Authorization in interoperable medical systems

by

Qais Tasali

B.S., Kabul University, 2009

MSE, Kansas State University, 2013

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2020

Abstract

Robust authentication and authorization are vital to next-generation distributed medical systems - the Medical Internet of Things (MIoT). Although future interoperable medical systems 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. For example, unauthorized access to the device(s) connected to a patient or a medical app (e.g. automated workflow) controlling these devices could result in patient harm, or even death. Furthermore, while in non-safety-critical systems confidentiality is generally prioritized over availability – an explicit "fail-closed" requirement – in medical cyber-physical systems (mCPS) availability must be prioritized over other security properties (because cessation of therapy may be lethal to the patient). This makes it challenging to craft least-privilege authorization policies which preserve patient safety and confidentiality even during emergency situations. Previous work has suggested a virtual version of "Break the Glass" (BTG), an analogy to breaking a physical barrier to access a protected emergency resource such as a fire extinguisher or "crash cart". In healthcare, BTG is used to override access controls and allow for unrestricted access to resources, e.g. Electronic Health Records. After a "BTG event" completes, the actions of all concerned parties are audited to validate the reasons and *legitimacy* for the override.

In this dissertation, we present a flexible authorization architecture for interoperable medical systems, and implementation and evaluation of the proposed architecture in the context of the Medical Device Coordination Framework (MDCF) high-assurance middleware. We also show how to handle emergency access control override natively within the attribute-based access control (ABAC) model, maintaining full compatibility with existing access control frameworks, putting BTG in the policy domain rather than requiring framework modifications to support it. This approach makes BTG more flexible, allowing for fine-grained facility-specific policies, and even automated auditing in many situations, while maintaining the principle of least-privilege. We do this by constructing a BTG "meta-policy" which works with existing access control policies by explicitly allowing override when requested, with well-defined procedures to return the system to a known secure state with minimal manual auditing. Finally, we formally verify that the resulting combined set of access control policies (as joined with the BTG meta-policy) correctly satisfies the goals of both the original policy set and of BTG. We show how to use the same verification methods to check new or modified policies in real time, easing the process of crafting least-privilege policies.

Authorization in interoperable medical systems

by

Qais Tasali

B.S., Kabul University, 2009

MSE, Kansas State University, 2013

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2020

Approved by:

Major Professor
Eugene Y. Vasserman

Copyright

Abstract

Robust authentication and authorization are vital to next-generation distributed medical systems - the Medical Internet of Things (MIoT). Although future interoperable medical systems 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. For example, unauthorized access to the device(s) connected to a patient or a medical app (e.g. automated workflow) controlling these devices could result in patient harm, or even death. Furthermore, while in non-safety-critical systems confidentiality is generally prioritized over availability – an explicit "fail-closed" requirement – in medical cyber-physical systems (mCPS) availability must be prioritized over other security properties (because cessation of therapy may be lethal to the patient). This makes it challenging to craft least-privilege authorization policies which preserve patient safety and confidentiality even during emergency situations. Previous work has suggested a virtual version of "Break the Glass" (BTG), an analogy to breaking a physical barrier to access a protected emergency resource such as a fire extinguisher or "crash cart". In healthcare, BTG is used to override access controls and allow for unrestricted access to resources, e.g. Electronic Health Records. After a "BTG event" completes, the actions of all concerned parties are audited to validate the reasons and *legitimacy* for the override.

In this dissertation, we present a flexible authorization architecture for interoperable medical systems, and implementation and evaluation of the proposed architecture in the context of the Medical Device Coordination Framework (MDCF) high-assurance middleware. We also show how to handle emergency access control override natively within the attribute-based access control (ABAC) model, maintaining full compatibility with existing access control frameworks, putting BTG in the policy domain rather than requiring framework modifications to support it. This approach makes BTG more flexible, allowing for fine-grained facility-specific policies, and even automated auditing in many situations, while maintaining the principle of least-privilege. We do this by constructing a BTG "meta-policy" which works with existing access control policies by explicitly allowing override when requested, with well-defined procedures to return the system to a known secure state with minimal manual auditing. Finally, we formally verify that the resulting combined set of access control policies (as joined with the BTG meta-policy) correctly satisfies the goals of both the original policy set and of BTG. We show how to use the same verification methods to check new or modified policies in real time, easing the process of crafting least-privilege policies.

Table of Contents

[†]This chapter includes results published as part of "A Flexible Authorization Architecture for Systems of Interoperable Medical Devices" [1]

[‡]This chapter includes results published as part of “Controlled BTG: Toward Flexible Emergency Override in Interoperable Medical Systems” [2]

List of Figures

List of Tables

Acknowledgments

My doctoral journey has been nothing short of amazing, and I have truly enjoyed every mile of the journey. Of course, this would have not been possible without continuous support of several individuals.

First of all, I would like to thank my major professor, Eugene Vasserman, for his insight and unwavering support. I would have never made it this far, if it wasnt for his guidance throughout my research in the past 5 years. I would also like to thank Prof. Torben Amtoft, Prof. Mitch Neilsen, and Prof. Caterina Scoglio, for serving as my dissertation committee members and helping me achieve excellency in my research through their expertise. I am also thankful to Prof. David A. Gustafson for his encouragement and helping me with the decision to pursue a doctoral degree.

Next, I would like to thank my colleagues from SyNeSec lab for their contribution to this dissertation, dearest friends who brought home to me here in Manhattan, and everyone else who became my family at Kansas State University. I am grateful to Chandan, a best friend for life, for his authentic interest in my research.

Finally, biggest thanks go to my family for their continuous support and unconditional love. My dear mom, Malika, for all of the sacrifices that she has made on my behalf to help me get to this point. My brothers, Hewad and Farhad, who have only ever pushed me to follow my passions. My sisters-in-law, Sailany and Marghalara, who are like sisters to me, for always believing in me. My spouse, Fatimah, for having been extremely supportive of me throughout this entire process. My daughters, Sailghai and Yosra, and my nieces, Salena, Sana, Maryam and Hela, who have continually provided the much-needed breaks from philosophy and the encouragement to finish my degree with expediency. I love you very much.

Dedication

To the dear memory of my father, Abdul Hakim Tasali.

Chapter 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),[4] first introduced in 2008. ICE is a medical system environment which can be created dynamically through a combination of interoperable heterogeneous medical devices and other integrated equipment. ICE allows centralized control and monitoring of devices connected to a patient through a medical applications, which can provide better service to patients, since clinicians can monitor many patients from a single location, and need not visit each patient as often. The system can also allow visiting or unaffiliated clinicians (e.g. GPs) to check on their patients.

These benefits come at the cost: open heterogeneous systems face many security challenges. Without proper authentication and authorization, there is nothing to stop an unauthorized or even malicious user from interfering with patient care, underscoring the critical importance of authorization and authentication (AA) in medical systems. While authentication and authorization in health record systems is well researched and mature, the same is not true about new and emerging standards. The ICE standard (ASTM F2761)[4] defines essential safety requirements for equipment comprising the patient-centric care network, it barely covers authentication and authorization, which is left as a de-facto open area for research. Most research conducted so far in the area of authorization in medical domain is focused on how to control access to Electronic Medical Records (EMR) and Electronic Health Records (EHR), which are static data collected from doctor-patient interaction in healthcare facilities.[5–10] The proposed solutions are not entirely generalizable to systems of interoperable medical devices due to 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. Currently, there are few implementations claiming compatibility with the ICE standard, including OpenICE[11] and the Medical Device Coordination Framework (MDCF).[12] Only one of these has implemented authentication.[13]

Traditionally, restricting access to resources can be achieved by either a) allowing access to selected resources and denying everything else by default (whitelisting), or b) denying access to selected resources and allowing everything else by default (blacklisting). While whitelisting is considered a more secure fail-closed option, blacklisting is less secure but *safer* in the context of fail-open (high availability) requirements. Non-safety-critical systems tend to prioritize confidentiality and authorization over availability, usually expressed in system authorization policies as a concrete "fail-closed" requirement. However, in certain domains, particularly in medical cyber-physical systems (mCPS), fail-closed is not always the safest approach: in some emergency situations, medical systems' availability must be prioritized over other security properties, leading to a non-traditional access control model. The Health Insurance Portability and Accountability Act (HIPAA)[14;15] in the United States also requires that availability of medical resources be prioritized over patient's privacy (also known as "fail-open" requirement) to ensure medical systems do not deny life-saving treatment to patients in unforeseen situations such as medical emergencies. Nevertheless, unauthorized access to device(s) connected to a patient or an app controlling these devices could result in patient harm, privacy violation, or even death.[1;12;16] This seemingly impossible situation is encountered in medical facilities all too regularly. As a result, defining an authorization policy that can follow the principle of least-privilege as closely as possible without compromising patient safety or confidentiality (e.g. protect patient information from unauthorized access) even during unforeseen situations is an unresolved and ongoing challenge. Previous work and industry practices have suggested a virtual version of the “Break the Glass” (BTG) concept,[17] an analogy to breaking a physical barrier to access a protected resource such as a fire extinguisher during a fire, or a “crash cart” or AED for a medical emergency. In healthcare, BTG is used to override access controls and allow for unrestricted access to resources, e.g. Electronic Health Records. After a “BTG event” completes, the actions of all concerned parties are generally audited, requiring detailed logging of what happens during BTG. Post-hoc analysis can determine the reasons and *legitimacy* for overriding access controls. Medical BTG has largely been treated in the literature as an all-or-nothing scenario: either unrestricted access is provided (BTG allowed; fail-open) or BTG is not supported (fail-closed). We show how to bridge this gap using an access control model and set of “BTG-compliant” policies which maintains the power and flexibility of policy-based dynamic access control decisions, provides structured logging and auditing functionality, and allows for automated system rollback to a known-secure state after the emergency has passed.

To simplify (for demonstration purposes) the resource categorization and permission assignment at an arbitrary facility, we use the following listed *three groups* for all available resources for an arbitrary clinician role:

1. Resources the clinician is authorized to access in order to fulfill their duties
2. Resources the clinician is not authorized to access, except to deal with unexpected situations
3. Resources the clinician is never authorized to access regardless of situation

Consider Nurse `N` (clinician) who is authorized (assigned permissions) to access resources (e.g. vitals) in group 1 throughout the normal performance of his or her duties. Under normal circumstances, `N` will be denied access to resources in groups 2 (e.g. exceeding soft limit on infusion pumps)[*] and 3 (e.g. access control policies). However, the healthcare facility understands the need for access (read/write) to resources such as infusion pumps

[*]Soft limits are lower or upper dosing limits that are not meant to be overridden and are considered a safety feature of infusion pumps.[18]

Resource Categorization and Access Control Rules			
Clinician	Group 1	Group 2	Group 3
N	allowed	denied unless BTG requested	denied

Table 1.1: Permission assignment based on our approach

in unexpected situations and thus allows `N` to access the resources in group 2 only during emergencies.

Group 3 represents resources which are not part of `N`'s job duties or workflow, and would be disallowed during normal operation. Moreover, group 3 may include critical resources for maintaining the security and integrity of the authorization system itself, e.g. the access control policies themselves and the database that contains them, which would be specifically restricted from access during emergencies (see Section 4.1.4). Therefore, `N` is not allowed to access group 3 during normal operation, nor during emergencies. We leave the flexibility for individual facilities to not restrict access to these resources. Such changes to global defaults can be implemented purely via policy alterations and require no framework modification.

While it is relatively straightforward to define an authorization policy for the groups 1 and 3 using traditional access control models, defining a policy for group 2 would require an understanding of all unexpected situations internal and external to a medical system – an impossible task by definition. Therefore, an alternative solution is to deny access to these resources by default except when the clinician initiates an emergency access session (i.e. BTG), which also makes the system "fail-open". If a system or application is developed to remain operational and allow access during unexpected situations when failure conditions are present, it is called to be "fail-open". A traditional "fail-closed" system will block access and/or prevent further operation if a failure is detected.

Developing an access control model for a given system entirely depends on system requirements and/or the effort that it takes to tailor or extend a given model to meet those requirements. In fact, access control models in the medical domain are so diverse that usually an access control model developed for one environment may not be useful or applicable in another environment. While a number of access control models are used in medical domain, RBAC and ABAC are the two most common.[9;19–21] They excel in protecting data in a closed environment, where all resources and users are known. However, they fall short when used for enforcement of access control in a dynamic environment, which requires taking into account information context, users, and objects that are not known prior to issuing a "deny" or "allow" decision. There are additional commonly-used access control models, such as ReBAC,[22;23] which are extensions of RBAC. These are mainly developed to overcome shortcoming of RBAC.

RBAC is mostly used for modeling organizational authorization/security policies in commercial systems, but it has limited flexibly and lacks dynamic access control capabilities.[20] Whereas frameworks that control these "system-of-systems" require more robust and dynamic access control. For example, the Medical Device Coordination Framework (MDCF), which is an open source MAP that facilitates interoperability between heterogeneous medical devices and is designed to be an open test bed for the conceptual architecture described by the Integrated Clinical Environment (ICE) interoperability standard, requires a flexible dynamic authorization architecture to control access to medical devices and apps (scripted medical workflows) considering context information. In addition, the MDCF also facilitates implementation of medical device coordination applications through its model-based development environment.[12]. These applications not only increase safety and effectiveness of medical care but also the efficiency of clinician workflows.[16]

The use of richer and more granular methods, such as ABAC and ReBAC, has been on the rise in recent years due to the increasing need for dynamic and context-aware access control. It is less common now to grant permission based on evaluating a single static attribute (e.g. 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.[24–26] 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 regulatory requirements but also offer several other benefits, such as up-to-date centrally managed authorization policies, consistency in authorization policies and less administrative work. RBAC requires administrators to reassign and revoke roles for a user when the users status changes, or even reassign permissions on individual resources. This means more administrative work, which also results in out-dated and less consistency in authorization policies.

1.1 Contributions

We enhance ABAC by introducing a new method of attribute inheritance – "user role inheritance" – into our access control mechanism. Since in the MDCF there is always an app intermediary between the clinician and the patient, we need to reason about the roles and permissions of the clinician and app simultaneously. This is more challenging than it initially appears, since the app and clinician are both principals in the system, and therefore have differing permissions, and moreover 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. 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. We solve this problem through *permission 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 *permanently* 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. The details of this important augmentation to traditional ABAC can be found in Section 3.4.

We present a new flexible authorization architecture and its implementation. Our architecture is based on ABAC due to its more flexible access control model (compared to unmodified RBAC) and dynamic access control capabilities. In addition, defining a relationship among principals (clinicians, devices, and medical applications) using attributes can be done whereas defining a relationship among devices, apps and clinician, while the first two can change states between subject and object dynamically as described in 3.4, using roles as suggested in RBAC framework is not possible.

Furthermore, we approach BTG from a more flexible standpoint, and demonstrate how to first "Break the Glass" and then "Fix the Glass" within systems of interoperable medical devices and applications, on a time-bounded, patient-by-patient basis. By scoping a BTG session to single patients rather than individual resources, and by allowing sessions to last as long as an emergency is active, we minimize the amount of manual auditing required after the session ends. BTG will last until a clinician explicitly signals the end of the event, rather than invoking BTG (and corresponding auditing burdens) for every instance of emergency access to every different device or health record. We also avoid system-defined "default" BTG duration windows, since these may easily be forgotten during an emergency, raising the possibility of an abrupt end to a BTG session, inconveniencing and confusing caregivers by disrupting their workflow.

When the access control framework is able to accommodate the obligations returned with an "allow-if-BTG" decision, the system institutes exceptional logging procedures, and allows a request that may otherwise be denied. Even when obligations cannot be fulfilled, the high-availability nature of medical systems may still require that the BTG request be allowed: overly strict enforcement of system obligation can prevent the timely delivery of life-saving treatment. Among the examples of emergency situations during which obligations may go unfulfilled is a denial of service attack on the facility's IT network, resulting in insufficient bandwidth to fulfill the increased BTG logging requirements. When obligations are met, the system is said to transition to a “controlled BTG” state. If some obligations cannot be met, the system fulfills as many as it is able, and transitions to an “uncontrolled BTG” state instead. In short, declaring an emergency will transition the system from a normal operating state to either a controlled or uncontrolled BTG state based on whether logging obligations were met.

Finally, by using a real-time resource access log analysis and enforcement of logging obligations, we limit the extent of uncertainty of the system state following a BTG session, and allow for recovery to a known safe and secure state. When an emergency is declared (BTG is invoked), the access control policy returns a series of logging obligations along with a more permissive authorization decision (i.e. the access control decision during a BTG session is almost always “allow”). When the authorization framework is able to accommodate the obligations returned with anallow-if-BTG decision, the system institutes exceptional logging procedures, and allows a request that may otherwise be denied. Real-time log analysis during an emergency access control override session can be effective in detecting anomalous user BTG accesses. We lay out a framework for investigating anomaly detection in the medical “Break the Glass” (BTG) procedure using statistical analysis. This can be achieved by integrating a semi-supervised learning model into anomaly detection system to flag anomalous BTG sessions. The model is trained in a supervised fashion with labeled and unlabeled data (pseudo-labeling) simultaneously. In statistical analysis, the statistical model is trained on non-anomalous data as an ensemble to calculate aggregated score that defines how likely a given BTG session is anomalous. The anomaly detection approach also uses users profiling approach based on users’ behavior (constructed from the users’ past interaction with the system) to detect deviation form normal behavior.

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.

- Our architecture is flexible-enough to support integration into most implementations of device interoperability standards, such as the Integrated Clinical Environment (ICE).

- We describe a new flexible medical "Break the Glass" (BTG) access control model which maintains compatibility with existing access control frameworks.

- We formally verify our model, showing that it allows for automatic return to a known-secure state even after temporarily granting emergency access requests.

- We demonstrate a number of alternate ways to construct access control and BTG policies such that it is all but impossible to mistakenly grant or withhold resource access (even during emergencies), backed by tool-based formal analysis of potential inconsistencies in the overall facility policy set.

- We provide guidance on constructing BTG policies which use the native logging facilities of the access control framework to facilitate automated, semi-automated, and fully-manual post-hoc auditing of emergency access requests and resulting resource use. The level of detail and accuracy would also improve by reducing reliance on human memory of the event.

- We show how to limit the extent of uncertainty of the system state following a BTG session, and allow for recovery to a known safe and secure state when needed using real-time resource access log analysis and enforcement of logging obligations.

1.2 Organization

Chapter 2 presents an overview of existing open ICE-compatible implementations, access control policies representation, state of the art for access control override, and show how our work is distinguished from other work. In Chapter 3 we present the first proof-of-concept implementation of authorization for systems of interoperable medical devices, an augmentation of traditional Attribute Based Access Control (ABAC) model with a novel method of attribute inheritance. Section 3.3 shows how the presented authorization model supports "plug-and-play" connectivity of new medical devices. The last Section 3.8 provides the testing methodology and results for the authorization system after integration into the Medical Device Coordination Framework.

In Chapter 4 we extend our study on safe and secure emergency access control override in interoperable medical systems. Sections 4.4 present an emergency access control model with policy evaluation and specification. A formal verification and validation of the model is presented in Section 4.5. The last Section 4.6 in this chapter provides guidance on how to facilitate revisions and preventing errors in access control policies.

Chapter 5 looks further into limiting the extent of uncertainty of the system state following an emergency access control override. This chapter focuses on real-time log analysis of resource access and enforcement of logging obligations.

Finally, Chapter 6 of this dissertation presents a summary of the main contributions of this work, and discusses on directions for future work.

Chapter 2

Background & Related Work

2.1 Interoperable Medical Systems

Interoperable medical systems are especially beneficial in multi-vendor environments with different devices on a shared IT network. In essence, by combining independent sensors and actuators with a coordinating entity (e.g. script or application running on commodity hardware), the system becomes more than the sum of its parts – a complex multi-featured device capable of operating in both open- and closed-loop control modes.[27] The promise of easy integration and avoidance of vendor lock-in would allow for significantly more flexible interoperable systems, able to carry out monitoring and even treatment functions as a group which no individual system component could accomplish by itself, and can significantly ease the burden on clinicians leading to more efficient and effective patient care.[16]

Simplifying connectivity increases the complexity of resulting "Medical Application Platforms" (MAPs), making them more difficult to understand and manage. Frameworks capable of creating and controlling these "system-of-systems" must be carefully designed to preserve patient safety *despite* of the increased complexity.[28] Their increased power requires greater assurance that they will not be misused (intentionally or unintentionally) to harm the patient(s) they are treating[4;12;16]. MAPs also present novel problems in terms of privacy and security. Each device and application within a MAP may require different levels of network

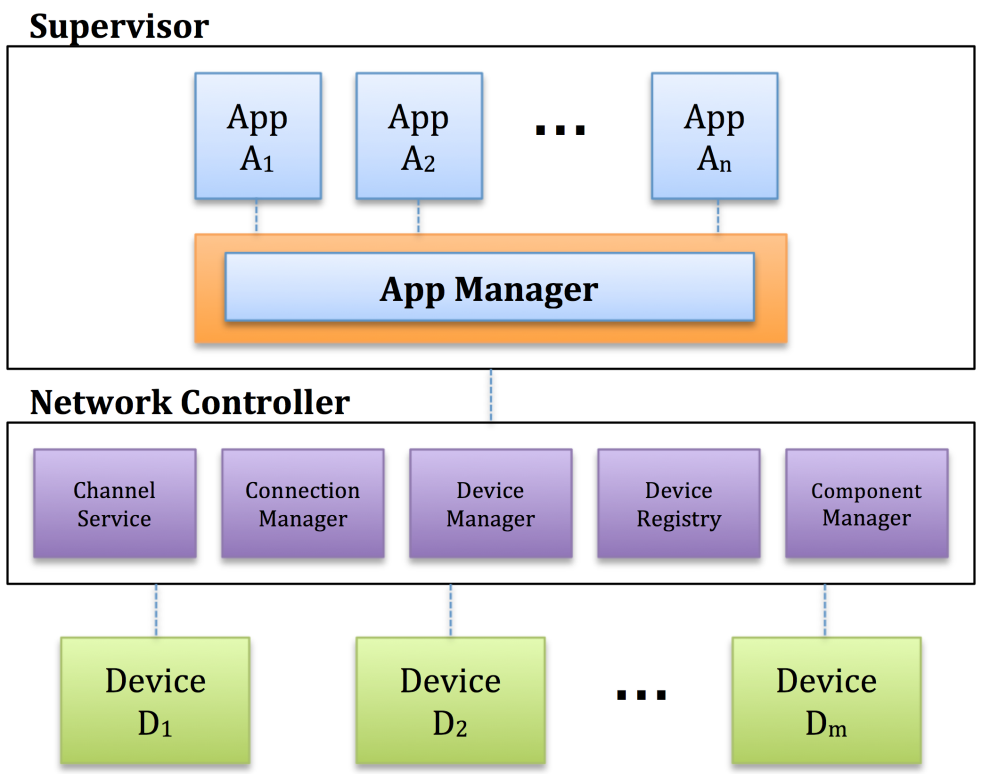


Figure 2.1: The architecture of the MDCF and its components

access and quality of service, complicating the resulting access control policies. Policy-based access control can be used to provide a comprehensive and flexible solution to the problem.

We are aware of only two open implementations which claim compatibility with the ICE standard, including OpenICE[11] and the Medical Device Coordination Framework (MDCF),[12] as of writing this dissertation. OpenICE was developed by the Medical Device Plug and Play Interoperability Program (MD PnP)[29] to automate peer-to-peer node discovery, data publishing and subscribing between nodes, and proprietary medical device protocol translations.[11] 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 in Chapter 3. 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).[30] We also use the modularity of the MDCF security framework to assist in our design and integration.[13;31] 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.

The MDCF is architected in logical units that closely follows the Integrated Clinical Environment (ICE) standard.[4] In addition to data logging and display capabilities provided by existing medical device connectivity features, the MDCF also 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)[32] and Data Distribution Service (DDS),[30] as message-oriented publish/subscribe middleware.[*] Each component is described briefly below and illustrated in Figure 2.1.

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

[*]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.[13]

- *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 Attribute Based Access Control Model

In ABAC, which is also known as Policy Based Access Control (PBAC), access to objects is granted by evaluating rules against attributes of user (subject), resource (object), action and other relevant attributes (environment) to a request. Attributes usually falls into 4 different categories and represents anything that may be defined and to which a value may be assigned, e.g. user role, resource type, work shift, etc. ABAC model supports conditional access rules (Boolean logic) that contain “IF, THEN” statements about subject, object and action. For example, *IF* the requester is a primacy physician for the patient, *THEN* allow read/write access to the patient’s sensitive data. As shown in the example, ABAC is dependant on the proper evaluation of attributes of the subject, attributes of the object, and a formal relationship defining the allowable operations for subject-object attribute combinations. These combinations are presented in a from of policy which express what is allowed or not allowed – policies can be granting or denying policies. Each access control policy may present a specific access control scenario.

Traditional identity based access control models, such as Role Based Access Control (RBAC),[33] are based primarily on the identity of a subject requesting access to a resource. In RBAC permissions are assigned to roles and roles are assigned to users. Access to a resource (object) is granted to a user (subject) only if the user is a member of a role that contains the requested permission. RBAC is considered insufficient to express real-world access control scenarios that may require multi-factor decisions. For example, in healthcare facility a decision may be dependant on clinician/user type, patient-clinician relationship, physical location, specialized training such as for Health Insurance Portability and Accountability Act (HIPAA)[14;15] patient records access, etc.

2.3 Representing Access Control Policies

The design of the Medical Application Platform concept[27] 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.[13] 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 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.[20] 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)[34;35] 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,[34] 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,[36] home automation gateways,[37] healthcare domain,[24;38;39] distributed systems,[35] 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)[40] 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.[41] 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 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.

2.4 Access Control Override

Restricting access to resources is the main goal of implementing authorization policy. Traditional access control mechanisms prevent the information from open access that could result in misuse, and only restricting privileges of users to what is needed to fulfill their tasks. Depending on environment (application domain) requirements access control rules are typically either defined too lax or too restrictive. A less restricted authorization policy gives users unnecessary access and defeats the purpose of least privilege. And a more restricted authorization policy than needed could stop users from fulfilling their tasks, which could be costly. Furthermore, healthcare is one such application domain where restricting access to only authorized users is not always the best and safest solution. For example, in medical emergencies allowing for access control override is considered a safer authorization approach over enforcing a deny decision, also discussed as part of HIPAA.[14;15] Overriding access control requires a mechanism within the authorization engine that enables it to reverse (or reevaluate) a returned decision and allow the access request which was initially denied. Certain user action(s) (e.g. explicitly requesting the override through either a software control or a physical hardware device/lever/button) are required as a prerequisite to overriding an access control decision.

2.5 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. There are numerous proposed solution that use different methods to achieve patient privacy.[24;42–44] Ray et al.[24] 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.[44] 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.[45] Various solutions have been proposed to override access restrictions in a controlled manner.[46;47]

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).[48] Authorization within ICE-compliant medical middleware has been not been studied, nor are the concepts covered in the associated standards,[4] 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,[13] but does not present an authorization architecture, except for a high-level design rooted in the Role-Based Access Control (RBAC) model.[31] 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.

Furthermore, the concept of access control override or Break the Glass (BTG) is not new. In one of the earliest papers on the topic, Povey[49] discussed unexpected risks resulting from static nature of authorization and proposes a new access control paradigm for constraining access in situations like medical emergencies where a user may need to exceed their normal privileges.

In a similar work, Rissanen, Firozabadi, and Sergot[50] suggest a mechanism for increased flexibility in access control by overriding denied access (when necessary) using the possibility-with-override concept. It is also suggested that the overrides should be audited using the access control policy. Ferreira et al.[51] proposed a BTG policy within an implemented access control policy (defined by healthcare professionals) and access control hybrid model for a hospital.

While the concept of BTG in their work is similar to the previous work done in the same area, the introduction of BTG policy within a hybrid access control model for presenting access control policy and its implementation in a healthcare facility is the main contribution of their work. In a follow-up work Fererira et al. extends the NIST/ANSI RBAC model with BTG and names it the BTG-RBAC model.[46] Their work is mainly focused on overriding access in a controlled manner using a state-based RBAC authorization infrastructure: in situations where a user is not allowed to access a resource and a deny decision would normally be returned, the BTG-RBAC model allows for a third decision option: BTG. Instead of "deny", a "BTG" decision is returned for an access request and allows the user to break the glass and access the requested resource.

The simplest implementation of a BTG-RBAC model allows for the start of BTG, but makes little or no allowances for how to terminate BTG. A more complete model addresses this limitation, and incorporates the concepts of BTG obligations as well as post-event auditing. BTG-RBAC requires policies to consider predefined values of BTG variables meant to keep track of the system BTG state, making it difficult for humans to reason about the policies they are writing. While in our work we consider 3 well-defined system states, a normal operating state and two emergency states, whereas transition from one state to another is controlled through evaluation of system-wide obligations. In addition, we test our policies for inconsistencies and incompleteness, and verify the access control model is expressed correctly.

Brucker and Petritsch[47] demonstrate a BTG model that allows for access control override on a permission basis, with different levels of emergency specified by policy. In their work, permissions are attached to emergency levels and need to be specified in emergency policies that are handled by a separate emergency policy manager and policy decision point.

While making significant strides toward flexible and controlled BTG, the system has some drawbacks. There is little built-in verification that BTG policies will behave as expected, especially when combined with other facility policies, which can be especially dangerous when they result in denial of availability at the time of emergency. (Like most policies, emergency level- and permission-specific policies are prone to errors like inter- and intra-policy inconsistencies, insufficient or excessive permissions, etc.) Prior work does not address the situation wherein the authorization system may fail to fulfill the returned obligations accompanying a decision from a BTG request, and it is therefore unclear whether a BTG request would be allowed or denied at that time.[46;47] We explicitly address these *unmet obligations* and show how they affect the overall authorization process. We handle them by forcing our system into an alternate BTG state which we call *uncontrolled BTG*.

Reviewing audit logs for unusual activity in safety critical systems is a common practice. Audit trails can be used to detect bugs, policy violation and fraudulent activities resulting in system crashes, unavailability of resources, network slowdowns etc. Emergencies requiring access control override are one concrete example of unusual activity in healthcare. Post-BTG audits are regular practice at medical facilities. A post-hoc audit of the override will suggest appropriate actions, including disciplinary, e.g., if the emergency was inappropriately declared or inappropriate records were accessed. To our knowledge there has been no previous published work in computer or information science proposing logging mechanisms for controlling BTG sessions in the medical domain or on logging and auditing requirements in relation to overall system performance.

In 2016 a conducted research on examining access logs collected from an Electronic Patient Record (EPR) system used largely in Hospital in Norway insists on analysis of access logs to be a very useful tool for learning how to reduce the need for exception-based access.[52] After reviewing the access log from eight hospitals in the Central Norway Health Region (CNHR), the authors discovered that the use of exceptional mechanisms in these hospitals is too common mostly due to the lack of a stricter form of access control, and the huge size of the log makes it impracticable to audit the log for misuse. Alizadeh et al.[53] has presented an approach for user behavior analysis by constructing user behavior profile and comparing it with an expected behavior using logs collected from a medical facility. The authors apply their approach to study the use of “Break the Glass” (BTG). A final “anamoly” score assigned to user’s profile determines the extent of difference in user’s behavior from an expected behavior. The application of their approach is not investigated in real-time log analysis.

Tasali et al.[2] presents a controlled BTG model for interoperable medical systems which is based on well-studied ABAC[20;21] model. In their work BTG is specified within policy and defined in terms of states in which the system is operating, and allows for overriding deny decisions automatically (instead of on a one-by-one basis) during a BTG session. This is achieved by constructing a BTG meta-policy which works with existing access control policies. In addition, previous work have noted the need for log analysis after access control override, proposing that logging requirements to be returned as obligations attached to a policy, but it is not immediately obvious what should happen if those obligations are not met. In this work, we discuss logging methods consistent with requirements for returning a system from an emergency state (BTG) to normal state, and show how our approach can be extended to partially or fully automate the post-BTG auditing process.[46;47;49;50;54]

Current practices require clinical facilities to have individuals or groups who are tasked with running access control audits. Access reports are usually run by IT or Security and reviewed by Privacy or Compliance in conjunction with clinical management who would understand if the access was appropriate. Auditors will review human- and machine-generated records for any flagged system activities either immediately after an emergency access event or during the next business day, and take appropriate actions. Automated log analysis may streamline some of this process while maintaining system-specific requirements. Log analysis tools can be used to provide a summary of user actions, resources accessed, obligations with their status (met/unmet), system state, etc. and modifications can be made to allow automation for a certain set of known activities commonly found in such logs.

Chapter 3

Authorization Architecture[†]

This chapter presents an authorization architecture for real-time "plug-and-play" interoperable medical systems. Figure 3.1 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

[†]This chapter includes results published as part of "A Flexible Authorization Architecture for Systems of Interoperable Medical Devices" [1]

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.

Devices connected to a Medical Application Platform[27] like the MDCF are expected to have preassigned attributes that can easily be transformed into access control attributes in authorization policies. These attributes can be preassigned by device vendors, app developers, and/or created 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), which results in decision changes without the necessity to change the device and user relationships defining any underplaying rule sets between the two. To make the process as easy as possible for the policymaker, not only are there 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.[20] Therefore, there is no need need to specify individual relationships between each device (or even device class) and each potential operator without sacrificing granular user permissions. We also use the eXtensible Access Control Markup Language (XACML)[34;35] as our back-end, due to its wide acceptance and tool support (it is an OASIS standard and one of the most widely used policy language) and portability of our framework. XACML is open but somewhat verbose and hard to read, so we use ALFA as a higher-level human-readable language for composing and editing policies, which are then translated to XACML for maximum flexibility of enforcement engine implementations.

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

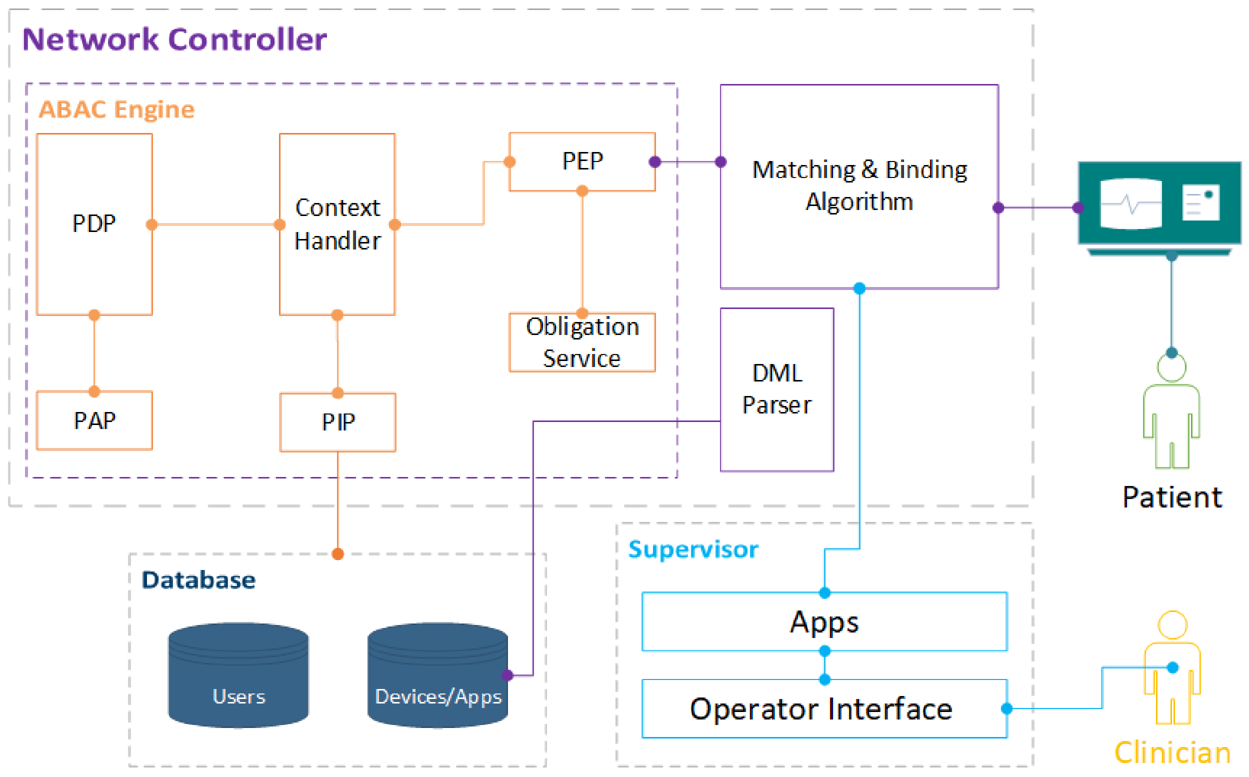


Figure 3.1: The basic architecture of the new authorization system after integration with the MDCF.

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.[31] 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 Current Workflow

We introduce patient Pamela and her primary nurse Nick in order to better illustrate the workflow of the authorization system. Pamela has had a medical surgery, and is now on pain relief medication (opioid delivered through a PCA pump) prescribed to her by her surgeon. 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[55] that displays Pamela's SpO2 (Blood Oxygen Saturation), EtCO2 (End-Tidal Carbon Dioxide), and RR (Respiratory Rate).

The MDCF workflow, as shown in Figure 3.5a 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 Chapter 3.

3.2 Authorization Policies

Authorization policies in the MDCF are divided into two categories based on the the type and initiator of an access request: user-app authorization policies and app-device authorization policies. Clinicians and apps are different entities, but both are considered actors, and so they each require distinct access permission before accessing any device. While access control rules for the category 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"
```

3.3 Plug-and-play Support

The authorization architecture can also support "plug-and-play" connectivity of new devices[29] 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.[56] 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

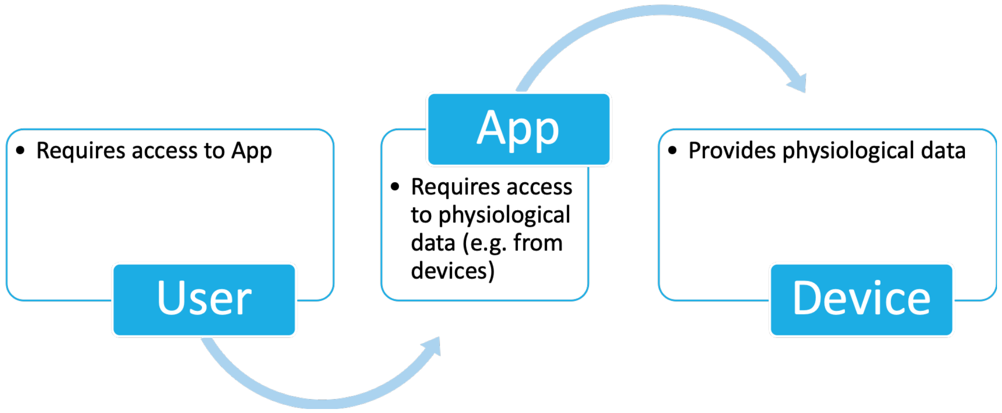

App
Requires access to physiological data (e.g. from devices)
Requires access to App
User
Provides physiological data
Device

Figure 3.2: App as an actor and resource.

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 3.4, but an extensive discussion of the plug-and-play in the MDCF is beyond the scope of this paper.

3.4 Attribute Inheritance

We treat clinicians and apps as distinct and independent actors, which makes our design challenging as well as unique. As a result, clinicians and apps 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.[33] 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 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).

```
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 3.3: An example written in ALFA – a subset of the policy used in evaluating our implementation

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).[*] 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 as shown in Figure 3.2. 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

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

```
<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 3.4: The XML generated from one line code (***target clause app.role == "aR1"***) in Figure 3.3.

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 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 3.3. 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.

3.5 Break the Glass (BTG)

Attribute inheritance should not be considered a substitute for Break the Glass (BTG) features,[47;51] 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.

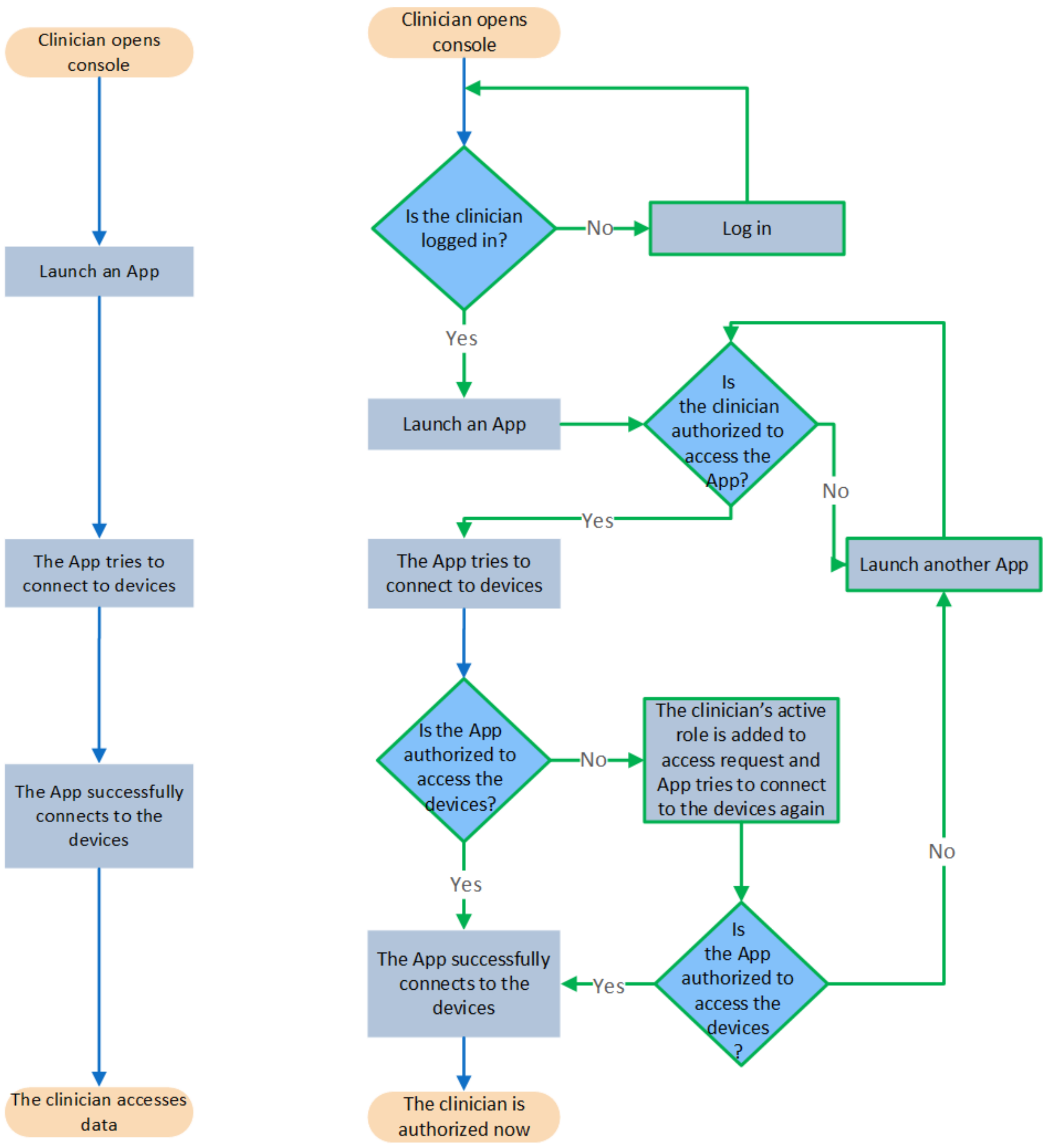
BTG allows overriding access controls and provides full (or sufficient) access to the system during an emergency, but attribute inheritance is a step in eliminating the need for a "global" override. Instead, it can be used to allow controlled, authorized access to medical resources (e.g. devices) during 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 BTG feature within our authorization architecture is further discussed in Chapter 4.

3.6 Modified Workflow

Here we show the new workflow, *with authentication and authorization*. Steps which are modified or added to the original workflow from Section 3.1 are italicized. Figures 3.5b and 3.5a provides a visual aid.

1. Clinician accesses the Clinician Console


Clinician opens console
Launch an App
The App tries to connect to devices
The App successfully connects to the devices
The clinician accesses data
Clinician opens console
Is the clinician logged in?
No
Log in
Yes
Launch an App
Is the clinician authorized to access the App?
No
Yes
Launch another App
The App tries to connect to devices
Is the App authorized to access the devices?
No
The clinician's active role is added to access request and App tries to connect to the devices again
No
Yes
The App successfully connects to the devices
Yes
Is the App authorized to access the devices ?
The clinician is authorized now

(a) Original workflow.

(b) Modified workflow. The boxes with green border represent the modified steps.

Figure 3.5: The workflows for a clinician accessing patient physiological data in the MDCF.

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 permit or denyto the Console*
6. *If permit 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 "permit", "deny", or "not applicable"*
12. *Shiro receives the authorization result from Balana*
13. *If Shiro receives "deny", 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 "permit" is returned, the app is allowed to connect to devices*

21. *If "deny" is returned, the request is reevaluated with the clinician's active role appended* [†]

22. *If "deny" 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)* [†]

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:

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 "deny", the app discards the change and displays a denied message to the clinician*

[†]This step is not yet implemented

7. *If "permit", 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.

3.7 Implementation Details

We chose Apache Shiro[57] as our Policy Enforcement Point (PEP) framework and WSO2 Balana[58] 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,[59] OACC[60] 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

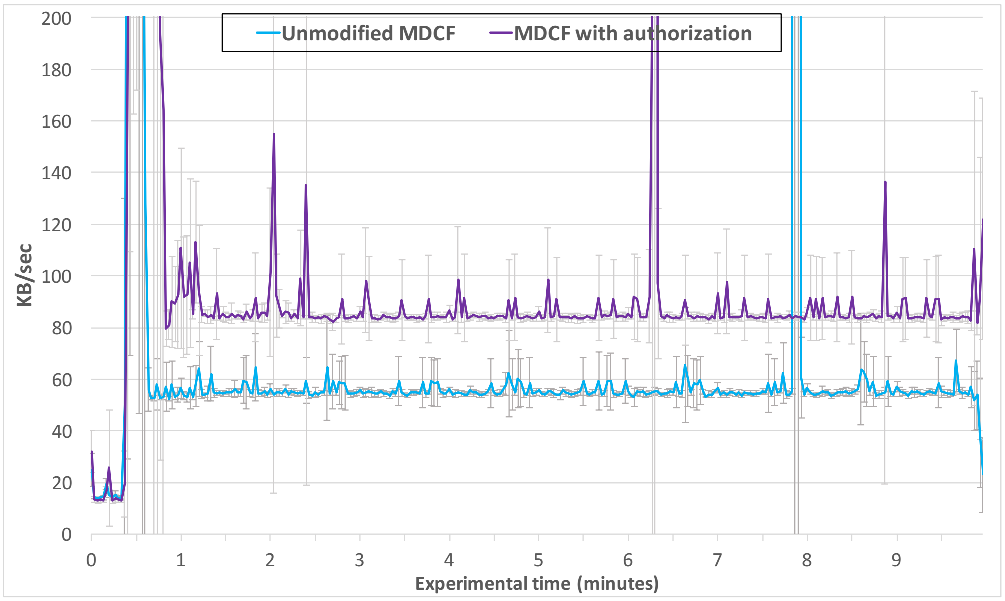
- Is initialized with a set of XACML policies,
- Takes as input the attributes of the app or device, and the details of the user requesting access, and
- Returns true if the user has access via any of the user's available roles.

3.8 Evaluation

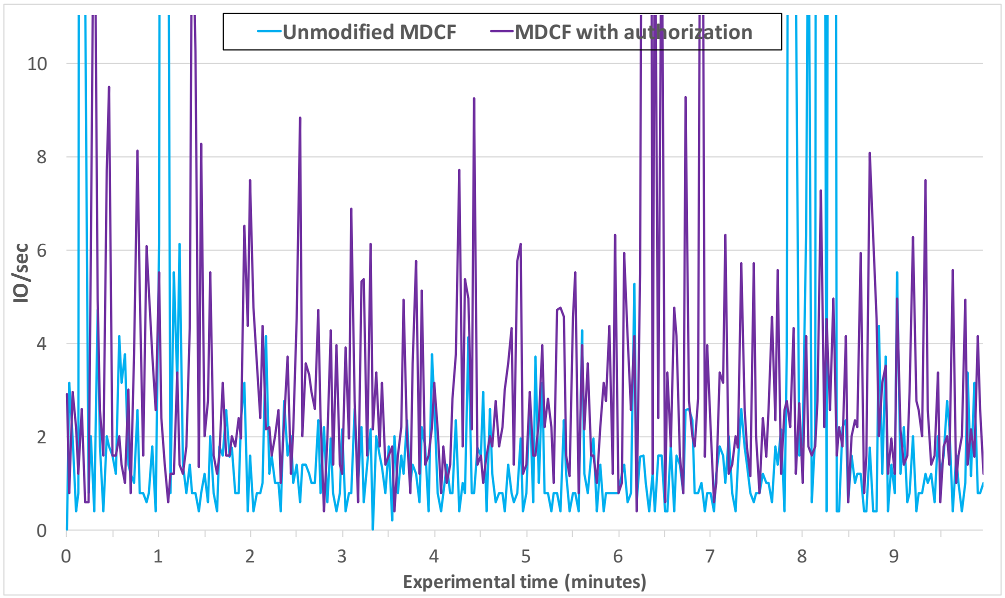
In this section we describe the testing methodology and results for the authorization system as integrated into the MDCF. The tests were designed to exceed the normal expected operational capacity of the system (also called stress-testing) to confirm safe usage limits and given specifications are met. 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,[4]) 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,[16] 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

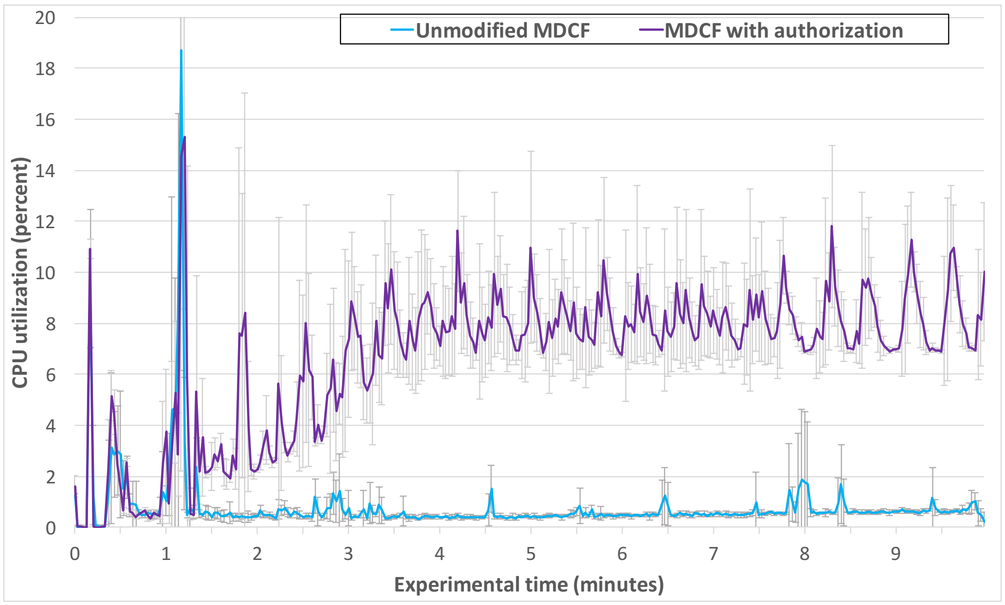


(a) Network (total-read) usage

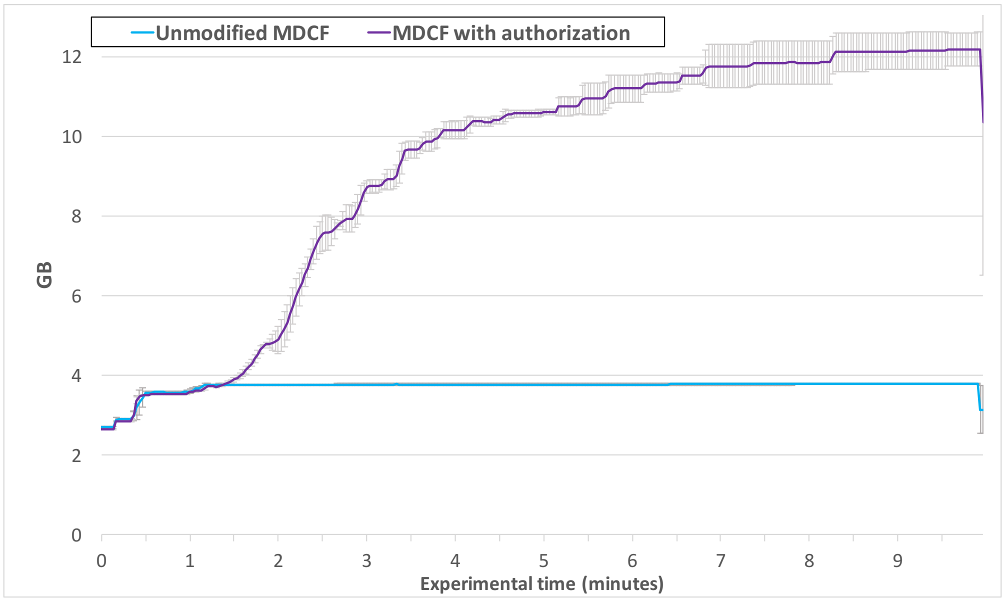


(b) Disk I/O with error bars omitted. The variability between baseline and modified versions is *not statistically significant*.

Figure 3.6: Disk and network I/O of the baseline versus authorization-enabled MDCF without Shiro caching



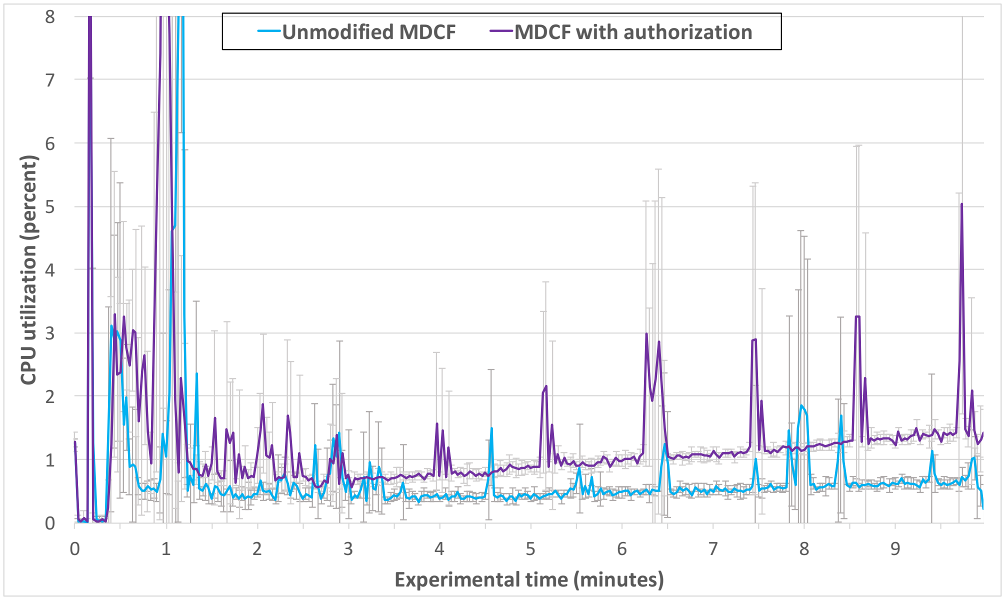
(a) CPU usage



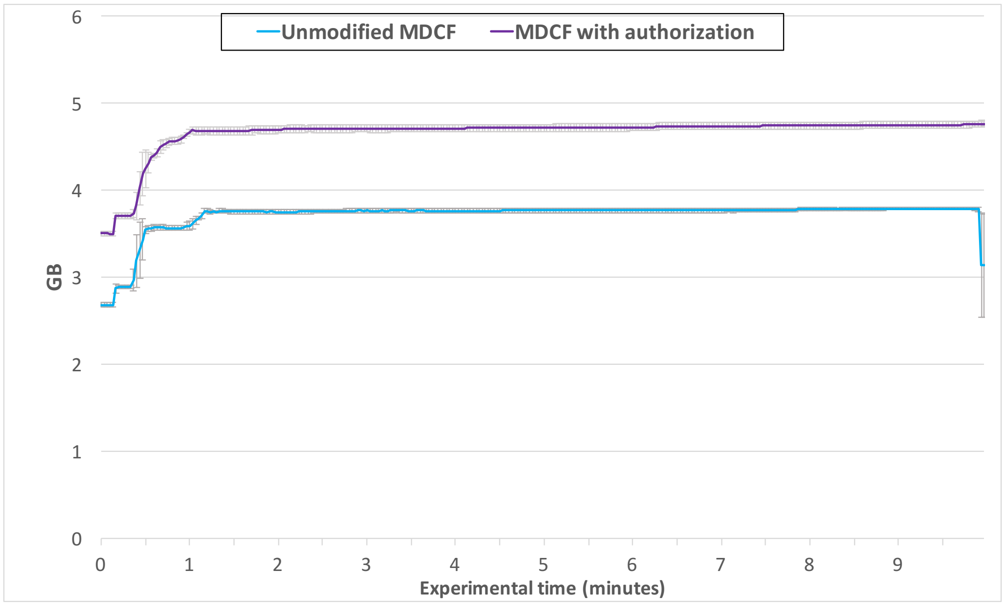
(b) Memory usage. Error bars are too small to be visible in some places.

Figure 3.7: CPU and memory usage of the baseline versus authorization-enabled MDCF without Shiro caching

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



(a) CPU usage



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

Figure 3.8: CPU and memory usage of the baseline versus authorization-enabled MDCF with Shiro caching

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 3.6 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 3.7 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 3.7a 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 3.8 shows a significant reduction of both CPU and memory overhead due to several simple optimizations: using the built-in Shiro caching API,[61] 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 3.8a, and memory overhead to 20%, also removing the memory growth over the time of the experiment, as shown in Figure 3.8b. 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.

Chapter 4

Bend-the-Glass: Controlled Emergency Access[‡]

Access control override is normally handled on a field-by-field basis in electronic medical record (EMR) systems, with a BTG request granting an exceptional one-time access to a single record. With minimal customization, our controlled BTG solution can be implemented as an extension to existing access control models which already allow for override *sessions*. The authorization architecture of Tasali, Chowdhury, and Vasserman[1] is used as a starting point. Since it uses dynamic systems of medical devices attached to a single patient, our BTG granularity is also per-patient, i.e. if an emergency is declared, access to devices connected to the patient suffering the emergency, as well as that single patient's electronic health records, are made available for the duration of the BTG session. Therefore, multiple access requests to a single patient's EMR (even to different fields), or access to that patient's devices, are all allowed on an emergency basis as part of the same BTG session. This continues until the BTG session terminates via explicit action of a clinician, who marks the session as completed. At that point, a system exiting a controlled BTG state can be automatically (or via semi-automated audit) rolled back to a known-secure state as shown in Figure 4.2.

Using a modified policy enforcement point design,[1] we can allow as-needed access control

[‡]This chapter includes results published as part of "Controlled BTG: Toward Flexible Emergency Override in Interoperable Medical Systems"[2]

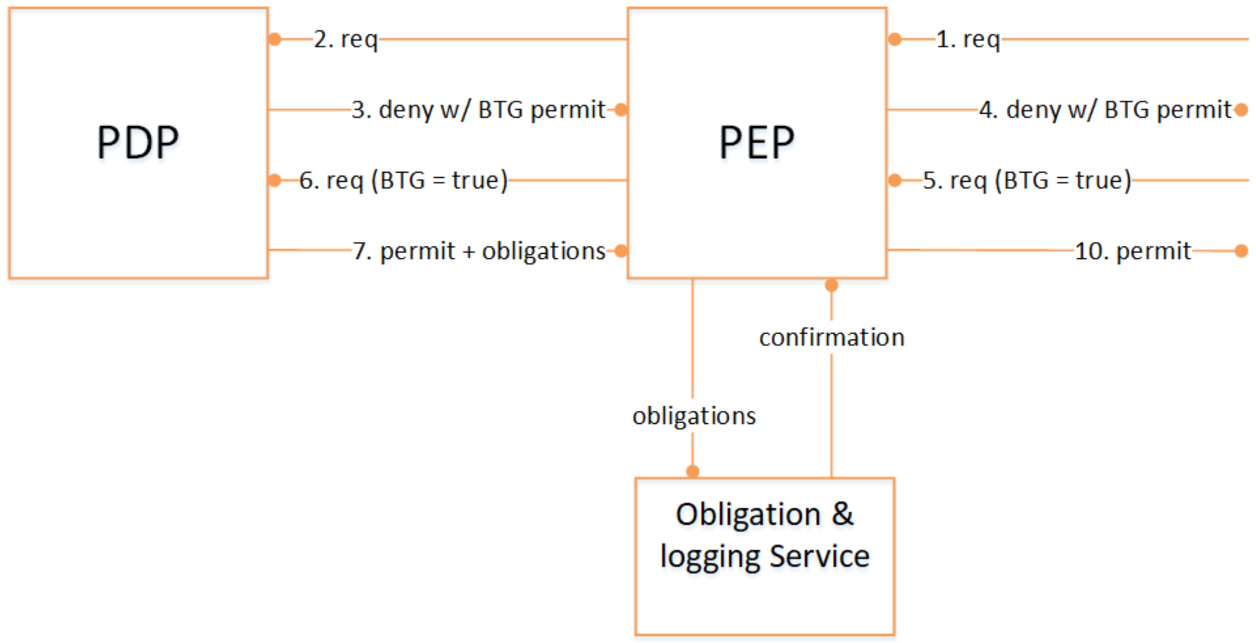


Figure 4.1: BTG request workflow. Parentheses indicate the state of the system through condition variables, and steps 3, 4, 7, and 10 denote returned decisions.

override. For any "deny" decision returned by the PDP, the clinician may choose to request access control override, resulting in a second access request being generated. That request is forwarded to the context handler and PDP for a second decision, now in a BTG context, as shown in Figure 4.1. In contrast to the first request, the PDP evaluates this request against a BTG policy. Other than the change of policy being used for the decision, the PDP behaves in a similar manner to the previous request. System status and other session attributes are used to determine what policies are used for evaluation by PDP. A returned decision by the PDP might include obligations that PEP will enforce prior to allowing access. Due to the importance of resource availability in medical emergencies, we must explicitly consider unmet obligations in our model. It is necessary to allow for access even if PEP fails to enforce the obligations. We allow for access to be successfully overridden even if the obligations are not met. See Section 5.3.1 for further details on how the PEP meets the requirement for an OASIS-compliant PEP.

4.1 Operating States

The BTG state machine for our system is diagrammed in Figure 4.2. Emergencies are time-sensitive and dynamic in nature, and the access control framework must take into account the changing conditions of the system as the emergency runs its course. for the purposes of implementation simplicity and policy flexibility, we treat BTG as a resource or a system state variable. Therefore, the access control framework does not need to periodically reevaluate whether or not BTG is in effect (which would change which policies apply for the purpose of making access control decisions). Instead, accessing the BTG resource triggers a permission check, allowing the system to make the transition at that time, if needed. Detecting other events, such as patient disconnection, is left to future work, and for the moment we assume that BTG sessions can end only when explicitly specified by a clinician. Whether the system can then be rolled back to "normal" state automatically or must be flagged for audit depends on whether it is in the "controlled" or "uncontrolled" state when BTG ends.

4.1.1 Normal State

This is the initial state for authorization system and remains the current operating state as long as for every user access request the Policy Decision Point (PDP) returns a decision and the user has not initiated an emergency session (BTG). Common decisions after evaluation of an access request include "permit", "deny", "not applicable", and "indeterminate". "Not applicable" means the PDP could not locate a policy

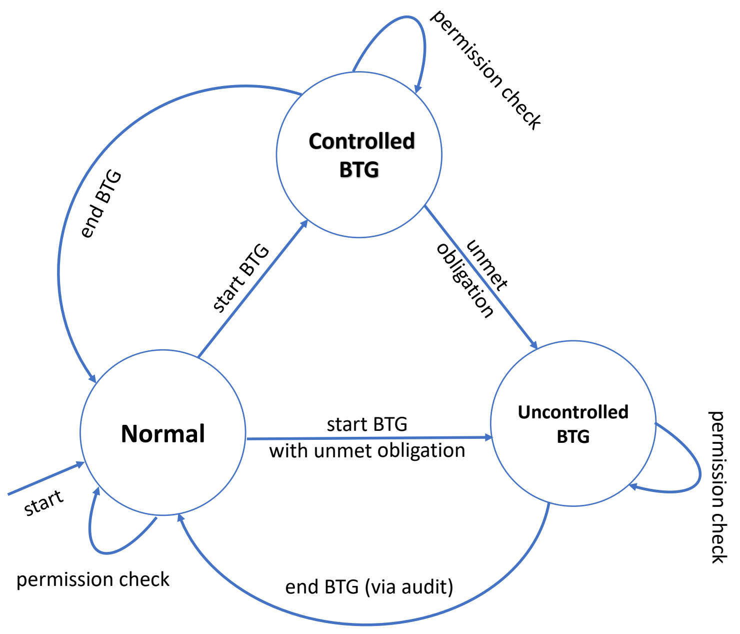


Figure 4.2: BTG state machine showing the normal state and two emergency (BTG) states, along with transition criteria.

that matches the request, and an “indeterminate” decision means an error was encountered during policy evaluation. Regardless of the decision, the system performs permission check for all subsequent access requests and remains in normal state until a state change is forced.

A “permit” decision can have obligations attached – the action is allowed on the condition that the obligations are fulfilled. It is important to note the differences between the two types of obligations returned by the PEP. We refer to the obligations that are returned after evaluation of a non-BTG request as *non-BTG obligations*, and to obligations that are required to be met in order to allow BTG as *BTG obligations*.

4.1.2 Controlled BTG State

The authorization system changes its state from normal to controlled BTG whenever the user access request is denied and the user overrides the access control by initiating a BTG session. In order for the system to change its state to controlled BTG, any returned BTG obligations after requesting BTG session need to be met. Obligations are facility- and policy-specific, but we expect them to include e.g. a requirement for more detailed logging prior to allowing a controlled BTG session. (This is further discussed in Section 5.1.) The BTG obligations are meant to help track the clinicians' actions for audit and system rollback purposes, and must include sufficient detail for such a rollback to be performed. If any required BTG obligations are not met, the system nonetheless allows the request but moves into uncontrolled BTG state. A system in a controlled BTG state returns to a secure (normal) state once the clinician explicitly signals an end of emergency. Independent of BTG state, the system evaluates every access request against existing (non-BTG) policies as it would if the system were operating in normal mode. Therefore, the only further change to the evaluation process is allowing an override in the first place, and even that override is subject to BTG policy evaluation.

4.1.3 Uncontrolled BTG State

As shown in the system state diagram in Figure 4.2, the authorization system transitions to uncontrolled BTG if 1) it is already in a controlled BTG state, but no longer able to fulfill some or all obligations that were returned with the access control decision that allowed BTG to start, or 2) it is in a normal state and the user overrides a "deny" decision, declaring BTG, and yet the system cannot meet some or all BTG obligations returned with the BTG allow decision. However, a system in uncontrolled BTG cannot return to controlled BTG – it can only return to normal state via a manual or semi-automated audit.

During uncontrolled BTG, the system performs permission checks similar to controlled BTG, such that the system evaluates each user access request against authorization policies and for any decision other than "permit", access control is overridden and the decision is logged. In addition, any returned obligation is fulfilled on a best-effort basis. Although the system is in uncontrolled state, it is still governed by the authorization rules – *the policies are never ignored, only overridden.*

Unlike the controlled BTG state, systems which end up in an uncontrolled BTG state may need to be manually audited before they can be returned to a normal state. Currently, this would be performed by an audit team either in real time or during the next business day. Only uncontrolled BTG sessions need to be handled manually, significantly reducing the workload and turnaround time of these audit teams. Furthermore, the level of automated logging provided by our system allows for structured log analysis, making even the manual process more straightforward and less error-prone, especially since automated logging reduces the reliance of human memory of these BTG incidents.

4.1.4 "BTG-restricted" Permissions

To ensure that the system retains its overall integrity and can be rolled back to a known secure state, some "deny" decisions cannot be overridden, including changes to the authorization policies themselves and certain other critical system resources (which may vary between facilities and can be explicitly specified in the BTG policy). In an example authorization policy in Figure 4.4, we protect resources in this "BTG-restricted" set from *any* access during a BTG session, even access which may be permitted when the system is in normal operating mode. We do not expect that security-critical resources such as the access control database would need to be accessed during BTG, especially since all other permissions can be overridden, removing the need to manipulate user roles and/or permissions.

This design is implemented purely via policy and does not require modification to the access control framework. A facility can write an alternate, more permissive BTG policy so that, e.g. access to the authorization policy database is allowed during BTG. This "permit" decision, however, would only apply to authorized users and cannot be overridden, i.e. no one can "Break the Glass" in order to alter access control policies.

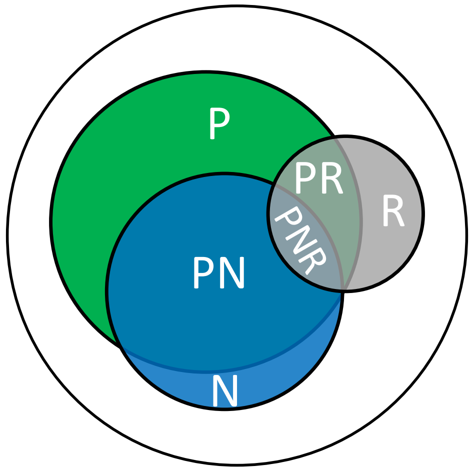


Figure 4.3: Permissions grouped by access. Each P, N and R represents group of permissions assigned to Physician, Nurse, and BTG Restricted, respectively.

4.2 BTG Policy Evaluation

An effective BTG policy is dependent on proper identification of resources at the time of policy evaluation. Thus, a method is needed to identify and categorize resources into their relevant access groups (sets). Additional attributes including resource group information are retrieved from a policy information point (PIP). We use an easy-to-implement method for resource identification that can be further extended for use with any domain. However, this method is dependent on a well-identified list of BTG-restricted resources, i.e. a limited number of resources marked as not for use during a BTG session – these resources can only be accessed in normal situations and policy decisions for those rules are not subject to BTG override. The method particularly helps identifying if a resource is available for BTG access by taking requested resource and list of BTG-restricted resources as inputs and returning a Boolean value. A false value means the resource is available for BTG access ("BTG-allow" group) while a true value means resources are not available during BTG ("BTG-restricted" group). Resource assignment into groups is not fixed and may change like other dynamic attributes. We take this into account and the system will query for group information only at policy evaluation time, returning the correct group for a resource even if the group is subject to change.

Figure 4.3 shows a diagram of how resources (grouped by access) work within our architecture. The outer circle represents the set of all available resources. The circles marked with letters P (physician) and N (nurse) represent the set of resources that can be accessed by physicians and nurses, respectively. The circle R represents the set of resources that are restricted during BTG, but can be accessed by authorized entities outside of BTG events. A nurse or physician is only allowed to access a BTG-restricted resource if it is included in the set(s) of their accessible resources (PR and PNR) and the authorization system is in normal operating mode. Access to resources in R, PR, and PNR is unconditionally denied during BTG (see Section 4.1.4). Note that we expect real-world policy sets PR and PNR to be null (empty), as they would contain e.g. the access control database itself, which should not ever require access during BTG since all other

	Resource	**System State**	**Decision**
1	P	any	permit
2	PN	any	permit
3	N	normal	deny
4	N	BTG	permit
5	PR	normal	permit
6	PR	BTG	deny
7	PNR	normal	permit
8	PNR	BTG	deny
9	R	any	deny
10	$\notin (P \cup N \cup R)$	normal	deny
11	$\notin (P \cup N \cup R)$	BTG	allow

Table 4.1: Sample access control table for a physician (the "Subject" column is omitted, as it is always "physician")

permissions can be overridden. (As Nurse and Doctor are chosen arbitrarily, the example is representative or any two roles in an organization.)

The resources within the outer circle are assigned to sets accessible by other clinicians or users not shown here. There may exist resources not assigned to any set (e.g. specification error causing resources to be "orphaned"). These are not included in the outer circle. Access control decisions for those resources cannot be made conclusively in either normal or BTG operating modes (see "indeterminate" in Section 4.1.1). One may argue that "indeterminate" decisions should be overridden to "permit" in a BTG context. This is a design decision, and is explored further in Section 4.6.

Table 4.1 shows access control requests and decisions for a physician accessing resources categorized as in Figure 4.3. For ease of illustration, we assume that all obligations are fulfilled by the PEP, as including state transitions would render the table unreadable.

- In row 1, the physician is allowed access to P regardless of the current state of the system since P is the set of resources explicitly allowed for physicians.

- In row 2, the physician is allowed to access PN regardless of the current state of the authorization engine since PN is a subset of P and so explicitly allowed to physicians.

- In row 3, the physician requesting access to N is denied unless the authorization system is in either controlled or uncontrolled BTG state (row 4). Resources in N are allowed access to only by nurses. Thus, a physician may need to break the glass to successfully gain access.

- In row 5, the physician request for PR is permitted only if the authorization system is in normal state.

- In row 6, access to PR is disallowed during BTG since PR is part of the BTG-restricted set.

- In row 7, as in row 5, the request is permitted only since the authorization system is in normal state.

- In row 8, access to subsets of R is denied during BTG.
- In row 9, access to R (not PR or PNR) is denied regardless of the current state of the system because those resources are not within the allowable physician set during normal operation, and are restricted during BTG.
- In row 10, the physician is requesting access to some resource not diagrammed in Figure 4.3, meaning permission to access them has not been explicitly granted, and therefore the request is denied in normal mode.
- Finally, the access request in row 10 also refers to resources not explicitly permitted, but during BTG it is allowed (row 11) since those resources are not within R.

The above method of identifying BTG-restricted resources (R) ensures that access to all resources outside (R) is granted via a normal or BTG request. Treating R as a blacklist simplifies writing least-privilege BTG policies.

4.3 BTG Policy Specification

Figure 4.4 shows an example policy, written in ALFA,[40] that contains rules for normal, controlled, and uncontrolled BTG access. The policy is meant to be very generic, and can be easily customized to any BTG access control scenario with more fine-grained rules and obligations. Furthermore, the BTG rules and policies are meant to be easily integrated into existing authorization policies.

The example policy is expressed as a `policyset` containing two policies. The first (`polSetFlowRate`) specifies the target clauses for resource `flowRate`, possible actions (read or write), and contains two rules that come with their own target clauses and conditions. If a user access request matches the policy `polSetFlowRate` target clauses, then the authorization engine checks enclosed rules within the policy. The ***`allowSettingFlowRate`*** rule allows for normal access if 1) the access request matches its target clause (requesting subject to be a nurse or physician), 2) the access request meets the conditions specified within the

```
namespace org.facility {
  import org.facility.BTG.Attributes.*
  import Attributes.*

  obligation log = "org.facility.normalLog"
  obligation btgAudit = "org.facility.btg"

  rule allowEmergencyAccess {
    target clause EMG.BTG == true
    condition not (
        integerOneAndOnly(resource.group)
          == "BTG-restricted"
          )
    permit
    on permit {
      obligation btgAudit {
        // example logging obligations
        log.message = "BTG␣access␣granted"
        log.accessTimestamp = currentTime
        log.requestingIP =
            subjectLocalIpAddress
        log.accessedResource =
            resource.resourceId
        log.resourceGroup = resource.group
        log.accessingSubject = user.userId
        log.systemState = authSystem.state
      }
    }
  }

  rule allowBtg {
    condition EMG.BTG == false
    permit
    on permit {
      obligation btgAudit {
        // ensure the followings are
        // met or true
        EMG.state = "Controlled-BTG"
        EMG.userAuthenticated = true
        EMG.btgLog = true
      }
    }
  }

  rule allowSettingFlowRate {
    target clause
        user.role == "nurse"
      or user.role == "physician"
    condition integerIsIn(
            integerOneAndOnly(user.userId),
          patient.assignedClinicianId
          )
    permit
    on permit {
      obligation log {
        // example logging obligations
        log.message = "Normal␣access␣granted"
        log.accessTimestamp = currentTime
        log.accessedResource = resource.resourceId
        log.accessingSubject = user.userId
      }
    }
  }

  policy polSetFlowRate {
    target
      clause resource.resourceId == "flowRate"
      clause action.actionId == "write"
      or action.actionId == "read"
    apply permitOverrides
    allowSettingFlowRate
    allowEmergencyAccess
  }

  policy polSetBtgFlag {
    target clause
        resource.resourceId == "btg"
    apply permitOverrides
    allowBtg
  }

  policyset authPolicySet {
    apply permitOverrides
    polSetFlowRate
    polSetBtgFlag
  }
}
```

Figure 4.4: Example BTG policy written in ALFA, reformatted for readability

rule (checking if care relation exists), and 3) the obligations are fulfilled by PEP. Otherwise, a deny decision is returned. The sequence is similar for all rules. Assuming either the condition or target clause in the *allowSettingFlowRate* rule cannot be matched to the request, the rule *allowEmergencyAccess* is evaluated. The target clause and condition in the `allowEmergencyAccess` rule ensures that the BTG flag is set to true and the requested resource (`flowRate`) is not part of the "BTG-restricted" resources group. The user request for access to the resource `flowRate` is permitted regardless of the authorization system state as long as any of the two rules return permit decisions. This is ensured by the rule combining algorithm *permitOverrides*, specified within the policy `polSetFlowRate`.

The second policy (`polSetBtgFlag`) controls access to BTG. Its target clause (`btg`) and consist of a single rule: *allowBtg*. That rule has only one condition which checks the status of the BTG flag. If the user requesting BTG access and the BTG flag is not set then a permit with obligations decision is returned to the PEP, which in turn ensures the obligations are fulfilled prior to allowing for a BTG access. If any of the returned obligations cannot be fulfilled then the BTG access is still granted and the authorization system changes its state to uncontrolled BTG. Otherwise, access is granted and the authorization system changes state to controlled BTG.

Access control requirements in general and the example policy in Figure 4.4 are kept simple on purpose. They provide a "generic" starting point for more complex and lengthy policies. They are formally verified in Section 4.5 in order to show how we may prove the correctness (or at least that certain heuristics/invariants hold) in arbitrarily complex policies. The example and formal verification are short to ease explanation, but the steps show are meant to scale up and test policies which too large and the interplay too complex to be done ad-hoc. Additional policies are listed in Appendix A.

4.4 Comparison to Real-World EHR-BTG

We analyze our policy structure design by comparing the available features and flexibility to real-world procedures for electronic health record access as implemented by one of the

```
namespace org.facility {

  obligation log =
      "org.facility.normalLog"
  obligation btgAudit =
      "org.facility.btg"

  rule BtgRule {
    target clause EMG.BTG == true
    condition not (stringIsIn
       ("Federally␣Funded␣Abuse␣
       Clinic", (patient.location)) &&
       (user.role == "PCP" ||
       user.role == "others"))
       && patient.BtgStatus == true
    permit
    on permit {
      obligation btgAudit {
        //obligations
      }
    }
  }

  rule readAndWriteRule {
    target clause user.role ==
       "clinician"
    condition
       integerIsIn(integerOneAndOnly(user.userId),
       patient.assignedClinicianId)
       ||
          user.hasAppt30DaysPriorOrAfter
          == true
    permit
    on permit {
      obligation log {
        //obligations
      }
    }
  }

  rule defaultDeny {
    deny
    on deny {
      obligation log {
        //obligations
      }
    }
  }

  rule setBtgFlag{
    condition EMG.BTG == false
    permit
    on permit {
      obligation btgAudit {
        //obligations e.g. BtgStatus,
           btgPeriod, etc.
      }
    }
  }

  policy polSetBtgFlag {
    target clause resource.resourceId
       == "btg"
    apply permitOverrides
    setBtgFlag
    defaultDeny
  }

  policy EHR {
    target clause resource.resourceId
       == "ehr"
         clause action.actionId ==
            "write" or action.actionId
            == "read"
    apply permitOverrides
    readAndWriteRule
    BtgRule
    defaultDeny
  }

  policyset EHRPolicySet {
    apply permitOverrides
    EHR
    polSetBtgFlag
  }
}
```

Figure 4.5: Sample BTG policy from a major medical group, written in ALFA and reformatted for readability

largest US health care groups. We studied their procedure for overriding access control from [medical group redacted for peer review]. The intention for this task was to verify that our approach meets the requirements for a real-world emergency access control policy deployed within healthcare facilities. Our BTG "meta-policy" works with existing access control policies by adding explicit override permissions which are granted or denied based on various dynamic factors, but the real-world example represents an explicit (rather than meta) BTG policy for medical record access during emergencies. (We stress that our BTG design is strictly more powerful and flexible than what is currently used with electronic health records, as it not only allows access to information, but also differential access to sensing and treatment device functions based on user identity and properties.) Figure 4.6 provides an emergency access control (BTG) matrix based on our analysis of BTG functionality in [medical group redacted]. Figure 4.5 shows a policy within our framework that satisfies the BTG requirements in Figure 4.6.

[Medical group redacted] uses Epic[*] for managing emergency access control (BTG) to patient information. A BTG decision is based on patient type (e.g. VIP, confidential), clinician type (e.g. emergency department (ED), primary care physician (PCP), etc.), and several other constraints such as timeframe. Access is either granted with a BTG warning, or not granted at all. We found that only access to information for patients at a Federally Funded Substance Abuse Clinic (42 CFR) is denied for primary care physicians and clinicians of type "other" (but allowed for mental health clinicians). For all other accesses to information, clinicians are required to provide a reason when invoking BTG. BTG is needed every 7 days to access information on VIP patients, who may have greater privacy concerns, but not needed for a clinician who is part of a care team for a patient, or patients whom the clinician has seen within the last 30 days, or patients scheduled to see the clinician within the following 30 days.

Figure 4.5 shows that [medical group redacted]'s EHR BTG requirements are easily expressed in our BTG meta-policy. The `BtgRule` ensures the requirements (e.g. PCP's access to information for a patient with 42 CFR condition should never be granted even if the

[*] https://epic.com/

	Patient Type		
Clinician Type	*Mental Health*	*Confidential*	*42 CFR*
Emergency	permit	permit	permit
Mental Health	permit	permit	permit
PCPs	permit	permit	**deny**

Figure 4.6: Sample access control matrix based on patient and clinician types, indicating when BTG access is allowed and when it is denied

status of BTG is valid) are met prior to granting access to information. All "normal" access requests will be evaluated against `readAndWriteRule`. This rule ensures access to requested information will only be granted if a care relation exists between patient and clinician, or if the patient has had appointments 30 days prior or 30 days after the access date. The rest of the policy matches our meta-policy in Figure 4.4 and is self-explanatory. In addition, while our approach requires expressing any system-wide requirements (e.g. invoking BTG once every 7 days) as obligations to avoid adding any complexity to existing policies, there is no compelling reason why they should be added as `target clause` or `condition`. We are not aware of any further obligations other than the ones listed in the policy that need to be met prior to granting access to information within [medical group redacted].

4.5 Verification and Validation

Access policies in medical domain are defined by clinical administrators, and translated by facility technical/IT staff into a set of polices expressed in a formal access control language. The nature of the policies is governed not only by clinical role but by job title and perhaps even regulatory and contractual requirements. There is a natural knowledge gap between clinicians, administrators, and IT staff, who are expert in their respective fields, but must work together to ensure that formally-written access control policies represent the intent of the facility administrators and the needs of the clinicians. None of the people involved in crafting these requirements and policies may simultaneously have a full understanding of the policy intent and simultaneous grasp of the richness and constraints of the language

Tool (col.) / Feature support (row)	Custom Attribute Types	Relational Rules	Static Verification	Dynamic Verification	t-way Testing	XACML Support
Alloy[62]	○	-	●	-	-	●
ACPT[63]	●	○	●	●	●	●
Margrave[64]	-	○	●	-	-	◐
SPIN[65]	-	-	●	-	-	-
ACCOn[66]	-	-	●	-	-	-

Figure 4.7: A brief comparison of the model checking tools we considered. A more complete treatment can be found in the work of Aqib and Shaikh.[3]

expressing the authorization policy. As a result, these authorization policies are either too expressive or not expressive enough. Defining BTG requirements within authorization policies adds more complexity, which can easily result in misconfigurations and faulty policies, introducing serious vulnerabilities. Therefore, rigorous verification and validation through systematic testing are required to ensure the security properties are satisfied, the access control model is expressed correctly in the authorization policy, multiple policies enforced simultaneously are consistent, and single policies are self-consistent. Two policies can be called inconsistent if they are both applicable to a specific access request and yet return different decisions, e.g. one returns "permit" while the other returns "deny".

4.5.1 Tool Selection

To formally verify and test authorization policies for our BTG framework, we explored a series of access control policy test tools that claimed to be ideal solutions for testing access control policies. However, we were only able to perform experiments with one tool due to the fact the other tools either not compatible with our model or missing certain properties/features that we required for thorough testing. Tools which we would consider good candidates for validating our work should support model-based verification in addition to properties enumerated below. The features marked **bold** are required while others are helpful but not crucial.

1. **Different types of attributes**: XACML has four categories for attributes by default, subject, action, resource, and environment categories. These categories are supported

by available XACML implementations such as WSO2 Balana.[58] We are looking for support for the default and additional categories, including customized contextual categories of attributes with discrete and continuous values, since the types of attributes used in practice (e.g. at a medical facility) are not limited to these four categories. Additional categories can be added as needed.

2. **Rules with relational expressions**: a simple XACML policy such as in Figure 4.4 can contain conditions composed of relational expressions (written as logical expressions).

3. **Static and dynamic verification**: Although we currently only use static verification, both are useful in detection and resolution of inconsistency and incompleteness in policies. In a realistic deployment, we envision a static policy check before they are enacted, and continuous dynamic checking to detect problems after deployment.

4. t-way combination tests: a policy developer can easily end up with hundreds of access control polices even for a relatively small size organization. t-way combinatorial testing is useful for generating smaller, more manageable test suites and reducing testing costs.[67]

5. Native support for XACML: our access control policies are written in ALFA and then translated into XACML for reasons of compatibility and portability. Therefore, the tool needs to support importing, processing, and exporting standard XACML policies.

6. Headless mode: in actual deployments, we view the tool as a background verification process rather than a dedicated step in testing new policies or policy changes. Continuous, unobtrusive operation is therefore very desirable, allowing the tool to be easily integrated into existing policy processing workflows as an additional, intermediary step which does not require explicit operator intervention (unless an inconsistency is discovered).

Finding a tool that could support all or most of the above listed features was a challenging task due to the differences in approaches taken by authors for testing and validation of access control policies. Our search resulted in more than 12 tools or approaches. We filtered these tools by methods used and only focused on the approaches that were based on model checking. This reduced the list to only 5 tools. Our next step in filtering was to check for the tools that use or support XACML for policy specifications. But filtering by XACML would have left us with a very limited number of tools. Therefore, we also looked into tools that did not support XACML but some other policy specification language. A brief summary of the tools or approaches based on model checking is given in Table 4.7. We refer the readers to Aqib and Shaikh[3] for a detailed survey of verification and validation tools and approaches for access control policies.

The Access Control Policy Testing (ACPT) tool,[63;68] developed by the National Institute of Science and Technology (NIST) comes closest to fulfilling our requirements above. It can be used not only to compose and generate access control polices, but also to verify and test these policies, supporting both static and dynamic verification. The tool uses Symbolic Model Verification (SMV) model checker[69] for property checking and Automated Combinatorial Testing for Software (ACTS)[70] for test suite generation. In addition, ACPT can export verified access control policies in XACML format. ACPT supports importing existing policies, but only in its specified format. Since our policies are written in XACML format instead, we implemented a simple translator for a subset of XACML, which allowed us to translate our polices and import them. We use the example policy set in Figure 4.4. ACPT allows for policies to be either merged or combined prior to verification or testing. Merged policies means the policies selected for testing or verification will be merged without any order among them. Combining policies means policies will be combined based on the algorithm specified (e.g permit-overrides). In addition, the tool also provides a "default deny rule" option for each individual policy to be combined. By selecting this option, a default deny rule will be added to the resulting policy.

```
(resource = "btg")&(BTG = "False")&(role = "sys_admin")&(group = "BTG-allow"
   ) ->decision = Permit

(resource = "auth_policy")&(BTG = "True")&(role = "sys_admin")&(group = "
   normal") ->decision = Deny

(resource = "flowRate")&(BTG = "False")&(role = "sys_admin")&(group = "BTG-
   restricted") ->decision = Deny

(resource = "btg")&(BTG = "False")&(role = "physician")&(group = "normal") -
   >decision = Permit

(resource = "auth_policy")&(BTG = "True")&(role = "physician")&(group = "BTG
   -restricted") ->decision = Deny

(resource = "flowRate")&(BTG = "True")&(role = "physician")&(group = "BTG-
   allow") ->decision = Permit

(resource = "btg")&(BTG = "True")&(role = "visitor")&(group = "BTG-
   restricted") ->decision = Deny

(resource = "auth_policy")&(BTG = "False")&(role = "visitor")&(group = "BTG-
   allow") ->decision = Deny

(resource = "flowRate")&(BTG = "False")&(role = "visitor")&(group = "normal"
   ) ->decision = Deny

(resource = "btg")&(BTG = "False")&(role = "nurse")&(group = "BTG-allow") ->
   decision = Permit

(resource = "flowRate")&(BTG = "True")&(role = "nurse")&(group = "BTG-
   restricted") ->decision = Deny

(resource = "auth_policy")&(BTG = "True")&(role = "nurse")&(group = "normal"
   ) ->decision = Deny
```

Figure 4.8: Results of ACPT heuristic testing reformatted for readability and with metadata header removed.

4.5.2 Testing Results

We use ACPT to validate the BTG policy from Figure 4.4 and verify that out model meets requirements for emergency override. Our testing set is comprised of two policies: regular and BTG. Figure 4.9 shows the list of requirements for our authorization policy in form of properties along with status for each of the properties returned by the tool after verifying it. The result confirms that all properties hold. To avoid confusion we use BTG (in capital letters) as a flag indicating status of the BTG state of the system and btg (in small letters) as a resource in policies. A property is specified by adding attributes from different categories and an expected decision (minimum one attribute and a decision is required).

The testing result from the same policy set is given in Figure 4.8. The tool generates test cases and return the testing results that can be used for identifying conflicts, inconsistencies and other type of faults in the given policies. A property like the ones in Figure 4.9 can be written for an identified fault and run with ACPT for verification. Note *not all possible cases are covered due to the use of t-way combinatorial testing.* We were able to perform 2-way, 3-way and 4-way combinatorial testing on the given policy set. For ease of reading, we only discuss 2-way testing in this chapter. The 3-way and 4-way testing results are provided in Appendix B.

A default deny rule is added to the polices. This ensures that in cases where not a single policy matching an access request a deny decision is enforced. The testing result shows access requests for resources not in R are allowed even if the clinician does not have access to these resources ("BTG-allow" group). Similarly, access requests to resources that are in R are denied if BTG is true. In cases where resources belong to the "normal" group (expected to be allowed according to user identity and/or job function) but a false decision is returned, it means not a single applicable policy could be found by decision point. Since for this test setting our goal was if not a single applicable policy could be found to be evaluated against a request then a default deny should be returned. In Section 5.3.1 we provided details on the PEP logic for access situations like these.

4.6 Facilitating Revisions and Preventing Errors

Mistakes such as misconfigurations or faulty policies can remain undetected even after some thorough verification and testing. Our approach is designed to make "misassigning" a BTG-restricted resource difficult, unless one also adds or modifies the resource-specific policy in addition to the overall BTG policy. "Misassinging" a BTG-restricted resource, which is similar to not including a resource in a blacklist, can introduce system vulnerabilities. We built on the concept of a policy fault model presented in[71;72] to check for misassigned re-

```
spec AG (((((role = "physician" & resource = "flowRate") & group = "BTG-
   allow") & DefaultAction = "write") & BTG = "True") -> decision = Permit)
    is true

spec AG ((((role = "physician" & resource = "flowRate") & group = "BTG-
   restricted") & BTG = "True") -> decision = Deny) is true

spec AG (((role = "physician" & group = "BTG-restricted") & BTG = "True") ->
    decision = Deny) is true

spec AG ((group = "BTG-restricted" & BTG = "True") -> decision = Deny) is
   true

spec AG (((((role = "physician" & resource = "auth_policy") & group = "BTG-
   restricted") & DefaultAction = "write") & BTG = "True") -> decision =
   Deny) is true

spec AG (((((role = "sys_admin" & resource = "auth_policy") & group = "BTG-
   restricted") & DefaultAction = "write") & BTG = "False") -> decision =
   Permit) is true

spec AG (((((role = "sys_admin" & resource = "auth_policy") & group = "BTG-
   restricted") & DefaultAction = "write") & BTG = "True") -> decision =
   Deny) is true

spec AG (((resource = "btg" & group = "normal") & BTG = "False") -> decision
    = Permit) is true

spec AG (((((role = "visitor" & resource = "flowRate") & group = "BTG-allow"
   ) & DefaultAction = "write") & BTG = "True") -> decision = Permit) is
   true
```

Figure 4.9: Results of ACPT verification of policy consistency, reformatted for readability. An inconsistency within a facility's policy corpus will cause at least one of the specifications to evaluate as "false" and the tool will provide a counterexample.

sources. We introduced new resources and misassigned them to the list of resources other than BTG-restricted resources. Testing results for these resources were either "indeterminate" or "not applicable". By re-running the tests and adding default deny rules to these policies, the results changed to "deny". However, access to resources, for which we also added new policies or modified their existing policies, the authorization engine returned decisions as expected. Therefore, the testing confirmed the need for a policy modification step for already misassigned resources before they can allow unwanted access.

4.6.1 Recovering from a BTG State

During a medical emergency, a clinician forces the authorization system to allow BTG access. Once the clinician ends the emergency session, the system is now ready to be returned to a normal state. To safely and securely roll back the authorization system to a normal state, an audit may be required. This depends on multiple factors, including whether the system was in a controlled or uncontrolled BTG state, the amount of logging information which was collected (e.g. via partially fulfilled obligations during uncontrolled BTG), as well as general facility policies. Traditionally an authorization system is reset to normal state followed by a post-hoc log analysis performed by a person or team (e.g. privacy and/or security auditor(s) in conjunction with a clinical manager). Note that *a manual audit cannot guarantee* that a system will be returned to a known-good state, and in some cases rollback of the system is not possible without a fully manual audit going beyond the logs. An *inappropriate rollback* may occur through human error, such as not noticing a particularly critical event – leading to a system which is not truly restored to correct operation even though it has been manually set to "normal" operating mode. An *incorrect rollback*, e.g. a system which is not restored to a secure state, may occur if there have been unnoticed or uncorrected modifications to authorization procedure or policy. In both cases, the state of the system, connected devices, EMRs, and even the authorization policy database must be carefully examined, and reconfiguration by facility biomedical engineers or IT staff may be required.

For this reason, we argue for the creation of a BTG-restricted resource set. While our framework is sufficiently flexible to define policies wherein *authorized* access to the authorization policy database may occur during BTG, we strongly recommend that *any* access to the BTG-restricted set be disallowed during BTG for the simplification of specifying policies and performing audits. The manual system examination and reconfiguration mentioned above is an extremely time-consuming and expensive process, requiring specialized expertise. Therefore, marking certain resources (e.g. safety, security properties of medical devices, authorization policies) as BTG-restricted simplifies the reasoning about system state following both controlled and uncontrolled BTG, with at least the certainty that the security subsystem was not modified. **The formally defined behavior of the policy enforcement point described in Section 5.3.1, along with the access control policy, ensures a BTG-enforced session will not alter the security state of the overall system.**

It is important that access controls to resources in R are specified properly. Constructing a list of BTG-restricted resources (R) is easily achieved considering the limited number of resources a facility can mark as BTG-restricted resources – recall that our example BTG policy specified that resources in set R are only accessible by authorized individuals, as enforced by the authorization system, in the normal state. Verifying policies prior to implementation can also help in detecting faulty policies as discussed in Section 4.5.

Chapter 5

BTG Audits Using Obligations and System Changes

In Chapter 4 we propose a solution that allows for automated BTG audit during a controlled BTG state (Section 4.1). In Section 4.6.1 we also discuss how audits, which are tedious and introduce aspects of human error, require manual intervention only if the system is in an uncontrolled BTG state. In this chapter we further investigate how this can be automated using log analysis to detect access events and real-time system changes.

Patient health information (e.g. electronic health record) is considered confidential data and need to be protected from misuse. To this end, medical systems use a complex composition (policy) of fine-grained access control mechanisms to enforce separation of privilege and and ensure patient privacy. Such traditional mechanisms perform well in preventing misuse of patient health information by restricting users' access to that which is needed to fulfill their tasks. For example, in the Role-Based Access Control (RBAC) model,[19] clinicians are assigned roles based on their job descriptions (roles are usually tied very closely to job titles).[73] Each role is assigned one or more permissions. In order to ensure the authorization policies are consistent and represent the intent of the healthcare facility administrators and the needs of the clinicians, who are expert in their respected fields, they must work together to define access control policies following the principle of least-privilege to preserve patient safety and privacy. A least-privilege authorization policy limits users access to minimum permissions that are needed to perform their routine authorized activities. For example, a registered nurse should not be able to prescribe a medication, but should be able to start a prescribed medication for a patient.

During a medical emergency, however, clinicians need immediate access to patients information or medical components to provide the best possible care, and the strict enforcement of these policies during an emergency may prevent timely delivery of life-saving treatment. Therefore, an access control override mechanism is incorporated into healthcare facility authorization framework to override a denied decision and allow for an access request for dosage change or any other activity that could save patient's life during a medical emergency (which would have been denied otherwise).

Many governments have developed data protection legislation, such as the Healthcare Insurance Portability and Accountability Act[14;15] (HIPAA) in the United States, and the General Data Protection Regulations[74] (GDPR) in Europe. Healthcare providers must adhere to these rules, and ensure compliance to the established regulations for electronic data transmission, such as HIPPA and GDPR. HIPPA requires that availability of medical resources to be prioritized over patients' privacy or other security properties, in order to ensure medical systems do not deny life-saving treatment to patients in unforeseen situations.[14;15] This requirement, also known as "fail-open", makes it more challenging to craft least-privilege authorization policies (see Chapter 4).

Emergency access control override procedures such as BTG can be implemented without any modification to access control mechanism. For example, pre-staged emergency user accounts could be used to allow clinicians to override access control. Such emergency accounts are created in advance for one-time use and carry "sufficient" permissions for emergency use.[47] Administrators need to make sure these accounts are accessible in a reasonable manner, and clinicians are trained on how to use them. Furthermore, audit trials must be created for post-hoc analysis to determine appropriate actions and accountability in case of misuse. By performing a post-hoc audit, facilities try to determine the reasons (legitimacy) for overriding access control.

Emergency access sessions are normally logged for post-hoc audit and review to protect patients' privacy. Information such as the reason for access control override, user access session and any data access or modification should be clearly logged and communicated to the relevant workforce. Procedures for emergency access control override need to be well documented. And staff need to be trained how to use such mechanisms and made aware of the consequences of any inappropriate use. Moreover, a cleanup procedure is required to restore the system back to a safe operating state,[*] which includes tasks such as removal or changing passwords of any pre-staged emergency user accounts.[47]

Emergency access control override mechanisms defeat the purpose of least-privilege concept within medical domain, since during an override, patient data is no longer protected from misuse. For example, a clinician who is normally denied access to a VIP patient's electronic healthcare records can force the authorization system to operate in a "fail-open" mode, which is also known as uncontrolled BTG state 4.1,[2] by initiating a BTG access session and accessing the patient's data. Therefore, previous work has suggested constraining access during situations such as medical emergencies, when a user may need to exceed their normal privileges.[49] A very simple emergency access control policy may specify who can and who cannot request access control override during medical emergencies. In other words, the policy has to at least specify type of clinicians that are allowed to request BTG access. In some systems, clinicians are required to provide the reason for the BTG access before it is granted.[46;47;51] This is not enough and cannot prevent against misuse of the system for obtaining patient's confidential information. Confirmation of BTG by a clinician only self-reports the clinician's intention, and does not proactively protect against misuse.

Previous work has also suggested increased logging for tracing user actions and performing post-hoc audit.[46;47;50] Unfortunately, this does not proactively protect against misuse, but may provide for identification and punishment of a culprit after the fact. A recent survey of audit trails from a large group of hospitals in Norway reveals the use of BTG is a very common occurrence and thus generates logs at such a volume that reviewing them is a challenging

[*]Recall that a system is considered to be in normal operating state if it is following the principle of least-privilege, and access decisions are governed by the authorization policy and not overridden, as described in Section 4.1.

task.[52] Stark et al. propose a classification solution to the huge amount of transactions (logs from BTG access) from a university medical center in an attempt to decrease the number of transactions that need to be reviewed.[75] Their solution involves use of attributes (information within logs such as time/date, clinician job title, patient and clinician locations, etc.) and assigns them a score based on how likely they are to represent anomalous BTG access. For example, a BTG access transaction during a weekend night is scored higher (more likely anomalous) than a similar transaction that occurred during a weeknight.

Similarly, we propose a controlled BTG concept within policy domain in Chapter4. we show it is possible to construct a BTG "meta-policy" to allow users to exceed their normal privileges during a medical emergency while maintaining system safety and security protections. Three different operating states (normal, controlled BTG and uncontrolled BTG) represent the context of an override and a decision reevaluation is forced in the context of the override to provide "as-needed" access. They authors claim their approach significantly reduces the need for manual audit – only limiting manual audits to uncontrolled BTG accesses. Both work come short of presenting solution for automating log auditing for BTG accesses.

In this chapter we discuss how using real-time resource access log analysis and enforcement of logging obligations allows us to limit the extent of uncertainty of the system state following an emergency access session, and further allows for an *automatic recovery to a known safe and secure state* for most sessions. When an emergency is declared (BTG is invoked), the access control policy returns a series of logging obligations along with a more permissive authorization decision (i.e. the access control decision during a BTG session is almost always "permit"). When the access control framework is able to accommodate the obligations returned with an "allow-if-BTG" decision, the system institutes exceptional logging procedures, and allows a request that may otherwise be denied. Even when obligations cannot be fulfilled, the high-availability nature of medical systems may still require that the BTG request to be allowed, since overly strict enforcement of system obligations can prevent the timely delivery of life-saving treatment. Among the examples of emergency situations during which obligations may go unfulfilled is a denial of service attack on the facility's IT network, resulting in insufficient bandwidth to fulfill the increased BTG logging requirements. Currently, all emergency access control override (BTG) sessions are manually audited. This would be performed by an audit team either in real time or during the next business day. But in our work we identify only a limited number of BTG sessions that need to be handled manually, significantly reducing the workload and turnaround time of these audit teams. Furthermore, the level of automated logging provided by our system allows for structured log analysis, making even the manual process more straightforward and less error-prone, especially since automated logging reduces the reliance on human memory of these BTG incidents.

5.1 Logging

There has been limited work in the area of access control systems with BTG audit logging. Some previous work has noted the need for log analysis after access control override,[46;47] proposing that logging requirements be returned as obligations attached to a policy, but it is not immediately obvious what should happen if those obligations are not met. In this section, we discuss logging methods consistent with requirements for returning a system from a BTG to normal state, and show how our approach can be extended to partially or fully automate the post-BTG auditing process.

Reviewing audit logs for unusual activity is common in critical systems. Audit trails can be used to detect policy violation resulting in system crashes, unavailability of resources, network slowdowns etc. Emergencies requiring access control override are common in healthcare, and post-BTG audits are regular practice at medical facilities.[53;75] A post-hoc audit of the override will suggest appropriate actions, including disciplinary, e.g., if the emergency was inappropriately declared or inappropriate records were accessed.

Clinical facilities have individuals or groups who are tasked with running access control audits. Access reports are usually run by IT or Security and reviewed by Privacy or Compliance in conjunction with clinical management who would understand if the access was appropriate.[75] Auditors will review human- and machine-generated records for any flagged system activities either immediately after an emergency access event or during the next business day, and take appropriate actions. Automated log analysis may streamline some of this process while maintaining system-specific requirements. Log analysis tools can be used to provide a summary of user actions, resources accessed, obligations with their status (met/unmet), system state, etc. and modifications can be made to allow automation for a certain set of known activities commonly found in such logs. Some analyses can be quite voluminous due to the practice of requesting emergency medical override at individual edit or individual field granularity within current electronic health record systems. We are unaware of *timed BTG sessions*.

To attempt to streamline analysis without decreasing log granularity, levels are specified for the amount of logs to be generated. (This is in addition to tools intended to create log summaries and flag outlier events.) For example logs which only meet minimum requirements for audit would correspond to the normal state of the authorization system described in Section 4.1.1. The authorization system records request for access, returned decisions with evaluation details, changes to system resources and session specific information. Logging level set to "normal" ensures that the overall system performance is not significantly affected adversely by the amount of data processed, transferred, and stored. Logging granularity adjusts (via log "level" changes) as the system state changes. A state change from normal to BTG will require increased logging for auditing purposes (e.g. more frequent resource integrity checks). These logs should collect enough information to allow reconstruction of a timeline of what happened, and be able to restore the system back to a last known safe (normal) state. The logging level can be determined and set in a form of obligation within policies. When a decision is returned with logging obligations, the policy enforcement point (PEP)3.1 tries to fulfill the obligation and enforce the decision. If the PEP fails to fulfill an obligation, the original returned decision (e.g. "permit") is invalidated and returned to user as "deny". Contrasting this with BTG as currently practiced in the medical domain where availability is prioritized, we must allow the original decision ("permit") even if the returned obligations with the decision were not fulfilled. Our authorization system is designed to be flexible enough to allow for such steps to be taken automatically (as shown in Section 5.3.1) while maximizing the amount of useful data collected for later auditing.

When a BTG is requested, the policy decision point (PDP) will evaluate the BTG policy and return a decision (e.g. permit) with BTG obligations, which specifies logging requirements along with other requirements to be fulfilled prior to granting access. If all the requirements in form of BTG obligations are met then the system changes its state to a controlled BTG and will try to fulfill all the non-BTG obligations that are returned with decisions for other access requests during the same session.These obligations are not specific to BTG policy and will not affect the state of the system. Therefore, failing to fulfill one or all of non-BTG obligations will not result in a change in system state or logging level. However, if a logging or any other BTG-specific obligation is not fulfilled by PEP during a request for BTG, then the authorization system changes its state to uncontrolled BTG and prompts PEP to fulfill as many obligations as possible (both BTG and non-BTG) while allowing for access. This ensures logging requirements are set in accordance to the authorization state. For the sake of simplicity, from now on we refer to the levels of logging that are set for a controlled BTG state and uncontrolled BTG state as medium level and high level, respectively. The intent of differentiating between the levels of logging is to set requirements and specify what is necessary to log at each level.

Logging requirements are facility-specific, as each facility has its own customized system with facility-specific properties. Therefore, we want to avoid listing specific logging requirements but rather let them be specified as obligations in the facility-specific authorization policies. This design can be generalized to work with all logging requirements that can be specified by the policy language (e.g. can be written in a form of obligation in a policy) and are compatible with logging system components (e.g. the logging requirement can be translated into an input accepted by the logging system). We also understand that increased logging comes with a cost and in some situations may cause denial of service. Therefore, we suggest that logging requirements should be met only if there are enough system resources (network bandwidth) available to perform logging while not interrupting critical system processes.

Implementation and performance testing of a logging system is future work.

5.2 Post-BTG Audits

Medical emergencies requiring access control override are one concrete example of a usual activity in healthcare. Post-BTG audits are regular practice at medical facilities. A post-hoc audit of the override will suggest appropriate actions, including disciplinary, e.g., if the emergency was inappropriately declared or inappropriate records were accessed. To our knowledge there is no previously published work in computer or information science proposing logging mechanisms for controlling BTG sessions in the medical domain or on logging and auditing requirements in relation to overall system performance.

In 2016, research on examining access logs collected from an Electronic Patient Record (EPR) system used largely in a hospital in Norway found access logs to be a very useful tool for learning how to reduce the need for exception-based access.[52] After reviewing the access log from eight hospitals in the Central Norway Health Region (CNHR), the authors discovered that the use of exceptional mechanisms in these hospitals is too common mostly due to the lack of a stricter form of access control, and the huge size of the log makes it impracticable to audit the log for misuse. In a similar study of logs from a university medical center in Germany, Stark et al.[75] propose a solution to categorize transactions (BTG accesses) into two groups – A and B – representing normal transactions that need no manual audits, and anomalous transactions that need to be manually audited, respectively. Their proposed solution distinguishes anomalous transactions from normal transactions by investigating frequency of certain attributes (e.g. transaction timestamp, clinician job title) in the log. Each attribute is assigned a specific value range and a total value (sum of attributes score) for each transaction is calculated. For example, the "time and date" attribute has a range of 0 – 2 values. A value of 0 is assigned to daytime transaction while 1 is assigned to nighttime transaction. A maximum value of 2 is assigned to weekend transactions. The higher the total score for a transaction, the more likely it is for the transaction to be flagged as anomalous.

Using logs from a medical facility, Alizadeh et al.[53] presented an approach for user behavior analysis by constructing user behavior profiles and comparing them with expected behaviors. A final "anomaly" score assigned to a user's profile which determines the extent of the difference in that user's behavior from expected behavior. The authors applied their approach to study the use of "Break the Glass" (BTG). They do not investigate this approach in the content of real-time log analysis. In Chapter4 we present a controlled BTG model for interoperable medical systems which is based on well-studied ABAC model.[20;21] In our work, BTG is specified within policy and defined in terms of states in which the system is operating, and allows for overriding "deny" decisions automatically (instead of on a one-by-one basis) during a BTG session. This is achieved by constructing a BTG "meta-policy" which works with existing access control policies. In addition, previous work have noted the need for log analysis after access control override, proposing that logging requirements to be returned as obligations attached to a policy, but it is not immediately obvious what should happen if those obligations are not met.[46;47;49;50;54]

Current practices require clinical facilities to have individuals or groups who are tasked with running access control audits. Access reports are usually run by IT or Security and reviewed by Privacy or Compliance in conjunction with clinical management who would understand if the access was appropriate.[14;15] Auditors will review human- and machine-generated records for any flagged system activities either immediately after an emergency access event or during the next business day, and take appropriate actions. Automated log analysis may streamline some of this process while maintaining system-specific requirements. Log analysis tools can be used to provide a summary of user actions, resources accessed, obligations with their status (met/unmet), system state, etc. and modifications can be made to allow automation for a certain set of known activities commonly found in such logs.

5.3 Automating the audit process

In this section we present our approach of automating BTG audit using fine-grained structured logging and tracking system operating states. Our approach is compatible with any existing authorization framework that uses the Attribute Based Access Control (ABAC) model[20;21] and implements an emergency override mechanism. Furthermore, We discuss how our proposed solution integrates into an existing BTG model.[2] The BTG framework is based on a policy-based access control model (e.g. ABAC)[20] and has the properties enumerated below. The features marked in **bold** are required while others are helpful but not crucial for our work.

1. **Emergency override mechanism (BTG)**: our solution is dependant on existing implementation of emergency access override mechanism within an authorization framework to provide a base for integrating real-time log analysis. Authorization frameworks that lack emergency access override mechanism are outside the scope of this study.

2. **Definition of BTG within policy**: specifying BTG in the policy domain makes BTG more flexible, allows for fine-grained facility-specific policies and logging requirements, and facilitates for automation of auditing. This property is discussed in details in Section 4.3. A BTG procedure implemented outside the policy domain requires framework changes, and needs to be verified before we could reason about its compatibility with our approach.

3. Medical systems with high number of physiological data channels and real-time communication: while our solution is not limited to the medical domain – it is flexible enough to be applied to any other safety critical-system that use emergency access control override mechanism and prioritize safety over privacy – in this work we only focus on interoperable medical systems that come with high number of *physiological data channels*, *the real-time nature of the communication*, and *the large number of authorized users*.

4. Based on (compatible with) XACML: our logging requirements are predefined in the Abbreviated Language for Authorization (ALFA),[40] a high-level domain-specific language used in the formulation of access-control policies, and translated into XACML before use. Therefore, a framework that supports processing standard XACML policies is preferred. This is not a requirement, but helpful since XACML polices are represented in eXtensible Markup Language (XML) format and can be translated into

another policy language as long as the language can represent XACML policy elements. Also, the policy language need to support logging requirements returned to a policy decision enforcement point. For example, XACML has four default categories for attributes, subject, action, resource, and environment. These categories are supported by available XACML implementations such as WSO2 Balana.[58] We extend the default categories list with additional categories that include customized contextual and logging categories of attributes with discrete and continuous values, because the types of attributes used in practice (e.g. at a medical facility) are not limited to these four categories. Additional categories can be added as needed.

5.3.1 Obligations and Compatibility

Obligations are constraints that need to be met before a final decision is made by a policy decision point (PEP). For example, a permit decision returned to PEP along with obligations is changed to a deny decision if the any of the returned obligation cannot be met by PEP as shown in Section 4.1. During a BTG session, however, the state of the authorization system is used by a policy enforcement point to decide if obligations that cannot be fulfilled can be disregarded. To meet the "fail-open" requirement of medical systems, the authorization framework allows for the enforcement point and logging/obligations services to disregard obligations unless the authorization engine is in the normal state (see Section 4.1). Note that although the PEP is modified to meet the "fail-open" requirement, we still maintain XACML architecture compatibility – the enforcement point still functions as a compliant PEP.

According to OASIS XACML specifications, there are three different types ("flavors") of PEP: *deny-biased PEP, permit-biased PEP and base PEP.*[34] Each PEP flavor deals differently with situation where neither a permit or deny decision can be returned. **A deny-biased PEP** only allows an access request for which a permit decision is returned and all returned obligations can and will be fulfilled. In all other cases, it will deny the request. **A permit-biased PEP** denies an access request for which a deny decision was returned and all returned obligations can and will be fulfilled. In all other cases, it will permit access. i**In base PEP**, an access request is granted if a permit decision is returned by decision point and obligations can and will be fulfilled. Similarly, base PEP will deny an access request if a deny decision is returned by PEP and obligations can and will be fulfilled. In all other cases, an access request could either be allowed or denied. *It is up to developers to decide on how the base PEP will treat such access requests.* However, according to XACML specifications for base PEP, to be complaint it has to meet the following requirements:

- PEP permits access if PDP returns permit. If obligations are returned with the permit decision, then PEP permits access only if those obligations will be fulfilled by PEP.
- PEP denies access if PDP returns deny. If there are obligations returned with the deny decision, then PEP denies access only if those obligations will be fulfilled by PEP.
- In all other cases the PEP's behaviour is undefined. One such case is if an access request results in neither a permit nor a deny decision, e.g. if no rule can be found that matches the request – the policy simply does not specify what happens in case of that particular request.

In our approach, the PEP must *additionally* handle situations where *obligations cannot be fulfilled*. We use the state of the authorization engine as a pivotal factor in the decision process. By letting PEP query the state of the authorization engine, the PEP can be made to behave as follows:

- PEP permits access if PDP returns permit. If obligations are returned with the permit decision, then PEP permits access only if those obligations will be fulfilled by PEP.
- PEP denies access if PDP returns deny. If there are obligations returned with the deny decision, then PEP denies access only if those obligations will be fulfilled by PEP. An example application of this behavior of PEP is implementing access control based on the concept of blacklisting to ensure certain resources such as the resources in group R (as show in Figure 4.1.4) are denied access only if the returned obligations are met.

- In cases where the authorization system state is determined to be in either controlled BTG or uncontrolled BTG, PEP allows access only if PDP returns permit. If there are obligations returned with the permit decision, then PEP fulfills those obligations on a best-effort basis. If the obligations cannot be fulfilled, the PEP disregards them.

- In all other cases, PEP denies access.

The above list maintains the requirements for a complaint base PEP by only specifying its undefined behaviour to allow access during a controlled or uncontrolled BTG session only if PDP returns permit decision – recall that PDP is the policy decision point and PEP is the policy enforcement point (see Section 3). Obligations that cannot be fulfilled by PEP are disregarded. Note that when the authorization system is in controlled or uncontrolled BTG state, the BTG policy will be evaluated by PDP for a decision. For an access request if PDP returns deny after policy evaluation, then PEP denies access even if the authorization system is in a BTG state. This may seem counter-intuitive, but it is the intended behaviour since we require that BTG-restricted resources (resources that belong to set R)[2] must not be accessed to during a BTG session. An example policy has been provided in Figure 4.4.

5.3.2 System Architecture

A high level architecture of our approach is given in Figure 5.1. We choose Wazuh for log analysis because of its flexibility and open source.[76] It supports log analysis, file integrity checking, and policy monitoring. It was forked from the OSSEC host-based intrusion detection system (HIDS) project.[77] In practice, any system capable of preforming real-time log analysis integrity checking could be used. We only require a limited set of modules from Wazuh and the architecture presented in Section 3.1 in order to perform log analysis, shown in Figure 5.1. The main components of the architecture in Figure 5.1 are briefly described below:

- Interoperable medical system (a detailed description of the components of the interoperable medical system is provided in Section 3.1): the supervisor component forwards

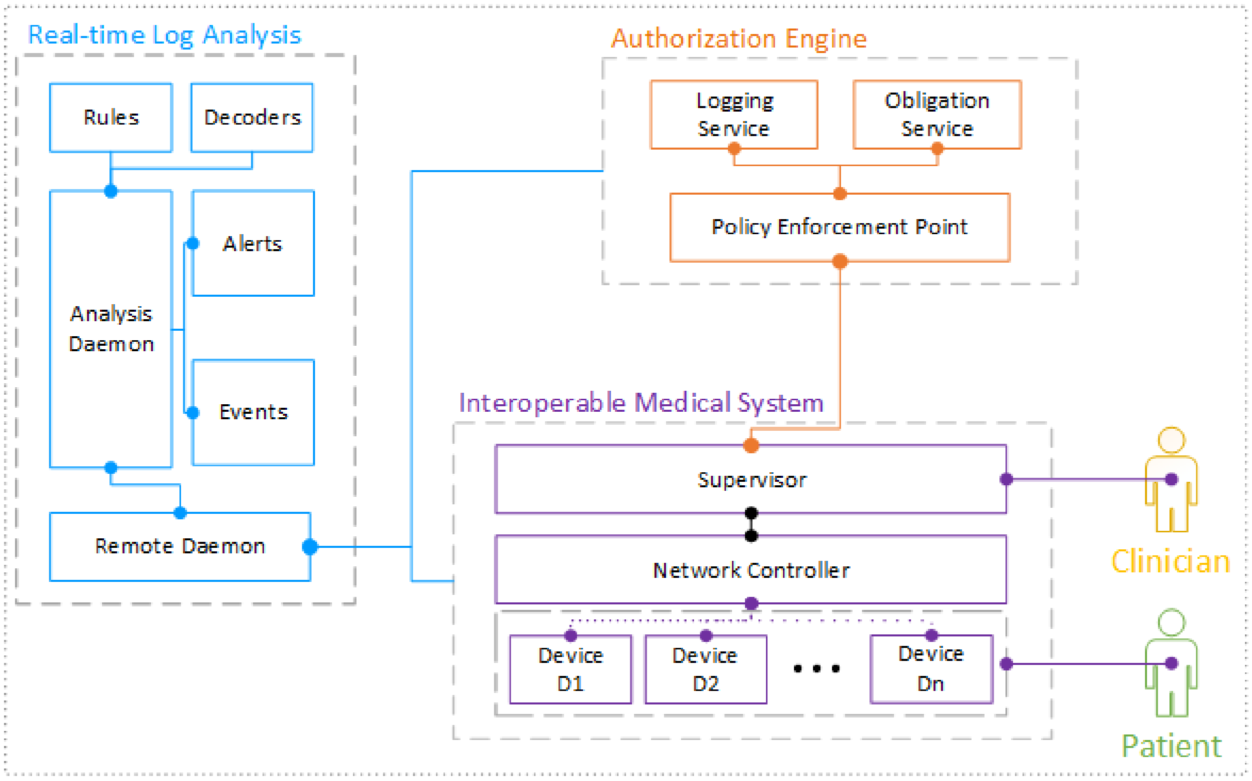


Figure 5.1: The integration of real-time log analysis with the authorization system.

access request to authorization engine for a decision. As shown in the Figure 3.5b an access request is created when a user (e.g. clinician) tries to access an application that communicates with a device attached to a patient via a data channel to get physiological data. The request is forwarded to authorization engine for decision.

- Authorization engine: the policy enforcement point (PEP) ensures a returned decision (after evaluation against a policy) is enforced along with its returned obligations (see Section 3). The *obligation service* component is responsible for: (a) fulfilling obligations (such as notifications to admins on BTG access request), and (b) returning a confirmation to the PEP. Logging levels, like any other obligations, are set in policy and returned to the PEP, as well. *Obligation service* handles all obligations except logging obligations. In order to ensure logging levels are set in accordance to the policy, *logging service*, an independent component, is required to communicate with all system components. The logging service handles logs for all components including obligation service within authorization engine and forwards the logs to the remote daemon for real-time log analysis.

- Real-time log analysis: the *remote daemon* collects logs and event data from all other system components. Agents are installed in the authorization engine and the medical system to collect system logs, configuration data, and forward them to the *remote daemon*. The *analysis daemon* receives data from *remote daemon* and analyzes it for changes in policy, systems states, and if any configurations are not compliant with policy. Rules are pre-written to identify events of interest and stored in *ruleset*. An alert will be generated if a received event (log record) matches a pattern. The *analysis daemon* uses *decoders* to process and extract information (e.g. user, system operating state, eventId, etc.) from received data.

```xml
<rule id="100005" level="11">
  <program_name>Balana</program_name>
  <match>CurrentSystemState:UncontrolledBtg</match>
  <description>system is in uncontrolled btg</description>
  <group>pci_dss_10.2.7,pci_dss_10.2.5,pci_dss_8.1.2,gpg13_4.13,gdpr_IV_35
    .7.d,gdpr_IV_32.2,hipaa_164.312.b,hipaa_164.312.a.2.I,hipaa_164.312.a
    .2.II,nist_800_53_AU.14,nist_800_53_AC.7,nist_800_53_AC.2,
    nist_800_53_IA.4,</group>
</rule>
```

Figure 5.2: Sample Wazuh rule for detecting uncontrolled BTG session.

All events generated during an access control session are analyzed in near real-time for abnormal activities. Information that are decoded and forwarded for analysis include but are not limited to *userId*, *patientId*, *deviceId*, *AppId*, *obligation(s) status*, *system operating state*, *resources accessed*, *access request parameters*, *policy evaluation results*, *target policy* and *timestamp*. Information are extracted from logs using customized decoders written for the authorization engine, medical devices and other components of the architecture in Figure 5.3.2. Decoded log events are later used by Wazuh ruleset for detection of software

```
May 18 01:56:36 AuthServer Balana[2665]: WARN CurrentSystemState:UncontrolledBtg
```

Figure 5.3: Sample log event.

misuse, configuration problems, application errors, system anomalies or security policy violations. We use customs rules in addition to the out-of-the-box set of rules in Wazuh to provide system state detection capabilities. An example rule to detect system transition to uncontrolled BTG is given in Figure 5.2. The rule level is set to 11 indicating the level of generated log event severity. Rule level values range from 0 (lowest) to 15 (maximum).[76] Each level indicates the severity of each triggered alert.An alerts is generated if a rule is matched against a log event. For example, the log event in Figure 5.3 is matched by the rule in Figure 5.2 and an alert of level 11 is generated. The list of groups in the rule are optional tags that define a behaviour. Similar rules are added to detect other events generated by the authorization engine, medical devices, etc. See the Wazuh website for a detailed documentation on rule definitions and how to write new customized rules.[76]

5.4 Evaluation

In this section we show how our implemented approach allows for logging and monitoring access control override mechanism – particularly an uncontrolled BTG session. We also show how our approach results in reduced administrative overhead by allowing administrators to focus only on a small subset of sessions and events. Our approach supports *automatic return of the system to a normal operating state* for all controlled BTG sessions, and uncontrolled BTG sessions, for which all BTG obligations (See Section 4.1) are met or no alerts of level 4 or above are generated.[†] Rules with level 4 or above identify some type of system error or software bad configurations. In the presence of any alerts with level 4 or above the system requires manual audit through human intervention.

We have run several simulated tests for generating reports and evaluating our imple-

[†]Implementation and performance testing of such a system is future work.

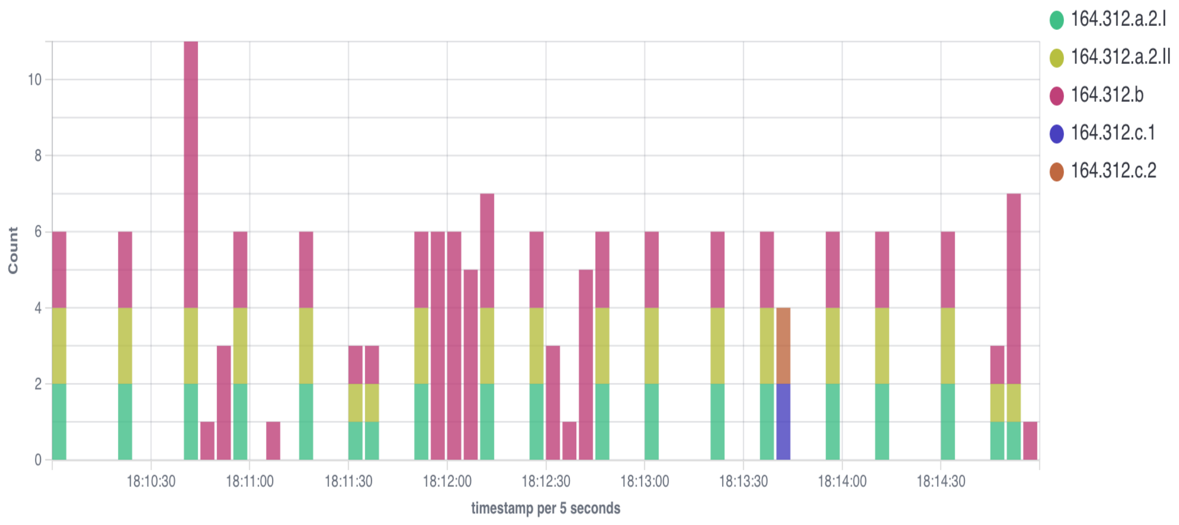


Figure 5.4: Events that resulted in HIPAA alerts. The color-coded labels reflect the specific regulation number within HIPAA.

mentation for monitoring access control override or BTG sessions. The simulation testing process was carefully designed to represent the possible states of the system (normal, controlled BTG and uncontrolled BTG) and generate specific types of alerts. For example, to demonstrate a state change from normal to controlled BTG, we generated test cases that would result in unfulfilled obligations. As discussed in Section 4.1, unfulfilled obligations forces the system to transition to controlled or uncontrolled BTG. Our tests have resulted in thousands of events generated an hour. For simplicity, in our evaluation we report only on log events and generated alerts from a testing window of 5 minutes.

A total of 78 alerts were generated during the selected 5 minute widow. The alerts were filtered from a set of 4320 log events. Figure 5.4 shows only the alerts that were triggered by HIPAA rules. In Figure 5.4 we see five different groups of alerts. Each group represent a specific HIPAA requirement. For example, group id 164.312.a.2.II represent the HIPAA requirement to establish a procedure for overriding access control during an emergency (also known as BTG). We refer the readers to HIPAA documentations[14;15] for a detailed description of each requirement.

Figure 5.5 shows a summarized graph of number of generated alerts and description for

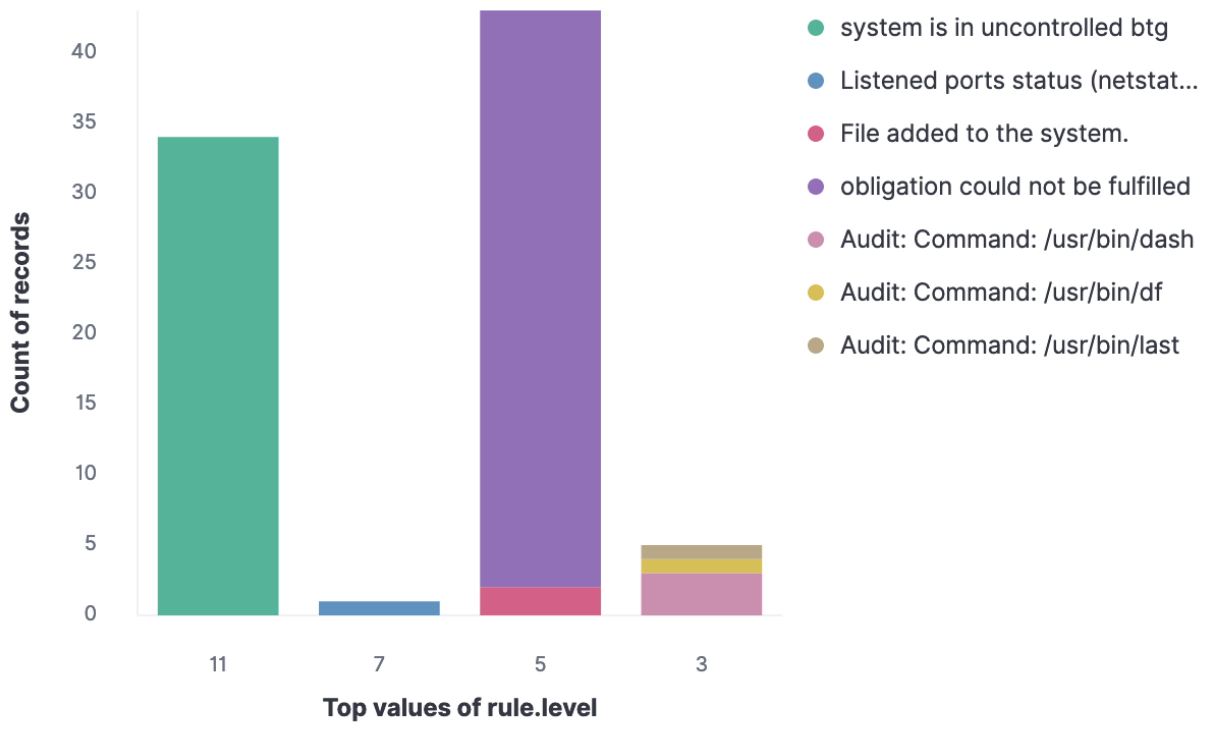


Figure 5.5: Top 7 rules with their counts and levels that triggered alerts. Only showing alerts from 5 minutes.

each rule resulting in the alerts. As shown, the number of uncontrolled BTG events triggering alerts of level 11 are one of the two more frequent events. The other more frequent event represent triggering alerts of level 5 is for unmet obligations. Other shown generated alerts belong to the auditing process, which is run during the uncontrolled BTG session to monitor system state. The auditing process is also responsible for detecting and logging changes to files (e.g. healthcare records, device configurations). Figure 5.5 shows two alerts from events of file addition (in red color) in the system.

Chapter 6

Conclusion

In this work we study authorization in interoperable medical systems. We explore different solutions with a focus on access control models that are applicable to the medical domain. We show how these models are not entirely generalizable to interoperable medical systems due to the core differences between interoperable and traditional systems – higher number of physiological data channels, real-time communication, and the sheer number of participants such as patients, clinicians, etc. We also investigate the use of emergency access control override mechanisms in medical systems and show how these mechanisms may compromise patients' privacy when not carefully designed. We the explore design strategies for emergency access control override mechanisms, placing them in the policy domain while minimized the required changes in already used authorization frameworks.

The Healthcare Insurance Portability and Accountability Act (HIPAA) requires healthcare provide to adhere to, and ensure compliance with, the established protocols for electronic data transmission. It also states that patient's safety must be prioritized over patient's privacy, which requires medical systems to follow "fail-open" concept. Clinicians should be granted access regardless of authorization policy to save a patient life.

We present 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, maintaining granularity and expressive power of past models, makes our access control model significantly different from prior models used in medical domain, which have historically protected mostly static electronic medical records. 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). We extend our study by proposing “Bend the Glass” access control model and show how to handle emergency access control override natively within the ABAC model, maintaining full compatibility with existing access control framework.

Current access controls override mechanisms in medical systems allow for uncontrolled access when needed, which may leave systems open to misuse and unnecessarily compromise patients’ privacy, resulting in irreversible damage. The controlled emergency access model in our work allows for a flexible emergency access control override while ensuring system safety- and security-critical resources are protected even during the Break the Glass state by managing system state and BTG obligations as specified in a well-structured formally verified and validated authorization policy. Furthermore, we show how the authorization architecture allows for the system to return to a known safe state and reduce or eliminate the need for manual audits when returning from a *controlled BTG* session. Currently, manual audits, which are tedious and introduce aspects of human error including faulty and incomplete memory of events. We reduce the requirement for full manual intervention in audits, mostly falling back on human operators only when the system has been in an “uncontrolled BTG” state. Even the latter can be automated using a system to simplify log analysis by automatically distilling data related to access events and real-time system changes. Additionally, we formally show that our example BTG “meta-policy” policy is expressed correctly and that policy specifications are satisfied.

6.1 Future Work

While our work makes some headway towards flexible authorization in interoperable medical systems, there are several important limitations, derived mostly from our limited access to non-PHI healthcare data. Given the difficulty in locating healthcare providers willing to participate in data sharing regarding their facility-specific authorization system policies, access control override mechanisms, authorization policies, and log data, we could only achieve a limited understanding of how our model would fit in a real healthcare setting. An in-depth qualitative and quantitative case studies of emergency access control override mechanisms in hospital settings is a subject for future work.

We have been able to restrict manual audits to uncontrolled BTG sessions – eliminating the need for manual audits for normal and controlled BTG sessions. We understand, however, investigating the effectiveness of anomaly detection during log analysis process for uncontrolled BTG sessions could further ease the human burden of post-hoc audits, and might entirely eliminate the need for most BTG sessions and fully automate auditing. Only sessions from uncontrolled BTG sessions that are tagged anomalous will require manual audits.

Another direction for future research includes investigation of system recovery procedure to determine if, using real-time resource access log analysis and enforcement of logging obligations, we can limit the extent of uncertainty of the system state following an emergency access session, and allow for recovery to a known safe and secure state. In our work we only propose a model. Implementation and performance testing of such a system is future work.

Bibliography

[1] Qais Tasali, Chandan Chowdhury, and Eugene Y Vasserman. A flexible authorization architecture for systems of interoperable medical devices. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 9–20. ACM, 2017.

[2] Qais Tasali, Christine Sublett, and Eugene Y. Vasserman. Controlled BTG: Toward flexible emergency override in interoperable medical systems. *EAI Endorsed Transactions on Security and Safety*, Online First, February 2020. doi: 10.4108/eai.13-7-2018.163213.

[3] Muhammad Aqib and Riaz Ahmed Shaikh. Analysis and comparison of access control policies validation mechanisms. *International Journal of Computer Network and Information Security*, 7(1):54–69, 2015.

[4] 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.

[5] Sofia K. Tzelepi, Dimitrios K. Koukopoulos, and George Pangalos. A flexible content and context-based access control model for multimedia medical image database systems. In *Workshop on Multimedia and Security: New Challenges*, pages 52–55. ACM, 2001.

[6] Mor Peleg, Dizza Beimel, Dov Dori, and Yaron Denekamp. Situation-based access control: Privacy management via modeling of patient data access scenarios. *Journal of Biomedical Informatics*, 41(6):1028–1040, 2008.

[7] Ramaswamy Chandramouli. A framework for multiple authorization types in a healthcare application system. In *Annual Computer Security Applications Conference (ACSAC)*, pages 137–148, 2001.

[8] Junzhe Hu and Alfred C. Weaver. A dynamic, context-aware security infrastructure for distributed healthcare applications. In *Workshop on Pervasive Privacy Security, Privacy, and Trust*, pages 1–8, 2004.

[9] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A role-based delegation framework for healthcare information systems. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 125–134. ACM, 2002.

[10] Jing Jin, Gail-Joon Ahn, Hongxin Hu, Michael J Covington, and Xinwen Zhang. Patient-centric authorization framework for sharing electronic health records. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 125–134. ACM, 2009.

[11] OpenICE User Introduction. https://www.openice.info/docs/1_overview.html. (Accessed on 01/26/2017).

[12] Andrew L. King, Sam Procter, Daniel Andresen, John Hatcliff, Steve Warren, William Spees, Raoul Praful Jetley, Paul L. Jones, and Sandy Weininger. An open test bed for medical device integration and coordination. In *ICSE Companion*, pages 141–151, 2009.

[13] Carlos Salazar and Eugene Y. Vasserman. Retrofitting communication security into a publish/subscribe middleware platform. In *International Workshop on Software Engineering in Healthcare (FHIES/SEHC)*, 2014.

[14] 45 CFR 164.312 - Technical safeguards, 2013. URL https://www.law.cornell.edu/cfr/text/45/164.312.

[15] U.S. Department of Health and Human Services Office for Civil Rights. HIPAA Administrative Simplification, March 2013.

[16] Andrew King, Dave Arney, Insup Lee, Oleg Sokolsky, John Hatcliff, and Sam Procter. Prototyping closed loop physiologic control with the medical device coordination framework. In *ICSE Workshop on Software Engineering in Health Care (SEHC)*, pages 1–11, 2010.

[17] National Electrical Manufacturers Association. Manufacturer disclosure statement for medical device security (MDS2). HIMSS/NEMA Standard HN 1-2013, 2013.

[18] Karen J Arthur, Ann Christine Catlin, Amanda Quebe, and Alana Washington. Changing smart pump vendors: Lessons learned. *Hospital pharmacy*, 51(9):782–789, 2016.

[19] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[20] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Arthur R. Friedman, Alan J. Lang, Margaret M. Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (ABAC) definition and considerations (draft). NIST Special Publication 800-162, 2013.

[21] Vincent C. Hu, D. Richard Kuhn, and David F. Ferraiolo. Attribute-based access control. *IEEE Computer*, 48(2):85–88, 2015.

[22] Syed Zain R. Rizvi, Philip W.L. Fong, Jason Crampton, and James Sellwood. Relationship-based access control for an open-source medical records system. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 113–124. ACM, 2015.

[23] Xin Jin, Ravi Sandhu, and Ram Krishnan. RABAC: Role-centric attribute-based access control. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 84–96. Springer, 2012.

[24] Indrakshi Ray, Toan C Ong, Indrajit Ray, and Michael G Kahn. Applying attribute based access control for privacy preserving health data disclosure. In *IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*, pages 1–4. IEEE, 2016.

[25] Ming Li, Shucheng Yu, Kui Ren, and Wenjing Lou. Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner

settings. In *International Conference on Security and Privacy in Communication Systems*, pages 89–106. Springer, 2010.

[26] Philip WL Fong. Relationship-based access control: protection model and policy language. In *Proceedings of the ACM conference on Data and application security and privacy*, pages 191–202. ACM, 2011.

[27] John Hatcliff, Andrew King, Insup Lee, Alasdair MacDonald, Anura Fernando, Michael Robkin, Eugene Y. Vasserman, Sandy Weininger, and Julian M. Goldman. Rationale and architecture principles for medical application platforms. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2012.

[28] John Hatcliff, Eugene Y. Vasserman, Todd Carpenter, and Rand Whillock. Challenges of distributed risk management for medical application platforms. In *IEEE Symposium on Product Compliance Engineering (ISPCE)*, 2018.

[29] Medical device "plug-and-play" interoperability program. http://mdpnp.org. (Accessed on 2/20/2017).

[30] Gerardo Pardo-Castellote. OMG data-distribution service: Architectural overview. In *International Conference on Distributed Computing Systems, Workshops*, pages 200–206. IEEE, 2003.

[31] Carlos Salazar. A security architecture for medical application platforms. Master's thesis, Kansas State University, 2014.

[32] Andrew L. King, Sanjian Chen, and Insup Lee. The middleware assurance substrate: Enabling strong real-time guarantees in open systems with OpenFlow. In *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 133–140. IEEE, 2014.

[33] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[34] 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, January 2013.

[35] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using XACML for access control in distributed systems. In *ACM Workshop on XML Security*, pages 25–37. ACM, 2003.

[36] Anna Carreras, Eva Rodríguez, and Jaime Delgado. Using XACML for access control in social networks. In *W3C Workshop on Access Control Application Scenarios*, 2009.

[37] Markus Jung, Georg Kienesberger, Wolfgang Granzer, Martin Unger, and Wolfgang Kastner. Privacy enabled web service access control using SAML and XACML for home automation gateways. In *International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 584–591. IEEE, 2011.

[38] Snezana Sucurovic. An approach to access control in electronic health record. *Journal of medical systems*, 34(4):659–666, 2010.

[39] A.A. Abd El-Aziz and A. Kannan. Access control for healthcare data using extended XACML-SRBAC model. In *International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–4. IEEE, 2012.

[40] Alfa, Mar 2015. URL https://www.axiomatics.com/solutions/products/authorization-for-applications/developer-tools-and-apis/192-axiomatics-language-for-authorization-alfa.html.

[41] Dianxiang Xu, Zhenyu Wang, Shuai Peng, and Ning Shen. Automated fault localization of XACML policies. In *Proceedings of the ACM on Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 2016.

[42] Jean Eyers, David M.and Bacon and Ken Moody. OASIS role-based access control for electronic health records. *IEEE Proceedings – Software*, 153(1), 2006.

[43] Jing Jin, Gail-Joon Ahn, Hongxin Hu, and Xinwen Covington, Michael J.and Zhang. Patient-centric authorization framework for electronic healthcare services. *Computers & Security*, 30(2):116–127, 2011.

[44] Thomas Hupperich, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. Flexible patient-controlled security for electronic health records. In *ACM SIGHIT International Health Informatics Symposium*, pages 727–732. ACM, 2012.

[45] Lillian Røstad. *Access control in healthcare information systems.* PhD thesis, Norwegian University of Science and Technology, 2008.

[46] Ana Ferreira, David Chadwick, Pedro Farinha, Ricardo Correia, Gansen Zao, Rui Chilro, and Luis Antunes. How to securely break into RBAC: The BTG-RBAC model. In *Annual Computer Security Applications Conference (ACSAC)*, pages 23–31. IEEE, 2009.

[47] Achim D Brucker and Helmut Petritsch. Extending access control models with break-glass. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 197–206. ACM, 2009.

[48] Eugene Y. Vasserman and John Hatcliff. Foundational security principles for medical application platforms. *Information Security Applications*, 8267:213–217, 2014.

[49] Dean Povey. Optimistic security: A new access control paradigm. In *New Security Paradigms Workshop (NSPW)*, pages 40–45. ACM, 1999.

[50] Erik Rissanen, Babak Sadighi Firozabadi, and Marek Sergot. Towards a mechanism for discretionary overriding of access control. In *International Workshop on Security Protocols (SPW)*, pages 312–319. Springer, 2004.

[51] Anna Ferreira, Ricardo Cruz-Correia, Luis Antunes, Pedro Farinha, E. Oliveira-Palhares, David W. Chadwick, and Altamiro Costa-Pereira. How to break access control in a controlled manner. In *IEEE International Symposium on Computer-Based Medical Systems (CBMS)*, pages 847–854. IEEE, 2006.

[52] Lillian Rostad and Ole Edsberg. A study of access control requirements for healthcare systems based on audit trails from access logs. In *Annual Computer Security Applications Conference (ACSAC)*, pages 175–186. IEEE, 2006.

[53] Mahdi Alizadeh, Sander Peters, Sandro Etalle, and Nicola Zannone. Behavior analysis in the medical sector: theory and practice. In *Proceedings of the Annual ACM Symposium on Applied Computing*, pages 1637–1646, 2018.

[54] Joint NEMA/COCIR/JIRA Security and Privacy Committee (SPC). Break glass procedure: Granting emergency access to critical ePHI systems. `https://hipaa.yale.edu/security/`, December 2014.

[55] Steve Barrett. The MDCF PCA Shutoff App 0.3 Documentation. `http://people.cs.ksu.edu/~scbarrett/pcashutoff-doc/`, 2015. (Accessed on 01/12/2017).

[56] Yu Jin Kim, Sam Procter, John Hatcliff, Venkatesh P. Ranganath, and Robby. Ecosphere principles for medical application platforms. In *International Conference on Healthcare Informatics (ICHI)*, pages 193–198, October 2015. doi: 10.1109/ICHI.2015.30.

[57] Apache Shiro Documentation. Apache Shiro — Simple. Java. Security. `https://shiro.apache.org/documentation.html`. (Accessed on 1/12/2017).

[58] Maduranga Siriwardena. Balana. `https://github.com/wso2/balana`, 2017. (Accessed on 1/12/2017).

[59] Pivotal Software, Inc. Spring security. `https://projects.spring.io/spring-security/`, 2017. (Accessed on 2/21/2017).

[60] Acciente, LLC. OACC — Java application security framework. `http://oaccframework.org/`, 2016. (Accessed on 2/21/2017).

[61] Apache Shiro Caching. Apache Shiro — Simple. Java. Security. `https://shiro.apache.org/caching.html`. (Accessed on 4/25/2017).

[62] Mahdi Mankai and Luigi Logrippo. Access control policies: Modeling and validation. In *NOTERE Conference (Nouvelles Technologies de la Répartition)*, pages 85–91, 2005.

[63] JeeHyun Hwang, Tao Xie, Vincent Hu, and Mine Altunay. ACPT: A tool for modeling and verifying access control policies. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 40–43. IEEE, 2010.

[64] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the International conference on Software engineering (ICSE)*, pages 196–205, 2005.

[65] Jianli Ma, Dongfang Zhang, Guoai Xu, and Yixian Yang. Model checking based security policy verification and validation. In *International Workshop on Intelligent Systems and Applications*, pages 1–4. IEEE, 2010.

[66] Loreto Bravo, James Cheney, and Irini Fundulaki. Accon: checking consistency of xml write-access control policies. In *Proceedings of the International conference on Extending database technology: Advances in database technology*, pages 715–719, 2008.

[67] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. IPOG: A general strategy for t-way software testing. In *IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*, pages 549–556. IEEE, 2007.

[68] NIST. Access control policy tool (ACPT). https://www.nist.gov/programs-projects/access-control-policy-tool-acpt, 2018. (Accessed on 3/13/2018).

[69] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *International conference on computer aided verification (CAV)*, pages 495–499. Springer, 1999.

[70] NIST. Automated combinatorial testing for software (ACTS). https://csrc.nist.gov/Projects/Automated-Combinatorial-Testing-for-Software, 2018.

[71] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *International conference on World Wide Web (WWW)*, pages 667–676. ACM, 2007.

[72] Dianxiang Xu, Roshan Shrestha, and Ning Shen. Automated coverage-based testing of XACML policies. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 3–14. ACM, 2018.

[73] Ravi Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *Proceedings of the third ACM workshop on Role-based access control*, pages 47–54, 1998.

[74] Paul Voigt and Axel Von dem Bussche. The EU general data protection regulation (GDPR). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

[75] Benjamin Stark, Heiko Gewald, Heinrich Lautenbacher, Ulrich Haase, and Siegmar Ruff. Misuse of 'break-the-glass' policies in hospitals: Detecting unauthorized access to sensitive patient health data. *International Journal of Information Security and Privacy (IJISP)*, 12(3):100–122, 2018.

[76] Wazuh. URL https://documentation.wazuh.com/3.12/user-manual/overview.html.

[77] Rory Bray, Daniel Cid, and Andrew Hay. *OSSEC host-based intrusion detection guide.* Syngress, 2008.

Appendix A

ALFA Policies

```
namespace healthcare_facility {
  import BTG_Attributes.*
  import Attributes.*

  obligation log = "obligations.normal"
  obligation btgAudit = "obligations.btg"

  // ehr specific btg access rule
  rule emg_ehr_rule {
    target clause EMG.BTG == true

    //42 refers to 42CFR
    conditio "clinician" == stringOneAndOnly(user.userType)
      && not (integerOneAndOnly(resource.group) == 3)
      && (integerOneAndOnly(patient.currentLocationId) ==
          integerOneAndOnly(user.currentLocationId)
      || integerOneAndOnly(patient.currentLocationId) == 9110)
      && not (integerOneAndOnly(patient.currentLocationId) == 42)
    permit
    on permit {
      // logging info
      obligation btgAudit {
        log.btgMessage = "emg_ehr sample 1"
      }
    }
  }

  // device data specific btg access rule
  rule emg_device_rule {
    // Any clinician can set infusion rate up to hard limit for a patient during
        an emergency if:
    //  a. system is in emergency state (controlled or uncontrolled BTG) due to a
        requested emergency access control override
    //  b. clinician is denied access to infusion rate settings
    //  c. clinician is physically located within the same facility as patient
    //  d. the requested resource (infusion rate) is NOT identified as emergency
        exempt (BTG-restricted) resource
    //  e. the patient and clinician are two different individuals

    target clause EMG.BTG == true

    condition "clinician" == stringOneAndOnly(user.userType)
      && device.devType == "pca"
      && not (integerOneAndOnly(resource.group) == 3)
      && (integerOneAndOnly(patient.currentLocationId) ==
          integerOneAndOnly(user.currentLocationId))
      && not (integerOneAndOnly(patient.currentLocationId) == 42)
      && not (integerOneAndOnly(patient.patientId) ==
          integerOneAndOnly(user.userId))

    permit
    on permit {
      // logging info
      obligation btgAudit {
        log.btgMessage = "emg_dev sample 1"
      }
    }
  }

  rule ehr_clinician_rule {
    target clause user.role == "nurse"
      or user.role == "physician"

    // making sure a care relation exits between patient and care giver
    condition integerIsIn(integerOneAndOnly(user.userId),
        patient.assignedClinicianId)
      && patient.hasOrHadAppointmentWithin30Days == true
      && integerIsIn(integerOneAndOnly(user.currentLocationId),
          patient.appointmentLocationId)

    permit
    on permit {
      // logging info
      obligation log {
        log.message = "ehr_clinician_rule sample 1"
      }
    }
  }

  rule ehr_patient_rule {
    target clause user.userType == "patient"

    // making sure the user is the patient whose ehr is being accessed
    condition not (stringOneAndOnly(resource.resourceType) == "clinician_note")
        && (integerOneAndOnly(user.userId) ==
            integerOneAndOnly(patient.patientId))

    permit
    on permit {
      // logging info
      obligation log {
        log.message = "ehr_patient_rule sample 1"
      }
    }
  }

  rule ehr_others_rule {
    target clause user.userType == "others"
            clause resource.resourceType == "med_history"

    // making sure the user is the patient whose ehr is being accessed
    condition integerIsIn(integerOneAndOnly(user.userId),
        patient.authorizedIndividuals)
    permit
    on permit {
      // logging info
      obligation log {
        log.message = "ehr_others_rule sample 1"
      }
    }
  }

  rule vitals_patient_rule {
    target clause resource.resourceType == "vitals"

    condition
        timeGreaterThanOrEqual(timeOneAndOnly(resource.resourceExchangeTime),
        timeOneAndOnly(currentTime))
      && (
          user.role == "remote_clinician"
          || user.userType == "device"
          || user.userType == "app"
          || (
            user.userType == "clinician"
            && user.status == "active"
            && integerOneAndOnly(user.currentLocationId) ==
                integerOneAndOnly(patient.currentLocationId)
          )
        )

    permit
    on permit {
```

```
    // logging info
    obligation log {
      log.message = "vitals_patient_rule sample 1"
    }
  }
}

rule personal_info_patient_read_rule {
  target clause resource.resourceType == "personal_info"

  condition ((
      user.role == "physician"
      || user.role == "nurse"
      )
    && user.status == "active"
    && integerIsIn(integerOneAndOnly(user.userId), patient.assignedClinicianId)
    && patient.hasOrHadAppointmentWithin30Days == true
    && integerIsIn(integerOneAndOnly(user.currentLocationId),
       patient.appointmentLocationId)
    )
    || ((integerOneAndOnly(user.userId) == integerOneAndOnly(patient.patientId))
    && user.userType == "patient")

  permit
  on permit {
    // logging info
    obligation log {
      log.message = "personal_info_patient_read_rule sample 1"
    }
  }
}

rule personal_info_patient_write_rule {
  target clause resource.resourceType == "personal_info"

  condition (not (resource.resourceName == "patient_vip_status")
    && ((
      user.role == "physician"
      || user.role == "nurse"
      )
    && user.status == "active"
    && integerIsIn(integerOneAndOnly(user.userId), patient.assignedClinicianId)
    && patient.hasOrHadAppointmentWithin30Days == true
    && integerIsIn(integerOneAndOnly(user.currentLocationId),
       patient.appointmentLocationId))
    )
    || (resource.resourceName == "patient_vip_status"
    && user.userType == "admin")

  permit
  on permit {
    // logging info
    obligation log {
      log.message = "personal_info_patient_write_rule sample 1"
    }
  }
}
```

```
rule device_alarm_rule {
  // only <<other>> med devices and apps can set or get alarm status

  condition ((integerIsIn(integerOneAndOnly(user.userId),
      patient.assignedClinicianId))
      && (integerOneAndOnly(user.currentLocationId) ==
          integerOneAndOnly(patient.currentLocationId))
    )
    //should we also consider device/app location and relation, or it is
        assumed to be in the same location (maybe for the time)?
    || user.userType == "device"
    || user.userType == "app"

  permit
  on permit {
    // logging info
    obligation log {
      log.message = "device_alarm_rule sample 1"
    }
  }
}

rule device_configuration_rule {
  // only admins can change the config on a device

  condition (integerOneAndOnly(user.currentLocationId) ==
      integerOneAndOnly(patient.currentLocationId))
  //should we also consider device/app location and relation, or it is assumed
      to be in the same location (maybe for the time)?
  && (user.userType == "admin"
  || user.userType == "app")

  permit
  on permit {
    // logging info
    obligation log {
      log.message = "device_configuration_rule sample 1"
    }
  }
}

rule device_infusion_rate_write_rule {
  target clause device.devType == "pca"
          clause action.actionId == "write"

  // An assigned clinician for a patient can set the infusion rate up to soft
      limit for the patient (e.g. write access for infusion rate for a PCA
      pump).
  //
  // or:
  //
  // An assigned physician can only set infusion rate above soft limit (but not
      beyond hard limit).
  //
  // or:
  //
  // An ER clinician can set infusion rate (up to hard limit) for a patient
      while in ER
  // (think of a new patient admitted to ER)

  condition ((integerIsIn(integerOneAndOnly(user.userId),
      patient.assignedClinicianId)
      && integerOneAndOnly(user.currentLocationId) ==
          integerOneAndOnly(patient.currentLocationId))
      && (
      (device.userProvidedInfusionRate <= device.infusionRateSoftLimit &&
          user.role == "nurse")
      || (user.role == "physician"
      && device.userProvidedInfusionRate <= device.infusionRateHardLimit)))
    ||
    (user.role == "er_clinician"
    && patient.currentStatus == "emg"
    && device.userProvidedInfusionRate <= device.infusionRateHardLimit)


  permit
  on permit {
    // logging info
    obligation log {
      log.message = "device_infusion_rate_write_rule sample 1"
    }
  }
}

rule device_infusion_rate_read_rule {
  target clause device.devType == "pca"
          clause action.actionId == "read"
```

```
// An assigned clinician for a patient can set the infusion rate up to soft
   limit for the patient (e.g. write access for infusion rate for a PCA
   pump).
//
// or:
//
// An assigned physician can only set infusion rate above soft limit (but not
   beyond hard limit).
//
// or:
//
// An ER clinician can set infusion rate (up to hard limit) for a patient
   while in ER
// (think of a new patient admitted to ER)

condition (device.infusionRate >= 0)
  && (((integerOneAndOnly(user.currentLocationId) ==
      integerOneAndOnly(patient.currentLocationId))
      && (user.userType == "clinician" || user.userType == "patient")
    )
  || (user.userType == "app"
  || user.userType == "device")
  )

permit
on permit {
  // logging info
  obligation log {
      log.message = "device_infusion_rate_read_rule sample 1"
    }
  }
}

rule device_pca_vital_read_rule {
  target clause device.devType == "pca"
         clause action.actionId == "read"
  // An assigned clinician for a patient can set the infusion rate up to soft
      limit for the patient (e.g. write access for infusion rate for a PCA
      pump).
  //
  // or:
  //
  // An assigned physician can only set infusion rate above soft limit (but not
      beyond hard limit).
  //
  // or:
  //
  // An ER clinician can set infusion rate (up to hard limit) for a patient
      while in ER
  // (think of a new patient admitted to ER)

  condition (((integerOneAndOnly(user.currentLocationId) ==
      integerOneAndOnly(patient.currentLocationId))
      && (user.userType == "clinician" || user.userType == "patient")
    )
    || (user.userType == "app"
      || user.userType == "device")
  )

  permit
  on permit {
    // logging info
    obligation log {
      log.message = "device_pca_vital_read_rule sample 1"
    }
  }
}

rule default_deny {
  deny
  on deny {
    // logging info
    obligation log {
      log.message = "device_pca_vital_read_rule sample 1"
    }
  }
}

policy ehr_clinician_policy {
  target clause resource.resourceCategory == "ehr"
          clause action.actionId == "write" or action.actionId == "read"
  apply permitOverrides
  ehr_clinician_rule
  emg_ehr_rule
}
```

```
policy ehr_patient_policy {
  target clause resource.resourceCategory == "ehr"
          clause action.actionId == "read"
  apply permitOverrides
  ehr_patient_rule
}

policy ehr_personal_info_policy {
  target clause resource.resourceCategory == "ehr"
          clause action.actionId == "read"
  apply permitOverrides
  personal_info_patient_read_rule
  personal_info_patient_write_rule
}



policy ehr_others_policy {
  target clause resource.resourceCategory == "ehr"
          clause action.actionId == "read"
  apply permitOverrides
  ehr_others_rule
}

policy vitals_patient_policy {
  target clause resource.resourceCategory == "ehr"
          clause action.actionId == "read"
            or action.actionId == "write"
  apply permitOverrides
  vitals_patient_rule
```

```
}

policy device_pca_alarm_policy {
  target clause resource.resourceName == "pca_alarm"
         clause action.actionId == "read"
           or action.actionId == "write"
  apply permitOverrides
  device_alarm_rule
}

policy device_pca_infusion_rate_policy {
  target clause resource.resourceName == "pca_infusion_rate"
         clause action.actionId == "write"
           or action.actionId == "read"
  apply permitOverrides
  device_infusion_rate_write_rule
  device_infusion_rate_read_rule
  emg_device_rule
}

policy device_pca_vitals_policy {
  target clause resource.resourceName == "pca_vitals"
         clause action.actionId == "read"
  apply permitOverrides
  device_pca_vital_read_rule
  emg_device_rule
}

policy device_pca_configuration_policy {
    target clause resource.resourceName == "pca_configuration"
            clause action.actionId == "write"
              or action.actionId == "read"
    apply permitOverrides
    device_configuration_rule
    emg_device_rule
  }

  policy default_deny_policy {
    apply permitOverrides
    default_deny
  }

  policyset ehr_policy_set {
    apply permitOverrides
    ehr_personal_info_policy
    ehr_clinician_policy
    vitals_patient_policy
    ehr_others_policy
    ehr_patient_policy
    device_pca_alarm_policy
    device_pca_infusion_rate_policy
    device_pca_configuration_policy
    device_pca_vitals_policy
  }
}
```

Appendix B

3- and 4-way Combinatorial Test Cases

Listing B.1: Results of ACPT heuristic 3-way merged testing: BTG policy

```
1: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

2: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(role
    = sys_admin)->decision = Permit

3: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Deny

4: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(role
    = sys_admin)->decision = Deny

5: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_restricted
   )&(role = sys_admin)->decision = Deny

6: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

7: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

8: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny

9: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Deny
```

```
10: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   physician)->decision = Permit

11: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = physician)->decision = Deny

12: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    physician)->decision = Permit

13: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = physician)->decision = Deny

14: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

15: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

16: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    physician)->decision = Deny

17: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = physician)->decision = Deny

18: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = physician)->decision = Deny

19: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   visitor)->decision = Deny

20: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = visitor)->decision = Permit

21: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   visitor)->decision = Deny

22: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = visitor)->decision = Deny

23: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = visitor)->decision = Deny

24: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = visitor)->decision = Deny

25: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   visitor)->decision = Deny

26: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = visitor)->decision = Deny

27: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Deny
```

```
28: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role = nurse
   )->decision = Deny

29: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = nurse)->decision = Permit

30: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   nurse)->decision = Deny

31: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = nurse)->decision = Deny

32: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Deny

33: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Deny

34: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   nurse)->decision = Deny

35: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Deny

36: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Deny
```

Listing B.2: Results of ACPT heuristic 3-way merged testing: flowRate policy

```
1: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

2: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(role
    = sys_admin)->decision = Deny

3: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Deny

4: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(role
    = sys_admin)->decision = Permit

5: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_restricted
   )&(role = sys_admin)->decision = Deny

6: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

7: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

8: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny

9: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Permit

10: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   physician)->decision = Permit

11: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = physician)->decision = Deny

12: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    physician)->decision = Deny

13: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = physician)->decision = Deny

14: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

15: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

16: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    physician)->decision = Permit

17: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = physician)->decision = Deny
```

```
18: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
    role = physician)->decision = Deny

19: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
    visitor)->decision = Deny

20: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
    role = visitor)->decision = Deny

21: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
    visitor)->decision = Deny

22: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
    role = visitor)->decision = Deny

23: (resource_id = auth_policy)&(BTG = True)&(resource_group =
    btg_restricted)&(role = visitor)->decision = Deny

24: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
    &(role = visitor)->decision = Deny

25: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
    visitor)->decision = Deny

26: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
    &(role = visitor)->decision = Deny

27: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
    role = visitor)->decision = Deny

28: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role = nurse
    )->decision = Deny

29: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
    role = nurse)->decision = Deny

30: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
    nurse)->decision = Deny

31: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
    role = nurse)->decision = Deny

32: (resource_id = auth_policy)&(BTG = True)&(resource_group =
    btg_restricted)&(role = nurse)->decision = Deny

33: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
    &(role = nurse)->decision = Deny

34: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
    nurse)->decision = Permit

35: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
```

```
   &(role = nurse)->decision = Deny

36: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Permit
```

Listing B.3: Results of ACPT heuristic 4-way combined testing: BTG policy

```
1: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

2: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   sys_admin)->decision = Permit

3: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
   = sys_admin)->decision = Deny

4: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(role
    = sys_admin)->decision = Permit

5: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Deny

6: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Permit

7: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
   = sys_admin)->decision = Deny

8: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(role
    = sys_admin)->decision = Deny

9: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_restricted
   )&(role = sys_admin)->decision = Deny

10: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = sys_admin)->decision = Deny

11: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

12: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

13: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

14: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    sys_admin)->decision = Deny

15: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny

16: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny

17: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Deny
```

```
18: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Deny

19: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   physician)->decision = Deny

20: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   physician)->decision = Permit

21: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = physician)->decision = Deny

22: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = physician)->decision = Permit

23: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   physician)->decision = Deny

24: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    physician)->decision = Permit

25: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = physician)->decision = Deny

26: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = physician)->decision = Deny

27: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

28: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

29: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

30: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

31: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   physician)->decision = Deny

32: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    physician)->decision = Deny

33: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = physician)->decision = Deny

34: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = physician)->decision = Deny

35: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
role = physician)->decision = Deny

36: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = physician)->decision = Deny

37: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   visitor)->decision = Deny

38: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   visitor)->decision = Permit

39: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = visitor)->decision = Deny

40: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = visitor)->decision = Permit

41: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   visitor)->decision = Deny

42: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    visitor)->decision = Permit

43: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = visitor)->decision = Deny

44: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = visitor)->decision = Deny

45: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = visitor)->decision = Deny

46: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = visitor)->decision = Deny

47: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = visitor)->decision = Deny

48: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = visitor)->decision = Deny

49: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   visitor)->decision = Deny

50: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    visitor)->decision = Deny

51: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = visitor)->decision = Deny

52: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = visitor)->decision = Deny
```

```
53: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Deny

54: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Deny

55: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role = nurse
   )->decision = Deny

56: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   nurse)->decision = Permit

57: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = nurse)->decision = Deny

58: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = nurse)->decision = Permit

59: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   nurse)->decision = Deny

60: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    nurse)->decision = Permit

61: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = nurse)->decision = Deny

62: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = nurse)->decision = Deny

63: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Deny

64: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Deny

65: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Deny

66: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Deny

67: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   nurse)->decision = Deny

68: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    nurse)->decision = Deny

69: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Deny

70: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Deny
```

```
71: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
    role = nurse)->decision = Deny

72: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
    role = nurse)->decision = Deny
```

Listing B.4: Results of ACPT heuristic 4-way combined testing: flowRate policy

```
1: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

2: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   sys_admin)->decision = Permit

3: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
   = sys_admin)->decision = Deny

4: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(role
    = sys_admin)->decision = Deny

5: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Deny

6: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Deny

7: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
   = sys_admin)->decision = Permit

8: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(role
    = sys_admin)->decision = Permit

9: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_restricted
   )&(role = sys_admin)->decision = Deny

10: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = sys_admin)->decision = Deny

11: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

12: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

13: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

14: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    sys_admin)->decision = Deny

15: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny

16: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny

17: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Permit
```

```
18: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Deny

19: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   physician)->decision = Deny

20: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   physician)->decision = Permit

21: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = physician)->decision = Deny

22: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = physician)->decision = Deny

23: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   physician)->decision = Deny

24: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    physician)->decision = Deny

25: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = physician)->decision = Deny

26: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = physician)->decision = Deny

27: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

28: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

29: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

30: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

31: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   physician)->decision = Permit

32: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    physician)->decision = Permit

33: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = physician)->decision = Deny

34: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = physician)->decision = Deny

35: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
```

role = physician)->decision = Permit

36: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(role = physician)->decision = Deny

37: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role = visitor)->decision = Deny

38: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role = visitor)->decision = Permit

39: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role = visitor)->decision = Deny

40: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(role = visitor)->decision = Deny

41: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role = visitor)->decision = Deny

42: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role = visitor)->decision = Deny

43: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role = visitor)->decision = Deny

44: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(role = visitor)->decision = Deny

45: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_restricted)&(role = visitor)->decision = Deny

46: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_restricted)&(role = visitor)->decision = Deny

47: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)&(role = visitor)->decision = Deny

48: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)&(role = visitor)->decision = Deny

49: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role = visitor)->decision = Deny

50: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role = visitor)->decision = Deny

51: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)&(role = visitor)->decision = Deny

52: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)&(role = visitor)->decision = Deny

```
53: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Permit

54: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Deny

55: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role = nurse
   )->decision = Deny

56: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   nurse)->decision = Permit

57: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = nurse)->decision = Deny

58: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = nurse)->decision = Deny

59: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   nurse)->decision = Deny

60: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    nurse)->decision = Deny

61: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = nurse)->decision = Deny

62: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = nurse)->decision = Deny

63: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Deny

64: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Deny

65: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Deny

66: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Deny

67: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   nurse)->decision = Permit

68: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    nurse)->decision = Permit

69: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Deny

70: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Deny
```

```
71: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Permit

72: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Deny
```

Listing B.5: Results of ACPT heuristic 3-way testing: combined policies

```
## For the combined policies, we assign default dummy values for unused
   attributes.

1: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

2: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(role
    = sys_admin)->decision = Permit

3: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Deny

4: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(role
    = sys_admin)->decision = Permit

5: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_restricted
   )&(role = sys_admin)->decision = Deny

6: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

7: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

8: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny

9: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Permit

10: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   physician)->decision = Permit

11: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = physician)->decision = Deny

12: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    physician)->decision = Permit

13: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = physician)->decision = Deny

14: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

15: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

16: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    physician)->decision = Permit
```

```
17: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
    &(role = physician)->decision = Deny

18: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
    role = physician)->decision = Deny

19: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
    visitor)->decision = Deny

20: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
    role = visitor)->decision = Permit

21: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
    visitor)->decision = Deny

22: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
    role = visitor)->decision = Deny

23: (resource_id = auth_policy)&(BTG = True)&(resource_group =
    btg_restricted)&(role = visitor)->decision = Deny

24: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
    &(role = visitor)->decision = Deny

25: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
    visitor)->decision = Deny

26: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
    &(role = visitor)->decision = Deny

27: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
    role = visitor)->decision = Permit

28: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role = nurse
    )->decision = Deny

29: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
    role = nurse)->decision = Permit

30: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
    nurse)->decision = Deny

31: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
    role = nurse)->decision = Deny

32: (resource_id = auth_policy)&(BTG = True)&(resource_group =
    btg_restricted)&(role = nurse)->decision = Deny

33: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
    &(role = nurse)->decision = Deny

34: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
    nurse)->decision = Permit
```

```
35: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Deny

36: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Permit
```

Listing B.6: Results of ACPT heuristic 4-way testing: Combined policies

```
## For the combined policies, we assign default dummy values for unused
   attributes.

1: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

2: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   sys_admin)->decision = Permit

3: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
   = sys_admin)->decision = Deny

4: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(role
    = sys_admin)->decision = Permit

5: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Deny

6: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Permit

7: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
   = sys_admin)->decision = Permit

8: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(role
    = sys_admin)->decision = Permit

9: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_restricted
   )&(role = sys_admin)->decision = Deny

10: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = sys_admin)->decision = Deny

11: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

12: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Deny

13: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Deny

14: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    sys_admin)->decision = Deny

15: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny

16: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny
```

```
17: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Permit

18: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Deny

19: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   physician)->decision = Deny

20: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   physician)->decision = Permit

21: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = physician)->decision = Deny

22: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = physician)->decision = Permit

23: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   physician)->decision = Deny

24: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    physician)->decision = Permit

25: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = physician)->decision = Deny

26: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = physician)->decision = Deny

27: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

28: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

29: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

30: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = physician)->decision = Deny

31: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   physician)->decision = Permit

32: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    physician)->decision = Permit

33: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = physician)->decision = Deny

34: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = physician)->decision = Deny
```

```
35: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = physician)->decision = Permit

36: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = physician)->decision = Deny

37: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   visitor)->decision = Deny

38: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   visitor)->decision = Permit

39: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = visitor)->decision = Deny

40: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = visitor)->decision = Permit

41: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   visitor)->decision = Deny

42: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    visitor)->decision = Permit

43: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = visitor)->decision = Deny

44: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = visitor)->decision = Deny

45: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = visitor)->decision = Deny

46: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = visitor)->decision = Deny

47: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = visitor)->decision = Deny

48: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = visitor)->decision = Deny

49: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   visitor)->decision = Deny

50: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    visitor)->decision = Deny

51: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = visitor)->decision = Deny

52: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
```

```
   &(role = visitor)->decision = Deny

53: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Permit

54: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Deny

55: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role = nurse
   )->decision = Deny

56: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   nurse)->decision = Permit

57: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = nurse)->decision = Deny

58: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = nurse)->decision = Permit

59: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   nurse)->decision = Deny

60: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    nurse)->decision = Permit

61: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = nurse)->decision = Deny

62: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = nurse)->decision = Deny

63: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Deny

64: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Deny

65: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Deny

66: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Deny

67: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   nurse)->decision = Permit

68: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    nurse)->decision = Permit

69: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Deny

70: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
```

```
   &(role = nurse)->decision = Deny

71: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Permit

72: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Deny
```

Listing B.7: Results of ACPT heuristic 4-way testing: Combined policies with default deny rules added

```
## The test result for the combined policies
## For the combined policies, we assign default dummy values for unused
   attributes.

1: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Non-applicable

2: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   sys_admin)->decision = Permit

3: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
   = sys_admin)->decision = Deny

4: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(role
    = sys_admin)->decision = Permit

5: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Non-applicable

6: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
   sys_admin)->decision = Permit

7: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
   = sys_admin)->decision = Permit

8: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(role
    = sys_admin)->decision = Permit

9: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_restricted
   )&(role = sys_admin)->decision = Deny

10: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = sys_admin)->decision = Non-applicable

11: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Non-applicable

12: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = sys_admin)->decision = Non-applicable

13: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   sys_admin)->decision = Non-applicable

14: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    sys_admin)->decision = Non-applicable

15: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Deny
```

```
16: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = sys_admin)->decision = Non-applicable

17: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Permit

18: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = sys_admin)->decision = Non-applicable

19: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   physician)->decision = Non-applicable

20: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   physician)->decision = Permit

21: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = physician)->decision = Deny

22: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = physician)->decision = Permit

23: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   physician)->decision = Non-applicable

24: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    physician)->decision = Permit

25: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = physician)->decision = Non-applicable

26: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = physician)->decision = Non-applicable

27: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = physician)->decision = Deny

28: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = physician)->decision = Non-applicable

29: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = physician)->decision = Non-applicable

30: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = physician)->decision = Non-applicable

31: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   physician)->decision = Permit

32: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    physician)->decision = Permit

33: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
```

```
   &(role = physician)->decision = Deny

34: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = physician)->decision = Non-applicable

35: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = physician)->decision = Permit

36: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = physician)->decision = Non-applicable

37: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role =
   visitor)->decision = Non-applicable

38: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   visitor)->decision = Permit

39: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = visitor)->decision = Deny

40: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = visitor)->decision = Permit

41: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   visitor)->decision = Non-applicable

42: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    visitor)->decision = Permit

43: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = visitor)->decision = Non-applicable

44: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = visitor)->decision = Non-applicable

45: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = visitor)->decision = Deny

46: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = visitor)->decision = Non-applicable

47: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = visitor)->decision = Non-applicable

48: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = visitor)->decision = Non-applicable

49: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   visitor)->decision = Non-applicable

50: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    visitor)->decision = Non-applicable
```

```
51: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = visitor)->decision = Deny

52: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = visitor)->decision = Non-applicable

53: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Permit

54: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = visitor)->decision = Non-applicable

55: (resource_id = btg)&(BTG = True)&(resource_group = normal)&(role = nurse
   )->decision = Non-applicable

56: (resource_id = btg)&(BTG = False)&(resource_group = normal)&(role =
   nurse)->decision = Permit

57: (resource_id = btg)&(BTG = True)&(resource_group = btg_restricted)&(role
    = nurse)->decision = Deny

58: (resource_id = btg)&(BTG = False)&(resource_group = btg_restricted)&(
   role = nurse)->decision = Permit

59: (resource_id = btg)&(BTG = True)&(resource_group = btg_allowed)&(role =
   nurse)->decision = Non-applicable

60: (resource_id = btg)&(BTG = False)&(resource_group = btg_allowed)&(role =
    nurse)->decision = Permit

61: (resource_id = auth_policy)&(BTG = True)&(resource_group = normal)&(role
    = nurse)->decision = Non-applicable

62: (resource_id = auth_policy)&(BTG = False)&(resource_group = normal)&(
   role = nurse)->decision = Non-applicable

63: (resource_id = auth_policy)&(BTG = True)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Deny

64: (resource_id = auth_policy)&(BTG = False)&(resource_group =
   btg_restricted)&(role = nurse)->decision = Non-applicable

65: (resource_id = auth_policy)&(BTG = True)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Non-applicable

66: (resource_id = auth_policy)&(BTG = False)&(resource_group = btg_allowed)
   &(role = nurse)->decision = Non-applicable

67: (resource_id = flowRate)&(BTG = True)&(resource_group = normal)&(role =
   nurse)->decision = Permit

68: (resource_id = flowRate)&(BTG = False)&(resource_group = normal)&(role =
    nurse)->decision = Permit
```

```
69: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Deny

70: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_restricted)
   &(role = nurse)->decision = Non-applicable

71: (resource_id = flowRate)&(BTG = True)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Permit

72: (resource_id = flowRate)&(BTG = False)&(resource_group = btg_allowed)&(
   role = nurse)->decision = Non-applicable
```