EAI Endorsed Transactions
on Security and Safety
Research Article EAI.EU

# Controlled BTG: Toward Flexible Emergency Override in Interoperable Medical Systems

Qais Tasali[1], Christine Sublett[2], and Eugene Y. Vasserman[1,*]

[1]Department of Computer Science, Kansas State University, Manhattan, KS 66506 USA {qtasali,eyv}@ksu.edu
[2]Sublett Consulting, San Mateo, CA 94402 USA csublett@sublettconsulting.com

## Abstract

**INTRODUCTION:** In medical cyber-physical systems (mCPS), availability must be prioritized over other security properties, making it challenging to craft least-privilege authorization policies which preserve patient safety and confidentiality even during emergency situations. For example, unauthorized access to device(s) connected to a patient or an app controlling these devices could result in patient harm. 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.

**OBJECTIVES:** Medical BTG has largely been treated as an all-or-nothing scenario: either a means to obtain unrestricted access is provided, or BTG is not supported. We show how to handle BTG natively within the ABAC model, maintaining full compatibility with existing access control frameworks, putting BTG in the policy domain rather than requiring framework modifications. This approach also makes BTG more flexible, allowing for fine-grained facility-specific policies, and even automates auditing in many situations, while maintaining the principle of least-privilege.

**METHODS:** We do this by constructing a BTG "meta-policy" which works with existing access control policies by explicitly allowing override when requested.

**RESULTS:** We present a sample BTG policy and formally verify that the resulting combined set of access control policies correctly satisfies the goals of the original policy set and allows expanded access during a BTG event. We show how to use the same verification methods to check new policies, easing the process of crafting least-privilege policies.

Received on 21 December 2019; accepted on 18 February 2020; published on 19 February 2020

**Keywords:** Break the Glass, Access control, Authorization, Medical IoT, CPS, XACML, ALFA



doi:10.4108/eai.13-7-2018.163213

## 1. Introduction

Non-safety-critical systems prioritize confidentiality over availability, usually expressed in system authorization policies as a "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. 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 [1–3]. This seemingly impossible situation is encountered in medical facilities all too regularly. As a result, defining an authorization

*Corresponding author. Email: eyv@ksu.edu

policy that can follow the principle of least-privilege as closely as possible without compromising patient safety or confidentiality 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 [4], 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 automated external defibrillators (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 (EHRs). After a "BTG event" completes, the actions of all concerned parties are generally audited, requiring detailed logging (currently performed post-hoc and manually) of what happens during BTG. Post-hoc auditing s used to determine the reasons and **legitimacy** for the override.

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. 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.

Popular access control models such as Role-Based Access Control (RBAC) [5] and Attribute-Based Access Control (ABAC) [6, 7] (also referred to as PBAC – policy-based access control) are the dominant models used in medical domain and excel in protecting data in a closed environment, where all resources (objects) and users (subjects) are known. RBAC is based on user identity (static roles assigned to users) and lacks flexibility and dynamic access control capabilities. For example, a user is only allowed access to a resource if it is included in the user's pre-assigned role(s). An RBAC role is usually a static organization position, but the medical domain requires decisions to be based on both static and dynamic information such as physical location, device status, clinician's relationship to patient, etc. ABAC allows for a more flexible and dynamic access control and is used mostly for information sharing in large enterprises where access control decision are based on evaluation of access control request against predefined characteristics (attributes) of user, resource, action and environment. Attributes in ABAC can be static (e.g. name) and dynamic (e.g. working shifts). However, ABAC also falls 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 (e.g. medical emergencies) prior to issuing a "deny" or "allow" decision.

In this work, 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. In previous BTG work (and in real-world deployments), overrides must be invoked for every instance of emergency access to every different device or health record. We 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. In this work, BTG will last until a clinician[1] explicitly signals the end of the event.

We base our work in the context of increasingly prevalent but difficult to secure interoperable medical systems, which enable intuitive "plug-and-play" functionality (e.g. the Integrated Clinical Environment (ICE) [8]), and can significantly ease the burden of medical product integration and testing. Policy-based access control provides a flexible solution to the problem of managing this complex security scenario, and allows us to tackle a wider problem of not only access to information such as EHRs, but also to functions of individual sensing and treatment devices which may allow different levels of access and/or control based on user identities.

Fine-grained access control requires increasingly sophisticated BTG policies to maintain patient safety. However, increasing policy sophistication brings with it the risk of unintended consequences (e.g. insufficient permission or excessive permission based on the situation). We therefore focus on constructing a BTG "meta-policy" which works with existing access control policies by explicitly allowing override when requested, with integrated verification functions to avoid unexpected (emergent) effects of multi-policy interaction in complex environments. We test our policies for inconsistencies and incompleteness, and verify the access control model is expressed correctly.

The core contributions of this work include:

[1] Any authorized caregiver including physician, nurse, technician, etc.

- Description, implementation, and verification of a new flexible medical "Break the Glass" access control model based on ABAC which maintains compatibility with existing access control frameworks.

- Comparison of our solution to a currently-implemented Break the Glass for Electronic Health Records used by one of the largest health-care providers in the United States, showing that our BTG system is strictly more flexible and expressive.

- Demonstration of 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.

Implementation and performance testing of the system is out of scope and left as a subject of future work.

## 2. Background

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 virtual device [9]. 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 [3].

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 [10]. Their increased power requires greater assurance that they will not be misused (intentionally or unintentionally) to harm the patient(s) they are treating [1, 3, 8]. MAPs also present novel problems in terms of privacy and security. Each device and application within a MAP may require different levels of network 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.

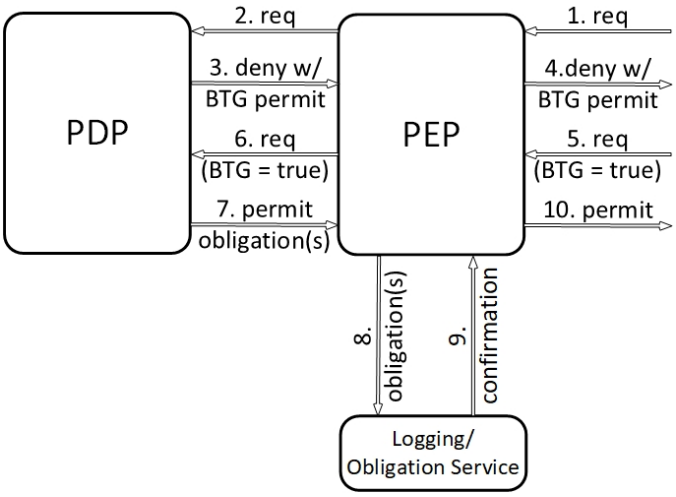


**Figure 1.** BTG request workflow. PDP is the policy decision point, and PEP is the policy enforcement point. Parentheses indicate the state of the system through condition variables, and steps 3, 4, 7, and 10 denote returned decisions.

### 2.1. Access Control

Access control policies for users, devices, and applications can be formally expressed in a policy language, such as the pervasive XML-based eXtensible Access Control Markup Language (XACML) [11]. Such a formally-written policy can be automatically interpreted and enforced in real time, reducing the burden of auditing, as the system will not permit policy-violating actions. Break the Glass must be invoked if a temporary policy violation is needed to care for a patient, and such events are later audited.

XACML is one of the most widely used policy languages. It defines a fine-grained attribute-based access control policy, and an architecture and processing model to describe evaluation of access requests. The main components of a XACML reference architecture [11] are: the Policy Decision Point (PDP), the Policy Enforcement Point (PEP), the Policy Access/Administration Point (PAP), and the Policy Information Point (PIP) and context handler. A subset of the components are shown in Figure 1. Numbering in the diagram represents the order of operations. A clinician accessing a resource will need to be authenticated and authorized. The access request is sent to the context handler in its native format along with required session information (attributes). The context handler generates a request context and forwards it to the PDP, which queries the PIP for any additional attributes needed for evaluation. PDP proceeds with policy evaluation and returns the decision along with any obligations to the context handler, which in turn translates the decision into a PEP-acceptable format. The final decision and obligations are forwarded to PEP for enforcement. (**Obligations** are

constraints – required actions on which the decision is contingent – and are enforced by the PEP.)

XACML provides different levels of access, a policy language, and a query language. When a query (access request) is evaluated, the returned result can be one of "allow", "deny", "indeterminate", or "not applicable". **Indeterminate** means there is an error in the query, and **not applicable** means no policy was located with a target that could match the information in the request. We take advantage of the different levels of access, and the flexibility of policy decision and policy enforcement points to design BTG for real-time interaction with dynamic, plug-and-play interoperable medical devices, systems, and applications.

**Access Control Override.** Controlling access to resources is the main goal of an authorization policy. Traditional access control mechanisms prevent misuse of information by restricting users' access to that which 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 restrictive authorization policy gives users unnecessary access and defeats the purpose of least privilege, while a more restrictive policy than needed could stop users from fulfilling their tasks. Healthcare is one application domain where restricting access by default only to authorized users is not always the best and safest solution. As discussed in the Health Insurance Portability and Accountability Act (HIPAA) [12, 13], systems should allow access during medical emergencies – access control override is considered a safer authorization approach over a strict fail-closed system.

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. While an access control override mechanism (BTG) allows for life-saving care in unexpected situations, it also leaves the system open to misuse, such as accessing patient's private records or even changing the dosages of life-sustaining medications. By performing a post-hoc audit, facilities determine the reasons (legitimacy) for overriding access control.

## 2.2. Related Work

In one of the earliest papers on real-time (time-of-emergency) Break the Glass (BTG), Povey [14] discusses 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 [15] suggest a mechanism for increased flexibility in access control by overriding denied access (when necessary) using the possibility-with-override concept. Additionally, auditing of overrides using the access control policy is also recommended.

Ferreira et al. [16] propose a BTG policy within an implemented access control policy (defined by healthcare professionals) and access control hybrid model for a hospital. The implementation allows for the start of BTG, but does not have a method to exit the emergency state. In follow-up work, Ferreira et al. [17] apply BTG concepts to the NIST/ANSI RBAC model and name it the BTG-RBAC model. 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. 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. This more complete model (compared to [16]) 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. In contrast to BTG as an alternative returned decision (vs. "allow" or "deny"), we define BTG in terms of states in which the system is operating. Instead of overriding "deny" decisions on a one-by-one basis, such decisions are automatically overridden during BTG events.

Brucker and Petritsch [18] demonstrate a BTG model that allows for access control override 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. (Emergency level- and permission-specific policies are prone to errors such as inconsistencies, insufficient or excessive permissions, etc.)

Nazerian et al. [19] extend RBAC into an Emergency RBAC (E-RBAC) model. The system has one of three states representing normal, emergency, and exception situations. The normal status is similar to RBAC with no access control override, whereas in emergency and expiation situations a user can override access control

only if the user has been preassigned an appropriate trust label (M for emergency and/or H for exception). Exception is an undefined emergency, similar to our "uncontrolled BTG", and requires an administrative role to assign or revoke permissions to/from user active (normal) role. The authors use Alloy to verify whether users have the right trust levels and if the newly assigned or modified roles meet static separation of duty constraints. Helal et al. [20] borrow from database concepts to propose Isolated enabled-RBAC (I-RBAC), in which an isolated environment with a copy of the original resources such as patient records is created for use in situations where a user wants to access a resource for which they do not have access permission. Any modified data will be updated in the "live" system (the origin source of data) after a security check performed on accumulated modifications by authorized user at a later time. This concept is similar in principle to our "uncontrolled BTG" followed by a manual audit. Both approaches suffer from the inflexibility of RBAC (instead of ABAC), and are limited to static resources (as opposed to attributes) and lack real-time evaluation of conditions at access time. These approaches are also meant as augmentations to existing access control framework, and lack the flexibility and incremental deployment possibilities of our meta-policy approach.

In an orthogonal piece of related work, Tasali, Chowdhury, and Vasserman [2] propose an ABAC-based authorization architecture for systems of interoperable medical devices. They implement and evaluate a proof-of-concept system within the Medical Device Coordination Framework [1], and partially address the lack of compatibility between current access control models, BTG, and plug-and-play dynamic medical systems composed of heterogeneous devices. Their work is only a partial solution to BTG, and we use it as a starting point for our work.

Prior work does not address the situation wherein the authorization system may fail to fulfill the returned *obligations* (constraints returned with a decision and enforced by the PEP) accompanying a decision from a BTG request, and it is therefore unclear whether a BTG request would be allowed or denied at that time [17, 18]. 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 emergency access even if PEP fails to enforce the obligations. Therefore, our access control model allows access **even if the obligations are not met**. We handle *unmet obligations* by forcing the system into an alternate BTG state which we call **uncontrolled BTG**, which increases the auditing requirements on actions taken during a BTG session, up to a potential full manual audit. The inclusion of uncontrolled BTG is meant to ensure availability even under resource (e.g. bandwidth, processing, etc.) constraints.

## 3. Design

Access control override is normally handled on a field-by-field basis in electronic health record (EHR) 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 [2] is used as a starting point. Our BTG granularity is per-patient (although the system does support multi-patient deployments), as the authorization system we describe is meant for dynamic systems of medical devices attached to a single 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 devices or EHR (even to different fields) 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.

We implement as-needed access control override by forcing decision reevaluation in the context of the override. For any "deny" decision returned by the PDP, the clinician may choose to override (effectively declaring an emergency), 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 1. In contrast to the first request, PDP evaluates the override against a BTG policy. System status and user session attributes including resource and action are used to determine what policies are used for evaluation by PDP.

Other than the change from normal to BTG policy used for the decision, PDP behaves in a similar manner to the previous request. However, the PEP logic is modified from [2] to allow user to request an access control override (e.g. initiate an emergency session). A returned decision may include obligations that PEP will enforce prior to allowing access. PEP delegates obligation fulfillment to an external entity we call "Obligation Service" and will only return a decision after a confirmation is received from Obligation Service, as shown in Figure 1. The Obligation Service ensures all obligations, including logging requirements, are fulfilled. It returns true if an obligation was successfully fulfilled, otherwise it returns false. The final decision to be enforced by PEP is dependant on the outcome of Obligation Service evaluation of obligations as well as the system operating state, further discussed below.

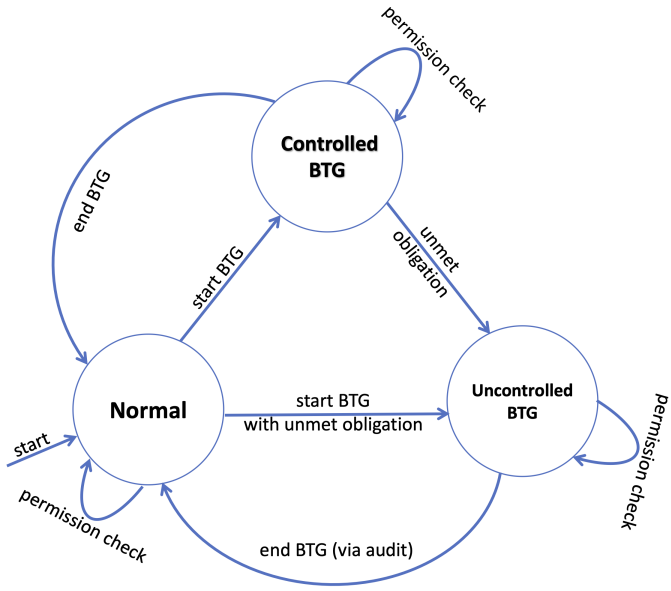


**Figure 2.** BTG state machine showing the normal state and two emergency (BTG) states, along with transition criteria. The system starts in "Normal" and moves into "Controlled BTG" when BTG is activated, as long all BTG obligations are met (otherwise the system transitions to "Uncontrolled BTG"). Normal state can be restored via automated or semi-automated audit from Controlled BTG, or via manual audit from Uncontrolled BTG.

## 3.1. Operating States

Our model has three system states – a normal operating state and two emergency (BTG) states. Transitions between states are controlled through evaluation of system-wide obligations. The BTG state machine for our system is diagrammed in Figure 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 an RBAC/ABAC resource or a system state variable. Therefore, the access control framework does not need to periodically reevaluate whether or not BTG is in effect. Instead, accessing the BTG resource triggers a permission check, allowing the system to make the transition at that time. For the moment, the system is designed to only exist a BTG state when explicitly requested 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.

**Normal State.** This is the initial state for the 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 "allow", "deny", "not applicable", or "indeterminate". ("Not applicable" means the PDP could not locate a policy that matches the request, and an "indeterminate" decision means an error was encountered during policy evaluation.) An "allow" 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**. Non-BTG obligations are evaluated every permission check, whereas BTG obligations are evaluated every system state change.

**Controlled BTG State.** The authorization system changes its state from normal to controlled BTG whenever the user overrides an access control decision by initiating a BTG session. In order for the system to change its state to controlled BTG, all returned BTG obligations need to be met. Obligations are facility- and policy-specific, and we expect them to include at least e.g. a requirement for more detailed logging prior to allowing a BTG session. The BTG obligations are meant to help track the clinicians' actions for audit and system rollback purposes, and must include sufficient detail.[2] A system in a controlled BTG state returns to a 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.

**Uncontrolled BTG State.** As shown in the state diagram in Figure 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 BTG obligations (returned when BTG was started), 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 returned BTG obligations, e.g. there is insufficient available bandwidth to ensure all activities are logged at a higher-than-normal level of detail. The system in an uncontrolled BTG state meets the "fail-open" requirement of the medical systems. 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

[2]Determining the specific details that satisfy this "sufficiency" condition are facility-dependent, and the subject of future work.

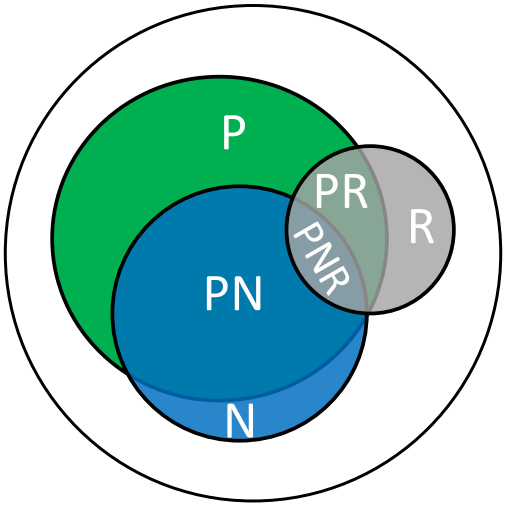


**Figure 3.** Permissions grouped by access: **P**, **N**, and **R** are groups of permissions assigned to **P**hysician, **N**urse, and BTG **R**estricted, respectively.

the system evaluates each user access request against authorization policies and for any decision other than "allow", access control is overridden. In addition, any returned non-BTG obligation is fulfilled on a best-effort basis. Although the system is in an uncontrolled state, it is still governed by the authorization rules – **the policies are never ignored, only overridden**.

**"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, certain patient information (e.g. VIP status) and certain other critical system resources (which may vary between facilities and explicitly specified in the BTG policy). A real-world example of a BTG-restricted resource is provided in Section 4. 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 (see Section 3.3). 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 "allow" decision, however, would only apply to authorized users and cannot be overridden, so access control policies cannot be altered at all during BTG.**

## 3.2. BTG Policy Evaluation

An effective BTG policy is dependent on proper identification of resources at the time of policy evaluation. Thus, resources need to be identified and categorized into their relevant access groups (sets), which are then stored in a database. Attributes including resource group information are retrieved from a policy information point (PIP) at the time of policy evaluation. To implement the concept of "BTG-restricted" permissions as discussed in Section 3.1, we define a 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. Resource assignment into groups is not fixed and may change like other dynamic attributes (e.g. conditional authorization based on the current time, and the clinician's assigned work hours). Therefore, 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.

Dynamically assigning resources in groups is similar to whitelisting and blacklisting. We approach resource identification from a more flexible perspective to allow for **controlled BTG** access. We identify resources that are available to a group of users as the "normal" set (whitelist), and resources that should be not available for access during unexpected situations (BTG) as BTG-restricted resources (blacklist). Resources that are otherwise not accessible to a user, but are not part of the blacklisted set, can be accessed during BTG.

Figure 3 shows a diagram of how resources (grouped by access) work within our architecture. The outer circle represents the set of all available resources available in a facility (a superset of resources accessible by our example subjects). The circles marked with letters **P** (physician) and **N** (nurse) represent the set of resources that can be accessed by physicians and nurses, respectively. (As Nurse and Physician are chosen arbitrarily, the example is representative of any two roles in an organization.) 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 3.1). **NR** is omitted for simplicity. 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 permissions can be overridden. If these

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

**Figure 4.** Sample access control table for a physician (the "Subject" column is omitted, as it is always "physician")

sets happen to be not empty, access is denied by default as we do not see any practical reason for allowing access otherwise. Even so, the policy enforcement point (PEP) can be customized to allow rather than deny.

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 3.1). One may argue that "indeterminate" decisions should be overridden to "allow" in a BTG context. This is a design decision, and is explored further in Section 5.3.

Figure 4 is a tabular view of access control requests and decisions for a physician accessing resources categorized as in Figure 3. For ease of illustration, we assume that all obligations are fulfilled by the PEP; the table would be rendered unreadable otherwise.

- In row 1 - 2, the physician is allowed access to *P* and *PN* regardless of the current state of the system since *P* and *PN* are the sets of resources explicitly allowed for 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 and 7, the physician request for *PR* and *PNR* is permitted only if the authorization system is in normal state.
- In row 6 and 8, access to *PR* and *PNR* is disallowed during BTG since *PR* and *PNR* are part of the BTG-restricted set.
- 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 3, meaning permission to access them has not been explicitly granted, and therefore the request is denied in normal mode.
- Finally, during BTG, access is allowed in row 11 since those resources are not within *R*. Contrast this to row 10 where BTG is not active and the resources are not explicitly permitted in a policy.

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, because any access to **R** would be unconditionally denied during BTG – this prevents inconsistencies or emergent properties in policy combinations from creating loopholes where in some cases, access to **R** is allowed during BTG.

## 3.3. BTG Policy Specification

Figure 5 shows an example policy, written in the Abbreviated Language For Authorization (ALFA) [21], that contains rules for normal, controlled, and uncontrolled BTG access. Resources in "BTG-restricted" are protected set from **any** access during a BTG session. 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 three 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 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

```
namespace org.facility {

  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 {
        //obligations
      }
    }
  }

  policy polSetBtgFlag {
    target clause resource.
         resourceId == "btg"
    apply permitOverrides
    allowBtg //invokes the rule
          below
  }

  rule allowBtg {
  //start BTG session if not
       in BTG already
    condition EMG.BTG == false
    permit
    on permit {
      obligation btgAudit {
        //obligations
      }
    }
  }

  rule allowSettingFlowRate {
    target clause user.role ==
         "nurse"
      or user.role == "
           physician"
    condition integerIsIn(
         integerOneAndOnly(
         user.userId),
      patient.
           assignedClinicianId)
    permit
    on permit {
      obligation log {
        //obligations
      }
    }
  }

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

  rule allowModifyAuthPolicy {
    target clause user.role ==
         "sysadmin"
    condition EMG.BTG == false
    permit
    on permit {
      obligation log {
        //obligations
      }
    }
  }
  policy polModifyAuthPolicy {
    target clause resource.
         resourceId == "auth
         -policy"
      clause action.actionId ==
           "write"
      or action.actionId == "
           read"
    apply permitOverrides
    allowModifyAuthPolicy
  }

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

**Figure 5.** Example BTG policy written in ALFA, reformatted for readability

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 as long as any of the two rules returns an allow decision. This is ensured by the rule combining algorithm *permitOverrides*, specified within the policy `polSetFlowRate`. Combining algorithms are specified in policy sets (referred to as policy combining algorithms) and policies (referred to as rule combing algorithm) to avoid potential conflicts within policies and rules, respectively. For example, given the results of evaluating a set of rules (within a policy) the policy combining algorithm *permitOverrides* ensures that a combined permit decision will be returned if there is at least one rule providing a permit decision. The sequence is similar for the policy `polModifyAuthPolicy`.

The policy `polSetBtgFlag` controls access to BTG. Its target clause requires the resource to be set to (`btg`) and the policy consists 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 an allow 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 5 are kept simple on purpose. They provide a "generic" starting point for more complex and lengthy policies. They are formally verified in Section 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 shown are meant to scale up and test policies which are too large to interplay and too complex to be done ad-hoc.

## 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 largest US health care groups. We studied their procedure for overriding access control from a large U.S. medical provider. 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 6 provides an emergency access control (BTG) matrix based on our analysis of BTG functionality in the health system. Figure 7 shows a policy within our framework that satisfies the BTG requirements in Figure 6.

| | Patient Type | | |
|---|---|---|---|
| **Clinician Type** | *Mental Health* | *Confidential* | *42 CFR* |
| **Emergency** | allow | allow | allow |
| **Mental Health** | allow | allow | allow |
| **PCPs** | allow | allow | **deny** |

**Figure 6.** Sample access control matrix based on patient and clinician types, indicating when BTG access is allowed and when it is denied

The health system uses Epic[3] 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 physicians (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 clinicians who are part of the patient's care team, or patients whom the clinician has seen within the last 30 days or scheduled to see within the following 30 days.

Figure 7 shows that the medical group'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 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 5 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 the medical group.

[3] https://epic.com/

```
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 7.** Sample BTG policy from a major medical group, written in ALFA and reformatted for readability

## 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

| Tool (col.) / Feature support (row) | Custom Attribute Types | Relational Rules | Static Verification | Dynamic Verification | t-way Testing | XACML Support |
|---|---|---|---|---|---|---|
| Alloy [22] | ○ | - | ● | - | - | ● |
| ACPT [23] | ● | ○ | ● | ● | ● | ● |
| Margrave [24] | - | ○ | ● | - | - | ◐ |
| SPIN [25] | - | - | ● | - | - | - |
| ACCOn [26] | - | - | ● | - | - | - |

**Figure 8.** A brief comparison of the model checking tools we considered. A more complete treatment can be found in the work of Aqib and Shaikh [27].

simultaneous grasp of the richness and constraints of the language 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 "allow" while the other returns "deny".

## 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. Many were either not compatible with our model or missing 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 simply helpful.

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 [28]. 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 5 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 [29].

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.

Our initial 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 5. 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 8. We refer the readers to Aqib and Shaikh [27] for a detailed survey of verification and validation tools and approaches for access control policies.

The Access Control Policy Testing (ACPT) tool [23, 30], 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 the Symbolic Model

Verification (SMV) model checker [31] for property checking and Automated Combinatorial Testing for Software (ACTS) [32] for test suite generation. In addition, ACPT allows for policies to be either merged or combined prior to verification or testing.

## 5.2. Checking Policy Consistency

The verification of safety requirements stated as properties can assure only the logical integrity of the policy rules against the specific safety requirements. This is a heuristic approach – although we use model checking, the model cannot provide complete coverage because of temporal logic within the policy, and so it may not cover all possible values of all rules (or all conditions in the rules).

Figure 9 shows the list of requirements for our authorization policy (Figure 5) in the form of high-level security properties along with the status for each of the properties returned by the tool after verifying it. The result confirms that all properties hold.[4] A policy is shown correct by checking a set of properties that the system should satisfy against. We verify our model against the below specified properties:

- Request to update or view *flowRate* by *physician* or *nurse* should always be granted when *BTG* is set to true, as shown in lines 1 and 2. *flowRate* is a medical resource which belongs to group BTG-allow and should be granted access if a clinician requests it during an active BTG session. These properties verify the rule *allowEmergencyAccess* from the policy set in Figure 5.

- The property in line 3 ensures that *physician* should never be granted access to "BTG-restricted" resources when *BTG* is set to true. This property particularly verifies the condition in the rule *allowEmergencyAccess*, which restricts emergency access (BTG) to resources that belong to "BTG-restricted" group. Line 4 further generalizes this check by replacing *physician* with *anyone*. A resource-specific version of this property (denying access for *physician* to modify *auth-policy* – a "BTG-restricted" resource – while BTG is enabled) is given in line 5.

- Line 6 checks that *sys-admin* should be allowed to update *auth-policy* when the system is not in *BTG* state. Line 7, on the other hand, checks that if BTG is active, then the same request should be denied. Only *sys-admin* is allowed to update or view *auth-policy* and since *auth-policy* belongs to "BTG-restricted" group, requesting access to it by *sys-admin* should be granted only if the system is in normal state. These properties verify the rule *allowModifyAuthPolicy*.

- The property in line 8 is used to verify that a request for setting *btg* (e.g. BTG access) flag by anyone should be granted unless the system is already in BTG state. This requirement is defined in the rule *allowBtg*.

- The property in line 9 verifies the BTG requirement in the rule *allowEmergencyAccess* – anyone with role *visitor*[5] requesting *write* access to *flowRate* should be granted if *BTG* is set to true and the resource *flowRate* belongs to *BTG-allow* group for the user. Recall the rule *allowEmergencyAccess* does not require a specific role (e.g. see physician and nurse in lines 1 and 2, respectively).

```
1 spec AG (((((role = "physician" & resource = "flowRate") & group = "BTG
      -allow") & DefaultAction = "write") & BTG = "True") -> decision =
      Permit) is true
2 spec AG ((((role = "nurse" & resource = "flowRate") & group = "BTG
      -allow") & BTG = "True") -> decision = Permit) is true
3 spec AG (((role = "physician" & group = "BTG-restricted") & BTG = "True"
      ) -> decision = Deny) is true
4 spec AG ((group = "BTG-restricted" & BTG = "True") -> decision = Deny)
      is true
5 spec AG (((((role = "physician" & resource = "auth_policy") & group = "
      BTG-restricted") & DefaultAction = "write") & BTG = "True") ->
      decision = Deny) is true
6 spec AG (((((role = "sys_admin" & resource = "auth_policy") & group = "
      BTG-restricted") & DefaultAction = "write") & BTG = "False") ->
      decision = Permit) is true
7 spec AG (((((role = "sys_admin" & resource = "auth_policy") & group = "
      BTG-restricted") & DefaultAction = "write") & BTG = "True") ->
      decision = Deny) is true
8 spec AG (((resource = "btg" & group = "normal") & BTG = "False") ->
      decision = Permit) is true
9 spec AG (((((role = " visitor " & resource = "flowRate") & group = "BTG
      -allow") & DefaultAction = "write") & BTG = "True") -> decision =
      Permit) is true
```

**Figure 9.** Results of ACPT verification of policy consistency **as specified by the facility**, 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.

## 5.3. Testing Results

We use ACPT to validate a **slightly modified version** of the BTG policy shown in Figure 5, and verify that our model meets safety requirements for emergency override. For example, the safety requirement in line 4 of Figure 9 (expressed in temporal logic) formalizes the rule `allowEmergencyAccess` in the example BTG policy in Figure 5, and is checked for any violations verified using the model checker.

[4] 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.

[5] A visitor could be a visiting doctor from a different healthcare facility who is holding temporary security credentials.

```
1  spec AG (((((role = "nurse" & resource = "flowRate") & group = "BTG
       -allow") & DefaultAction = "write") & BTG = "True") -> decision =
       Permit) is false as demonstrated by the following execution
       sequence:
2  Trace Description: Counterexample-
3       > ABAC_ polSetBtgFlag.decision = Deny-
4       > ABAC_ polSetFlowRate.decision = Permit
```

**Figure 10.** Policy inconsistency shown with a counterexample, reformatted for readability

```
1  (resource="btg")&(BTG="False")&(role="sys_admin")&(group="BTG
       -allow")->Permit
2  (resource="auth_policy")&(BTG="True")&(role="sys_admin")&(group
       ="normal")->Deny
3  (resource="flowRate")&(BTG="False")&(role="sys_admin")&(group="
       BTG-restricted")->Deny
4  (resource="btg")&(BTG="False")&(role="physician")&(group="
       normal")->Permit
5  (resource="auth_policy")&(BTG="True")&(role="physician")&(group
       ="BTG-restricted")->Deny
6  (resource="flowRate")&(BTG="True")&(role="physician")&(group="
       BTG-allow")->Permit
7  (resource="btg")&(BTG="True")&(role="visitor")&(group="BTG
       -restricted")->Deny
8  (resource="auth_policy")&(BTG="False")&(role="visitor")&(group=
       "BTG-allow")->Deny
9  (resource="flowRate")&(BTG="False")&(role="visitor")&(group="
       normal")->Deny
10 (resource="btg")&(BTG="False")&(role="nurse")&(group="BTG-allow
       ")->Permit
11 (resource="flowRate")&(BTG="True")&(role="nurse")&(group="BTG
       -restricted")->Deny
12 (resource="auth_policy")&(BTG="True")&(role="nurse")&(group="
       normal")->Deny
```

**Figure 11.** Results of ACPT's auto-generated heuristic testing, reformatted for readability and with metadata header removed

The reason for verifying a slightly modified policy in the previous paragraph can now be revealed: Figure 10 shows **faults in the seemingly trivial policy**. The issues are **inconsistency** and **incompleteness**. The `authPolicySet` was initially defined with the `denyOverrides` policy combining algorithm, which favors "deny" decisions, and was considered the safest for our example. During verification, however, the property on line 9 from Figure 9 returned false with a counterexample demonstrating **incompleteness** via a "not applicable" decision, since no matching rule was found. To resolve this, we added default deny rules to the polices to ensure that in cases where there is not a policy matching an access request, a deny decision is still enforced. This resulted in other properties failing due to conflicting decisions (**inconsistency**). For example, the property on line 2 ensures that a request for a resource in the "btg-allow" group should always be allowed provided that the system is in a BTG state (recall this property is defined in `allowEmergencyAccess` rule in Figure 5). During verification, `polSetFlowRate` returned "allow" and `polSetBtgFlag` returned "deny", resulting in a "deny" decision being returned by the combined policy set (`authPolicySet`) because of the combining algorithm `denyOverrides`, as shown in Figure 10. Replacing `denyOverrides` with `permitOverrides` resolved the inconsistency and the final verification result is shown in Figure 9. Conflicting and missing rules are hard to detect without a policy verification and evaluation engine, i.e. a clinician or policymaker may not foresee the consequences of a policy combination until an unexpected result occurs (likely during a medical procedure), and it may be difficult to determine the reason for the unexpected result after-the-fact, making the policies difficult to fix once deployed.

When a new resource is added to the list of available resources, it should be "flagged" for administrative review, and an access control policy for the identified resource should be written. Normally a resource without a written access control policy is considered non-existing and the policy decision point (PDP) will return an indeterminate or not applicable result if the resource is requested for access. In our design and implementation of a sample BTG policy we added default deny rules to avoid not applicable and indeterminate results in situations such as those. Also, if a well written access control policy (e.g. meta-policy) is written for resources such as medical devices or apps that share some properties, then a new policy is not necessary to be written. For example, by adding a new medical device that is either part of a new setup with identical configurations to an existing device or a replacement for an existing device in the system there needs to be no further actions to be taken except for approving the installation by an administrator.

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. "Mis-assinging" 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 [33, 34] to check for misassigned resources. 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.

In Figure 11, we perform conformance testing to validate compliance of implementation of the policy model. The tool automatically generates test cases from the model using combinatorial array generation technology [32], evaluates test requests against the policy, and returns the testing results – which can be used for identifying conflicts, inconsistencies, and other type of faults in the given policies.

Properties like those in Figure 9 can be written for an identified fault and verified using ACPT. In Figure 11 each of the test cases generated for the BTG policy in Figure 5 consists of certain attributes (resource, role, group, BTG) and a final decision:

- Users should be granted access to the resource *btg* (e.g. initiating a BTG session) if the BTG flag is set to false, confirmed by the test cases 1, 4, 7, and 10.
- A user should not be granted permission to access the resource *auth_policy* when the system is in a BTG state, validated by test cases 2, 5, and 12.
- Similarly, test cases 5, 7, and 11 validate that access to any resource in the group *BTG-restricted* will be denied during BTG.
- Test cases 3, 8, and 9 are used to validate access restrictions during the normal operating state (e.g. non-clinicians – system administrators – should not be able to set medication dispensing rates).
- Finally, in test case 6, we check that a user will be granted access to any resource in group *BTG-allow*, other than *btg* and *auth_policy*, only if the system is in a BTG state.[6]

Note that **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. Due to space limitations, we only report on 2-way testing results.

## 6. Conclusion, Limitations, and Future Work

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 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. Finally, we formally show that our example policy is expressed correctly and the policy specifications are satisfied.

While our work makes some headway towards flexible emergency access (BTG) within existing authorization frameworks, 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 sharing data on their facility-specific policies regarding access control override mechanisms, authorization policies, and log data, we could only achieve a limited understanding of how our proposed model would fit in a real healthcare setting. An in-depth qualitative case study of emergency access control override mechanisms from hospital settings is a subject for future work. Another potential direction for future research could include log auditing to determine if by 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. Therefore, implementation and performance testing of such a system is included in future work plans as well.

[6] Recall the policy `polSetBtgFlag` in 5 returns deny only if the BTG flag is set to True. In all other cases it returns permit regardless of what group the resource *btg* belongs to or what the role of requesting user is. Similarly, the policy `polModifyAuthPolicy`, which defines access control policy for authorization policies, returns permit only if the requesting user is a system administrator and the system is normal state. Otherwise, it returns deny.

## References


[1] King, A.L., Procter, S., Andresen, D., Hatcliff, J., Warren, S., Spees, W., Jetley, R.P. *et al.* (2009) An open test bed for medical device integration and coordination. In *International Conference on Software Engineering (ICSE) Companion*.

[2] Tasali, Q., Chowdhury, C. and Vasserman, E.Y. (2017) A flexible authorization architecture for systems of interoperable medical devices. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*.

[3] King, A., Arney, D., Lee, I., Sokolsky, O., Hatcliff, J. and Procter, S. (2010) Prototyping closed loop physiologic control with the medical device coordination framework. In *ICSE Workshop on Software Engineering in Health Care (SEHC)*.

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

[5] Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R. and Chandramouli, R. (2001) Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)* **4**(3).

[6] Hu, V.C., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R. and Scarfone, K. (2014), Guide to attribute based access control (ABAC) definition and considerations, NIST Special Publication 800-162.

[7] Hu, V.C., Kuhn, D.R. and Ferraiolo, D.F. (2015) Attribute-based access control. *IEEE Computer* **48**(2).

[8] (2008), Medical Devices and Medical Systems-Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE)-Part 1: General requirements and conceptual model, ASTM F2761.

[9] Hatcliff, J., King, A., Lee, I., MacDonald, A., Fernando, A., Robkin, M., Vasserman, E.Y. *et al.* (2012) Rationale and architecture principles for medical application platforms. In *International Conference on Cyber-Physical Systems (ICCPS)*.

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

[11] OASIS (2013), 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.

[12] (2013), 45 CFR 164.312 - Technical safeguards. URL https://www.law.cornell.edu/cfr/text/45/164.312.

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

[14] Povey, D. (1999) Optimistic security: A new access control paradigm. In *New Security Paradigms Workshop (NSPW)*.

[15] Rissanen, E., Firozabadi, B.S. and Sergot, M. (2004) Towards a mechanism for discretionary overriding of access control. In *International Workshop on Security Protocols (SPW)*.

[16] Ferreira, A., Cruz-Correia, R., Antunes, L., Farinha, P., Oliveira-Palhares, E., Chadwick, D.W. and Costa-Pereira, A. (2006) How to break access control in a controlled manner. In *IEEE International Symposium on Computer-Based Medical Systems (CBMS)*.

[17] Ferreira, A., Chadwick, D., Farinha, P., Correia, R., Zao, G., Chilro, R. and Antunes, L. (2009) How to securely break into RBAC: The BTG-RBAC model. In *Annual Computer Security Applications Conference (ACSAC)*.

[18] Brucker, A.D. and Petritsch, H. (2009) Extending access control models with break-glass. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*.

[19] Nazerian, F., Motameni, H. and Nematzadeh, H. (2019) Emergency role-based access control (E-RBAC) and analysis of model specifications with Alloy. *Journal of information security and applications* **45**.

[20] Helal, M.R. (2017) *Efficient Isolation Enabled Role-Based Access Control for Database Systems*. Ph.D. thesis, University of Toledo.

[21] Axiomatics (2015), Axiomatics language for authorization (ALFA), https://www.axiomatics.com/solutions/products/authorization-for-applications/developer-tools-and-apis/192-axiomatics-language-for-authorization-alfa.html. (Accessed on 2/21/2017).

[22] Mankai, M. and Logrippo, L. (2005) Access control policies: Modeling and validation. In *NOTERE Conference (Nouvelles Technologies de la Répartition)*.

[23] Hwang, J., Xie, T., Hu, V. and Altunay, M. (2010) ACPT: A tool for modeling and verifying access control policies. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*.

[24] Fisler, K., Krishnamurthi, S., Meyerovich, L.A. and Tschantz, M.C. (2005) Verification and change-impact analysis of access-control policies. In *Proceedings of the International conference on Software engineering (ICSE)*.

[25] Ma, J., Zhang, D., Xu, G. and Yang, Y. (2010) Model checking based security policy verification and validation. In *International Workshop on Intelligent Systems and Applications*.

[26] Bravo, L., Cheney, J. and Fundulaki, I. (2008) Accon: checking consistency of xml write-access control policies. In *Proceedings of the International conference on Extending database technology: Advances in database technology*.

[27] Aqib, M. and Shaikh, R.A. (2015) Analysis and comparison of access control policies validation mechanisms. *International Journal of Computer Network and Information Security* **7**(1).

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

[29] Lei, Y., Kacker, R., Kuhn, D.R., Okun, V. and Lawrence, J. (2007) IPOG: A general strategy for t-way software testing. In *IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*.

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

[31] Cimatti, A., Clarke, E., Giunchiglia, F. and Roveri, M. (1999) NuSMV: A new symbolic model verifier. In *International conference on computer aided verification (CAV)*.

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

[33] Martin, E. and Xie, T. (2007) A fault model and mutation testing of access control policies. In *International conference on World Wide Web (WWW)*.

[34] Xu, D., Shrestha, R. and Shen, N. (2018) Automated coverage-based testing of XACML policies. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*.