our policy language also allows a list of resources in a similar way as for the actions attribute. The effect attribute is either permit or deny. The conditions attribute is a list of Boolean expressions with the implicit assumption that the conditions are evaluated in Conjunctive Normal Form (CNF), i.e., an AND of ORs. The current example has only one equality condition. Other condition expressions are !=, >, <, >= and <=. If multiple conditions are defined, they all have to be fulfilled. The first condition states that the variable patientid must be equal to the id attribute of the subject or to the admin user. If other conditions exist under which access should be granted, then this can be realized with another policy. Figure 8 depicts a policy set holding a list of policies. The salience attribute defines the evaluation order of the different policies.

```
{
  "id": "policy_123",
  "description": "authorization policy for patient appointments",
  "version": 1.0,
  "salience": 100,
  "policy": {
   // List of resources including all appointments of a given patient
    "resources": "patientappointments::${patientid}/appointments",
    "actions": [ "read", "write", "update", "delete" ],
    "effect": "permit",
    "conditions": [ {
        "=": {
          // The subject is either the patient himself OR the system administrator
          "subject::id": ["\${patientid}", "admin"]
        }
      }, {
        "=": {
          // AND the subject is the organizer of the appointment
          "resource::organizer": ["subject::id"]
        }
      }
    ]
  }
}
```

Figure 7. A basic access control policy example expressed in the JSON language.

```
{
  "id": "policyset_1",
  "version": 1.0,
  "policyset": [
    {
      "id": "policy_1",
      "version": 1.0,
      "salience": 100,
      "policy": {
        ...
      }
    }, {
      "id": "policy_2",
      "version": 1.1,
      "salience": 50,
      "policy": {
        ...
      }
    }, {
        ...
    },
    ...
  ]
}
```

Figure 8. A list of policies evaluated in order defined by the salience attribute.

Our framework imposes a default policy that denies access to all resources. If multiple policies exist for a given resource, all with a permit effect, then access is granted if the conditions of at least one policy hold (i.e., a permit-overrides policy combination). If one or more policies declare a deny effect, then the first policy whose conditions are fulfilled will determine the outcome (i.e., a first-applicable policy combination). This means that the order of policy evaluation plays a role. The salience attribute will determine the order in which policies are evaluated, i.e., policies with a higher salience value will be evaluated first. If the salience attribute is not specified, a default value of 100 is assumed, and policies with the same salience value are evaluated in a non-deterministic order. A schematic overview of the policy evaluation is given in Figure 9.

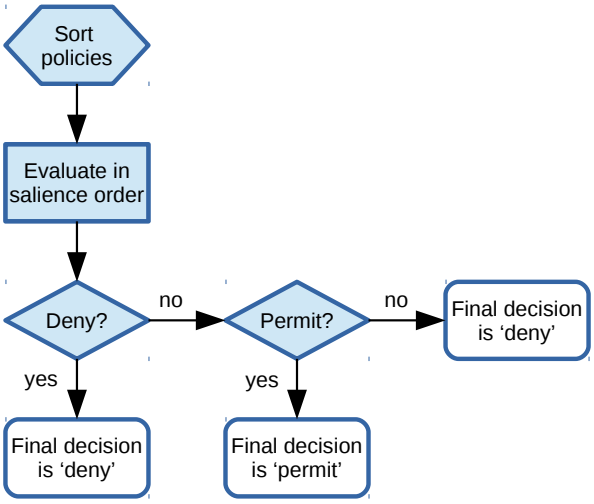

Figure 9. Policy evaluation flow.

In summary, the simplifications imposed by our JSON-based access control policy language with respect to the XACML specification are listed below:

- There is always a default and implicit deny policy
- Rule conditions are declared in conjunctive normal form
- No explicit definition of policy targets.
- Simplified policy combinations with salience values

A first advantage is that there is always a policy decision (i.e., permit or deny), whereas XACML policies may result in a indeterminate or notapplicable decision outcome. The conjunctive normal form has the advantage that if one of the constituents of the conjunction fails, then the whole conjunction is not fulfilled. In that case, the policy set (as depicted in Figure 8) can be reduced in complexity simply by removing that particular policy from the policy set. XACML policies typically define targets that indicate which conditions must be fulfilled before a policy set, policy or rule is evaluated. These conditions complement the ones of the rules that produce the permit or deny decision. Our policy language eliminates this dual constraint complexity, combining all the conditions in the corresponding conditions section of the policy (see Figure 7). Last but not least, these simplifications make it easier for policy editors to understand and reason about the correctness of a policy.

4.2. Policy Encapsulation with Feature Toggles and Circuit Breakers

To protect access to the data that each microservice stores and processes, the REST end-points are augmented with authorization policies to grant or deny access to the underlying data and functionality. These authorization policies evaluate decisions based on various attributes (e.g., the role of the subject requesting access). For these attributes, the following circumstances may arise:

- The values of all attributes used in the conditions of a policy are available in the data store of the microservice and require no further processing. Typical examples are attributes related to the local data resource itself (e.g., name of patient in medical records) or environment attributes (e.g., time and location of the request).
- Some attribute values are not available, but can be derived or computed from the data available in the data store (e.g., compute a risk score based on the anomalous nature of the request). The attribute may be computed on demand or cached if the processing is too resource intensive.
- Some policies depend on attributes whose values are being managed by other microservices, such as in the need-to-know example (granting access to medical records depends on the existence of a patient appointment; see Figure 10). However, requests for these attributes will also be subject to the authorization policies put in place for these third party microservices.

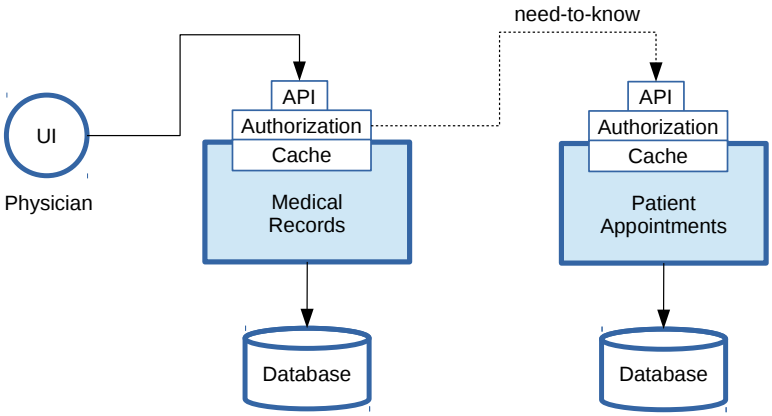

Figure 10. Policy evaluation depending on third party microservices.

Our framework relies on a feature toggle per policy to determine whether (1) attributes used in the conditions should be retrieved and/or computed on demand or whether cached results should be return and (2) whether a microservice accepts requests from other microservices to help resolve attributes. These feature toggles are put in place to avoid a given microservice being overloaded with authorization related requests, leaving no resources available for its core functionality.

The state of the feature toggles is updated at runtime by a single circuit breaker per microservice that monitors (1) the outcome of the authorization decision and (2) the latency incurred for computing the authorization decision. As partially depicted in Figures 3 and 4, the circuit breaker may return default deny decisions or cached decisions depending on the state in which it is. With this particular configuration, we aim for a fail fast scenario such that our framework is not flooded with access requests that would otherwise be denied if evaluated.

4.3. Delegated Policy Evaluation

Thus far, we discussed how an authorization policy of a microservice can be evaluated all on its own (possibly with returning cached results) or rely on another microservice to resolve missing information. However, as explained in Section 3, such partial and distributed policy evaluation may impose security risks. This section will discuss in more detail how delegated policy evaluation is implemented and illustrate the process by means of the three scenarios from the motivating use case.

4.3.1. Enforcing the Need-To-Know Principle through Delegation

The need-to-know concern imposed for accessing medical records requires that there is a future patient appointment, either scheduled by the patient himself/herself (and implicitly approved) or scheduled by the physician and explicitly approved by the patient. As illustrated in Figure 7, the Patient Appointment microservice offers a REST interface to retrieve all appointments for a given patient, as shown below:

```
''patientappointments::${patientid}/appointments''
```

When a read access request to a set of resources is issued (in this case, a list of appointments), only those resources are returned for which access is granted. A read access request can also target a specific resource, such as an appointment with a given identifier, as shown below:

```
''patientappointments::${patientid}/appointments/${appointmentid}''
```

In that case, either the appointment is returned if access is granted, or otherwise, nothing is returned if access is denied or the appointment does not exist.

The authorization policy in Figure 11 is put in place of the Patient Appointment microservice, stating that only the patient and the physician are allowed to read an existing and write a new appointment, and only the organizer of the appointment (patient or physician) can update an existing appointment. Let us assume that access to the medical records of a patient is granted according to the following simplified authorization policies:

- A patient can read his/her own medical records
- A physician (or any other healthcare professional) can read and write medical records of a patient only if he/she has established a therapeutic relationship with the patient
- A physician can only access the medical records he/she created himself/herself, unless the patient provided informed consent to share his/her medical records with other healthcare professionals

To enforce the need-to-know principle, the following policy with additional conditions must be checked before access is granted.

The policy in Figure 12 states that there must be an appointment in the future that the patient accepted, either implicitly as he/she scheduled it himself/herself or explicitly after someone else scheduled it. The patient acceptance is necessary as the mere existence of an appointment would allow the physician to schedule an appointment with a patient (with which he/she already has a therapeutic relationship), hereby bypassing the need-to-know access restriction of the Medical Records microservice.

A straightforward solution would be to augment the above policy to grant the Medical Records microservice read access to all patient appointments. However, this is a potential security or privacy risk as sensitive information is shared with other parties. In case of a data breach with a third party microservice, sensitive information might leak, even if the microservice from which the data originated is properly secured. Furthermore, such a solution would complicate the definition and the maintenance of the authorization policies whenever new microservices are added to the system.

Our solution is to define the need-to-know access restriction at the level of the Medical Records microservice, but to outsource the evaluation to the Patient Appointment microservice, hence the term 'delegated policy evaluation'. The policy evaluation then becomes a workflow of its own orthogonal to the data processing workflows on top of the core functionalities of the microservices.

```
{
  "id": "policyset_1",
  "version": 1.0,
  "policyset": [
   {
     "id": "appointmentpolicy_1",
     "version": 1.0,
     "salience": 100,
     "policy": {
       "resources": "patientappointments::${patientid}/appointments/${appointmentid}",
       "actions": [ "read", "write" ],
       "effect": "permit",
       "conditions": [
         {
           "=": {
             "subject::id": ["${patientid}", "physician"]
           }
         }
       ]
     }
   }, {
     "id": "appointmentpolicy_2",
     "version": 1.0,
     "salience": 100,
     "policy": {
       "resources": "patientappointments::${patientid}/appointments/${appointmentid}",
       "actions": [ "update" ],
       "effect": "permit",
       "conditions": [
         {
           "=": {
             "subject::id": ["resource::organizer"]
           }
         }
       ]
     }
   }
  ]
}
```

Figure 11. Authorization policy for the Patient Appointment microservice.

```
{
 "id": "needtoknow_1",
 "version": 1.0,
 "salience": 100,
 "policy": {
  "resources": "patientappointments::${patientid}/appointments",
  "actions": [ "read" ],
  "effect": "permit",
  "conditions": [
   {
    ">": {
     "resource::time": ["environment::time"]
    }
   }, {
    "=": {
     "${patientid}": ["resource::accepted"]
    }
   }
  ]
 }
}
```

Figure 12. Need-to-know authorization policy for the Medical Records microservice.

4.3.2. Authorizing Delegated Policy Evaluation

The previous need-to-know example illustrated how delegated policy evaluation can help with minimizing data breaches by outsourcing the policy evaluation rather than by sharing sensitive data across multiple microservices.

However, delegated policy evaluation may also impose risks as highlighted in the first example of the motivating use case. The Mobile Health Tracker platform submits aggregated healthcare results and contextual information (time and location) to the Medical Records system when certain (aggregated) health parameters are above or below critical thresholds. Such an example is illustrated in Figure 13.

From the point of view of the Medical Records microservice, this policy will again be evaluated through delegation to the Mobile Health Tracker microservice. For each new health parameter, the authorization policy should compute the average of the blood glucose values of the last 24 h. When this average value is above a threshold, defined by the physician, then access to all health parameters of the past 24 h is granted. However, if the physician can easily change the threshold, he/she might be able to retrieve all health parameters.

Our framework addresses this challenge by ensuring that the delegated evaluation of policies is subject to authorization, as well. More specifically, delegated policies that compute values on sensitive health parameters must be digitally signed by each owner of any resource being accessed. As the health parameters are owned by the patient, he/she must authorize the evaluation of the policy by digitally signing the policy. Our proof-of-concept implementation relies on identity-based cryptography [24] in which a publicly known string representing an individual is used as a public key so that each party can easily verify the signature based on the '${userid}' identifier. All authorizations of a policy are included in the corresponding field of the delegated policy.

```
{
  "id": "needtoknow_1",
  "version": 1.0,
  "salience": 100,
  "policy": {
   "resources": "mobilehealthtracker::${userid}/healthparameters",
   "actions": [ "read" ],
   "effect": "permit",
   "conditions": [
     {
       // Daily blood glucose average is above 300 mg/dL
       ">": {
         "daily_avg(resource::bloodglucose)": [300]
       }
     }, {
       // The physician and the Medical Records microservice have access to alarming health parameters
       "=": {
         "subject::id": ["medicalrecords::", "physician"]
       }
     }
   ],
   "authorizations": [
     {
       // Identity−based signature of user 'bob' authorizing the delegated evaluation of the policy
       "bob": "28da687ea01b36fac..."
     }
   ]
  }
}
```

Figure 13. User 'bob' authorizing the delegated evaluation of data-driven authorization policies.

4.3.3. Workflows as Pre- and Post-Conditions of an Authorization Policy Evaluation

Usually, authorization policies are implemented as a set of access rules that can be atomically distributed (possibly through delegation). However, in healthcare applications, certain authorization procedures (e.g., patient consent, approval delegation of authority, break-the-glass) involve complex procedures and workflows with human involvement. For example, rather than just evaluating a policy that verifies whether a patient has provided consent, the authorization system may explicitly ask for consent if missing rather than denying access.

To offer such a solution, our framework enables the optional definition of authorization workflows that are triggered before the default and implicit deny policy is evaluated.

Figure 14 illustrates two workflows. The first workflow initiates an authorization task for the patient to provide patient consent (if missing). The second workflow triggers the break-the-glass procedure that can override access to patient records, even without consent, but with the obligation for the subject requesting access to provide a motivation of why access was necessary.

```
{
 "id": "workflowpolicy_1",
 "version": 1.0,
 "policy": {
  "resources": "medicalrecords::${patientid}/records",
  "actions": [ "read" ],
  "effect": "permit",
  "conditions": [
   {
    "=": {
     "subject::id": ["resource::patientconsent"]
    }
   }
  ],
  "workflow": {
   "pre−condition": "medicalrecords::/authorizationtasks/${patientid}/patientconsent",
   "post−condition": "medicalrecords::/authorizationtasks/subject::id/breaktheglass"
  }
 }
}
```

Figure 14. Workflows of authorization tasks as a pre- or post-condition of an authorization policy.

4.4. Framework Implementation

The current proof-of-concept framework is implemented in Java on top of the Spring Cloud (http://cloud.spring.io/) framework and, more particularly, Service Release 3 (SR3) of the Dalston release train. It simplifies the development of microservices and offers capabilities to realize data integration and real-time data processing pipelines with the Spring Cloud Data Flow (http://cloud.spring.io/spring-cloud-dataflow/) toolkit. Our framework also leverages Hystrix (https://spring.io/guides/gs/circuit-breaker/), the circuit breaker implementation provided by Spring. Each of the microservices runs in a Docker CE Edge 17.05 container (https://docs.docker.com/edge/). The isolation of microservices simplifies the allocation and monitoring of memory and processor resources, as well as the communication between microservices. Access control usually involves authentication (verifying whether the identity of a subject is correct) and authorization. The focus of this research was on the latter. For authentication, we rely on ForgeRock's OpenAM 14.1 (https://www.forgerock.com/platform/access-management/) Identity and Access Management (IAM) framework. This IAM framework is also deployed in a Docker container with password-based authentication of the subjects handled by AM's RESTful interfaces. While OpenAM also provides authorization capabilities on top of XACML, it is not able to offer the functionality discussed in this paper.

5. Evaluation

This section elaborates on the scalability and performance overhead of our delegated policy evaluation for data-driven workflows. Our goal is to carry out a systematic scalability assessment using the Universal Scalability Law (USL) [25,26]. The USL combines (a) the initial linear scalability of a system under increasing load, (b) the cost of sharing resources, (c) the diminishing returns due to contention and (d) the negative returns from incoherency into a model that defines the relative capacity $C(N)$:

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)} \quad (1)$$

where N represents the scalability of the software system in terms of the number of concurrent requests, α represents the contention penalty and β defines the coherency penalty, with $0 \leq \alpha, \beta < 1$. To benchmark the scalability, N is incremented on a fixed configuration.

The authorization policies used in our setup are similar in complexity to the ones we described earlier. The performance evaluation does not include the time required for subjects to authenticate. All subjects are authenticated in advance. For a scalability assessment of OpenAM itself, we refer to our previous work [27].

In our experimental setup, we deploy all microservices in a Docker container on a Dell PowerEdge R620 server with 32 Intel Xeon E5-2650 CPU cores running at 2.00 GHz and 64 GB of memory connected to a 1 Gigabit network. We simulated an increasing number of concurrent access control requests from multiple users. We compare two performance criteria:

- The latency overhead of delegating the evaluation of the policy to another microservice;
- The performance trade-offs of policy delegation vs. local policy evaluation.

5.1. Feasibility of Delegated Policy Evaluation

Figure 15 shows the scalability of our delegated policy authorization framework with a growing number of concurrent entities that each submit requests at a rate of one per second. For this particular deployment and configuration, we reach near linear scalability up to about 1500 authorization requests per second. This experiment validates the technical feasibility for deploying our solution, especially when one considers that it is fairly trivial for our solution to scale horizontally over multiple servers using Spring's client-side Ribbon load balancer (https://spring.io/guides/gs/client-side-load-balancing/).

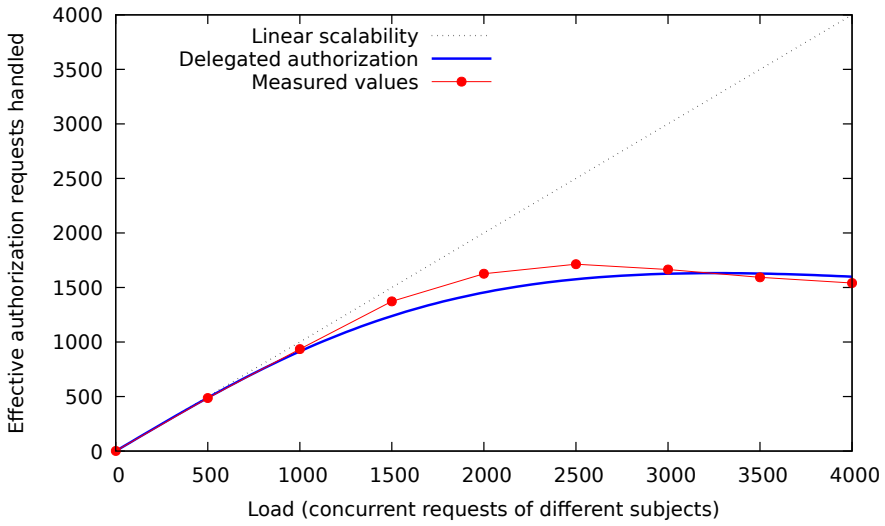

Figure 15. Systematic scalability analysis of delegated policy evaluation with the Universal Scalability Law (USL).

5.2. Performance Trade-Offs of Delegated Policy Evaluation

Figure 16 illustrates the added value of local policy evaluation versus delegated policy evaluation. This experiment specifically focuses on the authorization for data sharing between the Mobile Health Tracker microservice and the Medical Records service. The amount of health parameters to be processed on the left-hand side (data intensive policy) is 100-times larger than on the right-hand side (data non-intensive policy). This reflects a diabetes patient that manually measures his/her blood glucose values up to eight times a day, versus a diabetes patient with a Continuous Glucose Monitoring (CGM) system measuring as often as once per minute. In the local policy evaluation setting, all relevant data are first transferred from the Mobile Health Tracker service to the Medical Records service. The authorization

policy in Figure 13 is computed at the latter service. This is how the traditional policy evaluation engine works, centralizing all the data and attributes in order to evaluate the authorization policy. Given that the computational complexity to process the health parameters is the same for either microservice, Figure 16 clearly demonstrates on the left-hand side the effectiveness of evaluating the data-intensive policy where the required data reside.

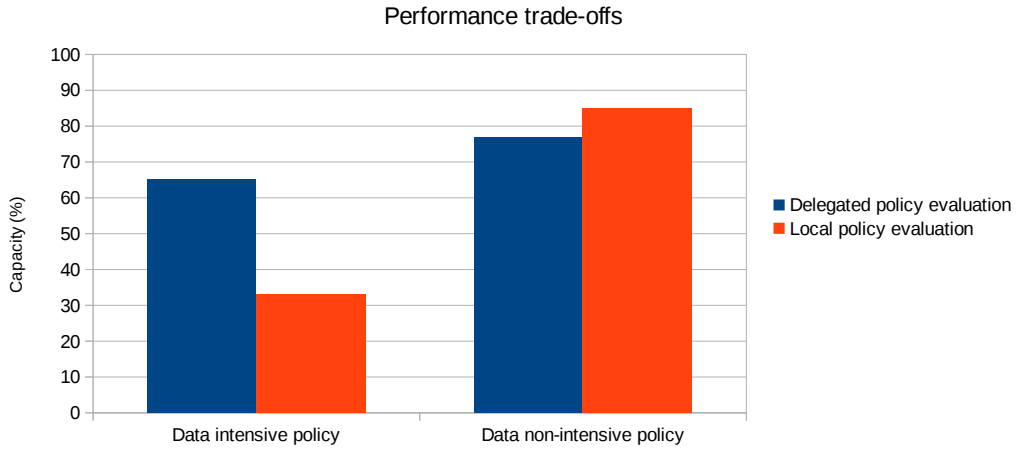

Figure 16. Performance trade-offs of delegated policy evaluation vs. local policy evaluation.

The performance improvement of delegated policy evaluation for policies that are less data intensive may not be so outspoken (or even less optimal, as shown on the right of Figure 16) as local policy evaluation. However, the fact that sensitive data do not leave the boundaries of the microservice is clearly an advantage from a security and privacy point of view. For scenarios where security and/or privacy breaches are not a concern, one can find a trade-off point where there is no significant difference between either approaches for policy evaluation. While this trade-off point can be determined empirically in a static setting, any additional load on the involved microservices may shift this trade-off point in either direction.

5.3. Qualitative Comparison with Related Work

As the related work does not offer the same delegated authorization capabilities as the ones proposed in our framework, a systematic quantitative assessment comparing performance improvements is not feasible. Instead, we carry out a qualitative comparison to contrast the strengths and weaknesses with other state-of-practice authorization frameworks.

A key characteristic that our framework shares with other authorization solutions is that users or subjects must be identified such that authorization policies can be correctly evaluated. The authentication step precedes any authorization decision and provides a set of subject attributes that can be used in authorization policy decisions. In principle, a dedicated microservice could add username/password authentication relying on a database or LDAPserver to store all user attributes and credentials. However, our framework treats the verification of identities as a separate concern. It implements federated Single Sign On (SSO) on top of ForgeRock's OpenAM (https://www.forgerock.com/platform/access-management/) identity and access management framework, as we have done in previous work [28], to create a federation of trust between identity providers for authentication and service providers enforcing authorization. It relies on established standards, such as OpenID Connect and SAML, to exchange user attributes stored at the identity provider with the microservices. Digitally-signed JSON Web Tokens (JWT) offered by the identity provider ensure the authenticity of the user attributes.

Two well-known authorization frameworks are XACML 3.0 (http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html) and OAuth 2.0 (https://tools.ietf.org/html/rfc6749). However,

there are fundamental differences between both solutions. OAuth is an access delegation framework that allows someone to do something on behalf of another user. From a high-level point of view, the protocol relies on access tokens that are obtained after user authentication and validated with each service request. In microservice deployments, OAuth access tokens are usually exchanged for signed JWT tokens at a token exchange point. These JWT tokens are then with each request passed downstream to each microservice where they are subsequently validated. However, the scope part of the OAuth protocol is not sufficiently rich to express access control conditions in a fine-grained manner. This is the goal of the XACML authorization language. XACML is a domain-specific language to define permissions that declare whether a given user can perform a given action on a particular resource. However, both OAuth and XACML can be combined to have the best of both worlds. When the OAuth authorization server verifies the validity of the access token, it can also submit a subsequent XACML request to an XACML policy engine when fine-grained authorization beyond OAuth scopes is required. Note that, contrary to our approach, both the access token validation and the policy evaluation are atomic operations, whereas our framework supports partial policy evaluation.

Several vertical domains have their own security and privacy standards. For example, in the finance world, the Payment Card Industry Data Security Standard (PCI DSS) specification (https://www.pcisecuritystandards.org/document_library) defines policies, procedures, architectures and other protective measures for credit card payments. For the healthcare domain, there are relevant international specifications such as the HL7Security and Privacy Domain Analysis Model [29] and working groups such as the European Federation for Medical Informatics (EFMI) (https://www.efmi.org/workinggroups/sse-security-safety-and-ethics) andthe International Medical Informatics Association (IMIA) (http://imia-medinfo.org/wp/security-in-health-information-systems/), which all aim to improve the security and privacy of medical data and the infrastructure deployed. However, a detailed discussion of these domain-specific initiatives and how relevant concerns can be mapped onto our framework is beyond the scope this paper.

5.4. Threats to Validity

One can argue that the above experiments provide anecdotal evidence of the benefits of the delegated policy evaluation. Indeed, it is not straightforward to generalize the obtained results or to draw strong conclusions that would also hold for other applications or use cases. The experimental results are subject to:

- The complexity of the authorization policies
- The computational power for each of the microservices
- The network capacity between each microservice

In our experimental setting, each microservice ran in a docker container with a virtual network connection between the microservices, creating ideal network operating conditions. By artificially reducing the bandwidth of virtual connections, the added value of delegated policy evaluation would be even more outspoken.

6. Conclusions

In this paper, we presented an attribute-based access control policy language in the JSON specification that simplifies certain complexities of the XACML industrial standard, and we discussed our support proof-of-concept authorization framework. A key advantage of our framework compared to the state-of-the-art is that it enables access control with delegated authorization policy evaluation for data-driven microservice workflows. Our framework targets applications where microservices each fulfill a particular functionality of the data processing pipeline, making it perfect for Internet of Things applications. Additionally, microservices can be scaled independently, allowing for a more flexible management of the individual components when compared to monolithic data processing

architectures. Furthermore, as each microservice manages its own data storage backend, the security or privacy impact of a data breach can be minimized.

Our authorization framework builds on top of well-known concepts, including feature toggles and circuit breakers, to isolate the policy evaluation from the core logic of the microservice. These building blocks help to monitor the behavior of the framework, allowing for certain performance tactics, such as caching of policy evaluation results, to improve the throughput of the framework.

By systematically benchmarking our framework against the universal scalability law, we were able to demonstrate the practical feasibility and performance of our solution in a healthcare scenario, and this under different experimental settings. Nonetheless, additional evaluation is necessary in order to generalize the current observations towards other application scenarios and operating conditions.

Part of our future work will focus on improving the expressivity of our JSON-based policy language, as well as extending the capabilities supported by runtime, without jeopardizing the minimal resource footprint of the framework and the flexibility it provides to easily customize policies at runtime.

Acknowledgments: This research is partially funded by the Research Fund KU Leuven. Work for this paper was supported by VLAIO (http://www.vlaio.be) through the SBO DiSSeCt (https://distrinet.cs.kuleuven.be/research/projects/DiSSeCt) project under Grant No. IWT 150038.

Author Contributions: Davy Preuveneers and Wouter Joosen conceived the research objectives; Davy Preuveneers performed the experiments and analyzed the data; Both authors contributed to the writing and reviewing of the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Atzori, L.; Iera, A.; Morabito, G. The Internet of Things: A Survey. *Comput. Netw.* **2010**, *54*, 2787–2805.
2. Knappmeyer, M.; Kiani, S.L.; Reetz, E.S.; Baker, N.; Tonjes, R. Survey of Context Provisioning Middleware. *IEEE Commun. Surv. Tutor.* **2013**, *15*, 1492–1519.
3. Preuveneers, D.; Berbers, Y. Internet of things: A context-awareness perspective. In *The Internet of Things: From RFID to the Next-Generation Pervasive Networked Systems*; CRC Press: Boca Raton, FL, USA, 2008; pp. 287–307.
4. Preuveneers, D.; Berbers, Y.; Joosen, W. SAMURAI: A batch and streaming context architecture for large-scale intelligent applications and environments. *JAISE* **2016**, *8*, 63–78.
5. Chen, M.; Mao, S.; Liu, Y. Big Data: A Survey. *Mob. Netw. Appl.* **2014**, *19*, 171–209.
6. Zikopoulos, P.; Eaton, C. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*, 1st ed.; McGraw-Hill Osborne Media: New York, NY, USA, 2011.
7. Toshniwal, A.; Taneja, S.; Shukla, A.; Ramasamy, K.; Patel, J.M.; Kulkarni, S.; Jackson, J.; Gade, K.; Fu, M.; Donham, J.; et al. Storm@Twitter. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14), Snowbird, UT, USA, 22–27 June 2014; ACM: New York, NY, USA, 2014; pp. 147–156.
8. Kulkarni, S.; Bhagat, N.; Fu, M.; Kedigehalli, V.; Kellogg, C.; Mittal, S.; Patel, J.M.; Ramasamy, K.; Taneja, S. Twitter Heron: Stream Processing at Scale. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15), Melbourne, Australia, 31 May–4 June 2015; ACM: New York, NY, USA, 2015; pp. 239–250.
9. Chen, G.J.; Wiener, J.L.; Iyer, S.; Jaiswal, A.; Lei, R.; Simha, N.; Wang, W.; Wilfong, K.; Williamson, T.; Yilmaz, S. Realtime Data Processing at Facebook. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16), San Francisco, CA, USA, 26 June–1 July 2016; ACM: New York, NY, USA, 2016; pp. 1087–1098.
10. Kiran, M.; Murphy, P.; Monga, I.; Dugan, J.; Baveja, S.S. Lambda Architecture for Cost-effective Batch and Speed Big Data Processing. In Proceedings of the 2015 IEEE International Conference on Big Data (Big Data) (BIG DATA '15), Santa Clara, CA, USA, 29 October–1 November 2015; IEEE Computer Society: Washington, DC, USA, 2015; pp. 2785–2792.

11. Kreps, J. Questioning the Lambda Architecture. Available online: https://www.oreilly.com/ideas/questioning-the-lambda-architecture (accessed on 16 August 2017).
12. Fowler, M.; Lewis, J. Microservices. 2014. Available online: https://martinfowler.com/articles/microservices.html (accessed on 16 August 2017).
13. Newman, S. *Building Microservices*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2015.
14. Preuveneers, D.; Joosen, W.; Ilie-Zudor, E. Policy reconciliation for access control in dynamic cross-enterprise collaborations. *Enterp. Inf. Syst.* **2017**, 1–21. Available online: http://dx.doi.org/10.1080/17517575.2017.1355985 (accessed on 30 September 2017).
15. Rahman, M.T.; Querel, L.P.; Rigby, P.C.; Adams, B. Feature Toggles: Practitioner Practices and a Case Study. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16), Austin, TX, USA, 14–15 May 2016; ACM: New York, NY, USA, 2016; pp. 201–211.
16. Nygard, M. *Release It!: Design and Deploy Production-Ready Software*; Pragmatic Bookshelf; ACM: New York, NY, USA, 2007.
17. Jha, S.; Sural, S.; Atluri, V.; Vaidya, J. Enforcing Separation of Duty in Attribute Based Access Control Systems. In Proceedings of the 11th International Conference on Information Systems Security (ICISS 2015), Kolkata, India, 16–20 December 2015; Jajoda, S., Mazumdar, C., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 61–78.
18. Botha, R.A.; Eloff, J.H.P. Separation of Duties for Access Control Enforcement in Workflow Environments. *IBM Syst. J.* **2001**, *40*, 666–682.
19. Sandhu, R.S. Lattice-Based Access Control Models. *Computer* **1993**, *26*, 9–19.
20. Sandhu, R.; Samarati, P. Access control: Principle and practice. *IEEE Commun. Mag.* **1994**, *32*, 40–48.
21. Sandhu, R.S.; Coyne, E.J.; Feinstein, H.L.; Youman, C.E. Role-Based Access Control Models. *Computer* **1996**, *29*, 38–47.
22. Jin, X.; Krishnan, R.; Sandhu, R. 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, DBSec 2012, Paris, France, 11–13 July 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 41–55.
23. XACML-V3.0. *eXtensible Access Control Markup Language (XACML) Version 3.0 Plus Errata 01. OASIS Standard incorporating Approved Errata*; OASIS Open: Burlington, MA, USA, 2017. Available online: http://docs.oasis-open.org/xacml/3.0/errata01/os/xacml-3.0-core-spec-errata01-os-complete.pdf (accessed on 27 September 2017).
24. Shamir, A. Identity-based Cryptosystems and Signature Schemes. In Proceedings of the CRYPTO 84 on Advances in Cryptology, Santa Barbara, CA, USA, 1984; Springer: New York, NY, USA, 1985; pp. 47–53.
25. Gunther, N.J. *Guerrilla Capacity Planning—A Tactical Approach to Planning for Highly Scalable Applications and Services*; Springer: New York, NY, USA, 2007.
26. Wikipedia. Universal Law of Computational Scalability—Wikipedia, 2014. Available online: https://en.wikipedia.org/wiki/Neil_J._Gunther#Universal_Law_of_Computational_Scalability (accessed on 10 February 2014).
27. Heyman, T.; Preuveneers, D.; Joosen, W. Scalability Analysis of the OpenAM Access Control System with the Universal Scalability Law. In Proceedings of the 2014 International Conference on Future Internet of Things and Cloud (FiCloud), Barcelona, Spain, 27–29 August 2014; pp. 505–512.
28. Preuveneers, D.; Joosen, W. SmartAuth: Dynamic Context Fingerprinting for Continuous User Authentication. In Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15), Salamanca, Spain, 13–17 April 2015; ACM: New York, NY, USA, 2015; pp. 2185–2191.
29. Blobel, B.; Ruotsalainen, P.; Lopez, D.; Gonzalez, C. How to Use the HL7 Composite Security and Privacy Domain Analysis Model. *Int. J. Biomed. Healthc.* **2015**, *3*, 12–17.

© 2017 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).