Dynamic Access Control: Preserving Safety and Trust for Network Defense Operations

Prasad Naldurg
Department of Computer Science
University of Illinois at Urbana Champaign
IL, 61801, USA
naldurg@cs.uiuc.edu

Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana Champaign
IL, 61801, USA
rhc@uiuc.edu

ABSTRACT

We investigate the cost of changing access control policies dynamically as a response action in computer network defense. We compare and contrast the use of access lists and capability lists in this regard, and develop a quantitative feel for the performance overheads and storage requirements. We also explore the issues related to preserving safety properties and trust assumptions during this process. We suggest augmentations to policy specifications that can guarantee these properties in spite of dynamic changes to system state. Using the lessons learned from this exercise, we apply these techniques in the design of dynamic access controls for dynamic environments.

Categories and Subject Descriptors

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

General Terms

Security, Design, Performance

Keywords

dynamic access control, access lists, capability lists, comparison, safety, trust

1. INTRODUCTION

Computer Network Defense (CND) operations have attracted increasing attention in recent years. In particular, there is growing interest in Response Actions (RAs), i.e., design of system mechanisms that can react to vulnerabilities and exposures and deploy countermeasures in real-time.

In traditional networked systems, access control policies and mechanisms specify and enforce the authorized use of shared resources. When the system is attacked, or when a vulnerability is detected, the security of its users or software components is often compromised. In such a situation, changing the access control matrix is often a suitable response action for system administrators. This has the effect of changing the authorizations and can prevent malicious users or software applications from accessing vulnerable (but not yet compromised) resources, or vice-versa.

In this paper we examine the issue of changing access rights dynamically. Once a threat is mitigated, e.g., by expelling the users, or by installing patches on software, it is also useful to be able to restore the original access control policies, i.e., to rollback the system to its original operating environment. We explore this in the context of two standard implementations of access control polices viz., access lists (ALs) and capability lists (CLs). We analyze the overheads of AL and CL based implementations with respect to storage requirements and algorithmic efficiency.

Though the two representations are equivalent in terms of expressibility [5], we show that they have different performance penalties and memory requirements with respect to dynamic change. Choosing one over the other, depending on the situation, can have a significant impact on the usability of the system. We also briefly describe later how this analysis helped us decide which implementation mechanism to pick in the case of dynamic access controls for dynamically changing environments.

As we show in this paper, changing these lists in an ad hoc manner can have unexpected side effects in terms of safety properties and trust assumptions. In order to preserve access control safety, we need to implement an atomic broadcast and commitment protocol in a distributed setting. It is well known that achieving this is difficult without synchronization assumptions. To preserve trust assumptions, we show with the help of a suitable formal notation how policy specification and enforcement mechanisms can be augmented with appropriate guards.

The rest of the paper is organized as follows: In Section 2, we compare ALs and CLs with respect to storage and performance overheads for dynamic access control. In Section 3, we discuss the impact of changing access controls in terms of safety properties and trust assumptions. In Section 4, we apply our analysis and justify our design choices for dynamic access control implementations for dynamic environments. In Section 5, we present related research, and conclude in Section 6.

SACMAT'03, June 2–3, 2003, Como, Italy.

2. DYNAMIC ACCESS CONTROL POLICIES

Access control is typically implemented with a conceptual access control rights-set (also called the access control matrix) that consists of $\langle subject, object, method\rangle$ tuples. Each tuple in the matrix asserts that a *subject* (or user or principal) has the right to access a *method* on a given *object* (or software-mediated resource). Requests to access a resource are allowed if and only if the tuple can be found in the rights-set. This is also called the **access control safety property**. The implementation mechanisms to enforce this property vary in practice. Usually, a software entity called a *reference monitor* guards the resource by intercepting requests and applying the membership test of access safety. If this test fails, access is denied. Therefore, the default behavior of most access control systems is to deny access unless an explicit right can be found in the system.

Restrictions are placed on the set of *subjects* that are allowed to add, update, or delete access rights from the rights-set. Mandatory Access Control (MAC) policies restrict this privilege to trusted system administrators. In Discretionary Access Control (DAC), this privilege is extended to the owner of an object. Most systems support a combination of MAC (for public or system resources) and DAC (for privately-owned resources) policies.

In a distributed system, shared resources (i.e., objects) can be on different physical machines connected over a network, and the access control enforcement mechanisms can also be distributed across the network. Each machine may have a reference monitor that intercepts both local and remote access requests to shared resources. A single centralized access control matrix to validate accesses is rarely implemented in practice [2]. In order to reduce performance overheads, these access rights-sets are also spread across the system.

Two different representations for storing the rights across the system are commonly used: access lists (ALs) and capability lists (CLs). ALs are lists of $\langle subject, method\rangle$ pairs per *object*, and correspond to the list of subjects that are allowed to access specific methods for a given object. CLs are lists of $\langle object, method\rangle$ tuples, and correspond to the list of objects and methods that can be accessed per *subject*.

Traditionally, in stand-alone systems where the sets of subjects (usually classified into groups or alternatively as roles) or objects do not change very often, the relative benefits of choosing one representation over the other are comparable [6, 5]. In an AL-based system, invoking a subject's right to access an object is simple and is usually accomplished by deleting specific rights from the object's AL. To locate an access right one may have to search through the entire AL. CLs are usually generated by system administrators and stored in protected shared memory. Each access request can either carry the capability itself (with sufficient cryptographic protection to prevent modification) or a pointer to the location of the capability in the CL, which is only accessible by the decision logic. CLs simplify the lookup of rights and speed up the the process of access control decisions. When capabilities are passed around or CLs replicated in different process address spaces, revocation of access rights becomes difficult.

It is not immediately clear which implementation mechanism is ideal to change the access rights in response to a perceived threat or vulnerability in a distributed setting, since there are no systematic studies that compare the relative benefits of the two approaches. In [10] the authors speculate that ALs are more popular than CLs because they efficiently answer the question "who accessed a given object?", whereas CLs are useful to answer the question "what else can a subject access?" [2]. While the first question is useful to protect the confidentiality of information accessed and directly relates to access control, the second question is useful to evaluate the flow of information in the system.

In the next subsection, our aim is to evaluate the performance overheads of changing access control rights in a distributed setting for both options. In Section 3, we study the impact of changing these lists in terms of safety properties and trust assumptions. For this paper, we focus on heterogeneous distributed systems that are organized as a single administration domain; i.e., all the different networked resources and users are managed by a single administrator. We leave interaction between administrative domains as future research.

2.1 Changing Access Control Rights

When a system is attacked or when a new vulnerability is discovered, administrators may receive notifications (either from other administrators, network monitoring software, software vendors, intrusion detection systems etc.) to disallow access to certain users, computers, software applications or specific methods for applications (e.g., disable execute for email attachments). We examine the cost of changing these rights for both AL and CL-based distributed access control implementations in terms of the sizes of sets of subjects, objects, and methods.

We assume that each subject $subject_i$ and object $object_j$ have a CL or AL associated with them. Let $|\mathcal{S}|$ and $|\mathcal{O}|$ be the cardinalities of the sets of subjects and objects, equal to the number of CLs or ALs in the system respectively. Table 1. summarizes the maximum number of lists that have to be processed (ALs or CLs), in order to remove a user, a resource, or a specific access right tuple from a distributed access control system. The numbers also correspond to the maximum number of network connections that have to be initiated by the administrator in order to decide whether an AL or a CL needs to be updated. The actual amount of time required to process the list depends on the data structures used to implement the lists, and the number of access rights in the system.

As shown in Table 1, to remove a user $subject_i$ in a distributed AL-based implementation, since the $\langle subject_i, method_k\rangle$ tuples are distributed across multiple access lists corresponding to different objects, each list must be examined in turn and the tuples purged appropriately. In order to remove a user in a CL-based implementation, only the capability list of a single user has to be deleted. In both cases, the maximum number of entries that may need to removed is equal to the number of distinct $\langle subject_i, object_j, method_k\rangle$ tuples in the system. This is also the number of entries that need to be stored for rollback in order to restore the original access rights.

To remove an object $object_j$ and all its associated rights, only that object's AL has to be removed from the system. In the case of CLs, all CLs corresponding to all subjects have to be examined, to find and delete the appropriate$\langle object_j, method_k\rangle$ tuples. In both cases the

RA	AL model	CL model
To remove user	$\mid \mathcal{O} \mid$	1
To remove object	1	$\mid \mathcal{S} \mid$
To remove a specific access right	1	1

Table 1: Maximum Number of Lists Processed

number of entries that need to be removed (or stored for rollback) is equal to the number of distinct access rights that have $object_j$ in their tuples.

To remove an individual permission for a particular object (e.g., disable the auto-execute option in a web browser etc.), in both AL and CL-based implementations, only one table has to be updated, corresponding to either the $subject_i$'s CL or the $object_i$'s AL.

To summarize, in order to remove an object's access rights from a distributed system, there is a significant cost asymmetry in favor of ALs. Similarly to remove a user from the system, using CLs is more efficient.

From this analysis, we observe that in terms of automating the process of changing access controls dynamically by removing entries from the ALs or CLs, if the sizes of the sets of users and objects are comparable, no implementation technique has a clear advantage. If we expect that our RAs are primarily removing access to certain objects or disabling specific object rights, even temporarily, clearly ALs are better. Given the rate at which bugs are being discovered and patches installed on existing software, we believe that this might very well be the case.

2.2 Using RBAC

CLs are more efficient when system administrators discover that certain user accounts are compromised and want to quickly throw a cage around the user, isolating their actions from the rest of the system. In contrast, in an AL based implementation, many lists may have to be located and updated. However, ALs can overcome this limitation if we use an aggregation mechanism such as Role Based Access Control (RBAC [8, 15]) to simplify administration of users and rights.

In RBAC, users can be associated with one or more roles. Each role is a placeholder for a set of permissions. The permissions consist of objects and methods that are authorized for that role. Users can be added and removed from roles, independent of the updates to the role-permission assignments. This asynchrony simplifies the management of large sets of users and restricts their behavior according to their "role" in an organization.

RBAC is very flexible and can be implemented naturally using ALs. In addition, it can also be used to implement both MAC and DAC policies. Instead of specifying individual subjects and permissions in each AL for each resource, we can aggregate entries according to roles. This has the effect of reducing the size of the ALs, and reducing the search space for changing access controls. CLs can also be organized according to roles. This reduces the total number of CLs in the system.

Instead of removing a user from a role-based AL, the user's access control permissions can be changed by revoking the user's current role and assigning the user to a special pre-defined role that has no access rights, or to a role that reduces the user's permissions to a restricted set of rights. This information can be relayed directly to the policy enforcement mechanisms (e.g., the reference monitors), who must use the user's new role and disallow any requests made by the user using their old role. While this eliminates the cost of changing the ALs, the communication cost of disseminating the user's new role has to be taken into account.

Another advantage of using pre-defined roles during response actions is the ability to formally analyze the access control behavior of the system a priori, even when the permissions of a user change dynamically. By restricting the permissions during dynamic state-changes in the system, it is possible to determine what guarantees can be made by the dynamic behavior of the system by formal analysis.

In the next section, we examine the possible side-effects of changing ALs and CLs as RAs in CND. We also describe techniques to augment existing specifications to overcome these limitations.

3. ENFORCING SAFETY AND PRESERVING TRUST

One of the problems with changing access controls dynamically, while the system is in operation, is the need to synchronize operations. In distributed systems, an administrator may initiate a response action (RA) to change access controls over the network. Depending on the type of changes requested, the administrator may have to update many ALs or CLs, distributed across different machines. As a result, some lists may get updated faster than others and the access control safety property many not be consistently enforced across the system. As we show in the next subsection, maintaining consistency in this situation is not easy.

Changing access rights dynamically to alter the access control behavior of a system can have unexpected side-effects. In a stand-alone operating system, or in a homogeneous distributed system (e.g.,Unix-based or Windows clusters), it may be possible to preserve existing trust assumptions by relying on the underlying protection model. Implementing protection domains is simplified by this support, and only network administrators can change access control lists or permissions. In a heterogeneous distributed system, the trust assumptions and trust validation have to be modeled explicitly into the system specifications, to prevent undesirable side-effects. We explore this in Section 3.2.

3.1 Enforcing Access Control Safety

In order to remove users, objects or particular methods from the system, a system administrator has to keep track of the different ALs or CLs stored in different parts of a distributed system. Removing objects in a distributed AL-based system, or subjects in a CL-based system while maintaining safety is straightforward. The access rights in a particular object or user's AL or CL are unique, and are not replicated across the system. Therefore, no consistency issues arise.

However, the problem emerges when an administrator needs to modify multiple ALs to remove subjects (or CLs to remove objects) in a distributed setting, and keep the lists consistent. A non blocking atomic commitment protocol [12] (such as a modified two-phase commit) is required to ensure that updates are consistent. To guarantee timeliness, when multiple update requests are sent, maintaining safety is reduced to implementing an atomic broadcast protocol [12], which needs to be both reliable, as well as deliver the update messages in total-order.

It is well known that there are no deterministic atomic broadcast algorithms for asynchronous systems. This is because the distributed consensus problem can be reduced to atomic broadcasts. However, there are many schemes [12] that account for clock drifts and periodically send out synchronize messages that work under the assumption of bounded drifts. The two protocols viz., non-blocking atomic commitment and atomic broadcast, can guarantee that the access control safety property is guaranteed even when the ALs and CLs change dynamically. The protocols may introduce a non-negligible overhead to change policies and their effectiveness as RAs must be evaluated in terms of these overheads.

Once again, RBAC can overcome the need to change access lists, but revoking a user's role and assigning the user to a new role requires similar consistency and synchronization guarantees.

3.2 Preserving Trust Assumptions

In addition to preserving consistency, trust assumptions related to changing ALs and CLs have to be examined carefully. Trust assumptions are incorporated into the access control safety property (allow access if and only if the access right exists) by including the concept of authorization as follows: allow access if and only if an authorized user added the access right to the system.

Enforcing this modified safety property is straightforward in a stand-alone system, or in a homogeneous distributed system (multiple machines running the same OS). Most existing distributed operating systems automatically provide support to enforce that the access control mechanisms can only be updated by authorized users (administrators in MAC, and owners in DAC).

in order to guarantee this property in a heterogeneous system, consisting of different operating systems and protection models, we need to examine all the mechanisms (or methods) in the access control system that can modify any of the entries in a conceptual access matrix. Since the authorization safety property has to be satisfied at every state of the system, we argue that any change in the state of the ALs and CLs can only be allowed if the user requesting the change can provide a proof of authorization. This property has to be enforced uniformly across the different machines in the system.

We describe a systematic technique to augment policy specifications with special clauses called guards that force users to present a proof of authorization, in the form of credentials, attesting that they have the right to change the access rights. This mechanism is independent of the underlying protection model and can guarantee that trust assumptions are always preserved by any proper implementation of the specification.

In particular, we describe how to generate authorization guards and show how they preserve the authorization safety property in all transitions that can modify the state of the access control implementations.

An access control system can be modeled as a state machine, with the sets of subjects, objects and access rights as its state variables, and the access control decision can be specified as follows:

state vars
S: **set of** SUBJECTS **initial** ϕ
O: **set of** OBJECTS
A: **set of** $\langle s : SUBJECT; o : OBJECTS; m : METHODS\rangle$

access control decision
$Allowed(s, o, m) \leftrightarrow \langle s, o, m\rangle \in A$

Notice that we do not explicitly model ALs or CLs. The set A is typically the union of different ALs or CLs in the the system. Once we define the state variables, the next step is to model the state transitions. State transitions are all system actions that can change the state variables. For example, a sample set of state-changing transitions may include:

$AddSubject(s') \longrightarrow S := S \cup \{s'\}$

$RemoveSubject(s') \longrightarrow S := S - \{s'\}$
$\qquad A := A - \{\langle s', \mathbf{o}, \mathbf{m}\rangle \mid \mathbf{o} \in O, \mathbf{m} \in METHODS\}$

$AddObject(o) \longrightarrow O := O \cup \{o\};$

$RemoveObject(o) \longrightarrow O := O - \{o\};$
$\qquad A := A - \{\langle \mathbf{s}, o, \mathbf{m}\rangle \mid \mathbf{s} \in S, \mathbf{m} \in METHODS\}$

$AddAccessRight(s', o, m) \longrightarrow A := A \cup \{\langle s', o, m\rangle\}$

$RemoveAccessRight(s', o', m') \longrightarrow A := A - \{\langle s', o', m'\rangle\}$

Note that it is very important to model all the state transitions in order to make any safety guarantees. This specification deliberately does not include any authorization checks. If this system was implemented, anybody is allowed to change the access rights matrix. As mentioned earlier, different sets of subjects are allowed (or authorized) to create and delete users, objects and methods. In a DAC system, users are allowed to create and own objects and add access rights to objects they own. For example, if $user1$ owns $file_{user1}$, then $user1$ can insert $\langle user2, file_{user1}, read\rangle$ into A. In an MAC system, users and objects can be added only by administrators.

In the next subsection, we present a method to automatically add guards to state transitions and preserve trust during the modification of access control implementations.

3.3 Trust Management

To preserve the trust assumptions, we augment an access control specification with special proofs of authorization. Henceforth, if a user wants to change an entry in A, the user is required to produce a proof attesting that he or she is allowed, by some trusted authority, to actually call the relevant method. The policy enforcer (e.g., a reference monitor) has to be suitably modified to check this proof. The proof check can be verified non-interactively, and is always decidable. This proof-checker is a guard, similar to Dijkstra's guarded commands [7, 16], and these guards can be applied to both ALs and CLs without loss of generality.

One way of generating a proof of authorization is by using an attestation from a trusted administrator that gives the holder of the attestation the capability to change an AL or CL entry, i.e., the permission to call a method to change the entry. This type of capability (also called a license [17]) or credential is an attestation of trust. These attestations should be protected against modification by unauthorized entities. They can be made unforgeable by the issuer by attaching a cryptographic digital signature. The signature should tie in the name of the issuer and the intended recipient to prevent modification and also provide non-repudiation of ownership.

Generating these credentials naïvely can cause management problems. Consider the set U of users who can issue signed capabilities, the set O of shared objects in the system and the set M of methods corresponding to access rights. The set of all licenses that can be presented to the policy manager in this system is exponential in size and is given by $C \subseteq U \times 2^{OXM}$).

A user can have many different credentials authorizing some or all methods with respect to a particular object and may present any subset of these to the enforcer to change an access right. The enforcer needs to decide whether the decision is consistent with the trust management implications of these attestations and this may be non-trivial. For example, the monotonicity of the privileges available after revocation may have to be maintained [17] to prevent undesirable behavior.

Instead, we argue that in the case of MAC or DAC policies, two simple types of credentials are sufficient to attest the identity of the entities (primarily subjects) and the ownership of one entity by another. These credentials cannot be delegated and are not available to any entity except the implementation logic. An example identity credential $typeof(Alice, administrator)$ asserts that the identifier $Alice$ is an administrator. The credential $owns(object, method)$ or $owns(user, object)$ attests that the method is "owned" or exported by the object or the object is owned by the user, respectively. We generate one credential per object and method. Therefore, the size of this credential set is equal to the number of unique access rights in the system. This simplifies the management of credentials and proof verification, though it may increase the number of credentials a user has to present.

With the help of an example, we show how we can augment our sample policy specification and enforce DAC and MAC trust authorizations. We also show how these authorization proofs guarantee only authorized behavior in our system. We introduce the set of credentials to our state variables and show an example state transition with its corresponding guard highlighted in boldface.

state vars
...
$\mathcal{C}$: **set of identity and ownership credentials**

transitions

```
$AddAccessRight(s, s', o, m)$
    $\wedge\, \mathbf{owns(s, o)} \in \mathcal{C} \wedge \mathbf{owns(o, m)} \in \mathcal{C} \longrightarrow$
    $A := A \cup \{\langle s', o, m \rangle\}$
```

In this specification, subject s owns object o and adds an access right to subject s'. By presenting the appropriate ownership credentials, s proves he or she has a right to add the access right to A. An example of a MAC policy where only an administrator is allowed to change the access rights is given next (notice the check for an appropriate typeof credential):

```
$AddAccessRight(s, s', o, m)$
    $\wedge\, \mathbf{typeof(s, admin)} \in \mathcal{C}$
    $\wedge\, \mathbf{owns(o, m)} \in \mathcal{C} \longrightarrow$
    $A := A \cup \{\langle s', o, m \rangle\}$
```

From the augmentations to the specifications, we claim that if the credentials are generated correctly and the administrators keys are not compromised, then the state transitions allowed in our modified system have the required authorization proofs necessary to guarantee the authorization safety property, even when the implementations are changed dynamically. This is because if a guard cannot be satisfied, the state of the system is unchanged. When the state changes, a proof of authorization was verified correctly. In addition to the technique we outlined to preserve the consistency in the previous subsection, we claim that our augmentations also ensure that only trusted entities can modify the ALs and CLs.

4. APPLYING DYNAMIC ACCESS CONTROL TO DYNAMIC ENVIRONMENTS

When the sets of subjects and objects do not change frequently, though the overheads of changing access controls in AL-based and CL-based systems seem equivalent, we observe that ALs are better when the changes involve updating rights for specific objects and object methods. In this section, we explore the ideas further in the context of dynamic environments where the sets of users and objects can change dynamically, and the access control matrix entries have short life times and may be updated frequently. In this situation, we find dynamic access controls are an important feature of the system design rather than just a mechanism to enable RAs in CND. In such cases, the performance overheads for changing the implementations can have a significant impact on the design decisions. We describe briefly how we can use our analysis to justify the choices for contrasting AL-based and CL-based implementations of dynamic access controls next.

Examples of dynamically changing environments include ad hoc networks, active networks, and smart spaces. In an ad hoc network, for example, where users bring in their own computers and connect together, it is often useful to associate ALs with the mobile resources themselves. Since users can come and go over a short period of time, it is not feasible for the participating subjects to carry CLs for all possible objects. Instead, the participants themselves can build dynamic trust relationships and assign permissions to each other by updating their own ALs.

In contrast, when the set of subjects can change over time, but the resources themselves remain more or less fixed, it is more efficient to use CLs. We have explored such a solution in the context of active networks in our previous work [4, 13] and developed a CL-based architecture to enable the dynamic installation and update of policies in real-time, to accommodate different sets of subjects using the same resources for different active network protocols over time. Our security architecture for active networks also incorporated the access control safety and authorization techniques discussed in this paper, in the framework of a CL based implementation. With this architecture, we were able to demonstrate how we can change the access control policies on software routers dynamically, and deploy a host of reactive security countermeasures, including dynamic firewalls and vaccines, without sacrificing safety guarantees.

Another area where we have explored the issues of changing access controls dynamically is in the context of smart spaces or active spaces [14]. An active space is a physical environment that is augmented with computing and communication resources and can be programmed and configured automatically to support different tasks and activities. An example of an active space is a smart room with many display and computation devices that can be configured as a meeting room, a lecture room, or a recreation room etc., only by changing the software in the room. In this context, we have developed an AL-based solution to address the access control issues in this room, where we have dynamically changing sets of users and to a lesser extent, objects. We chose an RBAC variation of ALs to scale to a large number of users.

5. RELATED WORK

In this section, we highlight recent relevant research and situate it in the context of our exploration of dynamic access control implementations. We are not aware of any systematic studies that evaluate different access control policy implementations with regard to changing these implementations in response to a perceived threat or vulnerability. However, our work on preserving trust assumptions is influenced by several recent research efforts.

Schneider defines a class of policies called Enforceable Policies [16] that can be enforced by execution monitoring. This class of policies is specified using a special automata called Security Automata and is concerned with the preservation of safety properties. Our guards can be validated in this framework.

While we have focused on policy implementation, many researchers have focused on high-level languages and frameworks for specifying different access control policies and models. Bertino et al [1] describe a logical framework for reasoning about different access control models. Specifically they address the task of evaluating different flavors of extensions to database authorization models (e.g., negative authorizations, multi-policy models, role based authorizations etc) and evaluate the power of expression of these models. Koch et al [11], analyze the interaction between different policy models and the behavior of their integration using the theory of graph transformations.

Jajodia et al [9] propose a language for expressing authorizations and enabling the enforcement of multiple access control policies and show how programs written in this language effectively capture the abstractions necessary to define different types of authorizations encountered in access control models. This is extremely useful when different parts of the system implement different access control policies. In our work, we have focused on a homogeneous system and we are looking into the impact of deploying dynamic policies when there are multiple access control policies in the system.

Weeks [17] provides a formal semantics for expressing trust management systems via a fixpoint lattice model for monotonic assertions. This model is useful to understand the trust management of capability-based assertions. Chander et al [5] provide a state transition approach to model the interaction of trust management and access control. The interaction of access control and trust management including the use of unforgeable credentials to provide authorization proofs, and the equivalence of ALs and CLs can be validated in their framework.

The capability-based KeyNote system of Blaze et al [3], provides a single language for both policies and credentials, based on predicates that describe the trusted actions permitted by holders of specific public keys (or other cryptographic identifiers). Our model integrates access control with a simple trust management mechanism. The main purpose of KeyNote is to express and evaluate policies and trust delegations that occur in PKI applications. KeyNote can be integrated into our framework for trust management for other types of dynamic policies that require more expressive credentials.

6. CONCLUSIONS

We investigate the overheads associated with changing access control lists to implement response actions in computer network defense. In particular, we compare and contrast the costs of enabling dynamic access policies in two standard implementation mechanisms, viz., access lists and capability lists. We also explore the preservation of safety properties and trust relationships during this process, and suggest augmentations to existing policy specifications to guarantee that properties are not violated. We describe our experiences with designing dynamic access controls in dynamically changing environment and summarize relevant related research. We hope that this work will enable designers of dynamic access controls understand the overheads in terms of performance and storage, as well as use our augmentations as templates to incorporate into their designs and preserve safety and trust relationships.

7. ACKNOWLEDGMENTS

The authors are grateful to Apu Kapadia, Geetanjali Sampemane, Seung Yi, Christopher Andrews and the anonymous reviewers for useful comments and suggestions.

8. REFERENCES

[1] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. In *Proceedings of the Sixth ACM Symposium on Access control models and technologies*, May 2001.

[2] M. Bishop. *Computer Security: Art and Science.* Addison-Wesley, 2003.

[3] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Security Protocols International Workshop, Cambridge, England*, 1998.

[4] R. H. Campbell, Z. Liu, M. D. Mickunas, P. Naldurg, and S. Yi. Seraphim: dynamic interoperable security architecture for active networks. In *OPENARCH 2000*, Tel-Aviv, Israel, March 26–27, 2000.

[5] A. Chander, D. Dean, and J. Mitchell. A state-transition model of trust management and access control. In *14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2001.

[6] D. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[7] E. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. Communications of the ACM, 18(8):453–457, 1975.

[8] D. F. Ferraiolo and D. R. Kuhn. Role-based access controls. In *In Proceedings of the 15th NIST-NSA National Computer Security Conference, Baltimore, MD, Oct*, 1992.

[9] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data, volume 26,2 of SIGMOD Record*, pages 474–485, 1997.

[10] P. Karger and A. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy, pp. 2-12*, 1984.

[11] M. Koch, L. V. Mancini, and F. Parisi-Presicce. On the specification and evolution of access control policies. In *Proceedings of the Sixth ACM Symposium on Access control models and technologies*, May 2001.

[12] S. Mullender. *Distributed Systems, Second Edition*. Addison-Wesley, 1995.

[13] P. Naldurg, R. Campbell, and M. D. Mickunas. Developing dynamic security policies. In *Proceedings of the 2002 DARPA Active Networks Conference and Exposition (DANCE 2002), San Francisco, CA, USA, IEEE Computer Society Press, May 29-31*, 2002.

[14] G. Sampemane, P. Naldurg, and R. Campbell. Access control for active spaces. In *Proceedings of 18th Annual Computer Security Applications Conference (ACSAC)*, 2002.

[15] R. S. Sandhu and E. J. Coyne. Role-based access control models. *IEEE Computer*, 29(2), Feb. 1996.

[16] F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[17] S. Weeks. Understanding trust management systems. In *2001 IEEE Symposium on Security and Privacy*, May 2001.