Enforcing Enterprise-wide Policies Over Standard Client-Server Interactions

Zhijun He, Tuan Phan, Thu D. Nguyen

{*zhijun, tphan, tdnguyen*}*@cs.rutgers.edu*

Department of Computer Science, Rutgers University, NJ 08854

Abstract. *We propose and evaluate a novel framework for enforcing global coordination and control policies over interacting software components in enterprise computing environments. This framework combines a per-node reference monitor with two existing coordination and control systems to enforce policies that, among other properties, are stateful and communal. Each reference monitor filters messages exchanged between the interacting software components similar to a firewall, passing only messages that are allowed by the policies in effect. This filtering approach decouples coordination and control from application implementation, allowing the coordination and control mechanism and application implementations to evolve independently of each other. We demonstrate the power of our framework by using it to specify and enforce an RBAC policy with delegation, revocation, and separation-of-duty over accesses to a cluster of NFS and SMB file servers without changing any client or server implementations. Measurements show that our framework imposes acceptable overheads when enforcing this policy.*

1. Introduction

Today's enterprise computing environments are increasingly comprised of heterogeneous components that were not constructed according to any single system design. Rather, independently constructed components are composed under the assumption that they will interoperate harmoniously because they implement standardized interaction protocols. Unfortunately, these components often do not work well together despite their adherence to standards because: (1) standards often define the mechanisms for interaction but provide little support for governing interaction, and (2) standards always leave some implementation details unspecified and so different implementations of the same standard are not always fully compatible. To exacerbate the problem, as personal devices proliferate, many organizations are being forced toward a decentralized control model, where shared infrastructures are maintained by professional staffs but personal devices are managed by their individual owners. Thus, even the composition of the enterprise system may not be well defined and understood. Needless to say, these systems are unreliable, insecure, and difficult to use and maintain [8, 14, 18].

A number of research efforts have sought to address the above problem by introducing mechanisms to enforce explicit policies that coordinate and control the activities of the disparate components within such environments, e.g., [4, 7, 11, 15]. Unfortunately, many of these mechanisms require the modification of existing software, which limits their applicability and slows their adoption. In this paper, we propose a novel approach for applying these mechanisms to software components that interact using standardized message passing protocols without requiring any changes to the applications. Our approach also allows new applications to be developed that are neutral with respect to specific coordination and control frameworks, effectively decoupling coordination and control from application implementation.

Specifically, we use a per-node reference monitor as a transparent "hook" for enforcing enterprise-wide policies over the message exchanges between interacting software components. Our reference monitor leverages the firewall capabilities prevalent in today's operating systems to intercept all packets sent to/from a node and a packet analyzer to reconstruct protocol-level messages that should be subjected to enterprise-wide policies. We then combine our reference monitor with two existing complementary systems, Law Governed Interaction (LGI) [11] and KeyNote [4], to implement an overall coordination and control framework.

We give an overview of our framework and explain how the three components, the per-node reference monitor, LGI, and KeyNote, fit together in Section 3, after we have presented an example policy in Section 2 to provide concrete context for the description. For now, we observe that the resulting mechanism can be used to explicitly specify and enforce policies that, among other properties, are *stateful* and *communal*. Stateful policies are those that are sensitive to the history of interactions. Example stateful policies include those encapsulating coordination principles such as delegation and revocation, as well as the Chinese Wall policy [5]. Communal policies provide for control decisions to be based on state accumulated across a community of interacting components, rather than just state accumulated at each individual component.

We demonstrate the power of our framework by using it to coordinate and control accesses to a set of file systems. Access control is a good representative problem because while a large body of research has accumulated to argue the importance of principles such as delegation and revocation [4, 10, 20], separation of duty, and auditing [16], most existing access control mechanisms are still based on the

access control matrix model, where large parts of the matrix are manually maintained by system administrators. For example, in UNIX, a simple coordination problem such as sharing a set of files between two people, say a professor and his TA, requires the intervention of a system administrator to set up and later remove a group. This can often lead to inappropriate control that can adversely affect the security of the system, e.g., the professor making his files world-accessible to share with his TA or the group remaining defined long after the interaction has ended.

Further, in today's client-server applications, access control is typically established by individual servers, where the policy and its enforcement are embedded in the code of the server. This *server-centric* approach has numerous disadvantages, including: (1) the difficulty of changing the policy and evolving the access control mechanism; (2) the lack of uniformity in what policies are enforced and enforceable by different servers; and (3) the inability to enforce communal policies—the revocation of access rights provides a ready example where communal access control is needed: it is often easier to detect client misbehavior by observing a client's interactions with multiple servers as opposed to individual pair-wise client/server interactions.

Thus, we describe how our framework can be used to specify and uniformly enforce the policy introduced in Section 2, which is a role-based access control (RBAC) policy with delegation, revocation, and separation-of-duty, across a community of NFS and SMB (SAMBA) file servers. We also quantify the performance overheads imposed by our framework, showing that the enforcement of our policy can incur non-trivial but (in our opinion) acceptable overheads.

In summary, our contributions include: (1) defining a framework for enforcing sophisticated coordination and control policies over communities of interacting software components that works external to the implementations of these components; this decoupling of coordination and control from application implementations will help to migrate advance coordination and control policies and mechanisms from the realm of research to practice; (2) demonstrating that our framework can be applied to practical client/server protocols such as NFS and SMB; and (3) showing that our framework can specify and enforce policies encompassing principles that have been deemed critical to enterprise security with acceptable overheads, even when applied to performance sensitive services such as NFS and SMB.

2. Motivating Example

We begin by describing an example policy for coordinating and controlling accesses to a set of file servers to concretely motivate the type of policies that we seek to support. We will show later how our mechanism can enforce this policy over accesses to a set of NFS and SMB (SAMBA) file servers *without requiring any changes to the client and server applications*.

To introduce our example policy, let us suppose that an enterprise has a set of registered principals (users) U, a set of roles R, and a relation $M : U \rightarrow R$ defining the roles that each principal can assume for access control purposes. Then, given a set of file servers, the enterprise might wish to establish a policy $\mathcal{P}_E$ with the following stipulations:

Authentication: To enable access from a client machine, a principal must: (1) present a certificate signed by the distinguished certification authority *ca* to authenticate that it is the owner of a public key K^{pub}, and (2) present a certificate signed by the private key paired with K^{pub} stating that it is running with uid U on the client machine with ip address I. An authenticated registered principal gains immediate access under all roles that it can assume. An unregistered principal, e.g., a guest, can also authenticate but does not have any access rights until it gains some delegated rights. This authentication must be renewed hourly.

Delegation: A principal p can gain delegated rights by presenting a certificate signed by *ca*, specifying that some registered principal p_d has delegated certain access rights to p under a role R_d. A registered principal p_d may delegate rights under any role R_d where $R_d \neq$ *Admin* and $(p_d, R_d) \in M$. The delegated rights may optionally have an expiration time and be limited to a file f or directory d, with the latter implying that the delegated rights can be used to access all files and directories under d.

Revocation: X access denials across any subset of servers within an hour will lead to suspension of all delegated rights for a particular principal. Y access denials within an hour will lead to suspension of all access rights for a principal. Suspended privileges can be reactivated through the presentation of two reactivation certificates signed by *ca*, specifying that reactivation permission has been obtained from two distinct principals who can assume the role *Admin*.

Per-file policies: The owner of each file can establish a per-file policy[1] specifying the roles that can access that file and their access rights.

While the above policy has been deliberately made simple for ease of presentation, it still demonstrates several important elements for establishing powerful yet flexible access control policies. First, its support for delegation empowers users to coordinate accesses to their files more flexibly yet securely without having to involve system administrators. For example, a professor can allow his TA to access course-related files by simply delegating access rights to an appropriate directory. Further, the professor can specify a

---

[1]The overload of the term policy here is somewhat confusing. Throughout our discussion, we always use the per-file prefix whenever there is any potential for confusion between the global policy and the per-file policies.
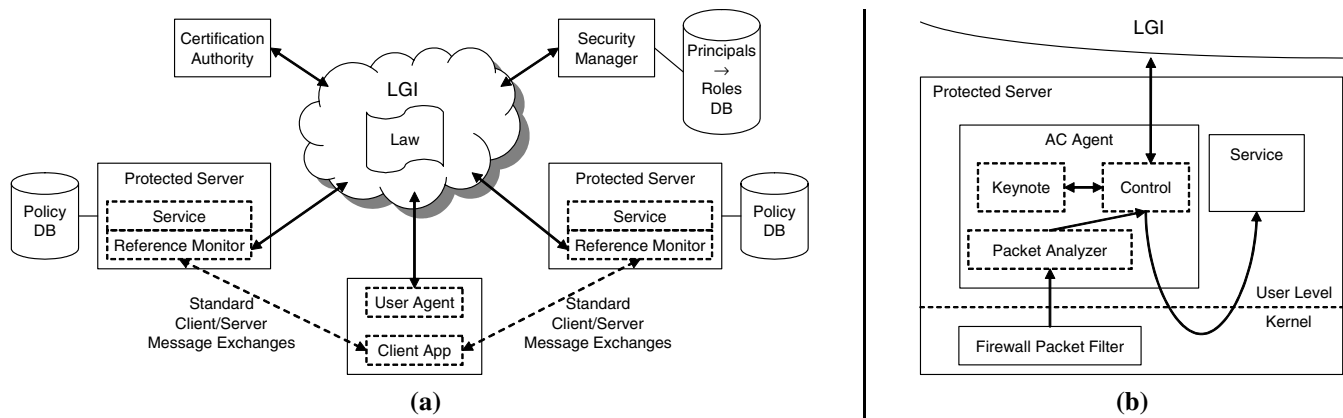
Figure 1. *Overview of our coordination and control framework: (a) Each server runs a software reference monitor that filters accesses to the server; (b) the reference monitor provides a unification point for the application of LGI and KeyNote to the coordination and control problem.*

delegation expiration time of the end of the term to ensure that the TA does not retain access rights forever. The professor can similarly delegate access rights to visitors and collaborators so that gaining access for these principals becomes just a matter of presenting a set of certificates.

Second, the stipulation that the role of *Admin* cannot be delegated demonstrates the use of a global constraint to prevent individual actions that may critically impact the dependability of the system. Delegation of the *Admin* role is dangerous because, among other things, *Admin*s likely have write access to configuration files that can affect the performance, reliability, and security of the system.

Finally, the revocation component allows the enterprise to establish an important restriction based on state derived from a principal's interactions with a community of servers. Regaining revoked privileges requires the intervention of two administrators, which is an example of the common separation-of-duty principle.

3. Overview

Figure 1(a) gives an overview of our framework, which consists of a set of reference monitors, a set of user agents, and a set of services comprising the enterprise security infrastructure. In general, each node hosting software components subjected to enterprise-wide policies, e.g., NFS clients and servers whose interactions must obey $\mathcal{P}_E$, would also host a reference monitor. This reference monitor filters all incoming messages, passing through only those allowed by the policies in effect. Figure 1(a) shows the reference monitors running only on the servers because this is sufficient to enforce policies that only coordinate and control accesses to server resources such as $\mathcal{P}_E$. Existing message passing applications such as NFS clients and servers will then interact as normal. However, this interaction is only possible if the client's user can "convince" the reference

monitor protecting the server to pass the client's requests through to the server.

Typically, a user can convince a reference monitor to accept his client's access requests by obtaining and presenting an appropriate set of credentials. The process of obtaining and presenting credentials is itself a sequence of interactions between the user agents, reference monitors, and security infrastructure that is subjected to control by enterprise-wide policies. In our framework, this process is coordinated using LGI, a mechanism for controlling message exchanges between a community of software agents according to an explicit *law*. Thus, the parts of a policy that are concerned with coordinating and controlling the gathering and presentation of credentials must be expressed as an LGI law. The law $\mathcal{L}_E$ that embodies this part of our example policy $\mathcal{P}_E$ is discussed in detail in Section 5.

The security infrastructure is typically comprised of components such as the certification authority and security manager shown in Figure 1. With respect to $\mathcal{P}_E$, these components perform functions such as certificate generation and mapping of principals to roles that they may assume.

Each user agent is some user's gateway to our framework, allowing the user to interact with the reference monitors and security infrastructure. With respect to $\mathcal{P}_E$, a user wishing to access a file server would use the user agent to authenticate himself, obtain the roles that he may assume, and present these roles to the server's reference monitor to convince it to permit the access.

Each reference monitor is a combination of a software firewall and a user-level access control (AC) agent (Figure 1(b)). The firewall intercepts all packets forming protocol-level requests that are subject to the policies in effect and passes them to the AC agent. The AC agent uses a packet analyzer such as Ethereal [6] to reassemble these packets back into meaningful protocol-level requests. For each request, the AC agent then extracts the information

needed for access control, namely the principal, the object to be accessed, and the access being attempted.

To determine whether a request should be permitted, the AC agent would gather all information that may affect the decision from the enterprise security infrastructure, the user agents, and possibly other reference monitors. With respect to $\mathcal{P}_E$, the AC agent would retrieve the roles that the requesting principal may assume from the principal's user agent. Critically, since this interaction and the interactions through which the user agent obtained the roles are controlled by $\mathcal{L}_E$, the AC agent can be confident that the user agent cannot provide false information—the reason for this confidence will become clearer once we provide more information about LGI in Section 4.1 and $\mathcal{L}_E$ in Section 5. The AC agent also contacts the policy database running on the protected server to retrieve any information specific to the object to be accessed; e.g., the per-file access control policy.

Once it has gathered all the necessary information, the AC agent uses KeyNote, a system designed to answer the question "does a set of credentials prove that a request complies with an access control policy," to decide whether the request should be permitted. For $\mathcal{P}_E$, the credentials correspond to the roles and the access control policy corresponds to the per-file policy. If the request is permitted, then the message will be forwarded to the server. Otherwise, a request denial is returned to the client.

Information can also flow from the AC agent to the security infrastructure and user agents. For example, the AC agent will inform the user agent when requests are denied so that $\mathcal{L}_E$ can enforce the revocation stipulations of $\mathcal{P}_E$.

For efficiency, we assume that an AC agent can cache information about principals and objects so that it does not need to communicate with remote services to mediate every access request. The AC agent may also batch the reporting of request denials.

It is important to observe that the AC agent is mainly an information conduit between LGI and KeyNote. In particular, *policy stipulations are never embedded in the code of the AC agent*. Rather, all policy stipulations are expressed as parts of an LGI law or as per-object KeyNote policies, which provides a clean separation between policy and mechanism. In essence, our approach uses LGI to coordinate the handling of credentials between the users, the security infrastructure, and the reference monitors, and uses KeyNote to mediate specific access requests. With respect to $\mathcal{P}_E$, this maps to the use of LGI to coordinate the management of the roles that users can gain and lose through delegation and revocation, and the use of KeyNote to mediate specific file access requests such as open, read, and write.

We chose LGI as the overall coordination and control mechanism because it supports a rich domain of stateful communal policies in a variety of environments [1, 2, 12, 13]. Further, LGI's inherently distributed nature is well-suited to our approach since we propose to have an AC agent per node for scalability.[2] We chose KeyNote because it was specifically designed to evaluate whether an access is authorized in a dynamic environment where principals can delegate rights to each other. We could have used LGI for this purpose as well. However, KeyNote targets a more constrained problem and so its evaluation engine provides well-defined compliance semantics and is more efficient than LGI. In fact, as shall be seen, we do not exploit the full capability of KeyNote. Thus, an even more constrained evaluation engine could have led to increased performance.

In many cases, a coordination and control policy will have components that are specific to objects being shared by the interacting software components as this allows the global policy to be augmented by the individual owners of the objects. For example, with respect to $\mathcal{P}_E$, each file has an associated policy that specifies access rights to the file by specific roles. To support the storage of these object-specific policies, we have implemented a simple policy database using the Berkeley DB [19]. This is the policy database mentioned earlier, which is responsible for servicing the AC agent's requests for per-object policies.

Because the policy database is implemented external to the software component (service) exporting the objects, there is a consistency issue between the policies stored in this database and the objects stored by the service. To minimize this problem, we expect that each server would run a policy database to hold the per-object policies for objects stored by services running on that server. Then, inconsistencies should only arise if the machine crashes unexpectedly in the middle of two related writes, say the write of a newly created object to the service and the corresponding per-object policy to the policy database. In this case, we would need some tool like `fsck` to synchronize the policy database with the object store after a crash.

4. Background and Related Work

In this section, we briefly describe LGI and KeyNote to provide context for the discussion of our experimental study in Section 5. We also discuss other related work.

4.1. Law-Governed Interaction (LGI)

LGI is a control mechanism that governs the message exchanges within a community of distributed agents according to an explicitly specified policy called a *law*. The messages exchanged under a given law $\mathcal{L}$ are called $\mathcal{L}$-messages, and

---

[2]It is easy to imagine a centralized reference monitor for all the servers of an enterprise but such a design is unlikely to scale well.

the group of agents interacting via $\mathcal{L}$-messages is called a community $\mathcal{C_L}$. For each agent x in a community $\mathcal{C_L}$, 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 and can represent such things as the role of an agent in this community and the privileges it carries.

Laws. The law $\mathcal{L}$ governing a community $\mathcal{C_L}$ is defined over a set of *regulated events* that can occur at members of $\mathcal{C_L}$, mandating the effect that such events should have—this mandate is called the *ruling* of the law for a given event. Regulated events include the sending and arrival of $\mathcal{L}$-messages, the coming due of an obligation (discussed below), and the submission of a certificate. The operations that comprise the rulings of the law for a regulated event are called *primitive operations*. They include operations on the control state of the agent where the event occurred (called the "home agent"), operations on messages, and the imposition of an obligation on the home agent.

Thus, a law $\mathcal{L}$ can regulate the exchange of messages between members of $\mathcal{C_L}$ based on the control state of the participants; it can mandate various side effects for a message exchange, such as modification of the control states of the sender and/or receiver of a message, and the emission of extra messages.

Law enforcement. The law $\mathcal{L}$ governing a community $\mathcal{C_L}$ is enforced by a set of trusted agents called controllers. Controllers are logically placed between all members of $\mathcal{C_L}$ and are used to mediate all exchanges of $\mathcal{L}$-messages between them. Each member x of $\mathcal{C_L}$ must *adopt* $\mathcal{L}$ at some controller $\mathcal{T}_x$, which maintains x's control state and imposes $\mathcal{L}$ over regulated events whose home agent is x. Each $\mathcal{L}$-message sent from an agent x to another agent y is then forwarded first through $\mathcal{T}_x$, then through $\mathcal{T}_y$, before it is delivered to y (if allowed by $\mathcal{L}$). Every $\mathcal{L}$-message exchanged 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 desirable.

The global nature of LGI laws requires that all members of a community $\mathcal{C_L}$ observe the same law $\mathcal{L}$. To ensure this homogeneity, a hash H of $\mathcal{L}$ is appended to all $\mathcal{L}$-messages sent from one controller to another. A controller then accepts an $\mathcal{L}$-message if and only if H is identical to the hash of its own law.

Finally, for controllers to trust one another, they may authenticate to each other using certificates signed by a certification authority acceptable to $\mathcal{L}$.

Writing laws. LGI laws are written using either a Prolog-based or a Java-based language. The two languages support the exact same semantics; here, we will briefly describe some relevant components of the Prolog-based language.

A Prolog-based law $\mathcal{L}$ is a program L, which, when presented with a goal e representing a regulated event at a given agent x's controller, evaluates the rule associated with e in the context of $\mathcal{CS}_x$ to produce a list of primitive operations representing $\mathcal{L}$'s ruling for e.

The regulated events most relevant to our case study in Section 5 include:

`sent(X, M, Y)`: an $\mathcal{L}$-message M sent by X to Y has arrived at X's controller.

`arrived(X, M, Y)`: an $\mathcal{L}$-message M sent by X to Y has arrived at Y's controller.

`certified(X, certificate(issuer(I), subject(Y), attributes(A)))`: X has presented a valid certificate issued by I. A is a list of attributes being certified about the subject Y.

`obligationDue(type)`: the self-imposed obligation of type `type` has fired. Typically, obligations are set to occur sometime in the future to ensure that some action is carried out in a timely manner.

In addition to the standard types of Prolog goals, the body of a rule may contain two distinguished types of goals. These are *sensor goals*, which allow the law to access the control state of the home agent, and *do goals*, which contribute to the ruling of the law. A sensor goal has the form `t@CS`, where t is any Prolog term. It attempts to unify t with each term in the control state of the home agent. A do goal has the form `do(p)`, where p is a primitive operation. It appends the operation p to the ruling of the law.

Commonly used primitive operations include:

`+T(v)`: add term T to the home agent's control state with value v.

`forward(x,m,y)`: send message m from x's controller to y's controller, triggering an `arrived(x,m,y)` event at y's controller.

`deliver(x,m,y)`: deliver the message m from x (sent via x's controller) to y.

`imposeObligation(type,t)`: impose an obligation with the specified `type` that will come due at time t.

Example. To see how the above pieces fit together, consider the simple example shown in Figure 2. Rule $\mathcal{R}1$ in this snippet specifies that each agent operating under law $\mathcal{L}_{ID}$ must have a name as certified by the certificate authority ca. Then, suppose that an agent x sends an $\mathcal{L}_{ID}$-message to another agent y. Recall that each such message must first be routed through x's controller $\mathcal{T}_x$. The arrival of the message at $\mathcal{T}_x$ corresponds to a `sent` event and so would trigger

```
R1. adopted(Any, cert([[certificate(issuer(ca),
        subject(Self), attributes([name(N)]))]]))
    :- do(+name(N)).

R2. sent(X, M, Y)
    :- name(N)@CS, do(forward(X, [from(N)|M], Y)).

R3. arrived(X, M, Y) :- do(deliver).
```

Figure 2. *Snippet of an example law $\mathcal{L}_{ID}$.*

Rule $\mathcal{R}2$. The ruling of Rule $\mathcal{R}2$ is to forward the message prepended with x's name to y at $\mathcal{T}_y$. The addition of x's name to the message ensures that spoofing is impossible. When the message arrives at $\mathcal{T}_y$, it would trigger Rule $\mathcal{R}3$, which leads to the delivery of the message to y.

4.2. KeyNote

KeyNote is a trust management system designed to answer the question: does a set of credentials C prove that a request r complies with a local security policy P? Specifically, KeyNote implements a general-purpose evaluation engine that can be invoked by an application each time it needs to answer the above question. For each invocation, the application passes to KeyNote a list of credentials, a set of policies, the requester's public keys, and a set of environment attributes. The result of each evaluation is an application-defined string such as "authorized" or "denied."

Policies and credentials are specified using KeyNote's assertion language, where each assertion is essentially a delegation of some rights. The only difference between policies and credentials is that each credential must be cryptographically signed whereas a policy is trusted by KeyNote and so is not signed; this means that KeyNote must cryptographically validate credentials but not policies during an evaluation. Attributes are also trusted and so are not validated during the evaluation.

Each KeyNote assertion takes the following form:

```
Authorizer: "POLICY"
Licensees: "<public key>"
Conditions: (file == "f" && operation == "read")
            → "authorized";
```

where the *Authorizer* field identifies the principal that is delegating some rights, the *Licensees* field specifies a set of receiving principals, and the *Conditions* field specifies the rights to be delegated along with a set of conditions on the environment attributes that must be true for the delegation to hold. In the example above, the string POLICY indicates that this is a root assertion representing an "intrinsic" (as opposed to delegated) set of rights; the Licensees field specifies the public key representing some receiving principal p; and the Conditions field specifies that p should be given "read" access to file f. In general, the set of licensees

can be specified using expressions formed from conjunction, disjunction, and threshold operations. Conditions may include string comparisons, numerical operations and comparisons, and pattern-matching operations.

Each time it is invoked, KeyNote performs a depth-first search through the given credentials and policies to find an assertion graph from one or more root assertions to an assertion that authorizes the request. Each assertion in this chain represents a delegation of rights and thus can only refine the authorizations conferred on it by the previous assertions in the graph. If no such chain can be found, then the answer to the compliance question is no, which is mapped to an application-defined string such as "denied."

4.3. Other Related Work

A number of research and industry efforts have explored the use of a centralized reference monitor to impose global access control policies [7, 21, 22]. In contrast, our system uses a distributed network of reference monitors, one per server, and uses LGI to coordinate between them and other security services. The inherently distributed nature of our mechanism provides three important advantages. First, the access control evaluation can be done at the server itself, avoiding the overhead of additional network communication. Second, our framework is more scalable and does not have a single point of failure. Tivoli is one centralized system that has addressed the second issue through the replication of the reference monitor [7]. However, such replication introduces the difficulty of maintaining consistent state in the presence of dynamic policies. In fact, Tivoli's access control model is mostly limited to static policies; while Karjoth showed that Tivoli can support the Chinese Wall policy, this involved extending Tivoli (through an extension API) with customized code. It was unclear how this extension interacted with the replication. Finally, while not addressed in this work, our framework can be used to enforce hierarchies of policies across coalitions of enterprises, addressing a problem that is rapidly rising in importance.

Our reliance on the ubiquity of software firewalls is similar to Bellovin's work on distributed firewalls [3]. However, Bellovin's work was concerned with dynamically adjusting firewall policies of individual machines as the environment around them changes.

One effort that is particularly related to our experimental study of access control for file systems is Miltchev et al.'s Distributed Credential File System (DisCFS) [9]. This work explores the use of KeyNote and IPSec to extend NFS's access control model to the general credential-based model of KeyNote. This work is related to our in that it seeks to introduce a richer access control model, including dynamic delegation, to file systems. It's different than our work, however, in that it required changes to the file system client

and server; indeed, it led to the new DisCFS file system. In general, a key contribution of our work is that we seek to provide a practical gateway for applying trust management systems and advance policies to standard client-server protocols without requiring changing the protocols or applications. Further, our use of LGI allows us to support *communal* policies that KeyNote was not designed for.

5. Case Study

In this section, we first show how our framework can be used to *explicitly specify* and *uniformly enforce* policy $\mathcal{P}_E$ across a set of heterogeneous NFS and SMB servers. We then evaluate the performance of our implementation. Finally, we draw on our implementation experience to discuss characteristics that would make protocols more amenable to coordination and control mechanisms that depend on the interception of messages such as ours.

5.1. Implementation

Our implementation includes a security manager, a user agent, an AC agent, a policy database, and a law $\mathcal{L}_E$. The security manager is an LGI-aware application that maintains a database of registered users and the roles that they can assume using the Berkeley DB. The user agent is a simple LGI-aware application that allows the user to present certificates for authentication and gaining delegated rights. The user agent is simple enough that it can be implemented as a Java Applet runnable from any Java-enabled browser. The AC agent is an LGI-aware application that uses a combination of iptables, Ethereal, and KeyNote to enforce access control on specific requests. The policy database is a simple wrapper around the Berkeley DB that holds the per-file policies.

The interactions between the user agents (one per user), security manager, and AC agents (one per server) are coordinated and controlled by $\mathcal{L}_E$. We show a small snippet of $\mathcal{L}_E$ in Figure 3 just to give the reader a feel for what the law looks like. The entire law, comprised of approximately 40 rules, can be found at http://p2p.cs.rutgers.edu. We also show one critical message exchange sequence coordinated by $\mathcal{L}_E$ in Figure 4.

In the remainder of the section, we first describe how policy $\mathcal{P}_E$ is expressed and enforced by the above components. Then, we describe how the AC agent derives the necessary information (principal, object, and access request) from protocol messages and its caching and buffering actions. Finally, we discuss implementation details specific to processing NFS and SMB protocol messages.

Authentication. To access a server protected by our system from some client machine, a user must authenticate himself to the system by running the user agent. On startup, the user

```
𝒫reamble:  authority(ca, keyHash(K_ca^pub)).
       alias(sm, "sm@A.com"). initialCS([]).

       ca is the distinguished certification authority trusted by this law. sm is the
       distinguished security manager.

ℛ1.  certified(UA, certificate(issuer(ca), subject(UA),
         attributes(name(NAME))))
       :- do(addAuthority(NAME,keyHash(SubjectHash))),
         do(add(meAsCA(NAME))).

       The user agent has presented a valid certificate doubly signed by ca and
       itself specifying that its name is NAME. Save the user's public key as an
       authorizer in order to accept future certificates signed by the user. Also save
       the certified name NAME.

ℛ2.  certified(UA, certificate(issuer(N), subject(UA),
         attributes(ip(IP),uid(UID))))
       :- meAsCA(N)@CS, do(forward(Self,
         attributes(ip(IP), id(UID), name(N)), sm)).

       The user agent is informing the security manager that its user is running
       with user id UID on the client machine with address IP.

ℛ3.  arrived(UA, attributes(ip(IP), id(UID), name(N)),
         SM) :- do(add(user(ip(IP), id(UID),
         lgiName(UA)))), do(deliver).

       When a message from UA containing a (IP, UID, N) tuple arrives at the
       security manager's controller, save the tuple (IP, UID, UA) to the security
       manager's control state and deliver the message.

ℛ4.  sent(SM, roles(R), UA)
       :- do(forward(SM, roles(R), UA)).

       The security manager is sending back the roles that a user may assume to
       the user's user agent.
```

Figure 3. *Part of the enterprise law $\mathcal{L}_E$.*

agent first connects to an LGI controller and adopts $\mathcal{L}_E$. Then, it presents two certificates: (1) a certificate signed by the distinguished certificate authority ca acceptable to $\mathcal{L}_E$ (Preamble in Figure 3) specifying that the user owns a particular user name N and public key K^{pub}; and (2) a certificate signed by the private key paired with K^{pub} specifying the user's *UID* and the client machine's *IP* address. The presentation of the 1st certificate, assuming that the certificate is valid, leads to the addition of N and K^{pub} to the user agent's control state (Rule $\mathcal{R}1$). The presentation of the 2nd certificate leads to the forwarding of the tuple (IP, UID, N) to the security manager (Rule $\mathcal{R}2$).

When the above message arrives at the security manager's controller, the pair (IP, UID) together with the user agent's LGI name will be inserted into the security manager's control state (Rule $\mathcal{R}3$). The user agent's LGI name is saved so that future queries from the AC agents can be forwarded to the user agent (see Subsection **Enforcement** below). An obligation will also be set to discard this entry if a keep-alive message is not received within time T_{smc}. Finally, a query message containing the name N will be generated and forwarded to the security manager.

If the security manager finds N in its database, it will reply with the list of roles that the user can assume (Rule $\mathcal{R}4$). When this reply arrives at the user agent's controller, the roles will be inserted into the user agent's control state (Rule

not shown). The reply is then forwarded to the user agent to inform it that the authentication process has been completed and the roles that its user may assume.

While the user agent is connected to its controller and operating under $\mathcal{L}_E$, an obligation is set to periodically send a keep-alive message to the security manager so that the user's authentication information will not be discarded from the security manager's control state.

Delegation. Registered principals can delegate their access rights to others by providing them with certificates signed by ca specifying the delegating principal P_d, the receiving principal P_r, the delegated role R_d, and, optionally, the delegated rights, a file or directory, and an expiration time. A receiving principal can then invoke delegated rights by presenting one or more delegation certificates through the user agent. If the delegated role is *Admin*, the delegation will be unconditionally refused. Otherwise, a message will be forwarded to the security manager to verify that the delegating principal may assume the delegated role. A positive reply from the security manager will lead to the delegated rights being added to the control state of the user agent. If the control state of the user agent contains a non-empty list of AC agents (see Subsection **Enforcement** below), then the newly acquired rights will be forwarded to these AC agents. Finally, if the certificate had an expiration time, an obligation is set up to discard the entry at that time.

For the remainder of the discussion, we refer to the roles pre-assigned to a user as his *intrinsic* roles and roles that the user has gained through delegation as his *delegated* roles.

Per-File Policies. Each file has an owner and an owner-specified access control policy describing the roles that can access the file and the corresponding access rights. A role may be an arbitrary string, e.g., "Admin," or a public key, in which case only the owner of the corresponding private key can assume that role. The access control policy is expressed as a Keynote policy. Currently, the set of possible access rights include {*read*, *write*, *search*} and each access evaluation can return one of {*granted*, *denied*}. The *search* right is equivalent to Unix's access right x for directories.

When a file is created, the AC agent assigns it a default access control policy specified by the owner. The owner can access and modify this policy as desired using a set of simple scripts that we have developed.

Enforcement. For each intercepted access, e.g., read or write, if the AC agent currently has no information about the requesting principal, it would send a message to the security manager (M1 in Figure 4). This message is routed through the AC agent's controller, triggering the sent event E1, which simply forwards the message (M2) to the security manager. At the security manager's controller, this message will be rerouted to the principal's user agent using the user agent's LGI name that was saved earlier (E2 and M3). At
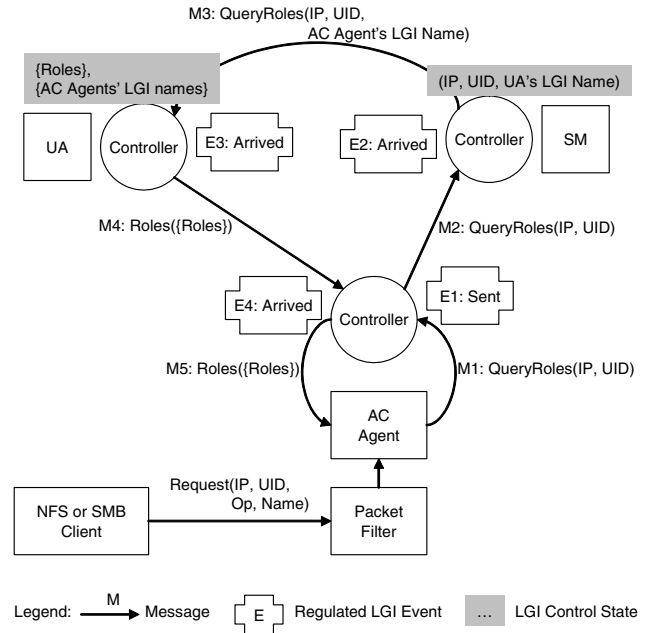

Figure 4. *Message exchange sequence for the AC agent to obtain the roles that the requesting principal may assume.*

the user agent's controller, this message will cause this particular AC agent's LGI name to be inserted into the user agent's control state (E3). A message containing all of the principal's roles is also sent to the AC agent (M4). Expiration times and file/directory restrictions for delegated roles are also included. At the AC agent's controller, the message is delivered to the AC agent (M5), which can now use the roles to mediate the intercepted request.

If the object being accessed is the file $/a/f$, then the AC agent also needs to access the policy database to obtain the per-file policies associated with $/$, a, and f. Finally, the AC agent would invoke KeyNote three times, twice to determine whether the principal has *search* right for $/$ and $/a$ and once to determine whether the principal has the necessary access right to $/a/f$.

For each evaluation, intrinsic roles are passed to KeyNote as attributes while delegated roles are passed as policy assertions as follows. Suppose that a principal identified by key K^{pub} can assume a delegated role of R_d and is attempting to access file f. In this case, the following assertion will be generated:

Authorizer: "R_d"
Licensees: "$< K^{pub} >$"
Conditions: (operation == "read" || operation == "write"
|| operation == "search") →"granted";

For the above rule to have an effect, however, some rule granting access rights to R_d would have to exist in f's pol-

Protocol	Regulated Messages
NFS	MOUNTPROC_MNT, MOUNTPROC_UMNT, MOUNT-PROC_UMNTALL, LOOKUP, SETATR, READLINK, READ, WRITE, CREATE, REMOVE, RENAME, LINK, SYMLINK, MKDIR, RMDIR, READDIR
SMB	SESSION_SETUP, TREE_CONNECT, OPEN, CLOSE, TREE_DISCONNECT, CREATE, CREATE_NEW, CREATE_TEMPORARY, CREATE_DIRECTORY, DELETE_DIRECTORY, DELETE, MOVE, RENAME

Table 1. *Controlled message types.*

icy. For example, suppose f's policy includes the following assertion:

> Authorizer: "POLICY"
> Licensees: "R_d"
> Conditions: (operation == "read") → "granted";

Then, the principal would gain *read* access to f through the authorization chain of POLICY → R_d → K^{pub}.

If the results of all KeyNote evaluations are *granted*, then the intercepted request will be forwarded to the server. If the result is *denied*, then the access is not authorized. In this case, either an error is generated and sent back to the client or the request is modified in such a way that is guaranteed to force the server to deny the request.

A count of denied accesses is maintained for each principal. Each time this count reaches a predetermined level, the AC agent sends a message to the principal's user agent. When this message arrives at the user agent's controller, the count will be added to an overall count maintained in the user agent's control state. An obligation will periodically clear this count. According to $\mathcal{P}_E$, if this count ever exceeds a certain level, a suspension tag will be added to the user agent's control state. This tag will prevent all further accesses. Regaining blocked privileges is done by presenting two reactivation certificates issued by distinct principals under role *Admin*.

Caching inside the AC agent. The AC agent caches three types of information: (1) Per-file access control policies are cached for an adjustable period T_{cp}. These policies are always discarded after T_{cp} regardless of client access patterns to ensure that the cached information does not become stale because of updates by the file's owner. (2) Principals' credentials are cached for an adjustable period T_{cc}. This period is extended by T_{cc} every time the AC agent receives a "keep alive" message from the principal's user agent. And (3) KeyNote evaluation results are cached using a tuple of (access type, IP, UID, object's pathname) as the key. These results are invalidated when the principal's credentials change and/or the object's policy is evicted from the policy cache.

Implementation details for NFS. Our system does not intercept all messages to minimize control overhead; we

wrote some simple iptables extensions to intercept only a subset of NFS message types and pass them to the user-level AC agent. Table 1 lists the message types that we do intercept and control. These map to important accesses such as read, write, open, create, delete, etc. We do not control purely informational requests such as an inquiry for the attributes of a file. In a highly secure environment, this might present a disclosure concern, which would necessitate the interception and control of all message types.

Recall that for each access, we must extract the principal, the object, and the access type. The principal and access type are easily extracted from NFS client requests. Extracting the name of the object being accessed is more difficult, requiring the mapping of an NFS handle to a pathname. Fortunately, Ethereal already implements the logic necessary to extract such mappings from requests such as LOOKUP and the corresponding replies and maintains them in a hash table. Evicting entries from this table can be difficult, however, because NFS versions 2 and 3 use persistent file handles. Any eviction may cause our system to not have the necessary mapping to evaluate a future request. Currently, we evict mappings that have not been used for a threshold amount of time despite this problem. If the system ever receives a request containing a handle that cannot be mapped, it returns a stale handle error, which becomes visible to the user. NFS version 4 allows the server to limit the use of a handle to some finite period, which solves this problem.

If an evaluation leads to a denial of an access, the AC agent will send an NFSERR_ACCES reply back to the client.

In our system, the NFS servers are set up with AUTH_UNIX authentication. All directories to be exported are set up as world-writable and exported with rw and async options. All users authenticated by the security manager can mount an exported directory. The local permissions of files and directories are set to world read, write, and access.

Implementation details for SMB. Similar to NFS, we do not intercept all SMB messages; Table 1 shows the message types that are intercepted. Interestingly, note that we do not need to intercept read/write requests. This is because the necessary access control is already performed at the open; for example, the server will itself refuse a write request on a file that was opened only for read.

SMB servers are set up to use user-level authentication. All shares are set up in such a way that only one particular user can access them. This user is private and only known to the AC agent. When an authenticated user tries to access an SMB share, before the request reaches the server, the user name and password will be replaced with the private user by the AC agent.

Similar to NFS, the AC agent maintains a map between file ids and file names. An entry will be evicted when the AC agent sees a close message for a particular file id. Unlike NFS, the AC agent also associates a connection with

Operation	Time
Keynote eval. of simple root policy	23 us
Keynote eval. of root policy + 1 delegation	41 us
Keynote eval. of root policy + 10 delegations	172 us
AC agent obtains a per-file policy from policy DB	
containing 50,000 entries	8.0 us
containing 500,000 entries	20.8 us
Authentication with the security manager under $\mathcal{L}_E$	509 ms
AC agent obtains roles from user agent	61 ms

Table 2. *Latencies for KeyNote evaluations, for AC agent to obtain per-file policies from the policy database, and for two LGI-mediated interactions.*

	Base	RM-AC	
Op.	Time (us)	Time (us)	% Deg.
NFS create	420	677	61.2
NFS write	170	257	51.2
NFS read	151	236	56.3
SMB create	1,042	1,246	19.6
SMB write	212	213	0.5
SMB read	175	176	0.6

Table 3. *Per-operation latencies.*

each specific principal and its roles. SMB servers are also set up to send keepalive packets every 1 minute. A connection will be considered as dead by the AC agent if it doesn't receive any packets in 3 minutes. The AC agent will silently reclaim all resources of a dead connection.

To deny an access, the AC agent will modify the request to ensure that the request will be refused by the SMB server based on its local access control policy. Then, when the server receive the request, it will reply with an access control error. For example, if the request is to open the file with read/write flags while the policy states that the user only has read permission, the write flag will be removed from the request. For users who tries to access a file which he has no permission, the file name will be changed to a local special file which is owned by a private user and is set up with permissions disabled.

5.2. Performance

We now quantify the performance impact of our approach. We first present micro-benchmark measurements for a client to access an NFS and SMB server when operating under policy $\mathcal{P}_E$. Then we present measurements using the modified Andrew benchmark [17] to evaluate the impact on normal usage of the protected file systems.

Our experimental setup was as follows. We had 1 LGI controller running on a 1.6GHz P4 PC with 512MB of memory. We had 1 NFS server, 1 SMB server, 1 client, and 1 security manager, each of which was running on a separate 2.8GHz P4 PC with 1GB of memory. All nodes ran Linux v2.6.9. The security manager used the Berkeley DB version 4.2.52 to hold the public keys and roles of registered principals. We populated this database with 500 entries to represent a medium-sized enterprise unit. Each file server also ran a policy database and our reference monitor, where the AC agent used iptables 1.2.9, the Ethereal packet analyzer version 0.10, and KeyNote version 2.3. The client and servers are configured to use IPsec Authenticated Header in tunnel mode to prevent IP spoofing, ensuring the integrity

of identifying a user by his (IP, UID) pair.

Micro-benchmarks. Table 2 gives the times to perform Keynote evaluations, to access a Berkeley DB, and to complete various LGI-mediated interactions. Of these, the costs of KeyNote evaluations are probably the most significant as they may be incurred frequently. These results show that the base cost for a KeyNote evaluation is fairly expensive but probably acceptable, particularly since we cache the results of these evaluations. Although our current policy only allows a single level of delegation, we also present the times for KeyNote evaluations when given longer delegation chains. These results show that very long delegation chains can lead to large performance overheads.

The cost of obtaining a per-file policy from the server is purely the cost of accessing the Berkeley DB database. This cost seems quite reasonable, particularly since we also cache these policies for 30 seconds. Finally, we show the authentication time and the time for the AC agent to obtain a user's roles for completeness; these times are not of real concern because they are typically incurred only when the user is initiating access.

Table 3 compares the worst-case per-operation latencies for create, read, and write operations performed under our framework (RM-AC) with the base case of no global access control. Recall that when a file is created, it is given a default access control policy. Thus, the time for `create` includes the time for creating a default policy and writing it to the policy database. The times for `read` and `write` do not include accesses to the policy database since the corresponding per-file policy has been cached (via the `create`).

For SMB, there is almost no overhead for read and write because our framework only has to control creates and opens; the server will itself refuse a write request on a file that was open only for read (and vice versa). The overhead for create is also not too high at 19.6%. For NFS, the overheads are much more significant. It is particularly high for create because each create involves several messages that must be intercepted for both control and for mapping handles to file names. Overheads for the first read and write to a file are significant because reads and writes must be intercepted and controlled. Subsequent accesses, however, should be more efficient because the AC agent caches the results of KeyNote evaluations. Also, when we consider the

IEEE
COMPUTER
SOCIETY

Phase	Base Time (ms)	RM-AC Time (ms)	% Deg.
NFS			
mkdir	49	85	74.9
copy	2,051	2,558	24.7
stat	512	688	34.3
grep	1,084	1,090	0.6
compile	21,282	21,837	2.6
SMB			
mkdir	88	110	24.1
copy	3,232	3,587	11.0
stat	239	244	2.0
grep	3,805	4,339	14.0
compile	21,707	21,859	0.7

Table 4. *Execution times for phases of the modified Andrew Benchmark*

common case of reading/writing larger amounts of data, the overheads become much lower. For example, the latency only degrades by 23% and 10% when writing 4KB and 8KB of data, respectively.

Modified Andrew Benchmark. Next we look at the impact of our approach at the macro-level using the modified Andrew benchmark. This benchmark has five phases as follows: (1) mkdir: creates a tree of directories, (2) copy: copies a 14,488KB collection of C source files into the directory tree created in phase 1, (3) stat: traverses the new tree and examines the status of each file and directory, (4) grep: reads every file in the new tree, searching for a string, and (5) compile: compiles and links the files. Table 4 shows the results for both NFS and SMB. First, observe that the overhead of our approach is quite low when file system activities are interspersed with non-trivial computation (and each KeyNote evaluation is likely amortized across a sequence of read/write operations because of caching effects): 1–3% for the compile phase and 1–14% for the grep phase. (The overhead for the grep phase running over NFS is especially low because the NFS client is still caching much of the content from the copy phase.) On the other hand, the overheads are consistent with our micro-benchmark measurements for the three phases that involve just file system activities, mkdir, copy, and stat.

We have also measured the impact of our framework on servers' throughput, e.g., when multiple clients are running the Andrew Benchmark against a single server. We do not show these results here because of space constraints. They are similar to the above results for latencies in that for reasonable workloads, e.g., multiple greps ran from multiple clients, the degradation in throughput is quite small.

5.3. Discussion

Our goal is to be able to enforce enterprise-wide policies on client-server interactions without requiring changing or

extending the application protocols. However, our implementation exposed several characteristics of protocols that can either ease our task or make it much more challenging. In particular, controlling NFS and SMB, which represent significantly different design points proved insightful.

First, NFS's use of a connection-less protocol (UDP) made it much easier for us to generate error messages when necessary. Thus, access denials were easy to implement. SMB's use of TCP made it difficult to insert messages directly into the connection; thus, we had to manipulate the access requests very carefully to implement denials. In general, protocol features for allowing external input on whether a request should be accepted or denied would significantly ease the implementation of an external access control mechanism such as ours.

On the other hand, the stateless nature of NFS, particularly with respect to the fact that there are no open and close operations led to several difficulties. First, we had to control each and every read and write access, which leads to greater overhead. Second, evicting mappings from file handle to file names was difficult. Thus, protocols that bound a stream of accesses to an object with distinguished requests (e.g., open and close) where access control can be limited to these distinguished requests should lead to better efficiency.

Finally, in both cases, we had to maintain a mapping of external object names (file names) to internal handles. We suspect that this is a representative problem for most servers, which may prove to be the trickiest challenge for applying our approach to an arbitrary protocol. Thus, protocols should limit the lifetimes of internal handles given out to clients, perhaps by using renewable leases for such handles. Also, it should be easy to map internal handles to external object names. (Clearly, this should be true for the principal's identity as well since we need the principal, operation, and the object being accessed to perform access control.)

In summary, our approach uses standardized message exchange protocols as the intersection points between our access control mechanism and applications. Thus, protocol characteristics can strongly affect the applicability of our approach. The above are some of the ways that protocol designers can help to make protocols more amenable to our approach.

Orthogonal to the above protocol-related issues is the issue of message interception in the presence of encrypted communication. Application-level encryption can present an unsurmountable obstacle to our framework (without requiring changes to the application). Encryption is not a problem, however, when the application relies on system-level encryption mechanisms such as IPSec and port redirection through ssh tunnels since we can intercept the messages after they have been decrypted but before they are forwarded to the application.

IEEE
COMPUTER
SOCIETY

6. Conclusion

In this paper, we have introduced a novel framework for enforcing enterprise-wide coordination and control policies over conglomerates of software components that interact using standardized message exchange protocols. Our framework can express and enforce sophisticated policies without requiring any changes to existing applications. Our approach also allows new applications to be developed that are neutral with respect to specific coordination and control frameworks, effectively decoupling enterprise-wide coordination and control from application implementation.

Our approach centers around the use of firewall-based reference monitors, one per server, as transparent "hooks" for intercepting and controlling message exchanges. We then combine these hooks with LGI and KeyNote, two existing complimentary systems, to provide a powerful and scalable coordination and control framework.

We demonstrate the feasibility and power of our framework by providing a concrete example, the unified enforcement of an enterprise-wide policy across NFS and SMB file servers. We show how our framework can be used to support an enterprise-wide policy that includes RBAC, delegation, revocation, and separation-of-duty. Measurements of our implementation show that our mechanism can incur non-trivial but acceptable overheads. We thus conclude that our approach can be used as a practical migration path for moving advance coordination and control policies and mechanisms from the realm of research to practice.

References

[1] X. Ao and N. Minsky. On the Role of Roles: from Role-Based to Role-Sensitive Access Control. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies*, June 2004.

[2] X. Ao, N. Minsky, and V. Ungureanu. Formal Treatment of Certificate Revocation Under Communal Access Control. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2001.

[3] S. M. Bellovin. Distributed Firewall. *;login: Special Issue on Security*, Nov. 1999.

[4] M. Blaze, J. Feigenbaum, and A. D. Keromytis. *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, chapter The Role of Trust Management in Distributed Systems Security. Springer-Verlag, 1999.

[5] D. Brewer and M. Nash. The Chinese Wall Security Policy. In *Proceedings of the IEEE Symposium in Security and Privacy*, May 1989.

[6] Ethereal. http://www.ethereal.com/.

[7] G. Karjoth. The Authorization Service of Tivoli Policy Director. In *Proceedings of the 17th Computer Security Applications Conference (ACSAC)*, Dec. 2001.

[8] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1), Jan. 2003.

[9] S. Miltchev, V. Prevelakis, S. Ioannidis, J. Ioannidis, A. D. Keromytis, and J. M. Smith. Secure and Flexible Global File Sharing. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, June 2003.

[10] N. Minsky and D. Rozenshtein. Controllable Delegation: An Exercise in Law-Governed Systems. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Oct. 1989.

[11] N. Minsky and V. Ungureanu. Law-Governed Interaction: a Coordination and Control Mechanism for Heterogeneous Distributed Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(3), July 2000.

[12] T. Murata and N. Minsky. Regulating Work in Digital Enterprises: A Flexible Managerial Framework. In *Proceedings of the Cooperative Information Systems Conference (CoopIS)*, Oct. 2002.

[13] T. Murata and N. Minsky. On Shouting "Fire!": Regulating Decoupled Communication in Distributed Systems. In *Proceedings of the International Middleware Conference*, June 2003.

[14] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, Mar. 2002.

[15] T. Ryutov and C. Neuman. Representation and Evaluation of Security Policies for Distributed System Services. In *Proceedings of the DARPA Information Servability Conference Exposition*, Jan. 2000.

[16] T. Ryutov and C. Neuman. The Specification and Enforcement of Advanced Security Policies. In *Proceedings of International Workshop on Policies for Distributed Systems and Networks (POLICY)*, June 2002.

[17] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming Aggressive Replication in the Pangaea Wide-Area File System. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Dec. 2002.

[18] M. Satyanarayanan. Digest of Proceedings. In *Proceedings of the 7th Workshop on Hot Topics in Operating System*, Mar. 1999.

[19] Sleepycat Software. Berkeley DB. http://www.sleepycat.com.

[20] L. Snyder. Theft and Conspiracy in the Take-Grant Model. Technical Report 147, Yale University, 1978.

[21] V. Varadharajan, C. Crall, and J. Pato. Authorization in Enterprise-wide Distributed System: A Practical Design and Implementation. In *Proceedings of the 14th Computer Security Applications Conference (ACSAC)*, Dec. 1998.

[22] T. Woo and S. Lam. A Framework for Distributed Authorization. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, Nov. 1993.