Relationship-Based Access Control for an Open-Source Medical Records System

Syed Zain R. Rizvi Philip W. L. Fong
University of Calgary
Alberta, Canada
{szrrizvi, pwlfong}@ucalgary.ca

Jason Crampton James Sellwood
Royal Holloway, University of London
Egham, United Kingdom
jason.crampton@rhul.ac.uk
james.sellwood.2010@live.rhul.ac.uk

ABSTRACT

Inspired by the access control models of social network systems, Relationship-Based Access Control (ReBAC) was recently proposed as a general-purpose access control paradigm for application domains in which authorization must take into account the relationship between the access requestor and the resource owner. The healthcare domain is envisioned to be an archetypical application domain in which ReBAC is sorely needed: e.g., my patient record should be accessible only by my family doctor, but not by all doctors.

In this work, we demonstrate for the first time that ReBAC can be incorporated into a production-scale medical records system, OpenMRS, with backward compatibility to the legacy RBAC mechanism. Specifically, we extend the access control mechanism of OpenMRS to enforce ReBAC policies. Our extensions incorporate and extend advanced ReBAC features recently proposed by Crampton and Sellwood. In addition, we designed and implemented the first administrative model for ReBAC. In this paper, we describe our ReBAC implementation, discuss the system engineering lessons learnt as a result, and evaluate the experimental work we have undertaken. In particular, we compare the performance of the various authorization schemes we implemented, thereby demonstrating the feasibility of ReBAC.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Control

Keywords

Medical records system, relationship-based access control, authorization graph, authorization principal, administrative model

1. INTRODUCTION

OpenMRS [4] is a production-scale, open-source electronic medical records system that has been deployed in many countries, including South Africa, Kenya, Rwanda, India, China, United States, Pakistan, the Phillipines, etc. Despite its tremendous success and wide deployment, OpenMRS has a limitation in its access control mechanism, which is an instantiation of ***Role-Based Access Control (RBAC)***. This limitation is the topic of the following posting in the developer forum [28].

> *The RBAC system provides a reasonably robust mechanism for restricting access to system behaviours; however, we do not yet have a mechanism for restricting access to specific data (e.g., you can see data for patient X, but not patient Y; or, you can see your patient's data except for specific lab results).*

An interpretation of the above limitation is that, while it is possible to restrict access of patient records to the role of doctors, it is not possible to restrict access of *my* patient record to *my* family doctor. RBAC satisfies the access control requirements of business domains in which data objects are "owned" by the organization, and thus all qualified personnel (i.e., of a certain role) may be granted access. In application domains in which privacy is a concern, the data objects are sometimes "owned" by individuals. There is now a need for finer-grained access control: e.g., my patient record shall only be accessible by the clinicians who are actually treating me. That is, access is granted on the basis of how the requestors are related to me.[1]

The above access control challenge is one of the primary motivations for the recently proposed ***Relationship-Based Access Control (ReBAC)*** models. Originally inspired by the access control models of social network systems (e.g., Facebook), ReBAC grants access based on how the access requester is related to the resource owner (e.g., friends, friends-of-friends). This is in contrast with RBAC, in which access is granted by considering the attributes of the requestor. Fong *et al.* proposed a series of general-purpose ReBAC models [19, 21, 10], in which ReBAC is envisioned to be applied to application domains other than social computing, with the healthcare domain being an archetypical example. While the idea of ReBAC has undergone a number of recent extensions in the literature [14, 13, 6, 16, 20, 33], *what remains to be seen is the adoption of ReBAC in a production-scale system for an application domain other than social computing. And this is the gap we attempt to bridge by extending the access control subsystem of OpenMRS to include an implementation of a ReBAC model.*

[1]Further examples of relationships are given in §7 and §9.

SACMAT'15, June1–3, 2015, Vienna, Austria.

http://dx.doi.org/10.1145/2752952.2752962.

In this paper, we report our experience of extending the access control subsystem of OpenMRS with a ReBAC model. The "diff" between our extension and the original OpenMRS code base consists of 25,754 lines (with no context lines). The extension involves 113 new files, 26 new database tables, and 15 web pages. Our contributions are the following.

1. We demonstrated for the first time that ReBAC can be incorporated into a production-scale medical records system, and did so with backward compatibility to the legacy RBAC mechanism.
2. We identified system engineering issues that one needs to address when one is to cleanly and efficiently implement ReBAC in a large system (§4, §5, §6 and §10).
3. We adapted, extended and implemented the advanced features of ReBAC that were recently proposed by Crampton and Sellwood [16]. The implemented features include a generalization of social graphs called authorization graphs (§5), a ReBAC analogue of roles called authorization principals (§7), and a Unix-style authorization mechanism for authorization principals (§8). Because OpenMRS supports a rich mechanism of privilege matching, the notions of authorization principals and authorization algorithms as proposed in [16] must be either adapted (§7) or extended (§8). Our novel extensions involve the proposal of two semantics of authorization (strict-grant vs liberal-grant), as well as a highly efficient principal matching algorithm based on the idea of lazy evaluation (lazy-match). What is pleasantly surprising is that, even after extensive adjustments in our implementation, the basic spirit of Crampton and Sellwood's design is preserved, thereby demonstrating the robustness of their proposals.
4. We designed and implemented an administrative model for ReBAC (§9). To the best of our knowledge, this is the first such implementation.
5. We empirically evaluated the performance of the various authorization schemes in item 3 above (§10). The evaluation is performed on a social network of 1.6 million nodes and 30 million directed edges. The lazy-match algorithm is found to offer competitive performance.

2. RELATED WORK

It has long been observed that the health domain requires an access control model that takes into account the relationship between the resource owner and the access requestor when an authorization decision is made [9, 29]. That was partly the reason that led to the proposal of an extension of OpenMRS to incorporate parameterized roles [18].

Relationship-Based Access Control was a term coined independently by Gates [22] and Carminati and Ferrari [11] to refer to a paradigm of access control in which authorization decisions are based on whether the resource owner and the access requestor are related in a certain way. Initially, ReBAC was envisioned to be applied to the domain of social computing. A seminal work with this application in mind was that of Carminati *et al.* [12].

Fong *et al.* proposed a series of general-purpose ReBAC models [19, 21, 10], and advocated the adoption of ReBAC for application domains outside of social computing. The health domain was envisioned to be an application domain in which ReBAC is particularly suited. ReBAC protection states are social networks. Modal logic and hybrid logic were proposed as policy languages for specifying ReBAC policies [19, 21, 10].

In UURAC (user-to-user relationship-based access control) [14], a policy is specified in a regular expression-based policy language. Access is granted if the resource owner and the access requestor are connected by a path made up of a sequence of edge labels satisfying the regular expression. An algorithm for finding a path that honors the regular expression is formulated. In a subsequent work [13], the protection states were extended to track relationships between user and resources (U2R) as well as between resources and resources (R2R). Another innovation is the provision for multiple policies to be applicable to the protection of a resource, and the design of conflict resolution policies (conjunctive, disjunctive and precedence) to arbitrate authorization decisions. The work proposes to employ ReBAC to regulate administrative activities, but does not provide details on how that is achieved. The administrative actions we proposed in §9 has clear semantics of how they are protected by security preconditions, and how their executions affect the protection state.

In [20], a temporal dimension is introduced into ReBAC, so that access control policies require entities to be related in a certain way in the past. The goal of this extension is to support the expression of social contracts in online communities. In [33], ReBAC is extended to account for geo-social network systems, and the hybrid logic policy language is extended to impose relationship constraints over people located in a certain geographical neighbourhood.

Crampton and Sellwood recently proposed a series of extensions to ReBAC [16]. The protection state is an authorization graph that tracks relationships among users, resources, as well as other abstract entities relevant to access control (e.g., groups, roles, etc). They also proposed a ReBAC analogue of roles called authorization principals. The run-time semantics of authorization principals are specified through path conditions, a language akin to regular expressions. An XACML-style conflict resolution mechanism was proposed to arbitrate authorizations when the access requestor is associated with authorization principals that grant conflicting authorizations. A UNIX-inspired authorization procedure serves as a framework for binding these technologies together. Our implementation has adopted the ideas of authorization graphs, authorization principals, and the UNIX-style authorization procedure. Yet we employ hybrid logic rather than path conditions for specifying the denotation of authorization principals. Detailed comparison with [16] will be given in the rest of this paper.

It has long been recognized that any practical access control system must provide ways to modify the authorization state or policy [24, 30]. While administrative access control models, which control modifications to policies, have been widely studied for the protection matrix and RBAC, this is a relatively unexplored area in the context of ReBAC. Fong's ReBAC model allows for changes to the protection state through the use of contexts [19], but we are not aware of any implementation of administrative features for ReBAC.

3. ReBAC GOES OPEN SOURCE

This section reviews the background materials needed for understanding the rest of the paper.

3.1 An Overview of ReBAC

In a series of papers [19, 21, 10], Fong *et al.* proposed a general-purpose access control model for Relationship-Based Access Control (ReBAC). This work is mainly based on the variant of the model discussed in [10].

The protection state of ReBAC is an edge-labelled, directed graph: directed edges represent interpersonal relationships, and each edge is labelled with a relation identifier to signify the type of relation (e.g., patient-of). In the original conception of ReBAC, vertices represent users, and thus the protection state is a social network. (In §5, we follow the proposal of [16], and generalize the social network to an authorization graph.)

A ***graph predicate*** determines whether particular conditions, relating the vertices in a graph, hold or not. We might, for example, define a predicate that returns true if two vertices are connected by an edge having a particular label. More formally, a ***graph predicate of arity*** k is a Boolean-valued function $GP(G, x_1, \ldots, x_k)$, where G is a graph that defines the current protection state of the system, and each x_i is a vertex in G. GP evaluates to 1 if and only if $(x_1, \ldots, x_k)$ belongs to a k-ary relation defined over G. A graph predicate of arity 2 is said to be a ***relationship predicate***. The relationship predicate $\mathsf{friend\text{-}of\text{-}friend}(G, x_1, x_2)$, for example, returns 1 iff there exists a path of length two or less in G connecting x_1 and x_2, where both edges in the path are labelled with the relationship type friend.

A ReBAC policy has the form "*grant access to* r *if* RP *evaluates to* 1", where r is a resource and RP is a relationship predicate [10]. An access request has the form (v, r), where user v wishes to access resource r. Access is permitted if $RP(G, u, v)$ evaluates to 1, where u is the owner of resource r. (In §7, we adopt a recent idea due to Crampton and Sellwood [16], and formulate ReBAC policies in terms of authorization principals — a ReBAC analogue of roles.)

A graph predicate $GP(G, x_1, \ldots, x_k)$ (with a relationship predicate as a special case) can be syntactically specified as a Hybrid Logic formula ϕ [10] with k free variables.[2] A local model checker is an algorithm that takes as input (i) a hybrid logic formula ϕ with k free variables, (ii) a protection state G, and (iii) k vertices $v_1, \ldots, v_k$, and then decides whether the k-ary graph predicate represented by ϕ is satisfied by $v_1, \ldots, v_k$ in G.

In the rest of this paper, knowledge of hybrid logic is not necessary for appreciating the contributions of this work. Nevertheless, examples of hybrid logic formulas will be shown to convey the realism of our design. Readers who are unfamiliar with hybrid logic can safely skip those examples.

3.2 The ReBAC Java Library

A reusable Java library of ReBAC technologies was released under open-source terms [8]. The library was developed and maintained separately from OpenMRS. The library was also packaged as a Maven module for easy integration with large projects.

[2]More specifically, a graph predicate of arity k can be represented by a hybrid logic formula ϕ with k free variables, such that ϕ is a Boolean combination of ***anchored formulas***. Each anchored formula is one in which the top-level operator is $@_x$, where x is one of the free variables. This is a generalization of the syntactic restriction adopted in [10] for relationship predicates (i.e., arity 2).

A main feature of the ReBAC library is the implementation of a local model checker for the hybrid logic policy language of [10]. This model checker is a cornerstone of the authorization mechanism in our OpenMRS extension, allowing us to determine membership in authorization principal (§7), as well as to test if an administrative action is enabled and/or applicable (§8).

Recall that the inputs to a local model checker include a graph and the abstract syntax tree (AST) of a hybrid logic formula. To allow the model checker to interoperate with different representations of graphs, we have defined a Java interface for graphs. For example, in the case of OpenMRS, relationship edges may come from three different sources (§5). A concrete class that makes appropriate queries to check for existence of each of the three kinds of relationships will implement the graph interface, thereby allowing the model checker to interoperate with OpenMRS.

Similarly, Java interfaces are declared for the AST nodes of hybrid logic formulas. This allows the model checker to work with different representations of hybrid logic formulas. One may ask why there is a need for different representations of hybrid logic formulas. A motivating example comes from OpenMRS. In OpenMRS, all data objects are stored as persistent objects (via Hibernate [25]). That includes AST nodes of hybrid logic formulas. Consequently, there are concrete representational demands on how AST node classes are declared (e.g., must be a subclass of a certain superclass). Declaring interfaces for AST nodes allows our model checker to interoperate with such representational idiosyncrasies.

Other features of the library include an XML parser for hybrid logic formulas that are stored as XML files.

4. ARCHITECTURE OF OpenMRS

This section introduces the architecture of OpenMRS, and explains how ReBAC is built on top of this architecture.

4.1 Interposition via AOP

Our ReBAC implementation is based on the source code of OpenMRS 1.10.[3] OpenMRS is built on the Spring Framework, which is a Java-based web application framework [32]. Core functionalities of OpenMRS are exposed as ***service-layer methods*** on the web application server. The HTML pages invoke the service-layer methods in order to query application data or alter application state. Access control is achieved by limiting access to the service-layer methods.

Each service-layer method is annotated with either one of two kinds of guard.[4] Intuitively, a guard is a specification of privilege requirements that must be satisfied by the requestor in order for the invocation of the service-layer method to be allowed.

1. `one-of(`P`)`: Here P is a set of privileges (i.e., positive permissions). The intended meaning is that the invoker of this method must have been granted one of the privileges in P in order for method invocation to be authorized.
2. `all-of(`P`)`: The invoker must have all of the privileges in P.

[3]The latest stable version of OpenMRS is 2.0, released on February 26, 2014.

[4]Annotation is achieved via the Java custom annotation mechanism [23, §9.7].

Formally, we write $Q \models g$ for a set Q of privileges and a guard g whenever Q ***satisfies*** g in the following sense:

$$Q \models \texttt{one-of}(P) \quad \text{iff} \quad P \cap Q \neq \emptyset$$
$$Q \models \texttt{all-of}(P) \quad \text{iff} \quad P \subseteq Q$$

Intuitively, if a requestor u has been "*granted*" a set Q of privileges, and $Q \models g$, where g is the guard of the method that u attempts to invoke, then invocation is authorized. (As we shall see in §8, there are two ways to interpret the word "*granted*", thereby yielding two authorization semantics.)

Spring uses aspect-oriented programming (AOP) [26] to implement interposition of authorization checks. The original authorization checking code is implemented as an "advice" (more precisely, a "before advice") that is "weaved" into the entry point of each service-layer method, thereby introducing additional behaviour on method entry. Thus every method invocation is intercepted by the RBAC authorization mechanism. To implement ReBAC, we introduced an additional authorization advice. The ReBAC authorization advice is an "around advice", which introduces additional behaviour at both the entry and exit of a method. Consequently every method invocation as well as method return is intercepted by the ReBAC authorization mechanism.

LESSON 1. *Physically localizing all authorization checks in an identifiable code unit (e.g., module, reference monitor, aspect, etc) greatly eases the extension of the authorization mechanism to incorporate ReBAC.*

In fact, the above lesson applies generally to all software systems that anticipate future evolution in their authorization mechanisms (incorporating ReBAC is but one possible evolution), and we have very positive experience with AOP in this regard.

4.2 Combining RBAC and ReBAC

Unmodified, OpenMRS enforces a Role-Based Access Control (RBAC) model [31], although the notion of sessions is not implemented. That is, all roles assigned to a user are activated when the user logs into the system. The likely reason is that the notion of role activation is probably too exotic for medical professionals, and the extra step of role activation in every log-in attempt would degrade care delivery efficiency. It has also been pointed out that the support for sessions is not essential to core RBAC implementations in certain application domains [27].

As discussed in §4.1, the original RBAC authorization checks are implemented as an advice. We implemented ReBAC authorization checks as a separate advice. The configuration is that the RBAC authorization checks are conducted first, and only when access is granted by RBAC will the ReBAC authorization checks be conducted. In summary, access is granted when both the RBAC and ReBAC mechanisms authorize access. We have also tailored configuration files in such a way that system administrators who do not use the new ReBAC features will not observe any difference between the original implementation and the extended one.

LESSON 2 (BACKWARD COMPATIBILITY). *Care must be taken to ensure that ReBAC features are backward compatible with the legacy access control model of the system.*

Crampton and Sellwood proposed a way of "encoding" RBAC in their extended ReBAC model [16]. This suggests an alternative means for integrating ReBAC and RBAC: implement only a ReBAC model, and simulate RBAC with ReBAC. Such an approach would be particularly fitting if the software application is written from scratch with a requirement to support both access control models.

4.3 Protection and Application State

In an application with a traditional access control model (e.g., RBAC), the protection state (e.g., role hierarchy, user-role assignment, etc) of the system is separate from its application state (i.e., application data). This is true of the original architecture of OpenMRS.

In social computing systems, however, the above is not necessarily true. For example, the interpersonal relationships articulated by users in a social network system is both application data and part of the protection state: authorization is granted based on the relationship between the resource owner and requestor. Inspired by social computing applications, ReBAC inherits this overlapping of protection and application state.

The above overlap is also present in the ReBAC extension of OpenMRS. Included in a patient record is a set of users (e.g., family members) related to the patient, as well as their relationships. This, for example, allows clinicians to anticipate hereditary conditions, or to identify compatible blood, organ and tissue donors. These relationships obviously belong to the application state of OpenMRS. Yet, as we shall see below, ReBAC authorization checks also make use of such relationships when an authorization decision is computed. That is, these relationships constitute part of the protection state.

The above overlap creates something of a dilemma. In the original OpenMRS architecture, patient relationships are accessible only via service-layer methods, thereby ensuring complete mediation. Yet, the ReBAC authorization advice also needs to access patient relationships. The advice will therefore need to invoke service-layer methods in order to access the relationships. As invocations of service-layer methods are intercepted by the authorization advice, this inevitably leads to an infinite loop.

All patient data, including patient relationships, are stored as Hibernate persistent objects [25]. These persistent objects are made accessible via Data Access Objects (DAOs). To break the infinite loop, we created direct access paths to patient relationships by configuring DAOs specifically for the ReBAC authorization advice, so that the latter may access patient relationships without mediation of authorization checks. The above experience leads to the articulation of the following general lesson for ReBAC systems.

LESSON 3 (APPLICATION AND PROTECTION STATE). *Data belonging to both the application and protection state of a system must be held in a data store which exposes two Application Programming Interfaces (APIs). One is mediated by authorization checks, the other is not. The mediated API is invoked by users, while the unmediated one is utilized internally for authorization.*

5. AUTHORIZATION GRAPH

In the early conception of ReBAC [10], the protection state is a social network of users: an edge-labelled, directed graph in which vertices represent users and edges model their interpersonal relationships. Crampton and Sellwood proposed an extension of ReBAC in which the protection state is an ***authorization graph*** [16]. The vertices model not only users, but also resources as well as other entities that are relevant to access control (e.g., groups). The edges capture relationships among users, objects and the aforementioned entities. Our ReBAC adaptation of OpenMRS implements the idea of authorization graphs.

When one applies ReBAC to an enterprise application domain (i.e., a domain other than social computing), a frequently raised question is: Where do the relationships come from? This rest of this section reports our answer to this question, as shaped by our experience with OpenMRS.

In OpenMRS, domain objects are all instances of the root class `BaseOpenmrsObject`, which has two subclasses `BaseOpenmrsData` and `BaseOpenmrsMetadata`. The instances of `BaseOpenmrsData` include users, patient records and their components, etc. Therefore, we take all instances of `BaseOpenmrsData` as the vertices of the authorization graph.

The authorization graph tracks binary relationships among instances of `BaseOpenmrsData`. During our development of the ReBAC extensions for OpenMRS, we identified three categories of relationships.

1. ***User-managed relationships***. These are relationships that are explicitly articulated and managed by end users. An example is friendship in Facebook.
   As we mentioned in §4.3, OpenMRS enables a clinician to document in a patient record the relatives of the patient. These interpersonal relationships are considered part of the authorization graph. More specifically, `BaseOpenmrsData` has a subclass `Person`. Recorded interpersonal relationships between instances of `Person` are considered to be edges in the authorization graph.
2. ***System-induced relationships***. The data structures of the system may contain relationships that are relevant to authorization. Examples include organizational structures, object ownership, object containment and provenance relationships. End users are not allowed to directly manipulate these relationships.
   In our ReBAC adaptation of OpenMRS, we have created an extension mechanism for administrators to introduce new system-induced relationships. Specifically, a system-induced binary relation is implemented as a Java class that performs queries into the run-time data structures of OpenMRS. Such a class implements the `ImplicitRelationIdentifier` interface, which defines a standard calling convention for performing relationship queries. At run-time, such a class will be dynamically loaded into the Java Virtual Machine, an instance of that class is created, and an appropriate method of that instance will be invoked when the authorization mechanism needs to check the system-induced relation. The administrator can install an extension class for each type of system-induced relationship.
   In our ReBAC adaptation of OpenMRS, a system-induced relation relates instances of `BaseOpenmrsData`, meaning that such relationships are not only among users, but they may also relate resources to resources, or persons to resources. As an example of the last case, we implemented resource ownership (owner) as a system-induced relation, relating a resource to its owner(s).
3. ***Access control relationships***. There are relationships that belong solely to the protection state: they are tracked solely for the purpose of access control, and have no relevance to the business logic of the application. Examples of access control relationships include role or group membership, records of access events (e.g., for implementing history-based policies, as in [20]), etc.
   In our ReBAC adaption of OpenMRS, access control relationships are defined among instances of the `Person` class. Manually adding or removing access-control edges in the authorization graph is an error-prone step. To reduce the cognitive burden of users, we have implemented an administrative model for ReBAC, thereby supporting a principled way for adding or removing access control relationships. See §9 for details.
   For example, say the family doctor of a patient may refer the patient to a specialist. Such a capability is only allowed if patient and a clinician are related by an access control relationship family-doctor. Once the referral is confirmed, the patient and the specialist will be related by the access control relationship referred-clinician, thereby enabling the specialist to access the patient's record.

LESSON 4. *In a ReBAC system, relationships come from three sources. Some relationships belong purely to the protection state (i.e., access control relationships): these are managed by system administrators. Other relationships are shared between the application state and the protection state. This latter kind may be further classified into (i) relationships that are explicitly articulated and managed by end users, and (ii) relationships that are induced by the system data structure (and thus cannot be manipulated directly by users and administrators).*

6. ACCESS REQUESTS

The ReBAC authorization advice needs three pieces of information to compute an authorization decision: (a) the resource r to which access is required, (b) the user u who wishes to have access (aka the "requestor"), and (c) the guard g of the service-layer method being invoked. Therefore, an access request in OpenMRS is characterized by a triple (r, u, g).

The ReBAC authorization advice can discover the identity of the requestor (u) and the service-layer method that is being invoked.[5] Using the Java Reflection API, the ReBAC authorization advice can then extract the guard (g) of the service-layer method. The last component of the access request, namely the resource r, is not directly available. OpenMRS was originally designed to use RBAC for authorization, and that explains why the identity of the resource is not explicitly made available for the authorization mechanism. In the following, we discuss how the requested resource r is identified in a systematic manner for the ReBAC authorization advice.

The ReBAC authorization advice has access to the arguments that are passed to the service-layer method, as well as the return value of that invocation. Depending on the

[5]The requestor can be identified by calling a public static method of the `Context` class in OpenMRS. The ReBAC authorization method is passed an argument of type `MethodInvocation`, which in turn provides access to the identity of the service-layer method that is being invoked.

kind of service-layer method, the target resource may be either (a) an argument or (b) the return value. There are two kinds of service layer methods in OpenMRS:

1. A ***setter*** method is one that operates on a given resource, which appears as one of the method arguments. That is, a setter produces side effects on the application state. The argument for which side effect is targeted is the resource that requires access control.
2. A ***getter*** method retrieves patient information (e.g., searching for the records of all patients with a given family name). The return value of a getter method is either (a) a single piece of patient information, or (b) a collection or a map of patient information. In the former case, the returned patient datum is the resource that requires access control, and in the latter case, every returned patient datum requires access control.

In the original design of OpenMRS, a naming convention is adopted to differentiate getter and setter methods, but there is no way for the ReBAC authorization advice to recognize which argument of a setter requires access control.

To address the above problem, we designed a custom annotation `@Resource` for identifying (a) whether a service-layer method is a setter or a getter, and (b) the target resource for each kind. In the case of setter methods, the `@Resource` annotation can be applied to a method parameter to indicate that that parameter corresponds to a protected resource.

T `m(`T_1 x_1`, @Resource` T_2 x_2`,` T_3 x_3`) { ... }`

The `@Resource` annotation is applied above to explicitly declare that the parameter x_2 of method m is a controlled resource. We systematically annotated the setter methods in the OpenMRS code base using the above annotation.

When the authorization advice is invoked, it employs the Java Reflection API to discover if any of the parameters of the invoked method is annotated by `@Resource`. If so, then it will pass the request (r, u, g) through the authorization procedure, where r is the value of the annotated parameter. Invocation of the method is only granted if authorization is successful.

Similarly, the `@Resource` annotation can also be applied to the method as a whole to declare that the method is a getter and thus the return value requires access control.[6]

`@Resource` T m`(`T_1 x_1`,` T_2 x_2`, ...) { ... }`

Again, we systematically annotated the getter methods in the OpenMRS code base using the above annotation.

Before an invoked method returns, the ReBAC authorization advice will check if the method has the `@Resource` annotation. If so, it will perform authorization checks on the return value. If the return value is a single piece of patient information r, then the request (r, u, g) will be subject to the authorization procedure, and a security exception will be raised if authorization fails. Otherwise, the return value is either a collection or a map of patient information. For every member r in the returned collection (resp. map), the request (r, u, g) will be subject to authorization check. A collection (resp. map) containing only those rs that pass authorization will be returned.

Lesson 5. *The legacy authorization subsystem of some applications may not have direct access to both the requestor and the resource of an access request. A ReBAC extension of such applications will need to provide means for run-time identification of these two entities.*

7. AUTHORIZATION PRINCIPALS

In an early conception of ReBAC [10], a ReBAC policy has the form "*grant access to r if RP*", where r is a resource and RP is a relationship predicate. There are two limitations to this design. First, access to resource r may be performed via many different forms of operations, and thus finer grained access control based on permissions (i.e., privileges in OpenMRS) is desirable. Second, there is no provision of permission abstraction (i.e., such as roles in RBAC) to ease administration. To overcome these limitations, Crampton and Sellwood [16] proposed an extension of ReBAC that is based on permission granting, and invented the notion of ***authorization principals***, which could be seen as a ReBAC analogue of roles, to ease administration. In our ReBAC extension of OpenMRS, we have adopted a variant of Crampton and Sellwood's proposal. In the following, we will first describe the scheme that we actually implemented, and then discuss how it differs from the original proposal of Crampton and Sellwood.

An authorization principal is defined via a principal matching rule of the form (AP, RP), where AP is the identifier of the authorization principal, and RP is a relationship predicate. Unlike a role in RBAC, in which membership in a role is defined statically (via the user-role assignment relation UA), the semantics of an authorization principal is dynamic. When a request to access resource r is issued (at run time), AP denotes the set of users u for which r and u satisfy RP, the relationship predicate that is associated with AP. This notion of authorization principals is actually familiar to us. For example, in Unix, there are three built-in authorization principals: "owner", "group", "other" (aka "world"); in Facebook, there are four built-in authorization principals: "me", "friend", "friend-of-friend", "everyone".

Note again that, in the original conception of ReBAC [10], the relationship predicate in a ReBAC policy specifies a desired relation between the resource owner and the access requestor. In contrast, the relationship predicate in a principal matching rule specifies a desired relation between the resource itself and the requestor. For example, the following principal matching rule specifies the principal $\mathsf{treating\text{-}clinician}$.

$$\Big(\mathsf{treating\text{-}clinician},\quad @_{\mathsf{resource}}\langle\mathsf{owner}\rangle\big(\langle\mathsf{family\text{-}doctor}\rangle\mathsf{requestor} \vee \langle\mathsf{referred\text{-}clinician}\rangle\mathsf{requestor}\big)\Big)$$

The rule says that the requestor is a treating clinician if she is either the family doctor or a referred specialist of the resource's owner. Note that the two free variables $\mathsf{resource}$ and $\mathsf{requestor}$ identifies the two parameters of the relationship predicates.

In our implementation, there is only one principal matching rule for each authorization principal AP: i.e., the princi-

[6]The annotation of getter methods is not absolutely necessary, as the above-mentioned naming convention already identifies getter methods. The annotation is performed as a convenience for the ReBAC authorization advice. The annotation of setter methods, however, is necessary in order for the ReBAC authorization advice to function properly.

pal matching rule defines a functional mapping from authorization principals to their corresponding relationship predicates. We write RP_{AP} for the relationship predicate of the authorization principal AP.

Permission abstraction is achieved by authorization rules of the form (AP, P), where AP is the identifier of an authorization principal, and P is a set of privileges. The meaning is analogous to the permission assignment relation PA in RBAC. That is, at run time, the members of authorization principal AP is granted permissions in P.

In our implementation, there is only one authorization rule for each authorization principal. We write P_{AP} for the set of privileges granted to authorization principal AP.

The scheme we implemented differs from the original proposal of Crampton and Sellwood in the following manners.

- Crampton and Sellwood use a formalism called path conditions to specify the relationship predicate RP. In our implementation, RP is specified via a hybrid logic formula. Path conditions and hybrid logic have incomparable expressiveness. There are certain relationship predicates that are expressible in hybrid logic but not path conditions, and vice versa. Extending our implementation to accommodate other specification formalisms for relationship predicates is a modular task.
- In Crampton and Sellwood's proposal, an authorization rule may grant either positive or negative permissions (i.e., allow or deny). Complying to the original design of OpenMRS, our implementation supports only positive permissions. Without negative permissions, the conflict resolution strategies proposed in [16] are not needed and thus not implemented. Extension of our implementation to accomodate negative permissions and conflict resolution is a tractable endeavour.
- An authorization rule of Crampton and Sellwood has an explicitly specified scope of applicability. Specifically, a rule is either applicable to all resources, or it is applicable only to a specific resource r. Our implementation supports only the first possibility (applicable to all resources).

A number of user interface elements have been introduced to ease the administration of authorization principals and privilege assignment. First, the specification of principal matching rules and the specification of authorization rules are performed in two separate web pages. Each web page is protected by separate privileges. This separation of duty allows a different group of administrators to be responsible for specifying each kind of rules. Second, we have developed a Javascript-based structure editor for specifying Hybrid Logic formulas (e.g., as relationship predicates in principal matching rules).

8. AUTHORIZATION MECHANISM

Inspired by the UNIX access control model [15], Crampton and Sellwood proposed an authorization mechanism for determining when a request is to be granted. We adapted their proposal for OpenMRS. Given an access request (r, u, g) directed against a protection state (i.e., an authorization graph) G, an authorization principal AP is said to be ***enabled*** iff $RP_{AP}(G, r, u) = 1$. Intuitively, the requestor u is a member of the enabled principals for the present access request. Thus, requestor u is granted the privileges in P_{AP}, for each enabled principal AP. Such privileges are then used for satisfying the privilege requirement of guard g.

On top of the above adaptations, we propose two novel extensions to their scheme: (a) liberal- and strict-grant semantics, and (b) eager- and lazy-match strategies.

Authorization Semantics. The presence of guards of the form `all-of(`P`)` present ambiguities in the precise manner in which authorization should be conducted. We therefore extend the proposal of Crampton and Sellwood by differentiating between two semantics of authorization.

1. ***Liberal-grant semantics***.
   - Let $\mathcal{E}$ be the set of all enabled principals.
   - Let $Q = \bigcup_{AP \in \mathcal{E}} P_{AP}$. That is, Q is the set of all privileges that are granted by at least one enabled principal.
   - Authorization is granted iff $Q \models g$.

   In liberal-grant authorization, the privileges required by g may come from any enabled principals. The assumption is that the requestor u can simultaneously "be" all the enabled principals.
2. ***Strict-grant semantics***.
   - Let $\mathcal{E}$ be the set of all enabled principals.
   - Authorization is granted iff there exists $AP \in \mathcal{E}$ such that $P_{AP} \models g$.

   In strict-grant authorization, the privileges required by g must originate from only one enabled principal. The idea is that the privilege requirements of g are satisfied only if there is an enabled principal who can "single-handedly" satisfy it.

The two semantics produce identical behaviour if the guard g is of the form `one-of(`P`)`.[7] They differ in behaviour only if the guard is of the form `all-of(`P`)`.[8] If a request is authorized in the strict-grant semantics then it is authorized in the liberal-grant semantics.[9]

Principal Matching Strategies. For each of the above semantics, we also developed two principal matching strategies.

1. ***Eager-match strategy***. This is the straightforward implementation of the two semantics, in which the set of all enabled principals is computed before an authorization decision is produced.
2. ***Lazy-match strategy***. This is an optimized implementation of the two semantics. The core idea is that the testing of relationship predicates during principal matching (i.e., determining which principals are enabled) is an expensive operation, and thus such checks should be avoided whenever possible. This idea is materialized in two ways. First, two principals may share the same relationship predicate. There is no point re-evaluating the predicate for both principals. When we determine what principals are enabled, the same relationship predicate is evaluated only once. Second, rather than computing the set of all enabled authorization principals, they are computed one at a time, and only for the principals that are relevant. If the

[7]If $g =$ `one-of(`P`)`, then $P \cap (\bigcup_{AP \in \mathcal{E}} P_{AP}) \neq \emptyset$ iff there exists $AP \in \mathcal{E}$ such that $P \cap P_{AP} \neq \emptyset$. That is, the two semantics agree in their authorization decisions.

[8]Suppose $g =$ `all-of(`$\{p_1, p_2\}$`)`. Suppose further $\mathcal{E} = \{AP_1, AP_2\}$. Say $P_{AP_1} = \{p_1\}$ and $P_{AP_2} = \{p_2\}$. Then liberal grant semantics will allow access but strict grant semantics will deny access.

[9]Suppose strict-grant allows access. There exists $AP \in \mathcal{E}$ such that $P_{AP} \models g$. In that case, $\bigcup_{AP \in \mathcal{E}} P_{AP} \models g$ as well, since $\models$ is monotonic. Thus, liberal grant allows access also.

Algorithm 1: Lazy-match, liberal-grant authorization of access request (r, u, g) against authorization graph G.

```
let P be such that g is either all-of(P) or one-of(P);
Q := ∅;
foreach AP do
    if (P_AP \ Q) ∩ P ≠ ∅ then
        if RP_AP(G, r, u) has been evaluated then
            reuse previous value;
        else
            compute value;
        if value is true then
            Q := Q ∪ P_AP;
            if Q ⊨ g then
                return "allow";
return "deny";
```

Algorithm 2: Lazy-match, strict-grant authorization of access request (r, u, g) against authorization graph G.

```
foreach AP do
    if P_AP ⊨ g then
        if RP_AP(G, r, u) has been evaluated then
            reuse previous value (which must be false);
        else
            compute value;
        if value is true then
            return "allow";
return "deny";
```

privileges associated with a principal do not contribute to the satisfaction of the guard in question, it is ignored, and its relationship predicate is not even evaluated. Otherwise, the principal is "relevant", and its relationship predicate is checked to see if the principal is enabled. Whenever a relevant principal is found to be enabled, the authorization engine checks to see if the required privileges are already present. If so, the search for enabled principals will be terminated. In summary, this "lazy" evaluation strategy opportunistically eschews unnecessary computation. The pseudocode listings for liberal-grant and strict-grant authorization using the lazy-match strategy are shown in Algorithms 1 and 2 respectively.

The two strategies produce the same authorization decision for any given access request.

We implemented a web interface for administrators to select between liberal- or strict-grant semantics, and between eager- or lazy-match strategy (i.e., four combinations).

9. ADMINISTRATIVE ACTIONS

Access control relationships in the authorization graph belong solely to the protection state. They are not application data. Their existence serve only the purpose of protection. One way of managing such relationships will be to place the burden entirely on the system administrators. (In our implementation, we have administrative web pages for administrators to manually add or delete edges of the authorization graph.) This, however, is not scaleable. Imagine the task of adding an edge in the authorization graph to indicate that the family doctor of a patient is referring the patient to a cardiologist (and thus the said cardiologist enjoys certain access rights that other cardiologists do not have over the patient's records). Such an action is common in the daily operation of a health service. It is completely impractical to go through the bottleneck of the system administrators every time such a referral is made. One way of making this scaleable is to delegate this operation to qualified users (e.g., the family doctor in the example), so that the latter may add this edge into the authorization graph. Yet, manual addition and deletion of edges can be error prone. First, business logic may dictate that multiple updates to the authorization graph must occur together (e.g., a person may have only one supervisor, and thus the addition of a new supervisor edge must be accompanied by the deletion of an out-of-date supervisor edge). If the user performs one update but forgets another, then the integrity of the authorization graph cannot be maintained. Second, business logic may dictate that an update can only occur if the user performing the update is qualified to do so (e.g., referral can only be made by a family doctor). Undisciplined updates of the authorization graph overlooks such security requirements.

The primary design objective of administrative actions is to provide a structured means for adding and removing access control relationships, so that such tasks can be performed safely by users other than system administrators. In our design, the declaration of an administrative action consists of the following components:

- **Action identifier**: A unique name is used for identifying the administrative action. For example, the referral action is identified by the identifier "Referral".
- **Enabling precondition**: Every administrative action is presumed to be performed by a user against a patient (e.g., a family doctor performing a referral for a patient). So every administrative action has two ***primary participants***, namely, the user who performs that action and the patient to which the action is targeted. The identifiers user and patient are used in the declaration for referring to the primary participants.
  Whether the action is ***enabled*** (see below) depends on whether the user and the patient satisfy a certain relationship predicate. Such a relationship predicate, called the enabling precondition, is specified as a hybrid logic formula with free variables user and patient. For example, the following hybrid logic formula can be used for requiring that referral can only be conducted by the family doctor of a patient.[10]

  $$@_{\mathsf{user}}\langle \mathsf{family\text{-}doctor} \rangle \mathsf{patient}$$

- **Participants**: Other than user and patient, there may be other participants involved in the action. They are called ***auxiliary participants***. The participant list enumerates the identifiers to be used for referring to auxiliary participants in the rest of the declaration.
  In the example of referral, there is only one auxiliary participant, "specialist", who is the specialist to which the referral is directed.
- **Applicability precondition**: Whether the administrative action is considered ***applicable*** (see below) depends on whether a certain condition holds among all the participants (both primary and auxiliary). Such

[10] This is only an illustration. We are fully aware that in real life it is not just the family doctor who can perform referral.

a condition is specified as a hybrid logic formula containing free variables that are user, patient, as well as the identifiers listed in the participant list above. The hybrid logic formula specifies a graph predicate of arity $\ell + 2$, where ℓ is the number of auxiliary participants. In the running example, we require that (a) the specialist is approved by the insurance company of the user, and (b) the user and the specialist must belong to the same health region. The above conditions are captured by the following hybrid logic formula.

$$\big(@_{\mathsf{patient}}\langle\mathsf{insurance}\rangle\langle\mathsf{approves}\rangle\mathsf{specialist}\big) \wedge$$
$$\big(@_{\mathsf{user}}\langle\mathsf{region}\rangle\langle-\mathsf{region}\rangle\mathsf{specialist}\big)$$

- **Effects**: The effects of an administrative action is a list of ***updates***. Each update is of the form "**add** $i(x, y)$" or "**del** $i(x, y)$". Here, i is a relation identifier (e.g., supervisor-of), and x and y are identifiers of participants (either primary or auxiliary). The keywords **add** and **del** indicates whether the update is an edge addition or deletion.
  For example, the referral action has one update:

  add referred-clinician(patient, specialist)

The enabling and applicability preconditions together specify the security constraints that must be met in order for the user to be allowed to perform the administrative action against the patient. The effects may involve updating multiple edges in the authorization graph. Grouping them together in one administrative action ensures that the updates are either performed together, or not at all. This in turn ensures the integrity of the authorization graph, and prevents errors on the part of the user who performs the updates.

At run time, the following sequence of events occur, which gives semantics to administrative actions.

1. When a user retrieves the record of a patient, the two primary participants are tested against the enabling precondition of every declared administrative action. An action for which the enabling precondition is satisfied is said to be ***enabled***. The set of enabled actions is computed.
2. When the patient record is displayed, a tab showing the list of all enabled actions is made available to the user. The user may choose to perform any of the enabled actions.
3. When the user signals to perform an enabled action, the list of auxiliary participants (if any) will be displayed to the user. The user must now instantiate each of the participants by selecting a person. This is facilitated by intelligent search features offered by OpenMRS.
4. Once the participants are selected, both the enabling and applicability preconditions are checked. The action is deemed ***applicable*** if the check succeeds. (We will explain below why the enabling precondition is checked again.)
5. If the action is applicable, then the effects of the action will be executed. Note that deleting a non-existent edge is an error. Similarly, adding an edge that already exists is also considered an error. Either all the updates are executed, or execution fails without any change to the authorization graph.

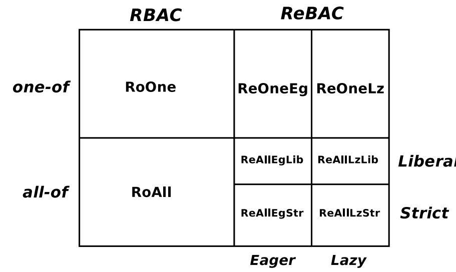

Figure 1: The 8 experimental configurations.

Note that the execution of effects is an atomic operation: either all updates are successfully executed, or else no update is performed. This is achieved by the transaction manager. Actually, the transaction begins at step 4 above. Including the check of both enabling and application preconditions into the transaction prevents time-of-check-to-time-of-use (TOCTTOU) race conditions [7, §6.2.1]. In addition, to prevent unintended roll-back, the preconditions should be crafted in such a way that the presence of an edge is verified in the preconditions if it is deleted in the effects, and the absence of an edge is confirmed in the preconditions if it is added in the effects.

To support policy engineering, we also developed administrative web pages for users to build a library of reusable hybrid logic formulas. Such formulas can be referenced in the declarations of administrative actions.

10. PERFORMANCE EVALUATION

An empirical study has been conducted to evaluate the performance of the various authorization schemes proposed in this paper. As our ReBAC-equipped version of OpenMRS is not yet deployed in any clinical setting, no production data set is available, and thus performance evaluation was conducted with synthetic data. Rather than performing "disembodied" simulation of the various authorization schemes, we measured the performance of those schemes within the infrastructure of OpenMRS, thereby capturing the overhead in a realistic implementation.

We compared the performance of the OpenMRS authorization mechanism in eight different configurations (Fig. 1). The two RBAC configurations (Ro*) correspond to OpenMRS with only the legacy RBAC authorization mechanism (i.e., ReBAC is turned off). These two configurations differ in whether requests are directed against `one-of` guards (RoOne) or `all-of` guards (RoAll). ReBAC authorization is turned on (and RBAC is turned off) for the remaining six configurations (Re*). Two of the ReBAC configurations correspond to `one-of` requests (ReOne*). They differ in whether eager- or lazy-match strategy is implemented. The last four ReBAC configurations correspond to `all-of` requests (ReAll*). In the case of `all-of` guards, there are two possible semantics (liberal- or strict-grant), as well as two possible matching strategies (eager or lazy), resulting in four configurations: ReAllEgLib, ReAllEgStr, ReAllLzLib, and ReAllLzStr.

RBAC Protection State. We randomly synthesized an RBAC protection state for the two RBAC configurations (Ro*). OpenMRS pre-compiles the role hierarchy into a flat space of roles. The RBAC protection state therefore con-

Users:	10,000	Privileges:	200	Roles:	67
Privilege-role assignment pairs:				469	
User-role assignment pairs:				50,000	

Figure 2: RBAC Parameters

tains a user-role assignment and a privilege-role assignment, but not a role hierarchy. Fig. 2 enumerates the parameters used for synthesizing the RBAC protection state. Justifications for the choice of these parameters are given in Appendix A.

ReBAC Protection State. For the six ReBAC configurations (Re*), we constructed an authorization graph out of a social network dataset, soc-Pokec, obtained from the Stanford Large Network Dataset Collection [5]. The graph has 1.6 million nodes and 30 million directed edges. This dataset is thus even bigger than what the OpenMRS community calls a high-density deployment.[11]

To construct the authorization graph, we identified 10,000 nodes with the highest in-degrees, and labelled them as users (i.e., clinicians).[12] The remaining nodes are patients. Consequently, a directed edge in the social graph can be one of four types: user-user, user-patient, patient-user, or patient-patient. According to the type of each directed edge, we then randomly labelled the directed edges using the relation identifiers of the Electronic Health Records System case study in [19, §5]. A detailed list of relation identifiers and the distribution of the edge labels can be found in Appendix B.

ReBAC Policies. The six ReBAC configurations (Re*) presumes the existence of ReBAC policies (authorization principals). We generated an authorization principal for each of the 67 roles. Authorization rules were formulated in such a way that each principal grants the same privileges as its corresponding role. Principal matching rules were in place so that every authorization principal is associated with a randomly generated hybrid logic formula. Specifically, from the two example formulas in the Electronic Health Records System case study in [19, §5], we extracted ten hybrid logic formulas for our experiment. For each principal, a formula was randomly selected from those ten formulas (with equal probability). See Appendix C for details.

Methods, Guards, and Requests. As the existing service-layer methods of OpenMRS will not work with the authorization graph synthesized above, we randomly synthesized service-layer methods for the purpose of this experiment. Each synthesized service-layer method takes a patient as an argument, and is invoked by a user (i.e., clinician). A guard is randomly generated for each method; `one-of` guards for the *One* configurations, and `all-of` guards for the *All* configurations. Each method has an empty body as we are only concerned about authorization overhead. For each of the eight configurations, we generated 200 method calls, with randomly selected clinicians and patients. The authorization times of the 200 method calls are then averaged and reported. Details can be found in Appendix D.

Results and Discussions. We conducted the experiment on a desktop machine with AMD FX-8350 8-core Processor (16 MB cache), 16 GB RAM (1866 MHz, DDR3), and an 840 EVO Solid State Drive running Windows 8 OS. The results are shown in Fig. 3.

[11]According to a thread in the OpenMRS developer forum [1], the number of patient records in various reported OpenMRS deployments ranges from 8,982 to 741,606.

[12]Our intuition is that clinicians are more connected than patients. Specifically, they have more incoming edges, for example, to indicate who is the attending clinician of whom.

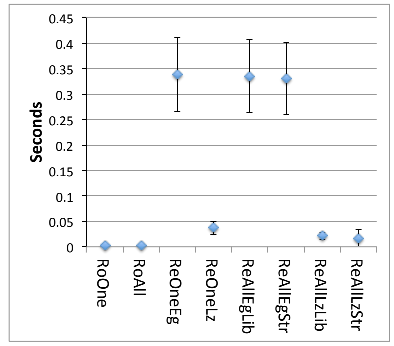

Figure 3: Average time for an authorization check (with 95% confidence interval).

The baseline RBAC configurations (Ro*) incur negligible running time. The three eager-match configurations (*Eg*) have authorization time averaging around 0.33 seconds, suggesting that the eager-match strategy is not practical. In contrast, all the lazy-match configurations (*Lz*) have competitive authorization times averaging around 0.016–0.037 seconds. In summary, matching strategy, and not authorization semantics, is the key determinant of performance.

In our experience, the main performance overhead comes not from backtracking within the hybrid logic model checker, but from database accesses. Due to the sheer size of the authorization graph (1.6 million nodes, 30 million edges), simply retrieving the neighbours of a given node takes 0.002 second in our preliminary experiments, resulting in unacceptable authorization times. Noticing this, we stored the access control relationships in a graph database (Neo4j [3]) instead of the original relational database (MySQL [2]), resulting in a 20-fold speed-up in neighbour retrieval time (0.0001 sec) and thus the fast authorization times reported above (Fig. 3).

LESSON 6. *A graph database offers more competitive ReBAC authorization performance than a relational database.*

Further details on the preparation of Neo4j for our experiments are discussed in Appendix E.

11. CONCLUSIONS AND FUTURE WORK

This ReBAC adaptation of OpenMRS is the first implementation of ReBAC in a production-scale electronic medical records system. We reported reusable engineering lessons for ReBAC deployment, presented extensions of advanced ReBAC features recently proposed by Crampton and Sellwood [16], designed and implemented the first administrative model for ReBAC, and evaluated the performance of authorization checks.

Our implementation can serve as a testbed for future extensions of ReBAC. A number of research opportunities are motivated by this implementation exercise. First, the way ReBAC interacts with the legacy RBAC mechanism is by way of conjunction: access is granted if both access control subsystem grant access. What are other ways in which RBAC and ReBAC can interact with one another to deliver advanced access control features? Second, authorization in

OpenMRS is performed through the satisfaction of privilege requirements known as guards. These privilege requirements interact with the design of other access control features (e.g., authorization principals, authorization algorithms, positive and negative permissions, conflict resolution) in an intimate manner. We have opted for simplicity in most of our design choices. Further studies on how advanced access control features can be implemented in the presence of OpenMRS-style privilege requirements is a research challenge. Third, while we have fashioned the first administrative model for ReBAC, the theory of ReBAC administrative models is an unexplored area. How does one perform, say, safety analysis in this administrative model [24]? Fourth, in the original proposal of ReBAC [19] relationships are contextual. For example, a referral relationship is effective only in the context of a certain medical case. Context creation and removal provide a clean mechanism for expiration of tentative relationships. How does one implement contexts in OpenMRS, especially with usability in mind?

Acknowledgments

This work is supported in part by an NSERC Discovery Grant (RGPIN-2014-06611) and a Canada Research Chair (950-229712).

12. REFERENCES

[1] Max number of records in an OpenMRS implementation. `http://listarchives.openmrs.org/Max-number-of-records-in-an-OpenMRS-implementation-td4662224.html`.

[2] MySQL. `http://www.mysql.com/`.

[3] Neo4J. `http://neo4j.com/`.

[4] OpenMRS. `http://openmrs.org/`.

[5] Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`.

[6] Aktoudianakis, E., Crampton, J., Schneider, S., Treharne, H., and Waller, A. Policy templates for relationship-based access control. In *Proceedings of the 11th Annual International Conference on Privacy, Security and Trust (PST'13)* (Tarragona, Catalonia, Spain, July 2013), IEEE, pp. 221–228.

[7] Anderson, R. *Security Engineering*, 2nd ed. Wiley, 2008.

[8] Anonymized. ReBAC Java Library.

[9] Beznosov, K. Requirements for access control: US healthcare domain. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'1998)* (Fairfax, VA, Oct. 1998), p. 43.

[10] Bruns, G., Fong, P. W. L., Siahaan, I., and Huth, M. Relationship-based access control: Its expression and enforcement through hybrid logic. In *Proceedings of the 2nd ACM Conference on Data and Application Security (CODASPY'12)* (San Antonio, TX, USA, Feb. 2012).

[11] Carminati, B., and Ferrari, E. Enforcing relationships privacy through collaborative access control in web-based social networks. In *Proceedings of the 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'09)* (Washington DC, USA, Nov. 2009).

[12] Carminati, B., Ferrari, E., and Perego, A. Enforcing access control in web-based social networks. *ACM Transactions on Information and System Security 13*, 1 (Oct. 2009).

[13] Cheng, Y., Park, J., and Sandhu, R. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *Proceedings of the 4th IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT'12)* (Amsterdam, Netherlands, Sept. 2012).

[14] Cheng, Y., Park, J., and Sandhu, R. A user-to-user relationship-based access control model for online social networks. In *Proceedings of the 26th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec'12)* (Paris, France, July 2012), vol. 7371 of *LNCS*.

[15] Crampton, J. Why we should take a second look at access control in Unix. In *Proceedings of the 13th Nordic Workshop on Secure IT Systems (NordSec'08)* (Copenhagen, Denmark, Oct. 2008).

[16] Crampton, J., and Sellwood, J. Path conditions and principal matching: A new approach to access control. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies (SACMAT'14)* (London, Ontario, Canada, June 2014).

[17] Ene, A., Horne, W. G., Milosavljevic, N., Rao, P., Schreiber, R., and Tarjan, R. E. Fast exact and heuristic methods for role minimization problems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT'08)* (Estes Park, CO, June 2008), pp. 1–10.

[18] Fischer, J., Marino, D., Majumdar, R., and Millstein, T. D. Fine-grained access control with object-sensitive roles. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09)* (Genoa, Italy, July 2009), vol. 5653 of *LNCS*, pp. 173–194.

[19] Fong, P. W. L. Relationship-based access control: Protection model and policy language. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy (CODASPY'11)* (San Antonio, Texas, USA, Feb. 2011), pp. 191–202.

[20] Fong, P. W. L., Mehregan, P., and Krishnan, R. Relational abstraction in community-based secure collaboration. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)* (Berlin, Germany, Nov. 2013), pp. 585–598.

[21] Fong, P. W. L., and Siahaan, I. Relationship-based access control policies and their policy languages. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT'11)* (Innsbruck, Austria, June 2011), pp. 51–60.

[22] Gates, C. E. Access control requirements for Web 2.0 security and privacy. In *IEEE Web 2.0 privacy and security workship (W2SP'07)* (Oakland, California, USA, May 2007).

[23] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. *The Java Language Specification*, Java SE 8 ed. Oracle America, Inc, Mar. 2014.

[24] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. Protection in operating systems.

Communications of the ACM 19, 8 (Aug. 1976), 461–471.

[25] Hibernate. `http://hibernate.org/`.

[26] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)* (Jyväskylä, Finland, June 1997), M. Aksit and S. Matsuoka, Eds., vol. 1241 of *LNCS*, Springer, pp. 220–242.

[27] Li, N., Byun, J.-W., and Bertino, E. A critique of the ANSI standard on role-based access control. *IEEE Security and Privacy 5*, 6 (Nov. 2007), 41–49.

[28] Mamlin, B. Re: Policy based access control for OpenMRS. `http://listarchives.openmrs.org/Policy-based-Access-Control-for-OpenMRS-td7220685.html`, Jan. 2012.

[29] Røstad, L., and Edsberg, O. A study of access control requirements for healthcare systems based on audit trails from access logs. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'2006)* (Miami Beach, Florida, USA, Dec. 2006), pp. 175–186.

[30] Sandhu, R., Bhamidipati, V., and Munawer, Q. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security 2*, 1 (1999), 105–135.

[31] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. Role-based access control models. *IEEE Computer 19*, 2 (Feb. 1996), 38–47.

[32] Spring framework. `http://projects.spring.io/spring-framework/`.

[33] Tarameshloo, E., and Fong, P. W. L. Access control models for geo-social computing systems. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies (SACMAT'14)* (London, Ontario, Canada, June 2014).

APPENDIX

A. RBAC PROTECTION STATE

We create 10,000 users for our experiments. Since OpenMRS has 184 distinct built-in privileges, we round up and thus create 200 privileges. We deduce several ratios from the "healthcare" database of [17]: (a) role-to-privilege ratio is 1:3; (b) average number of roles per user is 5; (c) average number of privilege per role is 7. From these ratios, we create 67 roles ($\approx 200/3$), 469 privilege-role assignment pairs ($\approx 67 \times 7$), and 50,000 user-role assignment pairs ($\approx 10000 \times 5$).

B. ReBAC PROTECTION STATE

From [19, §5], we extract the following relation identifiers. We indicate below the type of each identifier: e.g., an identifier of the patient-user type is identified by "p-u".

Rel. Id.	Type	Rel. Id.	Type
gp	p-u	register-ward	p-u
referrer	u-u	ward-nurse	u-u
appoint-team	u-u	agent	p-p
team	u-u		

Every directed edge in the social graph belongs to one of the four types: user-user, user-patient, patient-user, patient-patient. Based on the type of a given directed edge, a relation identifier of that type is random selected (with uniform distribution). Note that there is no relation identifier that has the type user-patient. For those edges, a dummy relation identifier is assigned.

C. ReBAC POLICIES

The Electronic Health Records System case study of [19, §5] has two formulas that we can use in our experiments. The first formula, specifying the patient-clinician relation, is constructed incrementally in four stages in [19, §5.1]. We take the subformulas constructed in the various stages as candidate formulas for our experiment.

$$\phi_1 = \langle \mathsf{gp} \rangle \mathsf{requestor}$$
$$\phi_2 = \langle \mathsf{gp} \rangle \langle -\mathsf{referrer} \rangle \mathsf{requestor}$$
$$\phi_3 = \phi_1 \vee \phi_2$$
$$\phi_4 = \langle \mathsf{gp} \rangle \langle -\mathsf{referrer} \rangle \langle \mathsf{appoint\text{-}team} \rangle \mathsf{requestor}$$
$$\phi_5 = \langle \mathsf{gp} \rangle \langle -\mathsf{referrer} \rangle \langle \mathsf{appoint\text{-}team} \rangle (\mathsf{requestor} \vee \langle \mathsf{member} \rangle \mathsf{requestor})$$
$$\phi_6 = \phi_3 \vee \phi_5$$
$$\phi_7 = \langle \mathsf{register\text{-}ward} \rangle \mathsf{requestor}$$
$$\phi_8 = \langle \mathsf{register\text{-}ward} \rangle (\mathsf{requestor} \vee \langle \mathsf{ward\text{-}nurse} \rangle \mathsf{requestor})$$
$$\phi_9 = \phi_6 \vee \phi_8$$

The last candidate formula is basically a minor adaptation of the formula expressing the agency relation in [19, §5.2].

$$\phi_{10} = \langle \mathsf{gp} \rangle \mathsf{requestor} \vee \langle -\mathsf{agent} \rangle \langle \mathsf{gp} \rangle \mathsf{requestor}$$

D. AUTHORIZATION REQUESTS

We generated 400 methods with `all-of` guards, and another 400 with `one-of` guards. The set P of privileges for each guard contains a minimum of one and a maximum of three privileges randomly selected from the 200 available privileges (Fig. 2).[13] In addition to the privileges we randomly generated a list of 400 clinician, and a list of 400 participants, to serve as participants in the authorization requests.

The methods were invoked in order (from 1 to 400) along with the corresponding clinician, patient pair. This process was uniformly conducted for all configurations, with the *One* configurations invoking the methods with the `one-of` guards, and the *All* configurations invoking the methods with the `all-of` guards. The first 200 method invocations were discarded as they were used for warming up the Java Virtual Machine. The performance of the remaining 200 method invocations were recorded.

E. Neo4j WARMUP

Retrieving the neighbourhood of a node in Neo4J normally start out slow then speeds up, and stabilize at an average of 0.0001 seconds after approximately 250 queries. Therefore, we randomly generated 250 distinct neighbourhood retrieval queries that were ran before the method invocations for each test configuration.

[13] The service-layer methods of OpenMRS never have a privilege set of size larger three.