Attribute Based Access Control for Healthcare Resources

Subhojeet Mukherjee
Dept. of Computer Science,
Colorado State University
Fort Collins, CO
subhomuk@colostate.edu

Indrakshi Ray
Dept. of Computer Science,
Colorado State University
Fort Collins, CO
iray@cs.colostate.edu

Indrajit Ray
Dept. of Computer Science,
Colorado State University
Fort Collins, CO
indrajit@cs.colostate.edu

Hossein Shirazi
Dept. of Computer Science,
Colorado State University
Fort Collins, CO
shirazi@gmail.com

Toan Ong
Anschutz Medical Campus
University of Colorado
Denver, CO
toan.ong@ucdenver.edu

Michael G. Kahn
Anschutz Medical Campus
University of Colorado
Denver, CO
michael.kahn@ucdenver.edu

ABSTRACT

Fast Health Interoperability Services (FHIR) is the most recent in the line of standards for healthcare resources. FHIR represents different types of medical artifacts as resources and also provides recommendations for their authorized disclosure using web-based protocols including O-Auth and OpenId Connect and also defines security labels. In most cases, Role Based Access Control (RBAC) is used to secure access to FHIR resources. We provide an alternative approach based on Attribute Based Access Control (ABAC) that allows attributes of subjects and objects to take part in authorization decision. Our system allows various stakeholders to define policies governing the release of healthcare data. It also authenticates the end user requesting access. Our system acts as a middle-layer between the end-user and the FHIR server. Our system provides efficient release of individual and batch resources both during normal operations and also during emergencies. We also provide an implementation that demonstrates the feasibility of our approach.

Keywords

FHIR, ABAC, Authorization, Access Control, REST

1. INTRODUCTION AND MOTIVATION

Medical institutions generate massive amounts of clinical data. Such data must be shared with various stakeholders including healthcare providers and researchers for the purpose of providing better care and making medical breakthroughs. Consequently, this necessitates efficient storage and retrieval of medical data. Moreover, since the data is used for different purposes and often retrieved by various entities, the data needs to adhere to some standards.

Fast Healthcare Interoperability Resources (FHIR) [22] specifies such a standard for fast and efficient storage/retrieval of health data. Units of health data in FHIR are referred to as "Resources". FHIR defines multiple such resources, such as Patient, AllergyIntolerance, where each resource can be linked to multiple other resources. Resources in FHIR can be queried using RESTful API calls. Multiple medical and software organizations [21] have adopted FHIR as their primary choice of storing and disseminating health data.

Inadvertent or malicious disclosure of data that contains Personally Identifiable Information (PII) to unauthorized individuals or organizations may have catastrophic consequences. Thus, medical institutions must comply with federal and state policies when they release sensitive medical data [9, 20]. The institutions are responsible for interpreting these rules and developing their own data release procedures. In addition, the institutions often have their own operational protocols and may provide additional security policies. Patients may be unwilling to disclose their health information to others for reasons of security and privacy. Consequently, patients should determine what data may be released to data requesters, the purpose for which such data will be used, and the period for which such data must be retained.

FHIR suggests basic features to ensure security, privacy, and authenticity of health data [25]. Some of the features suggested by FHIR are geared towards providing communication security, user authentication, and authorization. Communication security is ensured by the usage of secure Hypertext Transfer Protocol (HTTPS). User authentication and authorization is ensured by using O-Auth 2.0 [8] and FHIR defined security labels [26]. Moreover, FHIR also provides recommendations and guidelines [3] for ensuring authorized access to FHIR resources.

O-Auth is a popular web-based protocol used primarily for authentication and authorization. The general protocol flow of O-Auth 2.0 [8] requires the owner of the data to grant permissions explicitly to different parties who want to make use of their data. In the medical domain, the doctor will need the permission from the owner before viewing or editing some health data. The consequences can be catastrophic if the doctor is unable to get access to health data during an emergency. Moreover, if a researcher needs to get access to multiple patient data resources, she needs to obtain permissions from each resource owner. Also, O-Auth only succeeds if the data has already been created and O-Auth does not authenticate resource creators. FHIR does allow break-the-

ABAC'17, March 24 2017, Scottsdale, AZ, USA

DOI: http://dx.doi.org/10.1145/3041048.3041055

glass scenarios [26], where an individual can access FHIR resources in cases of emergency. However, such break-the-glass scenarios are provided only to authorized users. Due to lack of available implementations, it is unclear how to grant such allowances in a fine-grained manner. The deficiencies in O-Auth can be somewhat overcome by using FHIR security labels that provide access control. However, the set of security labels is finite and creating custom labels may result in cost enhancements, as indicated in [26].

Previous works which have been related to FHIR standards [1, 4, 15, 19] have made use of Role-Based Access Control (RBAC) or its derivatives to grant secure access to the resources. However, RBAC is not granular enough to provide decisions on the basis of attributes of the requester, such as age, hippa-compatibility, and name etc. Moreover, RBAC fails to take into account other factors like resource attributes and purpose of data release, while making access decisions [23]. The work that is most closely related to our work is User-Managed Access (UMA) [17]. UMA suggests using user grants and user-defined policies which decide whether a requesting party is granted access to perform certain actions on the data owned by a user. However, the focus of UMA lies primarily on resource owners trying to protect individual resources. UMA does not specify how policies should be defined to govern batch release of resources by making use of specific resource attributes, e.g. "Release all patient data to researcher A where patient-age > 40".

In this paper, we propose a system which acts as a middle layer between the client application and FHIR server, providing fine grained access to FHIR resources. Specifically, we make use of Attribute Based Access Control [11] (ABAC) to make decisions which govern access to FHIR resources. We present an approach that allows both incremental and batch release of FHIR resources to any requesting party, based on the policies defined by resource-owners. Our approach is owner-centric, where an *owner* is anyone who creates a FHIR resource [28] and can grant permissions to any requester any time after the creation of a resource. The major contributions of this work include authenticating users before allowing them access to FHIR resources and policies, demonstrating how access control policies and requests can be generated in coherence with RESTful API and FHIR standards, and releasing resources at the finest granularity and finally authorizing users to perform actions which abide by user-defined rules.

The rest of the paper is organized as follows. Section 2 provides a brief overview of FHIR and related efforts in securing access to resources stored on a FHIR server. Section 3 describes our system architecture and implementation. Section 4 describe the administrative operations in our system. Section 5 introduces customized XACML attributes needed for fine-grained access. Section 6 presents our adversary model and then performs an informal security evaluation. Section 7 concludes the paper by highlighting the contributions of this work and discuss possible future enhancements to our system.

2. BACKGROUND AND RELATED WORK

2.1 FHIR

HL7 Fast Healthcare Interoperability Resources (FHIR [22]) is the next generation standard for storing and disseminating health data. The units of data exchange in FHIR are *Resources*. Currently, FHIR defines 93 such resources such as Patient, Practitioner, Medication, and Observation. FHIR resources are standardized and can be referred to by their unique ids. When creating a resource, the FHIR server generates a unique id for that resource. FHIR resources can be accessed using RESTful API calls and returned units of data can be retrieved in XML or JSON format. A typical FHIR Rest call can be codified as "[http-verb] [base-uri]/[resource]/<optional id>?< optional parameters>". A FHIR query to read resource "Patient" with id "1234" should be "GET https://some-server/Patient/
1234". FHIR codifies HTTP verbs to perform the following operations:

- **GET**: Used generally to retrieve and search resources.
- **POST**: Used generally to create resources. Also used to update batch data, referred to as a *Bundle* (collection of resources) in FHIR terminologies.
- **PUT**: Update unique resources identified by their ids.
- **DELETE**: Delete FHIR resources.

2.2 Related Efforts in Securing Access to FHIR Resources

While much attention has been devoted to securing personal health records (PHR) systems [7, 16, 28], few have focused on FHIR as the back-end standard for medical resources. FHIR differs from traditional PHR systems by introducing a RESTful querying interface, and its own data modeling standards. This implies that an Access Control System running on top of FHIR must adhere to REST and the FHIR resource standards. Thus, a system that restricts access to FHIR resources, needs to incorporate the philosophies and standards behind FHIR resources and RESTful web services. Keeping this in view, we present a brief overview of the literature and industry adopted protocols that have been proposed to secure access to FHIR resources.

O-Auth 2.0 [8] and OpenId Connect 1.0 [6] are popular web-based authentication/authorization protocols. Both these protocols can be used to secure access to FHIR resources [18]. O-Auth [8] is widely used for authentication and authorization purposes. O-Auth 2.0 requires the owner of the data to grant explicit permissions to other requesting parties (human or computer applications) who want to access their data. In medical terms, this can be a patient granting access to a doctor to view or edit her health data. Section 1, highlights the general drawbacks of O-Auth 2.0. Additionally, O-Auth provides access control using scopes. Although this works seamlessly when an individual grants permissions to another individual, if a data owner wants to grant permissions on a set of resources to a requester, O-Auth 2.0 becomes efficient and impractical. OpenId connect is an authentication protocol which works on top of O-Auth 2.0 to verify the identity of a user. However, since OpenId is strictly integrated with O-Auth 2.0, it suffers from the same drawbacks as O-Auth.

The issues with O-Auth 2.0 and OpenId Connect can be mitigated using access control systems (ACS). However, till date, only a handful of works have focused on integrating FHIR with access control systems. Namli et al. [19] use standard hierarchical RBAC to restrict access to FHIR resources. The roles in this hierarchy define the functional relationships

between the patient and the data requester. Lamprinakos et al. [15] propose four different access control mechanisms for four different use cases. For short term access scenarios like a visit to a doctor, they propose sharing of QR codes which represent the patient's credentials. For long term access, they suggest using access control lists. Moreover, the authors in [15] provide an RBAC framework with a static set of permissions granted to each role. They also provide a break-the-glass scenario for emergency access to patient FHIR resources. Anwar et al. [1] develop an RBAC ontology by extending the traditional RBAC classes like *subject*, *action*, *resource*, *role*, *session*. They then use the Semantic Web Rule language (SWRL) to specify HIPPA compliant access control policies. Finally, they deploy an access decision engine which performs reasoning based on the access policies and the RBAC ontology. Dong et al. [4] leverage the concepts of RBAC by assigning roles which correspond to a *Circle-of-Care* (CoC) around the patient. CoC refers to the network of healthcare practitioners who are providing care to a patient at a given instance of time. The CoC is thus temporal in nature and can change as the patient's preferences change.

The works described above make use of RBAC as the primary model of choice for access control. However, RBAC suffers from its own deficiencies. Firstly, RBAC is not expressive enough to accommodate fine-grained access control that depends on factors such as the requester's age, hippa-compatibility, name etc. Secondly, RBAC grants similar access to people having the same roles. In spite of its advantages in scalability and management, RBAC is not considered quite suitable for the medical domain [23].

ABAC [11] model provides a more dynamic and granular approach to access control. Users are allowed to define access control policies using attributes of different entities like subjects, actions, resources and environments. Although ABAC has been used to secure access to RESTful web services [12, 13], to the best of our knowledge, it has not been used in context of securing FHIR resources. User-Managed Access (UMA) [17] does specify protocol standards for user centric access management of stored resources but, as mentioned in section 1, it does not cater to batch release of resources. Neither does it address the essential implementation details of an ABAC based security system on a FHIR server.

3. ARCHITECTURE AND IMPLEMENTATION

In this section, we describe the architecture and the implementation of our system. Figure 1 acts as a middle layer between the client application and the FHIR server. Specifically, it provides authentication and access control for fine-grained access to FHIR resources. Note that, this is done in conformance with FHIR specifications for deploying security systems [25].

Our architecture shown in Fig. 1 consists of four major components namely, the *Validation Server*, the *ABAC Engine*, the *PEP/Front-End* and the *Policy Admin*. All these components interact with each other to provide fine-grained access to resources stored at the FHIR server. The *Attribute database* stores attributes of registered users of the FHIR server. The *Owner database* stores the relationship that maps a FHIR resource to its owner. We now describe

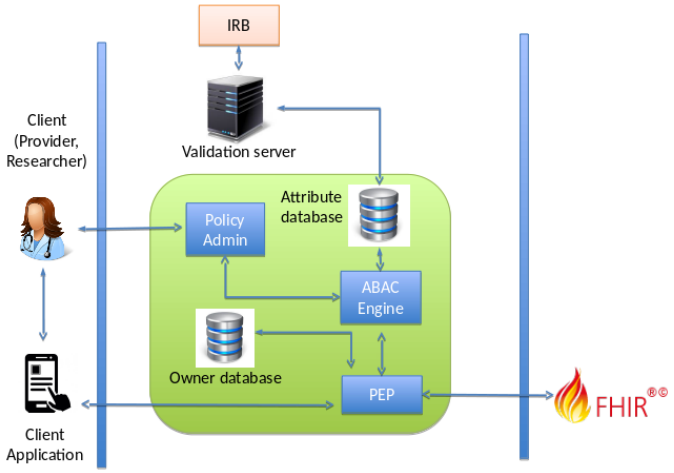

Figure 1: Solution Architecture

Please provide your details in this section. Keep id empty if you dont have one.

id:

role:

address.city:

name.given:

Submit Query

Figure 2: Validation Server Front-End Form

the components of our architecture and the platform used to design such components.

3.1 Validation Server

The *Validation Server* is responsible for validating credentials of a user who wishes to register herself. Registration refers to the process of a new or returning user entering her attributes in a front-end form shown in Figure 2. The attributes are populated from the *Attribute Database* (refer to Fig. 1) schema. The *Attribute Database* schema includes two default attributes namely, *id* and *role*. The *Attribute Database* is altered and other attributes are added dynamically as new policies are added. The `create policy` bullet in section 4.3 describes this process in details. Once the user has entered all the attributes, they are validated by the *Validation Server*, either by using pre-defined predicates or validating with the Institutional Review Board (IRB) in special cases. After validation, if the user did not enter her id at the time of registration (a new user), a unique *token/id* [1] is generated for the user. Following this, the entered attributes, along with the generated id, are transported (securely) back to the *Attribute Database*. The *Attribute Database* stores the attributes of the registered user, referenced by the unique id. In the case of a returning user, if the user id is not present in the *Attribute Database*, the corresponding attributes are not updated and a negative response is sent to the user. Other-

[1]Note that throughout this work we will be using these two terms, *token* and *id*, interchangeably

wise, if all of the above mentioned processes are carried out successfully, the user is provided with her unique token.

The *Validation Server* is implemented as a Java Servlet running on a Tomcat 7 server. The front-end pages are implemented as JavaServer Pages (JSP) and the *Attribute Database* is implemented as a back-end MYSQL table. All communications to and from the *Validation Server* are performed securely over HTTPS.

3.2 ABAC Engine

The *ABAC Engine* is responsible for validating the requests against existing policies and returning back the decision. The workflow of the *ABAC Engine* is similar to the one described in literature [11]. However, for the purpose of this work we customized the logic for the *PEP* and *PIP*. While the *PEP* is described as a separate component in the next section, the *PIP* was designed to fetch missing user attributes from the *Attribute Database* (refer to Fig. 1).

The *PAP* and *PDP* are the major components of the *ABAC Engine* and were obtained as parts of the *WSO2 Identity Server* [27]. The *PIP* was coded in Java and archived as a jar file, which was then integrated into the *WSO2 Identity Server* by making some configuration changes.

3.3 PEP

The *PEP* in our case is also the web service front-end for receiving all REST queries from a client application as shown in Figure 1. The *PEP* intercepts all calls to the FHIR resources, forwards the queries to the FHIR server layer, retrieves the requested data and creates XACML requests [5] from the incoming queries and the each retrieved FHIR resource.

The XACML requests are created by the *Context Handler* which is an independent component of the *PEP*. *Context Handler* generates a XACML request with the following attribute set: $\{user\text{-}id, action, resource\text{-}name, resource\text{-}owner, \{x | x \in resource\text{-}elements\}\}$. The $user-id$ is sent along with the request, the *action* can be either of "GET", "POST", "PUT" or "DELETE" and the $resource-name$ is supplied as a part of the query or as the root element of the request body. In cases where the request body is a *Bundle* (refer to section 2.1) each resource in the bundle is treated separately. The rest of the attributes are individual elements of the resource(s) to be created. Fig. 3 shows an example XACML request from a user with "id" 1234 to retrieve ("GET") a "Patient" resource having elements "patient-id", "patient-given-name" etc. The first three attributes of the XACML request are in coherence with the XACML request standards. The *resource-owner* attribute bears significant security implications. Each added policy to the *ABAC Engine* is also appended with a *resource-owner* attribute. This allows XACML requests for resources, owned by a particular user, to be evaluated against policies created by that user only. This eliminates the chances of requests being evaluated against global policies. For example, if the resource owner having id "c123" creates a policy which allows all researchers to view all patient resources, this policy would not be effective globally across the entire system. Therefore, a researcher requesting all patient data would only get access to patient data POSTed by owner "c123" and not the other owners since they do not explicitly allow this researcher to view resources owned by them. Finally the attribute set (*resource-elements*) is added to ensure the request is correctly evaluated against policies which include resource specific information like "patient-id", "patient-given-name" etc.

```
<Request>
/* :Attribute-Category :Attribute ID :Attribute Value */
:action :action-id :GET
:access-subject :id :1234
:resource :resource-id :Patient
:Patient :patient-id :abcd
:Patient :patient-given-name :John
.........
</Request>
```

Figure 3: Sample XACML Request

(a) Policy Admin Login (b) Policy Admin Options

Figure 4: Policy Admin Front-End

Each XACML request is then sent to the *PDP* component of the ABAC engine for evaluation and if the decision if "permit", the resource is sent out to the requesting user. The role of the *PEP* is described further in sections 4.1 and 4.2.

The *PEP* is coded as a Java Jersey Class [14] and built using Maven dependencies. The *PEP* interacts with the *ABAC Engine* by making calls to web services exposed by the *WSO2 Identity Server* using the Apache Axis 2 Client [2].

3.4 Policy Admin

The *Policy Admin* acts as gateway to the *PAP* component of the *ABAC Engine*. In order to manage policies the user needs to enter the unique token, provided by the *Validation Server*, on the "Welcome" page (Fig. 4a). A XACML request is then created using this token and validated by the *PDP*. If the user is allowed to manage policies, he/she is shown the "Options" page (Fig. 4b) which allows the user to create, view and delete policies which have been created by this user.

The *Policy Admin* is implemented as a Java Servlet running on a Tomcat 7 server. The front-end pages are implemented as JavaServer Pages (JSP). All communications to and from the *Policy Admin* are performed securely over HTTPS.

4. ADMINISTRATIVE OPERATIONS

Our system can be completely described in terms of three use cases namely, create/update resources, retrieve/delete resources and manage policies. The first and second use cases encompass the HTTP verbs PUT, POST, and GET, DELETE respectively. The third use case is introduced to allow the regular users manage their own policies.

It is to be noted that before performing any action the client needs to validate her attributes with the *Validation*

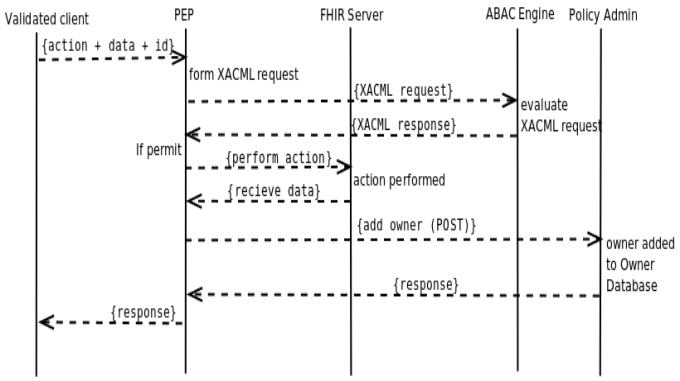

Figure 5: Create and Update Use Case

Server (refer to section 3.1) and obtain a unique id. A returning client (one who has been validated during a previous access attempt) can be redirected back to the *Validation Server* if she has null-valued attributes in the *Attribute Database*. This is because, as it will be observed later, when a new policy is added to our system, the *Attribute Database* schema is altered to accommodate previously absent but newly introduced attribute name fields. Initial values for these fields are kept null to indicate that they have not been updated by the corresponding clients. When the client returns, she is asked to complete her user profile by updating the null valued attributes. At this point, the user can choose to keep this value empty, in which case the field values are updated with white-space characters.

4.1 Creating and Updating Resources

FHIR allows creation and update of resources, using POST and PUT verbs (refer to section 2.1). Fig. 5 delineates the use case for creating and updating resources. The use case in Fig. 5 can be described using the following steps:

1. The client application sends a POST or PUT request to *PEP*. Along with the request, it also sends the resource as the body and the unique id obtained after successfully registering at the *Validation Server* as a custom HTTP header. If the user wishes to create new resources he/she must enter a specific role (termed as "Poster") in the "role" field of the *Validation Server* front-end form.

2. *PEP* makes a call to an internal *Context Handler* and generates a XACML request (refer to section 3.3) using the following attributes:

   - **user-id**: as received from the requester as an HTTP header.
   - **action**: "POST" or "PUT"
   - **resource-name**: resource to be created or updated. If its a *Bundle*, individual resource names are obtained from the request body.
   - **resource-owner** "admin"
   - **resource-elements** elements of the resource as supplied in the request body.

3. The XACML request is evaluated against the policies stored at the *PAP* of the *ABAC Engine*. The *PAP* stores a default policy for POST actions which is shown in Fig. 6. This policy allows any individual having the role of "Poster" to "POST" (create) any resource. The default policy does not support a "PUT" operation, since updating resources created by a different owner requires explicit permission from that owner. By FHIR guidelines, the HTTP verb POST can also be used to update resources. As a result, if the user attempts to POST on an existing resource created by some other owner, it is further verified whether the owner of that resource has explicitly allowed the user to POST on the resource he owns. This is done by creating a XACML request similar to the one shown in Fig. 6, except by changing the *resource-owner* to the owner of the resource which the requesting user attempts of the modify. This information is fetched from the *Owner Database*.

   A critical observation from the XACML request is that it does not contain the role of the requester. Because the role is a missing attribute, it is fetched by the *PIP* (refer to section 3.2) from the *Attribute Database* which is updated by the *Validation Server* at the time of user registration.

```
<Policy PolicyId = "DEF-POST"
                rule-combining-algorithm="deny-overrides">
  <Target>
  /* :Attribute-Category :Attribute ID :Attribute Value */
   AnyOf:
     AllOf:
      :access-subject      :Role               :Poster
      :resource            :xacml:resource-id  :.*
      :action              :xacml:action-id    :POST
      :admin               :resource-owner     :admin
  </Target>
  <Rule RuleId = "P" Effect="Permit">
  </Rule>
</Policy>
```

Figure 6: Default XACML POST Policy

4. The XACML response is sent back to the *PEP*.

5. If the response is a "permit", the *PEP* instructs the *FHIR Server* to perform the required action.

6. The FHIR server returns the created/updated resource back after performing the required action. The returned resource is then scanned to obtain the unique id assigned to it by the FHIR server or provided by its owner.

7. The *PEP* then requests the *Policy Admin* to keep a record of the resource that this owner has created in the *Owner Database*. When requests to update (PUT) certain resources are made, the id of the owner of that resource is retrieved from this table. A XACML request is then created to verify whether this requester is allowed to update the requested resource. This process is similar to the one described in bullet 3.

8. Finally the user is notified of the successful resource creation/modification.

4.2 Retrieving and Deleting Resources

FHIR allows retrieving and deleting resources using GET and DELETE verbs (refer to section 2.1). Fig. 7 delineates

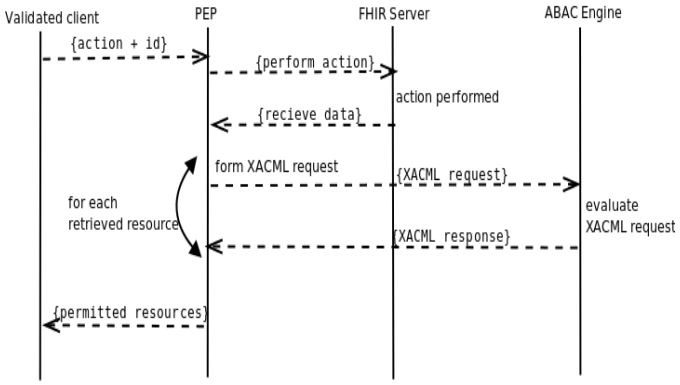

Figure 7: Retrieve and Delete Use Case

the use case for retrieving and deleting resources. The use case in Fig. 7 can be described using the following steps.

1. The client application sends a GET or DELETE request to *PEP*. Along with the request, it also sends the unique id, obtained after successfully registering at the *Validation Server*, as a custom HTTP header.

2. *PEP* forwards the request to the FHIR server. If the requested operation is a DELETE then the forwarded request is modified to a GET operation, in order to obtain the resource to be deleted.

3. The FHIR server returns the requested resource(s) either as a single resource or as a *Bundle* depending on the request query.

4. For each returned resource, the *PEP* makes a call to an internal *Context Handler* and generates a XACML request (refer to section 3.3) using the following attributes:

   - **user-id**: as received from the requester via an HTTP header.
   - **action**: "GET" or "DELETE".
   - **resource-name**: resource to be retrieved or deleted.
   - **resource-owner** owner of the resource as obtained from the *Owner Database*.
   - **resource-elements** elements of the resource as obtained from the FHIR server.

5. The XACML request is evaluated against the set of policies stored at the *PAP* of the *ABAC Engine*.

6. The XACML response is sent back to the *PEP*.

7. If the response is a "permit", the *PEP* buffers the resource as a future output.

8. Finally all permitted resources are returned back to the requester.

4.3 Policy Management

Fig. 8 delineates the use case for creating/updating, viewing and deleting policies at the *Policy Admin*. The use case in Fig. 8 can be described using the following steps.

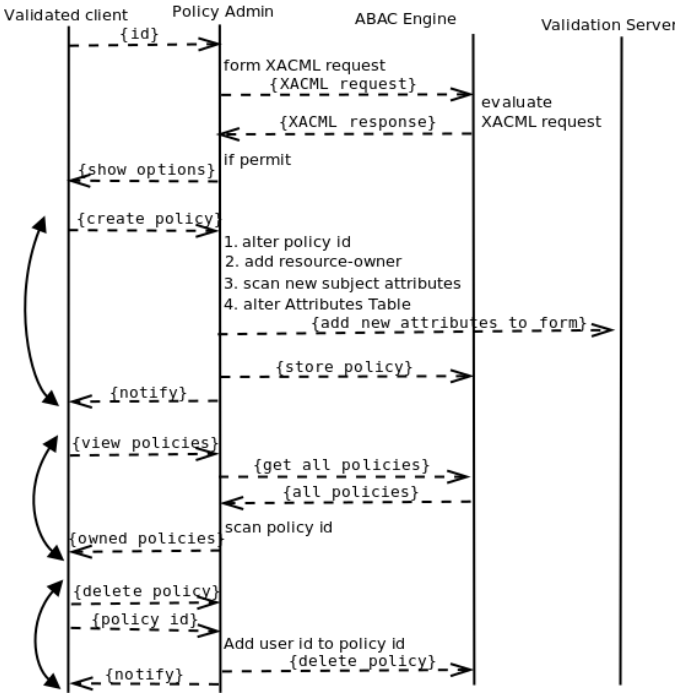

Figure 8: Policy Administration

```
<Policy PolicyId = "DEF-MANAGE-POLICY"
                rule-combining-algorithm="deny-overrides">
  <Target>
  /* :Attribute-Category :Attribute ID :Attribute Value */
   AnyOf:
     AllOf:
      :access-subject      :Role               :Poster
      :resource            :xacml:resource-id  :Policy
      :action              :xacml:action-id    :MANAGE
      :admin               :resource-owner     :admin
  </Target>
  <Rule RuleId = "P" Effect="Permit">
  </Rule>
</Policy>
```

Figure 9: Default XACML MANAGE Policy

1. The client provides her id at the *Policy Admin* frontend.

2. *PEP* makes a call to an internal *Context Handler* and generates a XACML request (refer to section 3.3) using the following attributes:

   - **user-id**: as received from the requester as an HTTP header.
   - **action**: "MANAGE"
   - **resource-name**: "Policy"
   - **resource-owner** admin
   - **resource-elements** Empty

3. The XACML request is evaluated against the set of policies stored at the *PAP* of the *ABAC Engine*. The *PAP* stores a default policy (shown in Fig. 9) for managing policies. This policy allows any individual having the role of "Poster" to "MANAGE" (create/update, view or delete) "POLICY"s.

4. The XACML response is sent back to the *PEP*.

5. If the response is a "permit", the *Policy Admin* presents the following options to the user.

6. **create policy**

   (a) Once the user enters a new policy, the supplied policy id is appended with the user's unique id as received from the *Policy Admin* front-end. This makes the policy unique to the user who creates it.

   (b) The *AllOf* node under the *Target* of the supplied policy is then appended with the *resource-owner* attribute and the value for that attribute is set to the user's unique id as received from the *Policy Admin* front-end. This allows XACML requests for resources, created by this user, to be evaluated against policies created by this user. If the initially supplied policy contains a *resource-owner* the user is prompted to resupply the policy without the *resource-owner* attribute.

   (c) The policy is then scanned for new attributes (missing in the *Attributes Database* schema) which describe requesting users.

   (d) The *Attributes Database* table schema is then altered to reflect the newly found attributes.

   (e) A request is made to *Validation Server* to add the newly found attributes to its front-end form. In this way information entered by new registering users are kept in-sync with the *Attributes Database* and the *Validation Server*.

   (f) Finally the newly entered policy is stored at the *ABAC Engine* and the user is notified. In case of any failure, the user is asked to notify the system administrator of back-end issues.

7. **view policies**

   (a) *Policy Admin* retrieves all policies stored at the *PAP* of the *ABAC Engine*. It compares if the retrieved policy ids end with the requesting user's id and buffers the policy as final output if it does.

   (b) It then displays all valid policies owned by the requesting user.

8. **delete policies**

   (a) *Policy Admin* expects the user to enter the policy id which the user intends to delete.

   (b) The id of the requesting user is then appended with the policy id and the resulting policy is attempted to be deleted.

   (c) The result of the delete operation is notified to the user.

5. XACML CUSTOM ATTRIBUTE SPECIFICATION

XACML requests generated by our system leverage standards exposed by both RESTful APIs and FHIR specifications. This implies that the XACML policies against which the XACML requests are evaluated, should also confirm to RESTful APIs and FHIR standards. Traditionally, XACML allows policies to be written in terms of four specific attribute types namely, *resource*, *action*, *subject* and *environment*. Our system supports writing policies that include all four. Additionally, we use custom attributes to write finer grained policies that relate to the details of the requesting user and FHIR resource standards. This allows to specify details about the requesting users and the resources they target in their queries. For example, the policy shown in Fig. 10 allows a "Researcher" from "Denver" to "GET" resources belonging to "Female" "Patient"s only.

```
<Policy PolicyId = "POLICY-1"
                rule-combining-algorithm="deny-overrides">
  <Target>
  /* :Attribute-Category :Attribute ID :Attribute Value */
   AnyOf:
     AllOf:
      :access-subject      :subject.role.none  :Researcher
      :resource            :xacml:resource-id  :Patient
      :action              :xacml:action-id    :GET
      :admin               :resource-owner     :1234
  </Target>
  <Rule RuleId = "P" Effect="Permit">
  <Condition>
     Function: string-equal
   /* :Attribute-Category :Attribute ID :Attribute Value */
      :access-subject      :subject.address.city  :Denver
      :Patient             :Patient.gender.none   :Female
   </Condition>
  </Rule>
</Policy>
```

Figure 10: Sample XACML Policy

We introduce three types of custom attributes, each aimed at uniquely identifying specific details about three entities: the subject/user/clients performing operations on a FHIR server, FHIR resources, and owners/creators of resources on the FHIR server. XACML standardized attributes are shown in Fig 10 using a "xacml" prefix. Examples include "xacml:resource-id" and "xacml:action-id".

5.1 Subject Custom Attributes

Subject Custom Attributes are categorized under the XACML standard "access-subject" category. Custom attributes referring to subjects requesting access to the FHIR server are represented using 3-layered *period*-delimited attribute-ids. Attributes "subject.address.city" and "subject.role.none", from Fig. 10 are examples of *Subject Custom Attribute*s. We require the first layer of this attribute to be the "subject". The second layer can be any specific attribute that the policy writer aims to refer to, for example, "address" or "role". The third layer represents a subclass of the second layer attribute, e.g., "city" is to "address". However, in cases where the third layer is not required it can be stated as "none". Further, although we specify a three layered format for *Subject Custom Attributes* it is left open to the choice of the system administrator if they wish to extend it to any number of layers beyond that.

5.2 Resource Custom Attributes

The XACML custom attributes referring to FHIR resources are also represented using a 3-layered *period*-delimited format. Attribute "Patient.gender.none" from Fig. 10 is an example of *Resource Custom Attribute*s. *Resource Custom Attributes* are categorized under the resource name. For example, an attribute referring to a Patient resource should

belong to the category "Patient". It is obligatory to keep the first layer of this attribute as the name of the resource, e.g. "Patient", "AllergyIntolerance" etc. The second layer can be any attribute of the FHIR resource as standardized by FHIR [22], e.g., "gender". The third layer represents a subclass of the second layer attribute, e.g., "city" is to "address" for a "Patient" resource. However, in cases where the third layer is not required it can be stated as "none". Moreover, although we specify a three layered format for *Resource Custom Attributes* it is left open to the choice of the system administrator if they wish to extend it to any number of layers beyond that.

5.3 Resource Owner Custom Attributes

The XACML custom attributes referring to the owner of a resource are represented using the attribute-id "resource-owner" and categorized as an "admin" attribute. The "resource-owner" is added automatically by the *Policy Admin* after the user enters a valid policy conforming to XACML 3.0 standards. The user supplying the policy is discouraged to provide this attribute in the policy. If it is provided, our policy manager prompts the user to resubmit the policy without the "resource-owner" attribute.

6. SECURITY DISCUSSION

6.1 Adversary Model

As a precursor to analyzing the security features provided by our system, we aim to model our adversary in this subsection. In particular, we wish to clearly delineate the capabilities of the adversary based on the assumptions and available security services offered by some previously established protocols used in our approach.

One of the primary observations from our framework is that communication between all endpoints is carried out securely using the HTTP over TLS (HTTPS) protocol [24]. Moreover, client identity verification services offered by HTTPS are used when communicating within all (except the resource requester) participating entities in the protocol. For example, when the *PEP* communicates with the *FHIR server* or the *Policy Admin* it presents them with its certificate. Although certificates used for this work are self signed (we manually updated the trustStores at each end-point), this can easily be extended to include certificates signed by well known signing authorities.

In its ideal form[2] TLS provides adequate communication security and prevents any attempted man-in-the middle attacks. This means, the only way for an adversary to deceive our system is to impersonate the end user. The attacker can do this in any of the two following ways:

1. Create a falsified identity at the time of registration.
2. Get access to or guess the id of any valid registered user.

Using either of the above techniques the attacker can attempt to subvert the security our protocols and thereby lead to the following misuse cases:

- Create false resources.
- Update existing resources created by others.
- Retrieve or delete resources.
- Create crafted policies, which might in turn allow the attacker further access into the system.
- View policies created by other users or the admin.
- Delete policies created by other users or the admin.

[2]Although, attacks of TLS have been documented previously [10], we assume the usage of an all-patched version of TLS.

6.2 Assumptions

While doing our security analysis we make the following assumptions:

1. Endpoints are never compromised. This means two things:
   (a) Data in storage is secure from any unauthorized viewing or modification.
   (b) Participants are infallible and perform their duties honestly devoid of any malicious intents.
2. End users secrecy is maintained; they do not leak or loose unique ids provided by the registration server.
3. User supplied policies are correct and clearly express the intent of the policy provider.

6.3 Security Analysis

We first analyze the security of our system under the assumption that the attacker successfully impersonates a valid user (possess a valid user id) using either of the two ways mentioned in section 6.1. We then show that our framework allows custom enhancements which can significantly constrict the capabilities of any adversary to impersonate as a valid/registered user. Finally, we prove that our framework is successful in blocking adversaries who cannot impersonate a valid/registered user. For the first and the final analysis, we discuss the possibilities of the adversary to successfully accomplish the misuse cases enumerated in subsection 6.1.

6.3.1 Adversary Possessing a Valid User ID

- **Create false resources:** If the adversary possesses a valid user id listed as a "Poster" in the *Attribute Database*, the default POST policy (shown in Fig. 6) allows the adversary to create false resources.
- **Update existing resources created by others:** An adversary in possession of a valid user credential cannot modify resources owned by other users unless the owner explicitly allows the user to do so. Update operations can be carried out using HTTP verbs PUT and POST (refer to section 2.1). Our framework does not allow users, with valid credentials, to PUT resources owned by a different user unless explicitly allowed by that user. In the case of POST operations, we first validate if the user is attempting to update an existing resource by checking the *Owner Database*. If so, we create a XACML request with the *resource-owner* attribute set to the `id` of the actual owner. Since requests are first evaluated against the target section of a XACML policy (which also contains the *resource-owner* attribute (section 5.3)) this request will be rejected if no valid owner policy has allowed this user to update their resource.

- **Retrieve or Delete resources which he does not have permissions to:** An adversary with valid credentials can view or delete resources if permitted by an appropriate policy.
- **Create (update) crafted policies, which might in turn allow the attacker further access into the system:** An adversary in possession of a valid id can create policies but since policies are appended with the *resource-owner*'s id, these policies only affect the release of resources owned by the user whose credential (id) has been compromised. This means, our system does not allow an attacker, in possession of a single registered user id, to affect any and every stored resource.
- **View policies created by other users or the admin:** It is often desirable for a system to prevent unauthorized disclosure of system policies. Because stored policy ids are appended with ids of those who create them, our system filters policies on the basis of these ids the displays them to the users. This allows a user to view only those policies created by him/her.
- **Delete policies created by other users or the admin:** Similar to unauthorized disclosure of policies, unauthorized deletion of policies are also prevented by appending the user id with the policy id. When the adversary attempts to delete a random policy, the policy id provided by the adversary is appended with her user id. If this resultant id not is found in the system that policy is not deleted.

By the above made observations we can thus claim that in most cases an adversary possessing a valid user credential (id) can at the worst affect resources owned by the user whose credentials have been compromised. It should be however noted that when it comes to aggregate queries the adversary can corrupt eventual result by creating or updating false resources.

6.3.2 Impersonating a Valid/Registered User

As mentioned previously, attackers can impersonate valid users by either registering with the *Validation Server* or guessing/obtaining the ids. Our framework allows the two enhancements to limit the capabilities of an impersonating adversary. Firstly, we allow system developers to create strong user credential verification logics on the *Validation Server*. In addition, we allow coordination with external verification agencies like the Institutional Review Board (IRB). Secondly, TLS allows us to securely transport the randomly generated id to the registering user. Thus, following *Assumption* 2 from section 6.2, the only way an adversary can get unauthorized access to our system is by brute-forcing the randomly generated id. In other words, an adversary, incapable of registering as a valid user, is bounded by the computational complexity of brute-forcing the random ids. Increasing the length of the identifier, therefore, increases the security offered by our system.

6.3.3 Adversary Lacking a Valid User ID

- **Create false resources:** If the adversary does not possess a registered user id, the *Attribute Database* does not list this user as a "Poster" and hence does not allow creation of false resources.
- **Update existing resources created by others:** It was previously observed in section 6.3.1 that an adversary in possession of a valid user credential can only update resources owned by the user whose id he/she possesses. If the adversary does not possess any registered id, the *Attribute Database* does not have an entry for this user. As a result, the *PIP* does not obtain any user attributes related to this user and the hence the adversary is not allowed to update any resources stored at the *FHIR server*.
- **Retrieve or Delete resources which he does not have permissions to:** Similar to the update scenario, since the *Attribute Database* does not have an entry for this user, none of the stored policies allow this user to view or update any policies.
- **Create (update), view or delete policies:** In order to get access to the policy database the adversary needs to have an entry in the *Attribute Database* listing her id having the role of a "Poster". However, as seen earlier, if not permitted by the *Validation Server*, an entry is not created for this user in the *Attribute Database*. As a result, this user is not listed as a "Poster" in the *Attribute Database* and hence does not get access to administer any policies. In this context, it is to be noted that we do not allow users to administer policies without having the allowance to create data items. This is because we believe the user should only be able to administer policies pertaining to her resources in order to avoid conflicting policies from multiple users on same resources.

6.4 Motivating Scenario

Having analyzed the security of our system, we aim to present the effectivity of our system in dealing with real world security situations. For this, we make use of simple hypothetical access scenario pictured below.

> Doctor A is a general physician and doctor B is a specialist. Doctor A treats patient C. Incidentally doctor B is also a patient of doctor A. Thus doctor A also treats patient B. Doctor A creates resources for patient A and patient B on the FHIR server. Doctor A allows patient B to view her resource. Doctor A refers patient C to doctor B. Doctor A allows doctor B to view and update patient C's resources. Doctor A allows any researcher from organization X to view resources of all patient who hail from the city "Mounds".

With respect to the above mentioned scenario, we will evaluate the security of our system by subjecting it to both benign and malicious access attempts from different individuals in the set of all actors from the motivating scenario. An overview of the access scenarios in tabulated in table 1.

6.4.1 Scenario 1

In the first scenario doctor A attempts to create patient resources without being validated by the *Validation Server*. In this case, doctor A has not yet obtained a unique token from the *Validation Server* and if doctor A tries to forge a token, the *Attribute Database* would be missing the forged token and hence doctor A would not be identified as a "Poster".

Scenario Index	Actor	Operation	State	Type	Response
1	Doctor A	Create patient resources	Not Validated	Malicious	Deny
2	Doctor A	Create patient resources	Validated	Regular	Allow
3	Doctor A	View patient resources	Policy not in place	Malicious	Deny All
4	Doctor A	View patient resources	Policy in place	Regular	Allow self created resource only
5	Patient B	View self patient resource	Policy not in place	Malicious	Deny
6	Patient B	View self patient resource	Policy in place	Regular	Allow
7	Doctor B	View self patient resource	Policy allows patient B to view not Doctor B	Malicious	Deny
8	Patient B	Edit patient C resource	Policy not in place	Malicious	Deny
9	Doctor B	Edit patient C resource	Policy in place	Regular	Allow
10	Researcher from city "kolhn"	View all patient resources	Policy in place	Malicious	Deny all
11	Researcher from city "Mounds"	View all patient resources	Policy in place	Regular	Release patient (B and C) created by Doctor A

Table 1: Security Scenarios

6.4.2 Scenario 2

In scenario 2 doctor A validates with the *Validation Server* and obtains a unique token. He is thus identified as a "Poster" and our system allows him to POST new resources.

6.4.3 Scenario 3

In this scenario doctor A attempts to view the *all* patient resources. However, he is denied access to any Patient resource. This is because he requires to provide a policy which allows himself to view patient records.

6.4.4 Scenario 4

Doctor A provides a security policy as follows:

```
<Policy PolicyId = "POLICY-A"
                rule-combining-algorithm="deny-overrides">
  <Target>
  /* :Attribute-Category :Attribute ID :Attribute Value */
   AnyOf:
     AllOf:
      :access-subject      :xacml:subject-id   :A
      :resource            :xacml:resource-id  :Patient
      :action              :xacml:action-id    :GET
      :admin               :resource-owner     :A
  </Target>
  <Rule RuleId = "P" Effect="Permit">
  </Rule>
</Policy>
```

The policy implies that doctor A will be able to view Patient resources which are created by him. The "resource-owner" attribute guarantees that even if doctor A queries all Patient resources only those Patient resources (B and C) created by him are evaluated against this policy and returned back to him by the *PEP*. Another noticeable factor in the above mentioned policy is that the "PolicyId" is appended with doctor A's id which is "A". This ensures that when performing policy administration doctor A would only be able to view/modify/delete this policy and not any other policy with the id "POLICY".

6.4.5 Scenario 5

If patient B attempts to access her records he would not be allowed access since doctor A, the creator of her resource has not supplied a policy allowing patient B to view her resource.

6.4.6 Scenario 6

Doctor A creates a policy on the *Policy Admin* as follows:

```
<Policy PolicyId = "POLICY-PATIENTB-A"
                rule-combining-algorithm="deny-overrides">
  <Target>
  /* :Attribute-Category :Attribute ID :Attribute Value */
   AnyOf:
     AllOf:
      :access-subject      :xacml:subject-id   :B
      :resource            :xacml:resource-id  :Patient
      :action              :xacml:action-id    :GET
      :admin               :resource-owner     :A
  </Target>
  <Rule RuleId = "PB" Effect="Permit">
  <Condition>
     Function: string-equal
   /* :Attribute-Category :Attribute ID :Attribute Value */
      :Patient             :Patient.id.none     :B
      :access-subject      :Subject.role.none   :Patient
   </Condition>
  </Rule>
</Policy>
```

This policy allows patient B to view only Patient resources corresponding to her id.

6.4.7 Scenario 7

Doctor B, as mentioned previously, is also a patient of doctor A. However, according to the policy defined in the previous scenario an individual with id "B" and role "Patient" can view the resource of patient B. As a result doctor B is not allowed access to her Patient resource when he tries to access it as a "Doctor".

6.4.8 Scenario 8

Since patient C is also treated by doctor B, doctor A allows doctor B to edit/view patient C's resources by supplying the following policy:

```
<Policy PolicyId = "POLICY-PATIENTC-DOCTORB-A"
                rule-combining-algorithm="deny-overrides">
  <Target>
  /* :Attribute-Category :Attribute ID :Attribute Value */
   AnyOf:
     AllOf:
      :access-subject      :xacml:subject-id   :B
      :resource            :xacml:resource-id  :Patient
  AnyOf:
        :action              :xacml:action-id    :GET
        :action              :xacml:action-id    :PUT
        :action              :xacml:action-id    :POST
      :admin               :resource-owner     :A
  </Target>
  <Rule RuleId = "PB" Effect="Permit">
  <Condition>
     Function: string-equal
   /* :Attribute-Category :Attribute ID :Attribute Value */
      :Patient             :Patient.id.none     :C
      :access-subject      :Subject.role.none   :Doctor
   </Condition>
  </Rule>
</Policy>
```

However, the above supplied policy does not allow doctor B, assuming the role of a patient, to perform any action on the resource.

6.4.9 Scenario 9

The policy shown in the previous section allows doctor B to modify or view the resource belonging the patient C.

6.4.10 Scenario 10

Doctor A approves the request made by an organization from "Mounds", and allows any researcher from that city to view all resources POSTed by him by supplying the following policy:

```
<Policy PolicyId = "POLICY-PATIENTC-RESEARCHER-A"
                rule-combining-algorithm="deny-overrides">
  <Target>
  /* :Attribute-Category :Attribute ID :Attribute Value */
   AnyOf:
     AllOf:
      :access-subject      :Subject.role.none  :Researcher
      :resource            :xacml:resource-id  :Patient
      :action              :xacml:action-id    :GET
      :admin               :resource-owner     :A
  </Target>
  <Rule RuleId = "PB" Effect="Permit">
  <Condition>
     Function: string-equal
   /* :Attribute-Category :Attribute ID :Attribute Value */
      :access-subject      :Subject.address.city   :Mounds
   </Condition>
  </Rule>
</Policy>
```

According to the policy an individual with the role of a "Researcher" would be allowed to access all Patient resources POSTed by doctor A if and only if he is a native of the city "Mounds". Thus a researcher from the city "Kolhn" does not get access to Patient resources created by doctor A.

6.4.11 Scenario 11

According to the policy from the previous scenario an individual with the role of "Researcher" from the city "Mounds" gets access to the Patient resources posted by doctor A.

7. CONCLUSION AND FUTURE WORKS

We presented a framework for allowing fine grained authorization and access to FHIR resources. Each user is authenticated through the validation server which may consult other identity providers for this purpose. In addition, the validation server allows a user to modify her attributes that will permit/deny future access. It also allows resource creators to provide fine-grained policies controlling access to those resources. In future, we plan to improve the process of authentication by integrating the logic of the *Validation Server* with protocols like OpenId Connect [6]. Future work also includes developing an interactive easy to use framework that helps users in specifying syntactically and semantically correct policies.

Acknowledgement

The work was funded in part by NIST under Contract Number 60NANB16D250.

8. REFERENCES

[1] Anwar, M., and Imran, A. Access Control for Multi-Tenancy in Cloud-Based Health Information Systems. In *Proceedings of the 2nd IEEE International Conference on Cyber Security and Cloud Computing* (New York, USA, Nov 2015).

[2] Apache Axis2 - Apache Axis2 User's Guide-Introducing Axis2. https://axis.apache.org/axis2/java/core/docs/userguide.html, Oct 2016. [Online; Accessed 24-January-2017].

[3] Caumanns, Jörg and Kuhlisch, Raik and Pfaff, Oliver and Rode, Olaf. IHE IT-Infrastructure White Paper: Access Control. Tech. rep., IHE International, Sept 2009.

[4] Dong, X., Samavi, R., and Topaloglou, T. COC: An Ontology for Capturing Semantics of Circle of Care. *Procedia Computer Science 63* (2015), 589–594.

[5] eXtensible Access Control Markup Language (XACML) Version 3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html, Jan 2013. [Online; Accessed 24-January-2017].

[6] Final: OpenID Connect Core 1.0. http://openid.net/specs/openid-connect-core-1_0-final.html. [Online; Accessed 24-January-2017].

[7] Grunwell, D., Gajanayake, R., and Sahama, T. The Security and Privacy of Usage Policies and Provenance Logs in an Information Accountability Framework. In *Proceedings of the 8th Australasian Workshop on Health Informatics and Knowledge Management (HIKM 2015)* (Sydney, Australia, 2015).

[8] Hardt, D. The OAuth 2.0 Authorization Framework. Request for Comments 6749, Internet Engineering Task Force, Oct 2012.

[9] Herdman, R., and Moses, H. Effect of the HIPAA Privacy Rule on Health Research. In *Proceedings of a Workshop Presented to the National Cancer Policy Forum* (2006).

[10] Holz, R., Sheffer, Y., and Saint-Andre, P. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). Request for Comments 7457, Internet Engineering Task Force, Feb 2015.

[11] Hu, V. C., Ferraiolo, D., Kuhn, R., Friedman, A. R., Lang, A. J., Cogdell, M. M., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K., et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication 800-162, National Institute of Standards and Technology, January 2014.

[12] Hüffmeyer, M., and Schreier, U. Efficient Attribute Based Access Control for RESTful Services. In *Proceedings of the 7th Central European Workshop on Services and their Composition* (Jena, Germany, February 2015).

[13] Hüffmeyer, M., and Schreier, U. RestACL: An Access Control Language for RESTful Services. In *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control* (New Orleans, LA, 2016).

[14] Jersey. https://jersey.java.net/, Jan 2017. [Online; Accessed 24-January-2017].

[15] Lamprinakos, G. C., Mousas, A. S., Kapsalis, A. P., Kaklamani, D. I., Venieris, I. S., Boufis, A. D., Karmiris, P. D., and Mantzouratos, S. G. Using FHIR to Develop a Healthcare Mobile Application. In *Proceedings of the 4th International Conference on Wireless Mobile Communication and Healthcare - Transforming Healthcare Through Innovations in Mobile and Wireless Technologies* (Athens, Greece, Nov 2014).

[16] Li, M., Yu, S., Ren, K., and Lou, W. Securing Personal Health Records in Cloud Computing: Patient-Centric and Fine-Grained Data Access Control in Multi-owner Settings. In *Proceedings of the 6th Iternational ICST Conference on Security and Privacy in Communication Networks* (Singapore, September 2010).

[17] Machulak, M. P., Maler, E. L., Catalano, D., and van Moorsel, A. User-managed Access to Web

Resources. In *Proceedings of the 6th ACM Workshop on Digital Identity Management* (New York, NY, USA, Oct 2010).

[18] Mandel, J. C., Kreda, D. A., Mandl, K. D., Kohane, I. S., and Ramoni, R. B. SMART on FHIR: a Standards-Based, Interoperable Apps Platform for Electronic Health Records. *Journal of the American Medical Informatics Association 23*, 5 (Feb 2016), 899–908.

[19] Namli, T., Postaci, S., Gençtürk, M., Dogaç, A., Yalçinkaya, A., and Taskin, C. Addressing the Adoptability Challenges of the PHR Systems: SharingCare. In *Proceedings of eChallenges e-2013 Conference* (Dublin, Ireland, October 2013).

[20] Ness, R. B., Committee, J. P., and of Epidemiology f, S. Influence of the Hipaa Privacy Rule on Health Research. *Journal of the American Medical Association 298*, 18 (Nov 2007), 2164–2170.

[21] Organizations Interested in FHIR - HL7Wiki. http://wiki.hl7.org/index.php?title=Organizations_interested_in_FHIR#List_of_Organizations_.28in_Alphabetical_Order.29, Nov 2016. [Online; Accessed 24-January-2017].

[22] Fast Healthcare Interoperability Resources Overview - FHIR v1.0.2. https://www.hl7.org/fhir/overview.html, Oct 2015. [Online; Accessed 24-January-2017].

[23] Ray, I., Ong, T. C., Ray, I., and Kahn, M. G. Applying Attribute Based Access Control for Privacy Preserving Health Data Disclosure. In *Proceedings of the 2016 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)* (Las Vegas, NV, February 2016).

[24] Rescorla, E. HTTP Over TLS. Request for Comments 2818, Internet Engineering Task Force, May 2000.

[25] Security - FHIR v1.0.2. https://www.hl7.org/fhir/security.html, Oct 2015. [Online; Accessed 24-January-2017].

[26] Security-labels - FHIR v1.0.2. https://www.hl7.org/fhir/security-labels.html, Oct 2015. [Online; Accessed 24-January-2017].

[27] WSO2 Identity Server: WSO2 Identity & Access Management | WSO2 Inc. http://wso2.com/products/identity-server/. [Online; Accessed 24-January-2017].

[28] Zhang, R., and Liu, L. Security Models and Requirements for Healthcare Application Clouds. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing* (Miami, FL, 2010).