On the Role of Roles: from Role-Based to Role-Sensitive Access Control

Xuhui Ao
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
ao@cs.rutgers.edu

Naftaly H. Minsky
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
minsky@cs.rutgers.edu

ABSTRACT

This paper maintains that for an access-control mechanism to support a wide range of policies, it is best to dispense with any built-in semantics for roles in the mechanism itself, leaving such semantics to be defined by particular policies.

The validity of this assertion is demonstrated by showing that a mechanism called Law-governed interaction (LGI), which has no built-in concept of roles, can support a wide range of policies that take roles into account. These include RBAC itself, its various generalizations, as well as concepts like budgetary controls, which seems to be quite inconsistent with RBAC. All such policies can be formulated, deployed, and enforced, via a single scalable, and fully implemented LGI mechanism.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Controls; C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Security

Keywords

role-based access control, security, access control policy specification and decentralized enforcement, law-governed interaction

1. INTRODUCTION

In current access-control (AC) literature, the term *role* is often being used for the representation of the position a given individual holds in a given organization, say as a doctor, or a nurse, in a hospital. The importance of such roles for the formulation of access-control (AC) policies has been put forward by Sandhu and his colleagues in their seminal work on "role-based access control" (RBAC) model [17, 16]. This model views roles as sets of permissions, and defines an AC-policy essentially as a pair of mappings: a mapping of users to sets of roles, and a mapping of roles to sets of permissions. The RBAC model also provides for a hierarchical organization of roles, and includes certain constraints on the combinations of roles that a given individual can possess at one time.

The RBAC model has been widely adopted by the industry and been broadly accepted by the research community—not as the last word on access control, but mostly as a foundation on which more sophisticated AC models are to be built. Consequently, we have seen a host of RBAC-based models that generalized RBAC in various ways. These generalizations include, but are not limited to: (1) parameterization of the roles of a subject by its other attributes [8]; (2) making authorization dependent on some dynamic relationship between the subject, and the object on which it operates [4]; (3) making the right associated with the role dependent on the *context*, such as time and location [13]; and (4) providing for dynamic, and temporal, control over role-permission activation and deactivation, and over user-role assignment and activation (TRBAC [5] and GTRBAC [10]). Similar generalizations have also been proposed in [14], and in [19].

Yet, in spite of the success of each of these RBAC-based models, no unified model emerged from this body of work, only a collection of distinct models, each covering it own range of application, in its own way. The reason for this, we believe, is that RBAC, used as the foundation by all these models, is based too rigidly on roles—requiring *every* permission granted to a subject to emanate from some role it possesses—and it employs an overly narrow interpretation of roles, simply as collections of permissions. Indeed, each of the above cited papers identifies an important authorization pattern that does not fit the RBAC model, and each attempted to adjust RBAC so it would fit this pattern. But such adjustments are often overly complex, and sometime not possible, as is exemplified by the two common patterns of authorization: *delegation* and *budgetary control*, discussed briefly below.

Delegation is the act of giving a right one has for himself to somebody else. It is a very important element of access control, as it provides policies with a critical of dynamism. The mechanism for doing delegation is usually quite simple, as befits such an important and commonly used operation. Under the traditional capability-based AC, in particular, delegation is done simply by passing a capability to

SACMAT'04, June 2–4, 2004, Yorktown Heights, New York, USA.

the delegatee. Under many distributed access control mechanism, delegation is done by issuing a *delegation certificate* [3].

Under RBAC, on the other hand, since a permission must be mediated via a role, a delegation of a permission requires, according to [20], the creation of a temporary *delegation role*, which needs to be assigned to the delegatee, and then activated by him. Besides the artificiality of creating a role just for the purpose of delegation, this makes the management of roles more complex. This may have a seriously adverse impact on efficiency, if delegations are done frequently, and if they are done in a distributed manner.

Budgetary control is an age-old mechanism for regulating activities, common in the real world as well as in computer systems. As a simple example, one may be given a "query budget" which specifies the maximal number of queries one can make to a given database. We do not see how such dynamically changing permissions can be made to fit into the RBAC framework. Indeed, to our knowledge, none of the published RBAC-based models attempted to support budgetary control.

It is our thesis, that for an access-control mechanism to support a wide range of policies, it is best to dispense with any built-in semantics for roles in the mechanism itself—be it the semantics of RBAC, or any other—leaving such semantics to be defined by particular policies. In other words, an AC mechanism should be *sensitive to roles* but not based on them. The effect that roles should have on the granting of permissions to a subject should be left entirely to specific policies, and not be hard-wired into the mechanism itself.

We will attempt to demonstrate the validity of this thesis by showing how a mechanism called law-governed interaction (LGI), which has no built-in concept of roles, can support RBAC itself—all four "reference models" of it [17]—as well as its various published generalization. Moreover, we show how budgetary controls, which seems to be quite inconsistent with RBAC, can be handled under LGI. All this, simply by formulating suitable policies (called "laws" under LGI), which can be easily deployed and enforced via a single efficient, scalable, and fully implemented LGI mechanism.

The rest of this paper is organized as follows: We start, in Section 2, with a brief overview of LGI, in order to make this paper as self contained as possible. In Section 3 we demonstrate how miscellaneous semantics of roles can coexist in a single policy, and how such policy can be established via a very simple LGI law. In Section 4 we show how a policy that mixes roles with budgetary controls can be supported under LGI, and how it can be enforced in a scalable manner in a distributed context. In the above two section we employ no RBAC features. So, in Section 5 we show how the RBAC model itself can be formulated as a law under LGI, and how it can be implemented in a manner that has some advantages over its conventional implementations. Finally, in Section 6 we cite two other mechanisms, quite different from LGI in many ways, but which also validate our thesis by their ability to implement a wide range of role-sensitive policies, without recourse to the strict RBAC model. We will briefly point out some of the differences between these mechanisms and LGI. We conclude in Section 7.

2. LAW-GOVERNED INTERACTION (LGI)– A BRIEF OVERVIEW

Operations on the control-state	
`t@CS`	returns true if term `t` is present in the control state, and fails otherwise
`+t`	adds term `t` to the control state;
`-t`	removes term `t` from the control state;
`t1` ← `t2`	replaces term `t1` from the control state with term `t2`;
Operations on messages	
forward(x,m,y)	sends message `m` from `x` to `y`; triggers at `y` an `arrived(x,m,y)` event
deliver(x,m,y)	delivers the message `m` from `x` to `y`
Miscellaneous	
`t@L`	returns true if term `t` is present in list L, and fails otherwise
imposeObligation(oType,dt)	causes the triggering of an `obligationDue(oType)` event after time interval `dt`.

Figure 1: Some primitive operations in LGI.

LGI is a message-exchange mechanism that allows an *open group* of distributed agents to engage in a mode of interaction *governed* by an explicitly specified policy, called the *law* of the group. The messages thus exchanged under a given law $\mathcal{L}$ are called $\mathcal{L}$-messages, and the group of agents interacting via $\mathcal{L}$-messages is called an $\mathcal{L}$-community, denoted by $\mathcal{C}_\mathcal{L}$, or simply by $\mathcal{C}$. This mechanism has been originally proposed by one of the authors in 1991 [11], and then implemented, as described more completely than here in various publications, including [12, 1, 2].

By the phrase "open group" we mean (a) that the membership of this group (or, community) can change dynamically, and can be very large; and (b) that the members of a given community can be heterogeneous. In fact, we make here no assumptions about the structure and behavior of the agents that are members of a given community $\mathcal{C}_\mathcal{L}$, which might be software processes, written in arbitrary languages, or human beings. Both the *clients* and the *servers* of the traditional distributed system terminology are viewed here as agents. All the members are treated as black boxes by LGI, which deals only with the interaction between them via $\mathcal{L}$-messages, ensuring conformance to the law of the community. (Note that members of a community are not prohibited from non-LGI communication across the Internet, or from participation in other LGI-communities.)

For each agent `x` in a given $\mathcal{L}$-community, LGI maintains, what is called, the *control-state* $\mathcal{CS}_x$ of this agent. These control-states, which can change dynamically, subject to law $\mathcal{L}$, enable the law to make distinctions between agents, and to be sensitive to dynamic changes in their state. The semantics of control-states for a given community is defined by its law, could represent such things as the role of an agent in this community, and privileges and tokens it carries. For example, under law $\mathcal{PR}$ to be introduced in next section, the term `role(doctor)` in the control-state of an agent denotes that this agent has the role of doctor.

2.1 The nature of LGI laws:

An LGI law is defined over a certain types of events occurring at members of a community $\mathcal{C}$ subject to it, mandating

the effect that any such event should have. Such a mandate is called the *ruling* of the law for the given event. The events subject to laws, called *regulated events*, include (among others): the *sending* and the *arrival* of an $\mathcal{L}$-message, the *coming due of an obligation* previously imposed on a given object, and the *submission of a digital certificate*. The operations that can be included in the ruling for a given regulated event, called *primitive operations*, are all local with respect to the agent in which the event occurred (called, the "home agent"). They include, operations on the control-state of the home agent and operations on messages, such as `forward` and `deliver`. Our middleware currently provides two languages for writing laws: Java, and a somewhat restricted version of Prolog. We employ prolog in this paper. In this case, the law is defined by means of a Prolog-like program `L` which, when presented with a goal `e`, representing a regulated-event at a given agent `x`, is evaluated in the context of the control-state of this agent `cs`, producing the list of primitive-operations `r` representing the ruling of the law for this event. In addition to the standard types of Prolog goals, the body of a rule may contain two distinguished types of goals that have special roles to play in the interpretation of the law. These are the *sensor-goals* (in the form `t@CS`), which allow the law to "sense" the control-state of the home agent, and the *do-goals* (in the form `do(p)`) that contribute to the ruling of the law. A sample of primitive operations is presented in Figure 1.

2.2 The Concept of Enforced Obligation:

Informally speaking, an obligation under LGI is a kind of *motive force*. Once an obligation is imposed on an agent—generally, as part of the ruling of the law for some event at it—it ensures that a certain action (called *sanction*) is carried out at this agent, at a specified time in the future, when the obligation is said to *come due*, and provided that certain conditions on the control state of the agent are satisfied at that time. The circumstances under which an agent may incur an obligation, the treatment of pending obligations, and the nature of the sanctions, are all governed by the law of the community.

Specifically, an obligation can be imposed on a given agent `x` at time t_0 by the execution at `x` of a primitive operation

```
imposeObligation(oType,dt)
```

where `dt` is the time period, after which the obligation is to come due, and `oType`—the *obligation type*—is a term that identifies this obligation (not necessarily in a unique way). The main effect of this operation is that unless the specified obligation is *repealed* before its due time $t=t_0+dt$, the *regulated event*

```
obligationDue(oType)
```

would occur at agent `x` at time `t`. The occurrence of this event would cause the controller to carry out the ruling of the law for this event; this ruling is, thus, the *sanction* for this obligation. Note that a pending obligation incurred by agent `x` can be *repealed* before its due time by means of the primitive operation

```
repealObligation(oType)
```

carried out at `x`, as part of a ruling of some event.

For example, under law $\mathcal{PR}$ in Figure 3, when a doctor or nurse `y` is appointed by the manager to be on duty, an obligation `dutyExpired` is imposed on `y`, to come due at 12 hours after that appointment. When this obligation comes due, it will cause `y`'s duty to be dismissed.

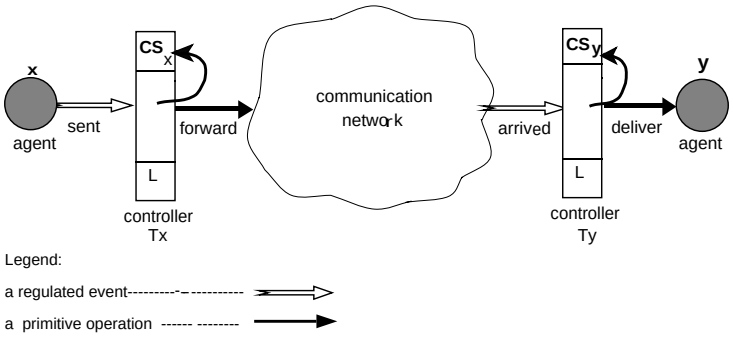

Figure 2: Enforcement of the law.

2.3 Distributed Law-Enforcement:

Broadly speaking, the law $\mathcal{L}$ of community $\mathcal{C}$ is enforced by a set of trusted agents called *controllers*, that mediate the exchange of $\mathcal{L}$-messages between members of $\mathcal{C}$. Every member `x` of $\mathcal{C}$ has a controller $\mathcal{T}_x$ assigned to it ($\mathcal{T}$ here stands for "trusted agent") which maintains the control-state $\mathcal{CS}_x$ of its client `x`. And all these controllers, which are logically placed between the members of $\mathcal{C}$ and the communications medium (as illustrated in Figure 2) carry the *same law* $\mathcal{L}$. Every exchange between a pair of agents `x` and `y` is thus mediated by *their* controllers $\mathcal{T}_x$ and $\mathcal{T}_y$, so that this enforcement is inherently decentralized. Although several agents can share a single controller, if such sharing is desired.

Controllers are *generic*, and can interpret and enforce any well formed law. A controller operates as an independent process, and it may be placed on any machine, anywhere in the network. We have implemented a *controller-service*, which maintains a set of active controllers. To be effective in a widely distributed enterprise, this set of controllers need to be well dispersed geographically, so that it would be possible to find controllers that are reasonably close to their prospective clients.

Finally, we point out that under the current implementation of LGI, a controller takes about 100 microseconds for every evaluation of a law of the size and complexity of laws such as the one introduced in Section 3.

2.4 On the basis for trust between members of a community:

For a member of an $\mathcal{L}$-community to trust its interlocutors to observe the same law, one needs the following assurances: (a) that the exchange of $\mathcal{L}$-messages is mediated by correctly implemented controllers; (b) that these controllers are interpreting the *same law* $\mathcal{L}$; and (c) that $\mathcal{L}$-messages are securely transmitted over the network. If these conditions are satisfied, then it follows that if `x` receives an $\mathcal{L}$-message from some `y`, this message must have been sent as an $\mathcal{L}$-message; in other words, that $\mathcal{L}$-messages cannot be forged.

These assurances are provided, broadly speaking, as follows: Controllers used for mediating the exchange of $\mathcal{L}$-messages authenticate themselves to each other via certificates signed by a certification authority specified by the value of the `ca` attribute in the `law` clause of law $\mathcal{L}$ (see, for example, Figure 3, in the case of law $\mathcal{PR}$). Note that

different laws may, thus, require different certification levels for the controllers used for its enforcement. Messages sent across the network are digitally signed by the sending controller, and the signature is verified by the receiving controller. To ensure that a message forwarded by a controller $\mathcal{T}_x$ under law $\mathcal{L}$ would be handled by another controller $\mathcal{T}_y$ operating under the *same* law, $\mathcal{T}_x$ appends a one-way hash [18] `H` of law $\mathcal{L}$ to the message it forwards to $\mathcal{T}_y$. $\mathcal{T}_y$ would accept this as a valid $\mathcal{L}$-message under $\mathcal{L}$ if and only if `H` is identical to the hash of its own law.

2.5 The deployment of LGI:

All one needs for the deployment of LGI is the availability of a set of trustworthy controllers, and a way for a prospective client to locate an available controller. This can be accomplished via one or more *controller-services*, each of which maintains a set of controllers, and one or more certification authorities that certifies the correctness of controllers. For an agent x to engage in LGI communication under a law $\mathcal{L}$, it needs to locate a controller, via a controller-service, and supply this controller with the law $\mathcal{L}$ it wants to employ. Once x is operating under law $\mathcal{L}$ it may need to distinguish itself as playing a certain role, or having a certain unique name, which would provide it with some distinct privileges under law $\mathcal{L}$. One can do this by presenting certain digital certificates to the controller. For the details of how to deal with the certificate, including its expiration and revocation in LGI, the reader is referred to [1].

2.6 The treatment of roles under LGI:

As we have pointed out, LGI has no built-in concept of role. But an LGI law may provide for the representation of roles, and may define their semantics. For instance under a given law $\mathcal{L}$ a term `role(doctor)` in the control-state of an agent x may signify that x is a doctor. It is important to realize that the LGI mechanism itself attaches no significance to this, or any other term. But law $\mathcal{L}$ may define how this term is acquired, and how it effects what its holder can do.

In this paper, for example, we use two techniques for an agent x to acquire its roles: First, by presenting a digital certificate with certain attributes, which the law in question accepts as a proof that x holds the specified role. This is the way roles are acquired under law $\mathcal{PR}$ introduced in Section 3. Second, by having another agent, operating under the same law, send a message to x appointing it to a certain role. This is the way roles are acquired under law $\mathcal{RB}$ of Section 5, where all users are assigned roles by the distinguished agent `r-admin`. In general, since the law defines the dynamic behavior of the control-state of an agent, under its interaction with other members of its community, it follows the law can define the dynamic behavior of those terms in the control-state that represent roles.

Moreover, as we have seen, the law of a community defines what an agent can, or must[1] do, depending, in part, on its control state. It follows, then that the law can determine the effect that role should have on the right, and even on the obligations, of an agent, in arbitrary way.

In summary, the law can define the process of acquiring roles, the dynamic behavior of these roles, and their effect on the ability of agents to operate. In other words, the law can define the semantics of roles, as we shall demonstrate in the following two sections.

[1]The "must" come from the concept of *obligation* in LGI .

3. SUPPORTING A MISCELLANY OF ROLE-SEMANTICS

We will show here how miscellaneous semantics of roles can coexist in a single policy, and how such policy can be established via a single mechanism—LGI, in this case—which does not have any built-in concept of role. For this end, we will formulate an example policy, called PR, that governs access to patient records in a hospital, and employs five roles: *doctor*, *nurse*, *manager*, *screening-nurse* and *pr-server* (for patient record server).

The PR policy is specified informally below. It will be restated later as an LGI-law, which can be readily deployed and enforced.

1. *An agent can acquire any of the above mentioned roles by presenting a digital certificate issued by a specific certification authority called* admin*, asserting that this agent has been appointed to the specified role. However, no agent can acquire more than one such role in one session.*

2. *A screening-nurse can appoint a doctor to be a patient's* attending-physician.

3. *A manager can appoint any doctor or nurse to be* on duty*, and this duty cannot last for more than 12 hours.*

4. *A doctor who is the attending-physician of patient y can delegate her/his access right to y's medical record to at most two nurses, who thus become the attending-nurses of y.*

5. *A subject x, who is a doctor or nurse, can access a patient y's medical record only if x is on duty and is the attending-physician or attending-nurse of y. And such access request can only be received by one who holds the role of a pr-server.*

A role, under this policy, is clearly not simply a collection of permissions. In particular, for a physician x to be able to access the record of a patient p stored on a server s, it is not enough for x to have a *doctor* role. He (or she) must also be appointed, by somebody playing the role of *screening-nurse*, as an *attending-physician* of p[2]; in addition, he must have been assigned for duty by somebody playing the role of *manager*; and he must not be on this duty for more than 12 hours. Moreover, the server s needs to have the *pr-server* role. This is quite a complex web of requirements that must be satisfied for one permission to be given.

In fact, we have designed this policy to feature a variety of role-semantics, gleaned from several different research papers, as follows: Point 1 of this policy is an example of dynamic separation of duties, as proposed by the RBAC model itself; Point 2 is an example of *appointment*, described in [19], and of *relationship*, described in [4]; Point 3 is an example of *temporal constraints*, as in TRBAC [5]; Point 4 is an example of *delegation* [20], and of *cardinality control* [7]; and Point 5 is an example of *context constraint* [14], and of *role of object* [13]. Note that we do not employ here the RBAC features of hierarchy of roles, nor do we distinguish

[2]Note that an *attending-physician* can be viewed as a relationship between the a doctor and a patient, although it can as easily be viewed as a role in its own right.

between the assignment of roles and their activation. But these features will be addressed in Section 5, when we implement the RBAC model itself under LGI.

This policy is formalized, in its entirety, as an LGI law $\mathcal{PR}$ displayed in Figure 3. Like all LGI laws, $\mathcal{PR}$ has two parts: First, is the *preamble* of the law, which, in this case, gives a name to this law, and the public key of the certification authority (CA) that is to be used for certifying the controllers interpreting this law. Its `authority` clause also specifies the public key of a CA—called *admin*—whose certification would be acceptable to it. The second is the *body* of the law, which is an ordered set of rules that defines a ruling for any regulated event that might occur at any agent. Each rule in this law is followed with an informal comments in italics. These comments, and the discussion below, should be sufficient to explain our technique, even for a reader not well versed with LGI laws.

By Rule $\mathcal{R}1$, an agent can claim a role `R`, getting the term `role(R)` into its control-state, by presenting a certificate, issued by *admin*, with this role as its attribute. And, as required by Point 1 of policy PR, only one such role can be claimed by an agent. Note, then, that the possession of any such role is represented here simply as a term in the control-state of agents.

Rules $\mathcal{R}2$ and $\mathcal{R}3$ allow the screening nurse to assign the patient identified by Pid to the doctor, who thus becomes the *attending-physician* of Pid.

By Rules $\mathcal{R}4$ and $\mathcal{R}5$, a manager can appoint other agents (specifically, doctors and nurses) to be *on duty*. However, by Rule $\mathcal{R}5$, by starting such a duty, one incurs an obligation to end this duty automatically, after 12 hours, as specified by Rule $\mathcal{R}6$. (*obligation* is an important proactive mechanism of LGI).

By Rules $\mathcal{R}7$ and $\mathcal{R}8$, a doctor x who is the attending-physician of patient Pid can delegate its access right to the medical records of Pid, to a nurse by appointing him/her as an attending-nurse of Pid; and he can thus appoint no more than two nurses, for any patient.

Note that ease and simplicity of delegation under this policy, compared to the way delegation is to be done in RBAC [20]. Note, also, that the cardinality-control over delegation, illustrated by this policy, can't be enforced efficiently via RBAC model in the distributed environment. This is because, it requires one to maintain the state of the delegator, which may change dynamically. This is, in fact, just one example of a wide range of dynamic and stateful policies, which are hard to enforce efficiently over distributed systems, unless the enforcement mechanism itself is decentralized, as it is under LGI. We will see a clearer example of such a dynamic policy in the following section.

Finally, by Rules $\mathcal{R}9$ and $\mathcal{R}10$, an agent x can send the access request for the record of patient Pid to a server s, only if: (1) x has the role of doctor or nurse (represented via the term `role(doctor)` or `role(nurse)` in its control state); (2) x is the attending-physician or attending-nurse of y (represented via the term `attending-physician(Pid)` or `attending-nurse(Pid)` in its CS); (3) x is on-duty (represented via the term `onDuty` in its CS); and (4) the server s has the role of pr-server (represented via the term

```
role(pr-server)
```

in its CS).

$\mathcal{P}$reamble:

```
law(name(PR),ca(publicKey1)).
authority(admin, publicKey2).
```

$\mathcal{R}1$.
```
certified(issuer(admin),subject(Self),
      attributes([role(R)])) :-
      not(role(R1)@CS),do(+role(R)).
```
Allow an agent to claim and activate its role by presenting a certificate issued by the admin, only if it hasn't activated any other roles in this session.

$\mathcal{R}2$.
```
sent(X,assign-patient(Pid),Y)
  :- role(screening-nurse)@CS, do(forward).
```
Only the screening nurse can assign the patient to the doctors.

$\mathcal{R}3$.
```
arrived(X,assign-patient(Pid),Y)
  :- role(doctor)@CS,
     do(+attending-physician(Pid)), do(deliver).
```
The arrivial of the patient assignment will be recorded into the doctor's control state.

$\mathcal{R}4$.
```
sent(X,onDuty,Y)
  :- role(manager)@CS, do(forward).
```
The manager can appoint any agent to be on-duty.

$\mathcal{R}5$.
```
arrived(X,onDuty,Y) :- do(+onDuty),
      do(imposeObligation(dutyExpired,[12,hour])).
```
The duty will expire after 12 hours.

$\mathcal{R}6$.
```
obligationDue(dutyExpired) :- do(-onDuty).
```
After the obligation dutyExpired fires, the onDuty term will be removed from the agent's CS.

$\mathcal{R}7$.
```
sent(X,delegate(Pid),Y) :- role(doctor)@CS,
      attending-physician(Pid)@CS,
      if (not(deleNo(Pid,N)@CS)) then
      (do(+deleNo(Pid,1)), do(forward))
      else (N<2, do(deleNo(Pid,N)<-deleNo(Pid,
      N+1)), do(forward)).
```
The attending doctor `X` can delegate its access right on its patients to the nurse, and for each patient, the doctor can't delegate its access right to more than 2 nurses.

$\mathcal{R}8$.
```
arrived(X,delegate(Pid),Y) :- role(nurse)@CS,
      do(+attending-nurse(Pid)), do(deliver).
```
The nurse Y becomes the attending nurse of patient Pid after delegated by the attending doctor X of Pid.

$\mathcal{R}9$.
```
sent(C,access(Op,m-record(Pid)),S)
  :- ((role(doctor)@CS,
      attending-physician(Pid)@CS);
      (role(nurse)@CS,attending-nurse(Pid)@CS)),
      onDuty@CS, do(forward).
```
The sending of the access request to the medical record of patient Pid will be authorized if it is from its on-duty attending-physician doctor or attending-nurse.

$\mathcal{R}10$.
```
arrived(C,access(Op,m-record(Pid)),S)
  :- role(pr-server)@CS,do(deliver).
```
The access request to patient's medical records can only be delivered to pr-servers.

Figure 3: Law $\mathcal{PR}$

4. DEALING WITH HIGHLY DYNAMIC AUTHORIZATION

Consider a hospital, whose management decided that some of its internal services–such as drug acquisition (from internal pharmacies), printing, file-services, databases, etc.– be paid with internal service currency (called *S-currency*), made available to various agents, in what we call their *budgets*. More specifically such services are to be governed by the following, informally stated, policy (called BC, for "budgetary control").

4.1 The Budgetary Control (BC) Policy:

We employ here three roles, allowing a single agent to hold no more than one of them at a time. These roles are: (a) *budget-officer* who would be allowed to grant S-currency budgets to agents; (b) *pharmacy*, who would be allowed to sell drugs; and (c) *doctors*, who would be allowed to purchase drugs. More specifically, this policy can be defined by the following four points:

1. *Roles are to be acquired by presenting appropriates certificates, signed by a specified certification authority (CA)—called* admin.
2. *S-currency can be granted by the budget-officer, into the budgets of various agents; and it can be moved from the budget of one agent to that of another.*
3. *Each service request must be paid off from the budget of the requester.*
4. *Drugs can be purchased, with S-currency, only by* doctors*, and only from servers playing the role of* pharmacy.

4.2 The Budgetary Control Law $\mathcal{BC}$:

Policy BC is implemented as law $\mathcal{BC}$ displayed in Figure 4. Rule $\mathcal{R}1$ of this law allows an agent operating under this law to acquire one role, by presenting an appropriate certificate. But once one role is thus acquired, the agent would be prevented from acquiring another one. This is identical to what we have done under law $\mathcal{PR}$.

By Rule $\mathcal{R}2$, a budget officer x can send a message

```
grantBudget(B)
```

to any agent y, which by Rule $\mathcal{R}3$, will be added the term `budget(B)` of the CS of y, upon its arrival.

Note that the term `budget(B)` in the CS of agents represents, under this law, what we have called S-currency, i.e., currency that can be used for paying servers for their services. Rules $\mathcal{R}4$ and Rule $\mathcal{R}5$ allow such currencies to be *moved* from one agent to another, via the

```
giveCurrency(B)
```

message. And note that this law provides no other means for effecting budgets, which means that S-currencies cannot be forged.

By Rule $\mathcal{R}6$, an agent can issue a service request

```
request(S,P)
```

for a service S, including payment P. This request will be forwarded only if payment P doesn't exceed the current budget of that sender; and, in case that the request is for purchasing a drug, if the sender has the role of a doctor. If the request is forwarded, then the corresponding payment will be deducted from the sender's budget. Finally, by Rule $\mathcal{R}7$, if the request is a drug purchase request, then only the agents playing the role of pharmacies is allowed to receive it. Other service requests received will be delivered to the receiver and the payment will be added into the budget of that receiver.

Preamble:

law(name($\mathcal{BC}$),ca(publicKey1)).
authority(admin, publicKey2).

$\mathcal{R}1$.
```
certified(issuer(admin),subject(Self),
     attributes([role(R)])) :-
     not(role(R1)@CS),do(+role(R)).
```
Allow an agent to claim and activate its role by presenting a certificate issued by the admin, only if it hasn't activated any other roles in this session.

$\mathcal{R}2$.
```
sent(X,grantBudget(B),Y)
  :- role(budgetOfficer)@CS, do(forward).
```
Only the budget officer can assign the inital budget to the agents.

$\mathcal{R}3$.
```
arrived(X,grantBudget(B),Y) :- do(+budget(B)).
```
The assigned budget will be recorded as term `budget(B)` *in the control state of the receiver.*

$\mathcal{R}4$.
```
sent(X,giveCurrency(B1),Y)
  :- budget(B)@CS, B>=B1, do(decr(budget(B),B1)),
     do(forward).
```
Any one can give part of its S-currency to others.

$\mathcal{R}5$.
```
arrived(X,giveCurrency(B1),Y) :- budget(B)@CS,
     do(incr(budget(B),B1)), do(deliver).
```
The S-currency given by others will be added into the the receiver's account.

$\mathcal{R}6$.
```
sent(X,request(Service,Payment),Y) :- (if
     (Service==drugPurchase) then role(doctor)@CS
     else true), budget(B)@CS, B>=Payment,
     do(decr(budget(B),Payment)), do(forward).
```
The service request must carry a payment in the form of S-currency, which is deducted from the sender's budget. Furthermore, the drug purchase requests can only be issued by the doctors.

$\mathcal{R}7$.
```
arrived(X,request(Service,Payment),Y)
  :- (if (Service==drugPurchase) then
     role(pharmacy)@CS else true), budget(B)@CS,
     do(incr(budget(B),Payment)), do(deliver).
```
The arrival of a service request increases the budget of the server by the payment. Furthermore, the drug purchase requests can only be received by the pharmacies.

Figure 4: Law $\mathcal{BC}$

4.3 Discussion:

Note that this policy is sensitive to roles, but it is also sensitive to the dynamically changing budget of whoever requests a service. We do not know of any reasonable way for modeling such a budgetary control via roles.

Moreover, as we have explained in [2], such a dynamic policy cannot be implemented scalably under an AC mechanism that employs a centralized, even if replicated, refer-

ence monitor. The implementation of this policy under LGI is efficient and scalable, precisely because LGI is inherently decentralized.

5. SUPPORTING STANDARD RBAC IN LGI

Here we show how LGI can support the standard RBAC model, including: the user-role assignment, user-role activation constraints that establish static and dynamic separation of duties, and the role hierarchy—all these via a single LGI law.

Both Ferraiolo et al. [7] and Park et al. [15] present the implementation of RBAC in distributed systems, e.g., in enterprise web servers. [7] assumes that the web server can get the user-role mapping information from its local RBAC database. In contrast, [15] assumes that the role server which hosts the user-role assignment information may be different from the web server, so one needs to solve the problem of how to secure the distribution of the user-role assignment information from the role server to the Web server. [15] proposes user-pull and server-pull modes, and implements them by integrating and extending cookies, X.509 certificate, SSL and LDAP techniques.

Like the RBAC implementation by Ferraiolo et al. [7] and Park et al. [15], we assume the existence of several servers (such as web-servers) operating within a single administrative domain, such as an enterprise. We also assume that each server specifies the role-permission mapping for its own objects; and that the role-hierarchy, the separation of duty constraints, and the user-role assignments are to be defined globally for the entire administrative domain. As to the user-role assignment information, similar to Park et al. [15], we assume the server (say, Web server) can't get it locally and those information needs to be pulled by the user from the centralized role administrator agent called *r-admin*. The secure distribution of those information is ensured by LGI's secure communication among agents and controllers.

We point out, however, that in one sense our implementation of RBAC is stronger than both [7] and [15]. The implementation of RBAC in [7] and [15], cannot enforce dynamic separation of duty constraints over users of several different servers. Here is how this limitation is explained in [7]:

> "Note that because the RBAC/Web implementation applies only to a single server, dynamic separation of duties is not implemented relative to users across multiple servers (e.g., in an environment where a collection of servers constitutes an administrative security domain), but only relative to a user on a single web server."

As we shall see later, our implementation of RBAC does not have this limitation.

5.1 The LGI Law that Implements RBAC

All the provisions of RBAC can be defined under LGI by law $\mathcal{RB}$, displayed in Figure 5 and 6 in its entirety. Before we get to a detailed discussion of this law, here are some broad comments about it. First, we use two central components: (a) a CA called `idCA` that issues identity-certificates for the various users, and which does not, itself, operate under LGI; and (b) an agent called `r-admin`, that maintains the global user-role mapping, and which would provide the various users (clients) with their static role assignment (this agent does operate under our law $\mathcal{RB}$). Second, the role hierarchy, and the static and dynamic constraints on role assignment and activation, are defined by law $\mathcal{RB}$ itself, and would thus be present in the control state of every client and server operating under this law. We are now in a position for taking a closer look at law $\mathcal{RB}$ itself.

Preamble:

```
law(name(RB),ca(publicKey1)).
authority(idCA,publicKey2).
alias(r-admin,"r-admin@enterprise.com").
initialCS([static-conflict([conflict(sr1,sr2),...])]).
initialCS([dynamic-conflict([conflict(dr1,dr2),...])]).
initialCS([role-hierarchy([senior(r3,r1),...])]).
```

$\mathcal{R}1$.
```
certified(issuer(idCA),subject(Self),
      attributes([myId(Id)])) :- do(+myId(Id))
```

Allow a client to establish its ID by presenting a certificate issued by `idCA` *with the attribute of* `myId`.

$\mathcal{R}2$.
```
sent(r-admin,assigned-roles(Roles,Id),C)
   :- static-conflict(SC-roles)@CS,
      not(is-conflict(SC-roles,Roles)), do(forward).
```

Make sure the role assignment conforms to the SSoD requirement.

$\mathcal{R}3$.
```
arrived(r-admin,assigned-roles(Roles,Id),C)
   :- myId(Id)@CS, do(+assigned-roles(Roles)),
      do(+activated-roles([])), do(deliver).
```

The arrivial of the role assignment.

$\mathcal{R}4$.
```
sent(C,activate-role(R),Self)
   :- assigned-roles(Roles)@CS,
      junior-member(R,Roles),activated-roles(ARS)@CS,
      dynamic-conflict(DC-roles)@CS,
      not(is-conflict(DC-roles,[R|ARS])),
      do(activated-roles(ARS) <-
      activated-roles([R|ARS])).
```

A client can choose to activate the role `R`*, which has been assigned to it or is junior to any assigned ones, under the restriction of the DSoD constraint.*

$\mathcal{R}5$.
```
sent(S,permission(R,OP,Obj),Self)
   :- do(+permission(R,OP,Obj)).
```

A server can assign the permissions of accessing its objects to different roles.

Figure 5: Law $\mathcal{RB}$

The `Preamble` of this law has the following clauses: The `law` clause specifies the name of this law, and the public key of the CA that is to be used for certifying the controllers interpreting this law. The `authority` clause specifies the public key of a CA—called *idCA*—whose certification would be accepted by this law for the authentication of the identities of the clients. The `alias` clause specifies the LGI-address of the *r-admin* agent, who is responsible for assigning roles to the clients (this is just for notational convenience). Finally, there are three `initialCS` clauses that contain terms to be included in the initial control state of every agent that adopts this law. These terms define the structure of the role ensemble at hand, as follows: (1) the term `static-conflict([...])` contains the list of role pairs that conflict statically; (2) the term `dynamic-conflict([...])` contains the list of role pairs that conflict dynamically; and (3) the term `role-hierarchy([...])`, contains the list of role pairs which has the direct senior-junior relations in the role hierarchy.

We discuss the body of the law by describing its treatment of the following issues: (a) authentication of the identity of clients (or subjects); (b) role assignment; (c) role activation;

(d) the specification, by individual servers, of their mapping of roles to permissions (i.e., their ACL); and (e) access request authorization.

5.1.1 Identity authentication:

Rule $\mathcal{R}1$ of this law allows a client to establish its identity by presenting an identity certificate issued by the CA called `idCA` in the preamble. The certified identifier would be recorded by the term `myId` in the control-state of this agent. This identity would be required for the client to be assigned roles, and thus to access any server.

5.1.2 Role Assignment:

By Rule $\mathcal{R}2$, the distinguished agent `r-admin` can send to a client c identified by Id, his role assignment `Roles`, subject to the static separation of duty constraint. This constraint is checked by the predicate `is-conflict(SC-roles,Roles)`, where `SC-roles` is the list of static conflict role pairs, maintained in the control-state of every agent under this law. The `is-conflict` predicate is defined, recursively, by Rules $\mathcal{R}9$ and $\mathcal{R}8$. It returns true if any two roles in the role list `Roles` conflict with regard of conflict role pair list `SC-roles`, taking into account the role hierarchy defined by `role-hierarchy` terms in the control state of every agent.

By Rule $\mathcal{R}3$, the arrival of the assigned roles to the client c with identity Id, will add two terms `assigned-roles(Roles))` and `activated-roles([])` into its control state, which represent the assigned roles of c, and its currently empty list of activated roles, respectively.

5.1.3 Role Activation:

By Rule $\mathcal{R}4$ a client c can activate any of its assigned role R, if the following two conditions are satisfied. First, if R is one of the assigned roles of c, as recorded in its `assigned-roles(Roles)` term, or if R is junior to one of the assigned roles in `Roles`, as checked by the predicate `junior-member(R,Roles)` defined by Rules $\mathcal{R}10$ and $\mathcal{R}11$. Second, if activation of R will not violate the constraint of the dynamic separation of duty. This constraint is checked by the predicate `is-conflict(DC-roles,[R|ARS])`, defined by Rules $\mathcal{R}9$ and $\mathcal{R}8$, where `DC-roles` is the list of dynamically conflict role pairs, and `ARS` is the list of roles that c have already activated in this session. Predicate `is-conflict(DC-roles,[R|ARS])` returns true if there exist any conflict roles in the list `[R|ARS]`, with respect to the conflict role pairs defined by `DC-roles` and the role hierarchy defined by `role-hierarchy` terms in c's control state. If both conditions above are satisfied, the R will be activated by being added into the argument of the term `activated-roles` in c's control state.

5.1.4 Mapping Roles to Permissions:

Each server can decide which kind of permissions it wants to associate with different roles. This it can do by sending the message `permission(Role,Op,Obj)` to itself. By Rule $\mathcal{R}5$, this message would be inserted as a term in the control-state of this server, indicating that the role `Role` has the permission of action `Op` on object `Obj` of that server.

5.1.5 Access Request Authorization:

By Rule $\mathcal{R}6$, a client c can send the access request

```
access(Op,Obj)
```

to any server. This rule would attach the term `myRoles(ARS)` to that message, where `ARS` is the list of activated roles of the sender. When the access request arrives at the server side, it will, by Rule $\mathcal{R}7$, be authorized only if any of the activated roles of the sender has the permission of that operation on that object. This is determined by the predicate `has-permission(ARS,Op,Obj)`, whose evaluation is carried out by Rules $\mathcal{R}12$, $\mathcal{R}13$ and $\mathcal{R}14$, which evaluates to true if any role in role list `ARS` or any role junior to the role in `ARS` has been assigned the permission `(Op,Obj)`.

$\mathcal{R}6$.
```
sent(C,access(Op,Obj),S) :- activated-roles(ARS)@CS,
      do(forward(C,access(Op,Obj,myRoles(ARS)),S)).
```
The access request sent from a client will carry its current activated roles `ARS`.

$\mathcal{R}7$.
```
arrived(C,access(Op,Obj,myRoles(ARS)),S)
   :- has-permission(ARS,Op,Obj),
      do(deliver(C,access(Op,Obj),S)).
```
The access request carrying the client's activated roles will be authorized, if that access permission is included in the privileges of its activated roles.

$\mathcal{R}8$.
```
is-conflict(conflict(C-role1,C-role2),Roles)
   :- junior-member(C-role1,Roles),
      junior-member(C-role2,Roles).
```

$\mathcal{R}9$.
```
is-conflict([H|T],Roles)
   :- is-conflict(H,Roles); is-conflict(T,Roles).
```
The above two rules check whether the role list `Roles` *violates the separation of duty constraint specified by the first argument conflict role pair list.*

$\mathcal{R}10$.
```
junior-member(Role,[H-role|T-roles])
   :- junior-equal(Role,H-role);junior-member(Role,
      T-roles).
```

$\mathcal{R}11$.
```
junior-equal(Role,H-role)
   :- Role==H-role; role-hierarchy(SList)@CS,
      senior(H-role,Role)@SList.
```
The above two rules check whether the `Role` *is a member of, or is junior to any member of the second argument role list.*

$\mathcal{R}12$.
```
has-permission([H-role|T-roles],Op,Obj)
   :- has-permisssion(H-role,Op,
      Obj);has-permission(T-roles,Op,Obj).
```

$\mathcal{R}13$.
```
has-permission(Role,Op,Obj)
   :- permission(Role,Op,Obj)@CS.
```

$\mathcal{R}14$.
```
has-permission(Role,Op,Obj) :- role-hierarchy(SList)@CS,
      senior(Role,Junior-Role)@SList,
      has-permission(Junior-Role,Op,Obj).
```
The above three rules check whether the permission `(Op,Obj)` *is assigned to the first argument role list, considering the role hierarchy relations.*

Figure 6: Law $\mathcal{RB}$–continue

5.2 Discussion

Let us explain now why our implementation of RBAC supports dynamic separation of duties even in a multi-server context—which the traditional implementation of RBAC in [7] and [15] was not able to do. This is basically because this constraint is enforced locally at the client side, via Rule $\mathcal{R}4$, by the controller associated with it. Specifically, once a client c activated one of its assigned role, say `r1`, and used that role to access the objects in server $s1$, c can't activate any other roles that dynamically conflicts with `r1`, no matter which other server it tries to access.

6. RELATED WORK

We have already cited several papers about the RBAC model, and about its various extensions. Here we cite two papers—by Bertino et al. [6] and by Jajodia et al. [9]—that introduce wide spectrum access control mechanisms that provide for the implementation of a wide range of role-sensitive policies, without recourse to the strict RBAC model. We view these papers as providing further validation of our thesis that the semantics of roles should not be hard wired into the AC mechanism.

There are numerous differences between the above two mechanism and LGI. We will mention here just one. The mechanisms introduced in [6, 9] have been designed to be enforced in centralized manner. And as we have argued in [2], such an enforcement cannot be done scalably for highly dynamic policies, like our budgetary control policy presented in Section 4. LGI, on the other hand, has been designed for decentralized enforcement, and it does support scalable implementation of such policies.

7. CONCLUSION

The thesis of this paper has been that although an access control mechanism should be sensitive to roles, it should not be based on them. This, for two reasons. First, roles can effect authorization in too many ways to be captured by any particular semantics, such as that of RBAC. We have suggested, therefore, that the semantics of roles in any particular application is best left to the policy governing it. The second reason for an access control mechanism not to be role-*based*, is that there are aspects of access control policies that have nothing to do with roles. Budgetary controls are a case in point.

We have demonstrated the validity of this view by showing that LGI, which has no built-in concept of roles, can support a wide range of policies that take roles into account. These include RBAC itself, its various generalizations, as well as concepts like budgetary controls, which seems to be quite inconsistent with RBAC. All such policies can be formulated, deployed, and enforced, via a single scalable, and fully implemented LGI mechanism. Moreover, we believe that our thesis is equally well validated by the work of Bertino et al. [6] and Jajodia et al. [9].

8. REFERENCES

[1] X. Ao, N. Minsky, and V. Ungureanu. Formal treatment of certificate revocation under communal access control. In *Proc. of the 2001 IEEE Symposium on Security and Privacy, May 2001, Oakland California*, pages 116–127, May 2001.

[2] X. Ao and N. H. Minsky. Flexible regulation of distributed coalitions. In *LNCS 2808: the Proc. of the 8th European Symposium on Research in Computer Security (ESORICS) 2003*, pages 39–60, October 2003.

[3] T. Aura. Distributed access-rights management with delegation certificates. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*, pages 211–235. Springer, 1999.

[4] J. Barkley, K. Beznosov, and J. Uppal. Supporting relationships in access control using role based access control. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control*, pages 55–65, October 1999.

[5] E. Bertino, P. A. Bonatti, and E. Ferrari. Trbac: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3):191–233, 2001.

[6] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In *Proc. of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, Monterey, California*, pages 116–127, June 2002.

[7] D. Ferraiolo, J. Barkley, and R. Kuhn. A role based access control model and reference implementation within a corporate intranets. *ACM Transactions on Information and System Security*, 2(1), February 1999.

[8] L. Giuri and P. Iglio. Role templates for content-based access control. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 153–159, 1997.

[9] S. jajodia, P. Samarati, M. L. Sapino, and V. S. Subramanian. Flexible support for multiple access control policies. *ACM Trans. on Database Systems*, 26(2):214–260, June 2001.

[10] J. Joshi, E. Bertino, B. Sahfiq, and A. Ghafoor. Dependencies and separation of duty constraints in gtrbac. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, 2003.

[11] N. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, Feb. 1991.

[12] N. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.

[13] M. Moyer and M. Abamad. Generalized role-based access control. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 391–398, 2001.

[14] G. Neumann and M. Strembeck. An approach to engineer and enforce context constraints in an rbac environment. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, 2003.

[15] J. S. Park, R. Sandhu, and G.-J. Ahn. Role-based access control on the web. *ACM Transactions on Information and System Security*, 4(1):37–71, 2001.

[16] R. Sandhu, D. Ferraiolo, and R. Kuhn. The nist model for role-based access control: Towards a unified standard. In *Proceedings of ACM Workshop on Role-Based Access Control*. ACM, July 2000.

[17] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[18] B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.

[19] W. Yao, K. Moody, and J. Bacon. A model of oasis role-based access control and its support of active security. *ACM Transactions on Information and System Security*, 5(4):492–540, 2002.

[20] X. Zhang, S. Oh, and R. Sandhu. Pbdm: A flexible delegation model in rbac. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, 2003.