A Flexible Authorization Architecture for Systems of Interoperable Medical Devices

Qais Tasali
Department of Computer Science
Kansas State University
Manhattan, KS 66506, USA
qtasali@ksu.edu

Chandan Chowdhury
Department of Computer Science
Kansas State University
Manhattan, KS 66506, USA
chandanchowdhury@ksu.edu

Eugene Y. Vasserman
Department of Computer Science
Kansas State University
Manhattan, KS 66506, USA
eyv@ksu.edu

ABSTRACT

Robust authentication and authorization are vital to next-generation distributed medical systems – the Medical Internet of Things (MIoT). However, there is yet no good authorization model for real-time multi-channel data from systems of heterogeneous devices providing multiple physiological parameters for clinicians who may change on a minute-by-minute basis. We present a flexible authorization architecture for interoperable medical systems, and an implementation and evaluation in the context of the Medical Device Coordination Framework (MDCF) high-assurance middleware.

Our framework is based on the well-studied Attributed Based Access Control model, but we introduce a new method of attribute inheritance that provides more fine-grained access control, supporting multiple different authorization levels for multiple physiological data channels from the same device, and rich and expressive policy specification which facilitates plug-and-play connectivity of devices – most do not require pre-specification of individual permissions. Our architecture is standards-compliant and modular, using the eXtensible Access Control Markup Language (XACML), and Axiomatics Language for Authorization (ALFA) for policy specification, and standalone authorization modules which can be integrated with other platforms such as OpenICE. We stress-test our implementation in a realistic distributed system configuration, and show that the unoptimized system introduces negligible network and storage overhead, and minimal memory and CPU overhead.

CCS CONCEPTS

•Security and privacy →Access control; Authorization; •Applied computing →Health care information systems; •Computer systems organization →*Embedded and cyber-physical systems;* Client-server architectures; Sensors and actuators; **•Hardware** →*Safety critical systems;*

SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA

ACM ISBN 978-1-4503-4702-0/17/06…$15.00.
DOI: http://dx.doi.org/10.1145/3078861.3078862

1 INTRODUCTION

Future interoperable medical systems hold the promise of improved patient care through aggregation and manipulation of multiple physiological parameters simultaneously, as well as closed-loop control and automation of common clinical tasks. An early standard for such interoperability is the Integrated Clinical Environment (ICE) [21], first introduced in 2008. ICE is a medical system environment created by combination of interoperable heterogeneous medical devices and other integrated equipment. ICE allows controlling and monitoring of devices connected to a patient from a centralized location through medical applications, which allows better service toward the patients as clinicians can monitor all of their patients from their office and need not visit each patient as often. The systems allow medical device coordination through applications (apps), scripted medical workflows which orchestrate the action of one or more connected medical devices, and can operate either in closed-loop or open-loop control mode [16]. By automating common tasks, apps go beyond the functionality of checklists and increase safety and efficiency of clinician workflows. For a more detailed treatment of the benefits of medical apps we refer the reader to King et al.'s case study of rapid prototyping one such application [27].

Although the Medical Application Platform (MAP) [16, 46] and ICE concepts carry the potential for improvement of accuracy, consistency, and reliability in the practice of medicine, they also introduce new concerns – novel risks to patients' safety and privacy [8, 27]. For example, unauthorized access to the device(s) connected to a patient or an app controlling these devices could result in patient harm, or even death [21, 29]. Although access control concepts are well researched and mature in traditional medical systems, the same is not true when applied to new and emerging standards. There are few current systems claiming compatibility with the ICE standard, including OpenICE [33] and the Medical Device Coordination Framework (MDCF) [29]. Only one of these has implemented authentication [41]. Most research conducted in this area so far has been focused on how to control access to Electronic Medical Records (EMR) and Electronic Health Records (EHR) [9, 17, 22, 35, 45, 48], which are static data collected from doctor-patient interaction in healthcare facilities. Among the access control models used in medical domain, Role Based Access Control (RBAC) [12] and Attribute Based Access Control (ABAC) [18, 19] mostly dominate. However, RBAC lacks flexibility and dynamic access control capabilities [18], leading to the development and use of richer and more granular methods, such as Relationship-Based

Access Control (ReBAC) [38] and Role-Centric Attribute-Based Access Control (RABAC) [24], which are extensions of RBAC rather than entirely different models [15, 30, 37].

The proposed solutions are not entirely generalizable to newer system concepts such as Medical Application Platforms (MAPs) [16, 46]. The ICE standard (ASTM F2761) defines essential safety requirements for equipment comprising the patient-centric care network, but barely covers authentication and authorization. Other challenges in new systems include the high number of physiological data channels, the real-time nature of the communication, and the sheer number of clinicians, each of whom generally only interacts with the system for a few minutes. Moreover, it is less common now to grant permission based on evaluating a single static attribute or role of the user. Instead, an authorization request is evaluated using several different attributes, such as type of action, access time and location, the relation between subject and object, etc. Because some or most of these attributes do not become static once defined, access decisions must be made dynamically as well. For example, time- and location-aware access control systems may allow a clinician to access patient data during their shift while in the hospital, but may not be able to access the same patient data after their shift is over or if they try to access the data remotely. Dynamic authorization management not only allows organizations to react quickly to changing healthcare regulation but offer other benefits such as up-to-date centrally managed authorization policies, consistency in authorization policies, and reduced administrative work.

In this paper we present a new flexible authorization architecture for real-time "plug-and-play" [1] interoperable medical systems [16, 21]. We use ABAC as a basis for its flexibility, but find ourselves with a novel challenge, requiring a new type of solution. Apps are a fundamentally different type of principal – there is always an app intermediary between the clinician and the patient, so we need to reason about the roles and permissions of the clinician and app simultaneously. The app and clinician have differing permissions, and different access patterns: the app is a long-running clinical workflow, but clinicians may come and go while the app operates. For instance, while the clinician who launches and initially configures the app may have one set of permissions, a later operator may have a completely different permission set. At any given time, an arbitrary operator may need access to some or all of the data used by the app, requiring evaluation of the operator's role separately from that of the app, since the operator may hold a subset or a superset of permissions for the app's data and functionality. We solve this problem through **attribute inheritance**: at app launch time, if the app lacks permissions to access some device(s) that are needed for proper functionality, the permissions are reevaluated by combining the app's and requesting operator's permissions. If the combined permissions allow the app to function properly, it *inherits the permissions of the operator who started the app*. More precisely, the app now runs with permissions which are the union of the app's inherent permissions and the permissions of the operating clinician. In a traditional access control workflow, all that is needed is a single check, at app launch time, whether the operator has permission to use *all* data, devices, and functions provided by the app. In our case, the app is its own subject, and requires authorization to access any device providing patient's physiological data. **Inheriting an attribute is done for the purpose of temporarily expanding permissions**, so attribute inheritance is a mechanism to achieve temporary permission elevation, and may be thought of as permission inheritance.

This work makes the following contributions:

- We show the first proof-of-concept implementation of authorization for systems of plug-and-play interoperable medical devices.
- Our system is sufficiently rich and fine-grained to accommodate principals in the form of devices, apps, and clinician of numerous arbitrarily-defined roles.
- We augment traditional ABAC with a novel method of attribute inheritance that not only achieves fine-grained access control and separation of duty requirements, but also helps in creating genericized policies that support plug-and-play of medical devices for immediate authorized use by clinicians, such as during an emergency.
- We show that our authorization system performs sufficiently well to support very frequent authorization events, such as for protecting dynamically produced data generated in real time.
- Out architecture is flexible-enough to support integration into most implementations of device interoperability standards, such as the Integrated Clinical Environment (ICE).

2 BACKGROUND

We are aware of only two open implementations which claim compatibility with the ICE standard, including OpenICE [33] and the Medical Device Coordination Framework (MDCF) [29]. OpenICE was developed by the Medical Device Plug and Play Interoperability Program (MD PnP) [1] to automate peer-to-peer node discovery, data publishing and subscribing between nodes, and proprietary medical device protocol translations [33]. OpenICE allows users to convert heterogeneous medical device data from supported devices into a common structure and protocol and exchange the data on a different machine using demonstration clinical applications. Like the MDCF, OpenICE does not have an authorization system. We choose the MDCF over the OpenICE to implement our proof-of-concept system solution. This is due to several factors, the most important being that MDCF has a pluggable communication protocol layer, and therefore does not rely exclusively on a particular third-party network protocol (DDS, in case of OpenICE [34]). We also use the modularity of the MDCF security framework to assist in our design and integration [40, 41]. A detailed comparison of the benefits and downsides of the two implementations is beyond the scope of this paper, but it is worth noting that our proposed authorization architecture is designed for any ICE-like architecture, and therefore it should be possible to integrate into OpenICE. Verifying the exact difficulty of integration into OpenICE, and thus testing the generality of out design, is left for future work.

2.1 MDCF

The Medical Device Coordination Framework [27, 29] is an open testbed for medical device integration and coordination. The MDCF is a Medical Application Platform (MAP) [16], architected in logical units that closely follows the Integrated Clinical Environment (ICE) standard [21]. It allows medical devices to be controlled by scripted

medical workflows – apps. The system is divided into two large sub-components according to the ICE architecture: 1) Supervisor (the app and user interface host), and 2) Network Controller (the communication abstraction layer). Communication is abstracted as "channels", allowing the MDCF to use different network communication library implementations, such as MIDdleware Assurance Substrate (MIDAS) [28] and Data Distribution Service (DDS) [34], as message-oriented publish/subscribe middleware.[1] Each component is described briefly below and illustrated in Figure 1.

Supervisor Components

- *The App Manager* manages the lifecycle of apps. It starts and stops the execution of apps, manages interactions (communication) between apps, and notifies clinicians of any medically adverse architectural interactions, if applicable.
- *The Clinician Service* provides an interface for configuring, instantiating, and selecting supervisor apps that are used with the clinician graphical user interface (GUI).
- *The Administrative Service* is the control provider for installation and management of apps.

Network Controller Components

- *The Channel Service* is the function set for direct interaction with the communication substrate, and contains code for interfaces used between the MDCF publish/subscribe middleware and any connected components. It contains interfaces for the messaging server, message senders, receivers, and connection listeners, as well as hooks for pluggable authentication providers.
- *The Connection Manager* is the means to create, manage, and destroy connections (abstracted as channels).
- *The Device Manager* configures devices for use with apps and maintains an internal view of device status.
- *The Device Registry* is the known device information store API. These may be device models for which information has been preloaded, or individual devices which are currently connected or have connected in the past.
- *The Component Manager* is analogous to the Device Registry, but is responsible for apps instead of devices.

2.2 Representing Access Control Policies

The design of the Medical Application Platform concept [16] like the MDCF, and the devices meant to interoperate within it, implies the expectation of generic but preassigned device and app attributes that can easily be transformed into access control attributes in authorization policies [41]. These attributes can be preassigned by device vendors, app developers, and/or the defaults can be overridden by the clinical facility for use in the environment where devices and apps are deployed. A facility administrator, creating policies for use by the clinical environment, generates a set of authorization rules in a machine-readable, but human-*usable*, language, creating device and clinician/operator attributes (which can be modified at any time, even as the system is running). Runtime changes to these attributes can result in decision changes between two access requests without the necessity to change the device, app or user relationships

[1] Within the Java code, components that have a registered sender/receiver object for a channel are the only components that can send or receive messages in that channel, limiting access to the channel by permitting or denying subscription requests [41].

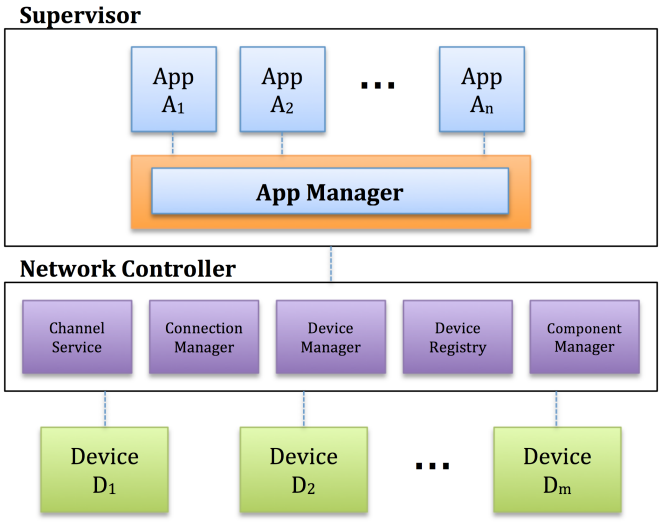


Figure 1: The architecture of the MDCF and its components

defining any underlying rule-sets. To make the process as easy as possible for the policymaker, not only are there predefined defaults for common roles and property sets, but authorization policy generation can be partially or fully automated for common classes of devices and apps – device instances do not require dedicated policies, and in many cases devices with analogous functionality or new models or product lines, even from different manufacturers, can reuse policies written for similar devices. Moreover, policies implemented in ABAC are only limited by the language used to express them, and the richness of the available attributes [18]. Therefore, there is no need to specify individual relationships between each device (or even device class) and each potential operator without sacrificing granular user permissions. We use the eXtensible Access Control Markup Language (XACML) [31, 32] as our back-end, due to its standardization (OASIS) and wide acceptance and portability (it is one of the most widely used policy language).

XACML. The eXtensible Access Control Modeling Language [32], written in XML, is used to define a fine-grained attribute-based access control policy. It can also be used to express an architecture and a processing model that describes how to evaluate access requests according to the rules predefined in access polices. XACML is designed to be suitable for a variety of application environments, such as social networks [8], home automation gateways [25], healthcare domain [10, 37, 44], distributed systems [31], etc.

ALFA. A major goal of XACML is to promote common terminology and interoperability between authorization implementations by multiple vendors. It is very general and expressive, but is verbose and hard to read. The Abbreviated Language for Authorization (ALFA) [2] is a Domain Specific Language (DSL) for XACML. In contrast to policies written in XACML, ALFA provides a friendlier and more usable syntax, similar to C#. Access control polices written in "raw" XACML are complex, and make it difficult to find faults which may be inadvertently introduced [47]. Therefore, we chose ALFA as a high-level policy language due to its increased user-friendliness over XACML, but kept the XACML back-end to

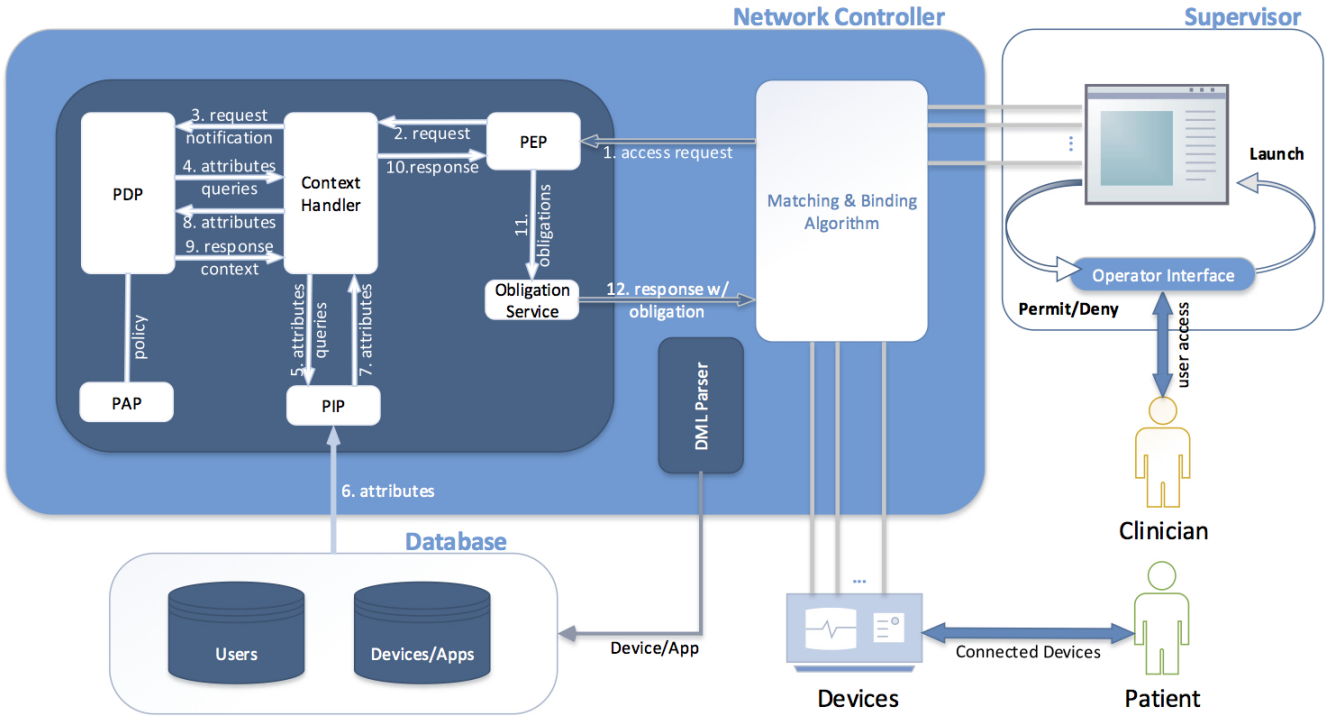


Figure 2: The basic architecture of the new authorization system after integration with the MDCF. Numbering represents the order of operations.

maximize portability. For a quantitative comparison, one of our ALFA policies is 30 lines, while a XACML policy generated from *one line* of code (***target clause app.role == "aR1"***) taken from the ALFA policy is 16 lines. On average, the policies which we wrote consisted of 3903 non-whitespace characters in XACML, while the same policies in ALFA had 528 non-whitespace characters.

2.3 Current Workflow

To better illustrate the workflow of the authorization system, we introduce patient Pamela and her primary nurse Nick. Pamela has had surgery, and is now on pain relief medication (opioid delivered through a PCA pump). Nick wants to access real-time telemetry from sensors monitoring Pamela to watch her vital signs for indications of an accidental opioid overdose, and to change the dosage (set the PCA Pump level). To do so, Nick opens the Clinician Console and launches the `PCAShutoff` app [6] that displays Pamela's SpO2 (Blood Oxygen Saturation), EtCO2 (End-Tidal Carbon Dioxide), and RR (Respiratory Rate).

The MDCF workflow for setting up a PCA pump interaction with the `PCAShutoff` app is:

(1) Clinician accesses the Clinician Console
(2) Console connects to the App Manager to fetch and display the list of available apps
(3) Clinician selects and launches the `PCAShutoff` app from the list of available apps
(4) Console relays the request to the App Manager
(5) App Manager launches the app
(6) Now-running App requests the list of required devices from Device Manager
(7) Via the Connection Manager, the App requests the Device Manager to connect to devices and request physiological data (and displays it to the clinician)
(8) Clinician changes the infusion dose (PCA pump level) in `PCAShutoff` app
(9) App forwards the new level to the PCA pump
(10) PCA pump changes the dose to the new value
(11) App requests updated values from the devices (and displays them to the clinician)

Without authentication or authorization, all these steps can be performed by anyone at any access level, such as any app, any clinician, etc. For example, anyone can access the Clinician Console and request a change to the PCA pump level. Anyone on the network can even connect to the PCA pump and directly request that it increase the medication level. We detail the workflow with added authentication and authorization in Section 3.

3 DESIGN OF THE AUTHORIZATION SYSTEM

Figure 2 shows the main architectural components of the authorization system. All XACML requests for access are sent to the Policy Enforcement Point (PEP). It forwards the request to the context handler in its native request format, which may include attributes for subjects, resource, action, environment and/or any other custom categories. Once the context handler receives the request for access, it generates a request context, which may include attributes, and

forwards the request context to the Policy Decision Point (PDP). The context handler will also handle queries for any additional attributes requested by the PDP. When additional attributes are requested, the context handler retrieves the requested attributes from the Policy Information Point (PIP). It is responsible for obtaining the requested attributes and returning the requested attributes to the context handler. After receiving the requested attributes, the PDP evaluates the policy and returns the response context to the context handler, where the response context is translated to the native response format of the PEP. After the response context is translated, it is sent to the PEP to fulfill the obligation. Obligations are additional constraints to an authorization decision and if PEP cannot fulfill any given obligations then it disallows access. The response context also contains the authorization decision and if the access is permitted, the PEP permits access to the resource. Otherwise, the PEP denies the access to the resource. The Policy Access Point (PAP) is responsible for policy creation.

The MDCF provides a channel abstraction for communication. To control access to the data in the MDCF communication substrate, we need to restrict access to channels, making decisions before either the user-app (clinician accessing an app) or app-device (app subscribing/publishing to a device) connection is created. Since the Network Controller is the component of the MDCF that manages all channel services and connections including creation and destruction of channels, it is the clear choice to host the authorization engine. When a clinician starts interacting with the console and launches an app, the App Manger within the Supervisor generates a user-app access request and forwards it to the Network Controller for evaluation. Only after the authorization engine within the Network Controller returns "permit" are the connections created.

There are two different phases of access control evaluation which take place before a clinician receives access to some features of an app or to physiological data from device(s) in the MDCF. The first phase is evaluation of clinician's access request for accessing app(s). When a clinician launches an app, a XACML launch access request is generated and forwarded to the Network Controller and is evaluated by the authorization engine. Once a "permit" decision for app launch is returned, the second phase is invoked to obtain authorization to access the device(s) that is/are required by the app. The app's requirements are identified internally and automatically by the MDCF's Matching and Binding algorithm. Authorization requests for required devices are generated and forwarded to the authorization engine. Note that the second phase comes into play only if the first phase returns a "permit" decision – if a clinician does not have access to an app, then there is no need to check if the app is authorized to subscribe/publish to required device(s) – the request is simply denied.

3.1 Authorization Policies

Authorization policies in the MDCF can be one of two categories: user-app authorization policies and app-device authorization policies. While access control rules for user-app authorization require use of several different categories of attributes, the app-device rules only requires subject and action attributes, and/or some conditions. For example, the below given rule will allow the `PCAShutoff` app having role ***aR*** (subject attribute) to set the data interval rate for a multimonitor device. (In this access control rule, app is a subject and the data interval rate – from a device – is a resource.)

```
Allow access to resource
  with attribute "dataIntervalRateForPCA"
    if Subject "app" has role "aR"
    and action is "set"
```

The authorization policies created for app access by clinicians have more complex rules. Each has several categories and conditions. The access rule in the example below will allow a clinician to access a patient's SpO2 reading only if 1) the clinician is a nurse who 2) is assigned to the primary physician of the patient, 3) has active role ***nR***, and 4) is working during his or her assigned shift.

```
Allow access to resource
  with attribute "SpO2"
       if Subject "clinician" has role "nR"
       and action is "get"
conditions:
  Subject "clinician" is "PrimaryPhysicianOfPatient"
  and Subject "clinician" is in their "shift"
```

3.2 Plug-and-play Support

The proposed authorization architecture can also support "plug-and-play" connectivity of new devices [1] by encouraging reuse of policies for other, similar device and app *types*. Administrators at a clinical environment with the MDCF deployment can define authorization polices for common classes of devices and apps, categorized based on common functionalities and components (capabilities). In order for the authorization system to restrict access to plug-and-play medical devices only to authorized users, all devices and apps are required to carry a set of attributes predefined in their Device Modeling Language (DML) schemas and to be parsed by the MDCF built-in DML parser [26]. Once a device or app is connected to the MDCF, the device registry and component manager in the Network Controller retrieve the DML (configuration) schema for the device or app and store it. The DML parser also retrieves access control attributes within the schema and automatically generates authorization policies based on the access control attributes. If automatic policy generation fails due to a new component or feature of the device or app not being recognized by the system, or errors resulting from missing or improper access control attributes, the administrator will receive a notification regarding the failure of the policy generation. They will then be asked to generate a custom policy for the device or app. More details on this can be found in Section 3.3, but an extensive discussion of the plug-and-play in the MDCF is beyond the scope of this paper.

3.3 Attribute Inheritance

Our design is unique in treating clinicians and apps as distinct and independent actors, and therefore they may have differing permissions for accessing devices. Our solution to this challenge, attribute inheritance, not only improves permission granularity without exposing the policy writer to additional complexity, but also allows **authorized** plug-and-play support *with only a few simple additional policies*. User role inheritance, has been in use for decades, first introduced in the RBAC framework [42]. Relationship among roles in a given organization are defined by a role hierarchy. In a typical healthcare facility, if we pick a specialist surgeon who is

```
namespace edu.ksu.santoslab.mdcf {
 import edu.ksu.santoslab.mdcf.mAttributes.*

 rule allowGet {
   target clause action.actionId == "GET"
   permit
 }

 rule allowSet {
   target clause action.actionId == "SET"
   permit
   condition exchange.exchangeTime >= user.shiftStart &&
             exchange.exchangeTime <= user.shiftEnd
 }

 policyset polMultiMonitorSample {
   target clause resource.resourceId == "*.pulserate.alerts.
         seperation_interval"
   apply denyUnlessPermit
   polMultiMonitorSampleSET
   polMultiMonitorSampleGET
 }

 policy polMultiMonitorSampleSET {
   target clause app.role == "aR1" or app.role == "aR2"
     clause user.role == "Critical_Care_Nurse"
   apply denyOverrides
   allowSet
 }

 policy polMultiMonitorSampleGET {
   target clause app.role == "aR2" or app.role == "aR3" or app.role
         == "aR4"
     clause user.role == "Cardiothoracic_Surgeon" or user.role == "
           Agency_Nurse"
   apply denyOverrides
   allowGet
 }
}
```

Figure 3: An example written in ALFA, a subset of the policy used in evaluating our implementation

```
<xacml3:Target>
  <xacml3:AnyOf>
    <xacml3:AllOf>
      <xacml3:Match MatchId="urn:oasis:names:tc:xacml:1.0:function:
      string-equal">
        <xacml3:AttributeValue
          DataType="http://www.w3.org/2001/XMLSchema#string">aR1</
      xacml3:AttributeValue>
        <xacml3:AttributeDesignator
          AttributeId="edu.ksu.cis.santos.mdcf.app.role-attrID"
          DataType="http://www.w3.org/2001/XMLSchema#string"
          Category="urn:oasis:names:tc:xacml:1.0:subject-category:
                access-subject"
          MustBePresent="false">
      </xacml3:Match>
    </xacml3:AllOf>
  </xacml3:AnyOf>
</xacml3:Target>
```

Figure 4: The XML generated from one line code (*target clause app.role == "aR1"*) in Figure 3

senior in position to a resident surgeon, then the specialist inherits any roles held by the resident. However, this type of role inheritance is not only impractical, but actually impossible in a system like the MDCF, where apps can also be actors (subjects) at certain times. On other hand, the relationships among clinicians, apps, and devices can be captured by using their attributes – yet another reason why we need Attribute-Based Access Control (ABAC).

To explain why user *role inheritance* (attribute inheritance) is needed, it is worth first understanding the main entities of the authorization system. A device in the MDCF is always considered a resource (object).[2] On the other hand, a clinician is always an actor (subject). An app can be an actor or a resource depending on the access scenario. It is a resource when a clinician tries to access it, but that same app can later assume the role of an actor (subject) when it tries to access data from device(s), e.g. patient physiological parameters. Therefore, an app inheriting a user role does not suggest the app will replace its own role with it. Instead, the user role is added as an extra attribute in the access request.

The app always plays the role of an interface between a user and a device, so it is illogical for an app to hold more than one role – there is no way of changing the active role for an app during a single session, unlike a clinician. Instead, based on the set of components and features offered by an app, it will be assigned to a specific role. Thus, all apps are categorized into common classes of features. In contrast, clinicians may be assigned more than one role, but can only have one active role at a time – we refer to this as the clinician's *active role*. Clinicians can switch between roles whenever needed. The clinician is always expected to have access to more resources than an app because an app is always restricted to certain attributes (e.g. one pre-assigned role based on the app's type, category, or feature set). Since neither an app nor a clinician can replace each others' roles, we will not benefit from permissions from user's active role unless the app inherits the clinician's role as an extra attribute when needed (in the second phase of our two-phase access control evaluation design). Note when we say the permissions set for a clinician's active role, we mean the resources to which the clinicians is granted access when the clinician's role is added to a XACML request as a required attribute in the second authorization step. Splitting the access control decision into two phases in this manner is a way to limit having to invoke attribute inheritance, using it only when it is needed, and allowing us to define a more fine-grained access control model without writing more complex policies. **This achieves separation of duty.**

This method of user role inheritance, or in general attribute inheritance, not only provides least privilege to both user and app but also can drastically cut the time required to make newly installed devices available for use (e.g. during an emergency). Note also that this *does not require bypassing the authorization system* – devices are available for authorized use.

To examine this concept in detail, consider the example wherein a clinician discovers the need for some physiological data to be collected from a patient, but none of the already-connected medical devices is able to provide the data due to lack of features or incompatibility with the app. The clinician connects a new plug-and-play medical device to the system that can collect the data from the

[2] This may change in future work, as devices and apps become increasingly similar.

patient and is compatible with the app. Since this is a new device, there is a high probability that it will not be available for immediate use due to lack of authorization policies. There are two possible ways to handle access permissions for a newly installed device in the MDCF, and we analyze both options below:

(1) *Use generic predefined authorization policies:* The administrator needs to generate a set of generic policies based on common features sets offered by each device. Similarly, the administrator ensures that apps which are fully or partially compatible with these devices are authorized to access these common features. However, we believe that, for safety, generic predefined authorization policies should only be limited non-safety-critical features of devices, so apps would not be authorized to access all features, but rather a conservative, safe subset. *Generic policies* which allow apps full device access may, in certain specific cases, violate the goals of the authorization system.
(2) *Generate new custom authorization policies:* The administrator introduces a new set of attributes, which will require generation of new authorization policies for each feature each device offers.

In option 1, a newly installed device becomes automatically available for use, but it is expected that any safety-critical features of the device, such as setting infusion rate for a PCA pump, will *not* be available until explicitly authorized by the system administrator since these features can only be used by explicitly authorized apps. This prevents their use in an emergency. Moreover, creating policies that explicitly authorize apps to use safety-critical features of a device defeats the purpose of *generic* predefined policies. Alternatively, in option 2, all features of the newly-installed device can be made available for use by an app if an administrator explicitly pre-authorizes the app to access these features by generating custom authorization policies. However, this requires time and deep understanding of healthcare facility-specific access control rules for these device-specific (often unique) features. Thus, neither option 1 nor option 2 is very effective in an emergency. Regardless of which option is chosen, the safety-critical features (capabilities) of a device – the ones most needed in an emergency – will not be available for immediate use after first-time device connection. *Attribute inheritance* provides the solution to the above problem in two steps, one addressing the clinician and the other the app. They are addressed in turn below.

If a clinician meets the access requirements for the use of some safety-critical features offered by a newly-connected medical device, or any devices that may be connected to the system at a later time (e.g. during an emergency), then authorization policies for all features (including critical features) can be governed by simple, generic policies written in advance, and the clinician will gain authorization for these features dynamically, as needed. For clinicians, administrators write policies based on a per-facility understanding of the clinician's role (responsibilities), and authorize clinician access to known devices. While this seems identical to option 2, apps introduce a layer of complexity which is still unresolved, and are less-well understood by administrators.

Clinician authorization alone does not solve the problem, since an app intermediary is still needed to access the data from a device, and the app may lack permission to interact with the device. Detailed per-app policies are more complex to write than per-clinician role policies, since app features, capabilities, and data access requirements are not as well understood, and may not even be known in advance. Nonetheless, if a clinician has been previously authorized to access safety-critical features of a device that just connected to the system (through per-clinician role policies), but the clinician is trying to access these features using an app that lacks access permissions, then if apply *attribute inheritance* again so the app *inherits the user role as an extra attribute* and therefore receives authorization to access the data from the device, both the device and app become available for immediate use. In other words, attribute inheritance allows authorized interaction in a new way: **even though neither the clinician nor the app *alone* are authorized, the clinician-app *pair* is authorized**.

Policies generated for attribute inheritance purposes do not fit into either option 1 nor 2 above, since we are neither writing custom policies at the time of device connection, nor we are generating generic policies that will allow any *single* actor full access to the new device. Instead, our approach is a hybrid third option and allows for full authorized access to the resource while maintaining the principles of least privilege and separation of duty. The sample access control rule below shows how user and app roles, with attribute inheritance, are combined to allow authorized access to a device with safety-critical features.

```
Allow access to resource
  with attribute "medicationInfusionRate"
    if Subject "app" has role "aR"
      and action is "set"
conditions:
      Subject "clinician" has role "nR"
```

The above policy shows how to merge clinician and app roles in order to allow the clinician-app pair to access the given resource, and also provide separation of duty. The rule tells us that an app with role (subject attribute) ***aR*** is authorized to set the infusion rate for a device (infusion pump) connected to the patient only if a clinician with role (subject attribute) ***nR*** is issuing the command. Authorization policies generated for an app that is compatible with (capable of connecting to) a given medical device should include app (subject) attributes, (in this example, ***aR***).

A far more extensive authorization example, written in ALFA, is given in Figure 3. It showcases policy-level details on usage of user and app roles (for the purpose of attribute inheritance). In the policy called "polMultiMonitorSampleSET", the user role "critical care nurse" needs to be inherited by any app having either role "AppSpO2" or "AppPulse", in addition to other required attributes, for the app to set the separation interval for pulse-rate alerts of a multimonitor device.

3.4 Break the Glass

Attribute inheritance should not be considered a substitute for Break The Glass (BTG) features [7, 14], but it does help achieve safe BTG. While the authorization system in a healthcare facility ensures the system is only accessed by authorized users, it may also prevent a clinician from delivering potentially life-saving care to patients during an emergency due to lack of permissions. In this case, saving the patient outweighs any risks associated from

overriding access controls, which can be partially deactivated by using BTG features of the system.

BTG allows overriding access controls and provides full (or sufficient) access to the system during an emergency, but attribute inheritance is a step in eliminating the need for a "global" override. Instead, it can be used to allow controlled, authorized access to medical resources (e.g. devices) during emergencies. This is achieved using a hybrid of dynamically and automatically generated app-device interaction polices (at the time of device access) and pre-defined clinician-app interaction polices.

We understand the importance and challenges of controlled BTG in the medical domain, but the scope of this work is deliberately limited to attribute inheritance and lacks a full treatment of BTG. Detailing the design of a Break The Glass feature within our proposed authorization architecture is future work.

3.5 Modified Workflow

Here we show the new workflow, *with authentication and authorization.* Steps which are modified or added to the original workflow from Section 2.3 are italicized. Figure 5 provides a visual aid.

(1) Clinician accesses the Clinician Console
(2) *Console asks for username and password*
(3) *Clinician enters username and password*
(4) *Console forwards the request to Shiro (PEP)*
(5) *Shiro verifies the entered credentials against the stored value and returns yes or no to the Console*
(6) *If yes is returned, the clinician is successfully authenticated and is logged in*
(7) Console connects to the App Manager to fetch and display the list of available apps
(8) Clinician selects and launches the `PCAShutoff` app from the list of available apps
(9) *Console receives the app launch request and forwards it to Shiro, along with the users' details*
(10) *Shiro takes the request, adds context (user's active role, type of request, timestamp etc.), and forwards it to Balana for an authorization check*
(11) *Balana checks the request against stored XACML policies and returns "yes", "no", or "not applicable"*
(12) *Shiro receives the authorization result from Balana*
(13) *If Shiro receives "no", it forwards it to the Console, and the clinician is denied app launch permission, ending the workflow (otherwise the workflow continues)*
(14) Console relays the request to the App Manager
(15) App Manager launches the app
(16) Now-running App requests the list of required devices from Device Manager
(17) *The access request is forwarded to Shiro*
(18) *Shiro forwards the request to Balana to confirm if the app has been authorized to connect to devices*
(19) *Balana checks the request against stored XACML policies*
(20) *If "yes" is returned, the app is allowed to connect to devices*
(21) *If "no" is returned, the request is reevaluated with the clinician's active role appended* [3]

[3]This step is not yet implemented

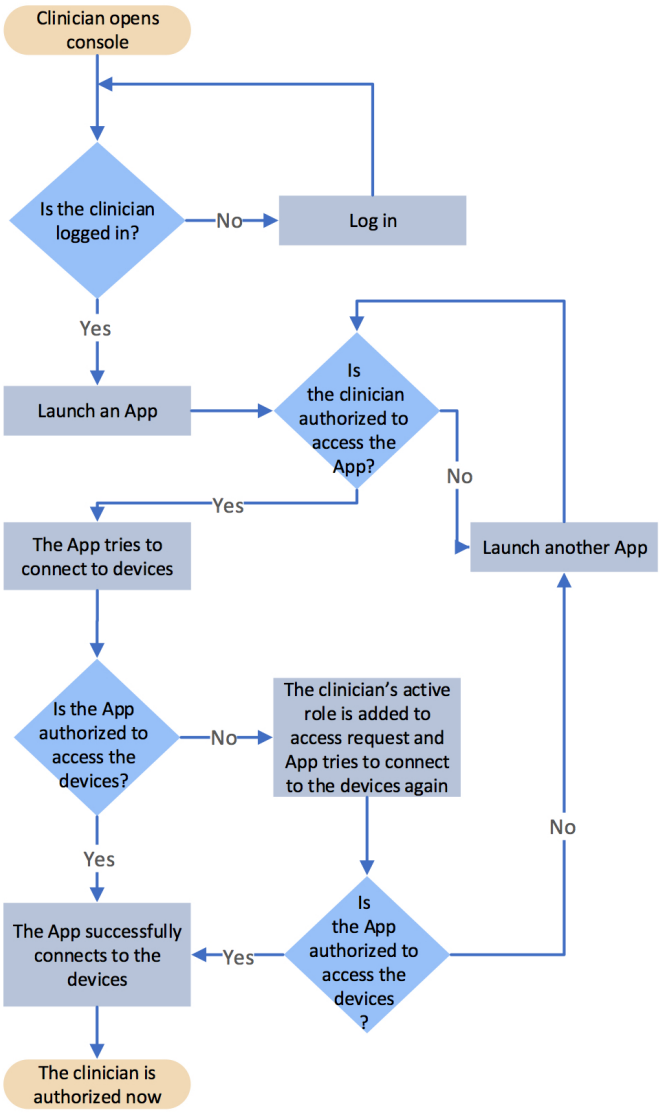
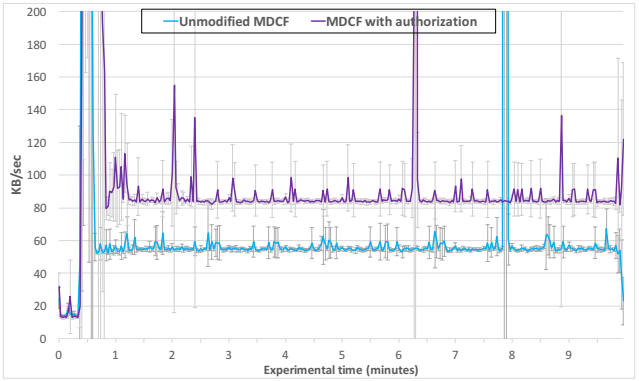


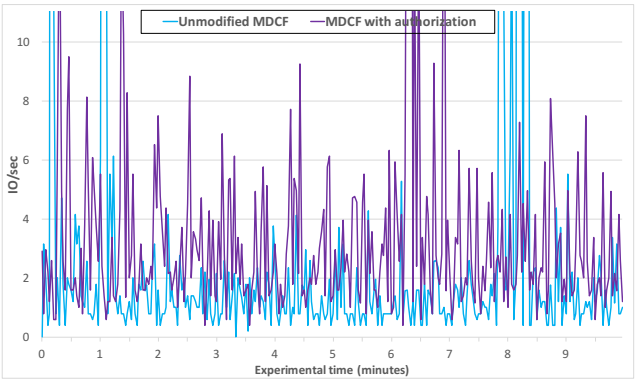
Figure 5: The workflow for a clinician accessing patient physiological data in the MDCF

(22) *If "no" is returned from request reevaluation, the final decision (deny) is returned to the Supervisor and the clinician is denied access (the app can start, but cannot perform useful work), ending the workflow (otherwise the app is allowed to connect to devices and the workflow continues)* [3]
(23) *The final decision (permit) with obligation(s) is returned to the Supervisor, and the clinician is allowed to launch the app*

If the clinician chooses to view physiological data for a patient or tries to make changes to either device or app parameters, the requested access request must be evaluated. For example, after the clinician has successful been authenticated and authorized to launch the application, the clinician may try to change the infusion rate for the PCA pump connected to a patient. The steps to authorize the clinician to change the infusion rate are:



(a) Network (total-read) usage



(b) Disk I/O with error bars omitted. The variability between baseline and modified versions is *not statistically significant.*

Figure 6: Disk and network I/O of the baseline versus authorization-enabled MDCF without Shiro caching

(1) Clinician changes the infusion dose (PCA pump level) in PCAShutoff app
(2) *The app requests for access to change the level*
(3) *An access request is generated*
(4) *Shiro adds context to the request and forwards it to Balana*
(5) *Balana checks if the clinician is allowed to change the level*
(6) *If "no", the app discards the change and displays a denied message to the clinician*
(7) *If "yes", the dose change request is sent to the PCA pump*
(8) PCA pump changes the dose to the new value
(9) App requests updated values from the devices (and displays them to the clinician)

Each action is checked for proper authorization. Authentication prevents malicious access to the Clinician Console, and authorization prevents users from doing things they are not authorized to do. For example, another nurse can be given permission to launch the `PCAShutoff` app to monitor the PCA pump and SpO2 level, but not to change the PCA pump level. An unauthenticated intruder in the network can send request to the PCA pump to change the level, but the message will be rejected as they are not authenticated. Authenticated malicious actors can likewise request the PCA pump to change the level, but the message will be rejected as unauthorized.

3.6 Implementation Details

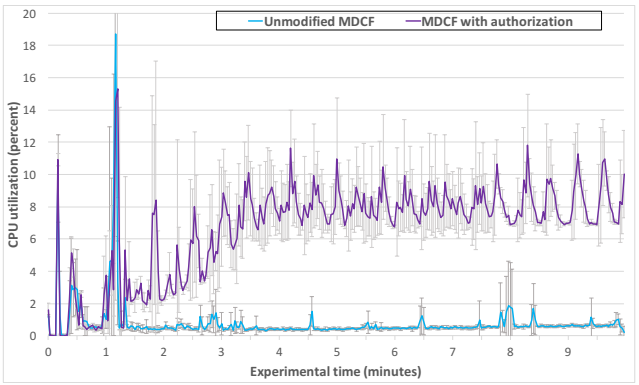
We chose Apache Shiro [5] as our Policy Enforcement Point (PEP) framework and WSO2 Balana [43] for the Policy Decision Point (PDP). Balana was a natural choice due to its maturity, rich feature set and flexibility for XACML. For the Policy Information Point (PIP), which stores details like usernames, passwords, groups, user-group relations etc., our requirements were that it be open source and compatible with Balana. Cost was also a significant consideration. We considered several options for PEP, such as Spring Security [36], OACC [3] and Shiro. We looked into Spring Security, but we are not currently using the Spring framework. OACC offers native Java compatibility, but it is relatively new (compared with Shiro) and does not come with pluggable authentication protocols, such as for LDAP. For PIP, as per our requirements, we decided on a simple SQLite database to store the PIP information. In a future version, the database will be replaced by a enterprise class user-management systems like LDAP or Kerberos. This will be a relatively easy change, since Shiro provides out of the box support for these kind of systems via configuration file parameter (for example, `securityManager.realms`). Also, we found it easier, at the proof-of-concept state, to use Shiro only for authentication, and delegate all authorization tasks to Balana. In future we may rethink our strategy and try to implement authentication using OACC.

The MDCF project already has a graphical Clinician Console for local or network access. Chromium-based and implemented in Dart, the code-base had pre-existing hooks for authorization calls – this is where we integrate our new code. As the PEP, Shiro receives all such requests from the Clinician Console. Shiro handles the authentication requests itself and acts as a mediator for authorization between the front-end and Balana. To keep the logic simple, we implemented two custom classes, `MDCFAuthenticator` and `BalanaAuthorizer`, by extending the `AuthorizingRealm` class of Shiro. `MDCFAuthenticator` is only responsible for authentication, and `BalanaAuthorizer` is only responsible for authorization.

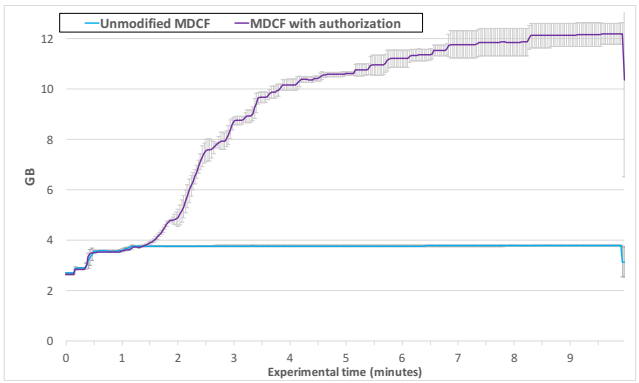
`MDCFAuthenticator` performs password-based authentication. If successful, an active role is assigned to the user. For simplicity, we currently use the first role found in the list of the user's available roles rather than ask the user. In future work, we will assign a role based on a saved user preference. After initial role assignment, the user can always change the active role using the Clinician Console.

When an action requires authorization, e.g. launching an app, the MDCF front-end sends the user details and action attributes (e.g. which feature of the medical application the user is trying to access) to the PDP, which performs the authorization check against available XACML policies and returns a response. Authorization is implemented as the `BalanaAuthorizer` Java class which

- Is initialized with a set of XACML policies,
- Takes as input the attributes of the app or device, and the details of the user requesting access, and
- Returns true if the user has access via any of the user's available roles.

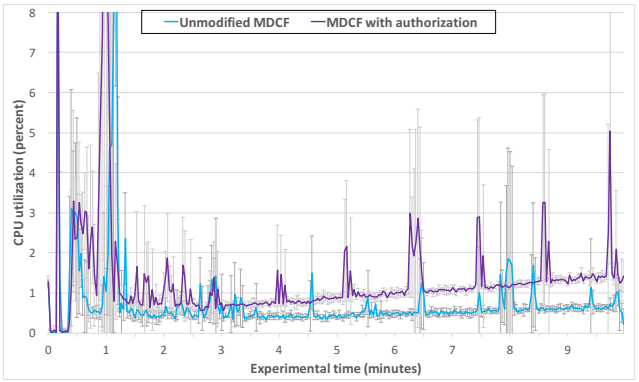


(a) CPU usage

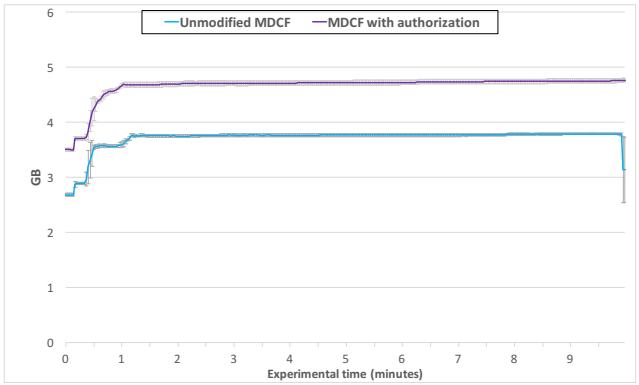


(b) Memory usage. Error bars are too small to be visible in some places.

Figure 7: CPU and memory usage of the baseline versus authorization-enabled MDCF without Shiro caching



(a) CPU usage



(b) Memory usage. Error bars are too small to be visible in some places.

Figure 8: CPU and memory usage of the baseline versus authorization-enabled MDCF with Shiro caching

4 EVALUATION

In this section we describe the testing methodology and results for the authorization system as integrated into the MDCF. The tests were designed to exceed the normal expected operational capacity of the system (also called stress-testing) to confirm safe usage limits and given specifications are met. Our goal was to maintain the same level of system performance (of the unmodified implementation of the MDCF) for the MDCF with authorization. (Note that phase two of the two-phase authorization design, including attribute inheritance, is not yet implemented and was thus excluded from the evaluation.) We compared the unmodified and modified (authorization-enabled) implementations of the MDCF with 64 medical devices, and observed the system for usage limits, particularly CPU, memory, network, and disk I/O. Since the intended use of the MDCF is for a single patient (as specified in the ICE standard [21]), our tests involved far more than the number of devices expected to be used simultaneously.

We used simulated (virtual) devices for our testing of the system to get around the lack of readily available medical hardware and the incompatibility of current physical medical devices with the MDCF [27], and due to the exceptional computing power afforded by our test harness when compared to physical devices. We used three different types of virtual devices: capnography (CO2), pulse oximetry (SpO2) and patient-controlled analgesia (PCA), each with multiple physiological output channels and control (input) channels. The clinician workflow also included use of a `PCAShuttoff` medical workflow script (app), which requires simultaneous use of three devices, one of each type. During the test, each running instance of the app was subscribed to a different set of virtual devices to simulate simultaneous treatment of multiple patients, each with an associated set of devices and a controlling `PCAShuttoff` app.

For consistency of testing environments between the pre-authorization and authorization-enabled versions of the MDCF, testing was partially automated. We ran the MDCF server using a Linux server (dual octa-core 64-bit Intel Xeon E5520 CPUs at 2.27 GHz, with 8 MB/core cache and 64 GB memory). The 64 devices were ran from two different machines with an identical configuration to the MDCF server. This allows stress-testing of the server without local interference from devices, i.e. devices and server computing resources are distinct and do not interfere with each other except

through communication. The 64 devices started connecting to the server after 20 seconds from the time the server began running. The initial peaks in the performance graphs are the result of 64 devices connecting to the sever simultaneously. Each device begins sending physiological data after successfully connecting and authenticating to the MDCF server. Once all devices were connected successfully, the user launches an app after successfully authenticating and verifying authorization. User interaction with the MDCF clinician console was timed, with the launch command issued at the 60th second of the experiment. The test was repeated 5 times for each version of the MDCF.

Figure 6 shows the difference(s) in network and I/O usage between the two implementations. The authorization-enabled MDCF (modified implementation) performed as expected, within normal parameters, even under stress testing. Furthermore, authorization imposed no statistically significant I/O overhead, and minimal to no network overhead (one standard deviation, or 68% confidence). The initial peak (between 0 and 1 minutes) results from the sudden connection of all 64 devices, as intended for stress testing. The average network usage for the unmodified and modified MDCF was 78.97 ± 28.91 KB/s and 116.52 ± 30.89 KB/s, respectively. Similarly, the average I/O usage for the modified and unmodified MDCF was 3.23 ± 4.15 IO/sec and 3.41 ± 1.00 IO/sec, respectively.

Figure 7 shows the CPU and memory utilization for the modified and unmodified versions of the MDCF, showing statistically significant overhead: 95% confidence interval for CPU and > 99% for memory. CPU utilization for the unmodified and modified MDCF averaged 0.77 ± 0.05 % and 6.63 ± 0.04 %, respectively. Note that the CPU visualization in Figure 7a is somewhat misleading, as it accounts for only a 5.86% (on average) overhead from the inclusion of authorization. Memory usage shows the unmodified MDCF using on average 3.62 ± 0.02 GB of the 64 GB available memory, whereas the modified system used on average 9.02 ± 0.12 GB, an increase of 5.4 GB: almost 250%. The reason for this (unexpected) memory overhead was the undocumented excessive use of JDBC connections to the authorization server: each authorization request created a new, *persistent* connection. Since the authorization engine needs to access the database for each access request, we end up with far too many JDBC connections, which persist throughout the experimental run, accounting for not only the memory overhead but also its steady increase over time. In fact, the Clinician Console requests data update from the server at the rate of 16 queries per sec, resulting in about 1000 new JDBC connection objects per minute, which also explains the unexpected CPU overhead.

Figure 8 shows a significant reduction of both CPU and memory overhead due to several simple optimizations: using the built-in Shiro caching API [4], and limiting JDBC to one persistent connection, brought CPU usage to within statistical indistinguishability from the unmodified MDCF (< one standard deviation difference), as shown in Figure 8a, and memory overhead to 20%, also removing the memory growth over the time of the experiment, as shown in Figure 8b. CPU utilization and memory usage for the cache-enabled modified MDCF averaged 1.32 ± 0.05 % and 4.55 ± 0.35 GB of the 64 GB available memory, respectively. Our initial experiment resulted in overhead of 5.8% CPU utilization and 5.4 GB memory (on average) from the inclusion of authorization, whereas the optimized modified MDCF resulted in an overhead of 0.55% CPU utilization and 0.93 GB memory, on average.

5 RELATED WORK

The differences between healthcare facilities and their individualized, unique access control requirements have resulted in many proposed access control models, all suitable for healthcare. Though a detailed discussion of the access control models themselves is not in the scope of this paper, here we provide several examples of how some of these models work.

Most research on access control in medical domain has focused on Electronic Health Record systems (EHRs), storing data which is accessed or changed only occasionally. The variety of EHRs makes it unlikely that a one-size-fits-all access control model will be used. A small sampling of proposed access control models for distributed EHRs can be found in [11, 20, 23, 37]. Each proposed solution uses different methods to achieve patient privacy. Ray et al. [37] use ABAC to ensure the disclosure of Protected Health Information (PHI), in response to requests from researchers, conform to various policies imposed by patients. Hupperich et al. [20] discusses the problems with some current proposed solutions for privacy, such as the use of smart cards for EHR authorization, and propose a flexible secure architecture based on attribute-based encryption and scalable authorization secrets to enable patient-controlled security and privacy. Moreover, availability of resources during emergencies is an active area of access control research [39]. Various solutions have been proposed [7, 13] to override access restrictions in a controlled manner.

None of the proposed models provides a solution and/or addresses the need for an access control model for real-time patient data generated by medical devices within a heterogeneous, interoperable environment such as standardized by the ASTM Integrated Clinician Environment (ICE) standards or one following the concepts of Medical Application Platforms (MAPs) [46]. Authorization within ICE-compliant medical middleware has been not been studied, nor are the concepts covered in the associated standards [21], which do not provide any authentication or authorization requirements or specifications. Salazar discusses authentication and authorization requirements for MAPs, and designs out a proof-of concept authentication framework scheme within the MDCF [41], but does not present an authorization architecture, except for a high-level design rooted in the Role-Based Access Control (RBAC) model [40]. Salazar's main contribution are limited to ensuring the trustworthiness of medical devices connecting to the MDCF, creation and integration of a flexible authentication system into the MDCF, and evaluation of the implemented system. We show that RBAC is insufficient to fulfill the requirements for dynamic access control required for an ICE-compliant system, and provide an alternative design based on ABAC.

6 CONCLUSION

In this paper, we presented the design, architecture, and evaluation of a flexible authorization architecture for systems of interoperable medical devices with a proof-of-concept implementation within the Medical Device Coordination Framework (MDCF). Our work is a first attempt to implement an authorization system within an

ICE standards-compliant medical middleware and provide access control to real-time data generated by the systems of interoperable medical devices. Our unique approach to attribute inheritance makes our access control model significantly different from prior models used in medical domain to protect mostly static electronic medical records, without sacrificing granularity or expressive power. Attribute inheritance also provides clinicians with authorized emergency access to medical devices, especially interoperable "plug-and-play" devices. Evaluation results show that our authorization architecture performs well, scales to many devices with many distinct physiological data channels, and is sufficiently flexible to integrate with other implementations of the ICE standard or Medical Application Platforms (MAPs). A more thorough and rigorous comparative analysis of our architecture with other flavors of RBAC/ABAC, e.g. RABAC, as well as with different performance characteristics of alternate design choices for the PEP, PDP, and policy languages, is left for future work. Additional future work includes a more complete exploration of the benefits of attribute inheritance, including how it can contribute to implementing a controlled Break The Glass procedure (along with "Fix The Glass").

ACKNOWLEDGMENTS

This work was funded by NSF grant 1253930. The authors also wish to thank Matthew French for his patience in explaining the inner workings of the MDCF.

REFERENCES

[1] Medical device "plug-and-play" interoperability program. http://mdpnp.org. (Accessed on 2/20/2017).
[2] Axiomatics language for authorization (ALFA). https://www.axiomatics.com/solutions/products/authorization-for-applications/developer-tools-and-apis/192-axiomatics-language-for-authorization-alfa.html, 2015. (Accessed on 2/21/2017).
[3] OACC — Java application security framework. http://oaccframework.org/, 2016. (Accessed on 2/21/2017).
[4] Apache Shiro — Simple. Java. Security. https://shiro.apache.org/caching.html. (Accessed on 4/25/2017).
[5] Apache Shiro — Simple. Java. Security. https://shiro.apache.org/documentation.html. (Accessed on 1/12/2017).
[6] S. Barrett. The MDCF PCA Shutoff App 0.3 Documentation. http://people.cs.ksu.edu/~scbarrett/pcashutoff-doc/, 2015.
[7] A. D. Brucker and H. Petritsch. Extending access control models with break-glass. In *SACMAT*, 2009.
[8] A. Carreras, E. Rodríguez, and J. Delgado. Using XACML for access control in social networks. In *W3C Workshop on Access Control Application Scenarios*, 2009.
[9] R. Chandramouli. A framework for multiple authorization types in a healthcare application system. In *ACSAC*, 2001.
[10] A. A. El-Aziz and A. Kannan. Access control for healthcare data using extended XACML-SRBAC model. In *ICCCI*, 2012.
[11] J. Eyers, David M.and Bacon and K. Moody. OASIS role-based access control for electronic health records. *IEEE Proceedings – Software*, 153(1), 2006.
[12] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *TISSEC*, 4(3), 2001.
[13] A. Ferreira, D. Chadwick, P. Farinha, R. Correia, G. Zao, R. Chilro, and L. Antunes. How to securely break into RBAC: The BTG-RBAC model. In *ACSAC*, 2009.
[14] A. Ferreira, R. Cruz-Correia, L. Antunes, P. Farinha, E. Oliveira-Palhares, D. W. Chadwick, and A. Costa-Pereira. How to break access control in a controlled manner. In *CBMS*, 2006.
[15] P. W. Fong. Relationship-based access control: Protection model and policy language. In *CoDASPY*, 2011.
[16] J. Hatcliff, A. King, I. Lee, A. MacDonald, A. Fernando, M. Robkin, E. Y. Vasserman, S. Weininger, and J. M. Goldman. Rationale and architecture principles for medical application platforms. In *ICCPS*, 2012.
[17] J. Hu and A. C. Weaver. A dynamic, context-aware security infrastructure for distributed healthcare applications. In *Workshop on Pervasive Privacy Security, Privacy, and Trust*, 2004.
[18] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (ABAC) definition and considerations (draft). NIST Special Publication 800-162, 2013.
[19] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. Attribute-based access control. *IEEE Computer*, 48(2), 2015.
[20] T. Hupperich, H. Löhr, A.-R. Sadeghi, and M. Winandy. Flexible patient-controlled security for electronic health records. In *SIGHIT*, 2012.
[21] Medical devices and medical systems-essential safety requirements for 5 equipment comprising the patient-centric integrated clinical environment 6 (ICE)-part 1: General requirements and conceptual model 7. ASTM F2761, 2008.
[22] J. Jin, G.-J. Ahn, H. Hu, M. J. Covington, and X. Zhang. Patient-centric authorization framework for sharing electronic health records. In *SACMAT*, 2009.
[23] J. Jin, G.-J. Ahn, H. Hu, and X. Covington, Michael J.and Zhang. Patient-centric authorization framework for electronic healthcare services. *Computers & Security*, 30(2), 2011.
[24] X. Jin, R. Sandhu, and R. Krishnan. RABAC: Role-centric attribute-based access control. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, 2012.
[25] M. Jung, G. Kienesberger, W. Granzer, M. Unger, and W. Kastner. Privacy enabled web service access control using SAML and XACML for home automation gateways. In *ICITST*, 2011.
[26] Y. J. Kim, S. Procter, J. Hatcliff, V. P. Ranganath, and Robby. Ecosphere principles for medical application platforms. In *ICHI*, 2015.
[27] A. King, D. Arney, I. Lee, O. Sokolsky, J. Hatcliff, and S. Procter. Prototyping closed loop physiologic control with the medical device coordination framework. In *ICSE/SEHC*, 2010.
[28] A. L. King, S. Chen, and I. Lee. The middleware assurance substrate: Enabling strong real-time guarantees in open systems with OpenFlow. In *ISORC*, 2014.
[29] A. L. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. P. Jetley, P. L. Jones, and S. Weininger. An open test bed for medical device integration and coordination. In *ICSE Companion*, 2009.
[30] M. Li, S. Yu, K. Ren, and W. Lou. Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner settings. In *SECURECOMM*, 2010.
[31] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First experiences using XACML for access control in distributed systems. In *ACM Workshop on XML Security*, 2003.
[32] OASIS. 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, 2013.
[33] OpenICE User Introduction. https://www.openice.info/docs/1_overview.html.
[34] G. Pardo-Castellote. OMG data-distribution service: Architectural overview. In *International Conference on Distributed Computing Systems, Workshops*, 2003.
[35] M. Peleg, D. Beimel, D. Dori, and Y. Denekamp. Situation-based access control: Privacy management via modeling of patient data access scenarios. *Journal of Biomedical Informatics*, 41(6), 2008.
[36] Pivotal Software, Inc. Spring security. https://projects.spring.io/spring-security/, 2017. (Accessed on 2/21/2017).
[37] I. Ray, T. C. Ong, I. Ray, and M. G. Kahn. Applying attribute based access control for privacy preserving health data disclosure. In *IEEE-EMBS BHI*, 2016.
[38] S. Z. R. Rizvi, P. W. Fong, J. Crampton, and J. Sellwood. Relationship-based access control for an open-source medical records system. In *SACMAT*, 2015.
[39] L. Røstad. *Access control in healthcare information systems*. PhD thesis, Norwegian University of Science and Technology, 2008.
[40] C. Salazar. A security architecture for medical application platforms. Master's thesis, Kansas State University, 2014.
[41] C. Salazar and E. Y. Vasserman. Retrofitting communication security into a publish/subscribe middleware platform. In *FHIES/SEHC*, 2014.
[42] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.
[43] M. Siriwardena. Balana. https://github.com/wso2/balana. (Accessed on 1/12/2017).
[44] S. Sucurovic. An approach to access control in electronic health record. *Journal of medical systems*, 34(4), 2010.
[45] S. K. Tzelepi, D. K. Koukopoulos, and G. Pangalos. A flexible content and context-based access control model for multimedia medical image database systems. In *Workshop on Multimedia and Security: New Challenges*, 2001.
[46] E. Y. Vasserman and J. Hatcliff. Foundational security principles for medical application platforms. *Information Security Applications, LNCS*, 8267, 2014.
[47] D. Xu, Z. Wang, S. Peng, and N. Shen. Automated fault localization of XACML policies. In *SACMAT*, 2016.
[48] L. Zhang, G.-J. Ahn, and B.-T. Chu. A role-based delegation framework for healthcare information systems. In *SACMAT*, 2002.