A Framework for Distributed Authorization*

(Extended Abstract)

Thomas Y.C. Woo   Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

1 Introduction

Security is an important concern in the design and implementation of many services. Typical security considerations include the following (among others): (1) only authorized clients can obtain service; (2) proper charges are levied on services performed; and (3) correct records are kept for all services requested and delivered. These considerations give rise to the following problems: *authentication*, *authorization*, *accounting* and *auditing*.

Authentication is the most basic, as well as the most studied among the four problems. Much work has recently been done on authentication [1, 3, 4, 10]. Its main issues are fairly well-understood. In fact, several implementations of distributed authentication are available, e.g., Kerberos from MIT [2, 8] (which has also been integrated as part of the OSF DCE Security Service [7]), SPX [9] from DEC, and KryptoKnight [5] from IBM.

On the other hand, issues of authorization, accounting and auditing have remained relatively unexplored. In this paper, we focus on distributed authorization. We examine the major issues involved and propose a framework for constructing a distributed authorization service. Our framework has two central ideas, namely, (1) a language-based approach (called *generalized access control list* or GACL in short) for specifying authorizations; and (2) authenticated delegation. GACL is a significant extension of ordinary ACL. In particular, it provides constructs for explicitly stating inheritance and defaults. Authenticated delegation is not new. For example, it has been discussed in one form or another in [3, 4, 6]. Most of these works, with the exception of [6], are focused on the authentication aspect. Our study of authenticated delegation is for authorization. Our idea is similar to the notion of *proxy* in [6]. The framework is the result of our attempt in exploring the theory and practice of constructing a distributed authorization service which parallels existing distributed authentication services. Since our focus is on authorization, we will discuss accounting and auditing issues only to the extent that they are relevant to authorization.

Due to length limitation, we have omitted most of the details (e.g., the precise syntax and semantics of the language GACL, detailed specifications of protocols); we plan to present them together with our implementation experience in a separate forthcoming paper.

*Research supported in part by NSA INFOSEC University Research Program under contract no. MDA 904-92-C-5150 and by National Science Foundation grant no. NCR-9004464. Published in *Proceedings ACM Conference on Computer and Communications Security*, Fairfax, Virginia, November 1993. Postscript files of this and other papers of the Networking Research Laboratory are available from http://www.cs.utexas.edu/~lam/NRL.

2 Distributed Authorization Service

In the following, we will refer to a service that a client would ultimately like to obtain as an *end service*; and a server implementing such a service as an *end server*.[1]

Our research aims at abstracting and separating out the authorization functions as a distributed "core" service, which performs authorization on behalf of end servers. A client desiring service from an end server

[1] This terminology is adapted from [6], where the notion of an end server is defined in the context of a proxy, and is much more specific. Our notion of an end server is informal, and is intended mainly for differentiating user-oriented services from system-oriented services.

must first contact an authorization server (and possibly an authentication server before that) to obtain authorization.

Two key problems need to be addressed in constructing a distributed authorization service:

- Representation problem — The commonalities in authorization requirements of end servers should be identified, and an appropriate representation abstraction designed to capture these commonalities. In our research, we adopt a language approach. Our specification language GACL can be used to specify most commonly encountered authorization requirements, and efficient algorithms can be constructed for their evaluation.

- Protocol design problem — To offload authorization from end servers to authorization servers, secure protocols are needed for interactions among clients, authorization servers and end servers. These protocols make transparent the decoupling of authorization services from end services.

In the next section, we describe two key concepts underlying our framework designed to address the above problems. Then, in Section 4, we provide a high-level overview of our framework for distributed authorization.

3 Two Key Concepts

Terminology. To differentiate between our language of generalized access control list from a particular generalized access control list, we will refer to the former as GACL and the latter as "gacl". A similar convention (acl and ACL) is adopted in referring to ordinary access control lists. □

3.1 Informal Introduction to GACL

ACL has long been used for specifying authorization requirements. An acl is typically associated with an object and consists of a list of pairs; each pair is made up of a subject identifier and a set of access rights. A subject s is granted access r to object o if and only if the acl associated with o contains a pair (s, R) such that $r \in R$. Denial is implicit, i.e., it is implied by the absence of positive authorization in the list. As an example, consider the following acl for a file f: (Alice and Bob are individuals while Dept is a group)

$$\mathsf{f} : (\mathsf{Alice}, \{\mathsf{read}, \mathsf{write}\}), (\mathsf{Bob}, \{\mathsf{read}\}), (\mathsf{Dept}, \{\mathsf{write}\})$$

This acl specifies that Alice can be granted read and write accesses to f, and denied any other access to f. Similarly, Bob only has read access while all members in group Dept only have write access.

The key advantage of ACL is its straightforward semantics which is easy to understand. However, it is not very expressive. Several extensions have been proposed, e.g., allowing explicit *negative* authorizations. Most of these extensions are, however, ad-hoc and have often been introduced without a well-defined semantics.

We believe that ACL is the right abstraction to use in an authorization service. However, it must be extended to be effective. To this end, we propose the GACL language. GACL is much more expressive than ordinary ACL. The main features of GACL include the following:

- It provides constructs that can express in a straightforward way most commonly encountered authorization requirements. For example, the structural properties, *closure*, *inheritance* and *defaults*, identified in [11], can be directly expressed in GACL.

- It allows *incomplete* authorizations. That is, it is possible that for some request, neither grant nor denial can be determined. A *failure* is returned in this case. This is preferred over the "denial by default" style of authorization because a failure may suggest an error in a specification. On the other hand, the language allows an authorization administrator to explicitly specify a catch-all "denial by default" if so desired.

- It has an implementation independent semantics, thus allowing implementations of varied complexity and permitting interoperability across different authorization servers.

- It provides a declaration section that gives an authorization administrator additional flexibility in expressing authorization requirements.

GACL can be viewed as a practical "approximation" of the logical language of *policy base* introduced in [11]. In what follows, we give an informal introduction to GACL by examples. This hopefully would provide sufficient background for discussions on the architectural and protocol aspects of our framework in Section 4.

An example is specified using GACL in Figure 1. Each gacl is labeled by an object name and consists of two parts: (1) a *declaration* part identified by the

P.exe	**declare**	ordered
	list	⟨[Alice,Bob],[−execute]⟩,
		⟨[Dept],[execute]⟩,
		highload ⇒ ⟨[*],[−execute]⟩,
		inherit P.src::⟨[*],[write]⟩
P.doc	**declare**	anonymous
	list	P.exe::⟨[_x],[execute]⟩ ⇒ ⟨[_x],[read]⟩,
		⟨[Dept],[−write]⟩,
		always inherit Doc::⟨[*],[−read]⟩,
		demand inherit Doc::⟨[*],[write]⟩
P.src	**declare**	
	list	⟨[Research],[read,write]⟩,
		⟨[−Dept],[−write]⟩
Doc	**declare**	ordered
	list	⟨[Dept],[read]⟩,
		⟨[DocSys ∧ Research],[*]⟩,
		default::⟨[*],[−*]⟩

Figure 1: Specification of an Example using GACL

keyword **declare**; and (2) a *list* part identified by the keyword **list**.

The declaration part contains a (possibly empty) list of predefined keywords that provide information for interpreting the gacl. In this paper, we discuss in detail only two such keywords: "ordered" and "anonymous". An "ordered" declaration specifies that the list part of the gacl is an ordered list. That is, in determining authorization, its entries should be examined in a sequential order starting from the first to the last. By default, a gacl is interpreted as "unordered". For an unordered gacl, all entries in its list part should be examined "together" in making an authorization determination. This would be made clearer as we examine the example more closely below.

The list part contains a list of entries, some of which resemble those of ordinary acl's, while others are new. We informally explain their meanings below using the example in Figure 1.

An Example

Consider a set of objects {P.exe, P.doc, P.src, Doc}. P.exe, P.doc and P.src together constitute a software package with P.exe being the executable, P.doc the documentation and P.src the source. Doc is a centralized documentation control system in which P.doc is a part. Alice and Bob are individual users while Research and Dept are groups. DocSys is a server responsible for maintaining the documentation control system (e.g., performing version control). Though DocSys is not an actual user, it is considered a user in our framework. We consider only three types of access, namely, read, write and execute. *highload* is a system predicate whose (boolean) value is continuously updated by some system component that monitors the load of the system. For brevity, in the following, we refer to an entry by its position in the list. For example, with respect to gacl P.src, entry 2 refers to the entry ⟨[−Dept],[−write]⟩.

Consider gacl P.exe in Figure 1. Entries 1 and 2 are similar to those in ordinary ACL. Entry 1 specifies that both Alice and Bob are not permitted to execute P.exe, while entry 2 specifies that members of group Dept are allowed to execute P.exe. Entry 3 specifies that if the value of *highload* is true, then no subject is allowed to execute P.exe. (* stands for all subjects.) Entry 4 specifies that any subject who can write P.src can inherit the same access (i.e., write) to P.exe. Since "ordered" is declared, these entries should be examined in order from entries 1 to 4 in determining authorization. For example, Alice will be denied execute right for P.exe even if she belongs to Dept or has write access to P.src.

Consider gacl P.doc in Figure 1. Entry 1 specifies that any subject who has execute right for P.exe can also read P.doc. (_x is a variable that can be instantiated to any subject.) Entry 2 specifies that members of Dept cannot write P.doc. Entry 3 specifies that any subject who is denied read access to Doc will inherit the same denial to P.doc. Entry 4 specifies that any subject who has write access to Doc can inherit on demand the same access to P.doc. That is, a demand inheritance is activated only if no other write authorization has been specified in other entries. For example, members of Dept would not be able to inherit their write access to Doc (even if they do have it) because of entry 2. Note that gacl P.doc is unordered, thus its entries must be considered together in making a determination. For example, if Alice has execute right to P.exe (cf. entry 1) but is denied read access to Doc (cf. entry 3), then a read request from Alice for P.doc would generate an error as entries 1 and 3 together specify contradictory read authorizations for Alice.

The "anonymous" declaration does not affect the semantics of authorization. It indicates that an end server is willing to accept authorizations certified by an authorization server even without precise knowledge of the client making the request. For example, if a client other than Alice or Bob presents itself only as a member of Dept without saying who it is, it will still be acceptable to the end server and be granted read access.

Consider gacl P.src in Figure 1. Entry 1 specifies that members of Research can read and write P.src. Entry 2 specifies that any subject not belonging to Dept is denied write access to P.src. Again, gacl P.src is unordered. Thus a write request for P.src from any member of Research who is outside of Dept would generate an error.

Consider gacl Doc in Figure 1. Entry 1 specifies that all members of Dept have read access to Doc. Entry 2 illustrates authorizations for compound subjects. A *compound subject* can informally be understood as a subject who has authority to act as each of its component subjects. Thus, entry 2 specifies that any subject who has authority to act both as DocSys and as a member of Research can be granted all accesses to Doc. Typically, a compound subject is constructed by *delegation*. For example, a member of Research who has obtained delegation from DocSys to act on behalf of DocSys is an instance of the compound subject DocSys $\wedge$ Research. Entry 3 specifies that by default, every subject should be denied all accesses. Since "ordered" is declared, this default serves as a negative catch-all, and provides the "denial by default" semantics of ordinary ACL. Defaults are typically used in an unordered gacl; its activation is then similar to that of demand inheritance. For an ordered gacl, the keyword **default** is optional; it serves as a comment. For example, the semantics of gacl Doc is unchanged if the **default** modifier is dropped from entry 3.

3.2 Authenticated Delegation

The basic idea of an authenticated delegation is fairly straightforward. Consider two processes P and Q. After performing mutual authentication, P and Q share a secret channel k.[2] If P wants to delegate to Q, it can generate a new secret key k_d and send it to Q via channel k. Since channel k is integrity-protected and secret, only Q can receive k_d. Thus, any message later received by P that has been encrypted by k_d must have come from Q, and can be accepted by P as according to the delegation.

Indeed, Q can further delegate to another process R by generating a new delegation key k_a and providing R with k_a and a *delegation certificate*. A delegation certificate is of the form

$$cert = \{k_a, T, L, other\text{-}info\}_{k_d}$$

where T is a timestamp and L a lifetime. If $cert$ is presented to P, P can easily verify (by the encryption k_d) that it has been issued by its delegate Q. And R can further prove that it is the legitimate "owner" of $cert$ by demonstrating its knowledge of k_a using an *authenticator* of the form $\{T'\}_{k_a}$, where T' is a timestamp.

In our framework, authenticated delegation is used in two protocols: (1) In the contracting protocol between an end server and an authorization server. This allows the delegation of authorization function from an end server to an authorization server. (2) In the protocol between a client and an authorization server. This provides the client with the authority to present its request to the desired end server. We explain both uses in greater details in Subsection 4.2.

A similar scheme but with the name *proxy* is used in [6].

4 Overview of Our Framework

4.1 Architecture

Figure 2 shows the architecture of our framework. Below, we give a functional description of the various servers in the figure. An operational description of these servers is provided in Subsection 4.2.

- Service Locator — A service locator assists clients in locating servers implementing a particular service. A service locator obtains such information either statically from some configuration file or dynamically from registration messages sent out by active servers. A service locator functions in a manner similar to a name server[3] or a remote procedures registry. It responds to a client's request with a list of end servers that implement the requested service, and possibly also a list of authorization servers for the end servers (for end servers that have elected to offload their authorization functions).

- Authentication server — An authentication server performs two basic functions: (1) To authenticate users during their initial sign-on and supply them with an initial set of credentials. (2) To enable mutual authentication between clients and servers. We note that all communications should be authenticated, including those between clients and servers (e.g., clients and group servers,

[2] For simplicity, we use the session key distributed in the mutual authentication to refer to the channel.

[3] Indeed, it can be easily implemented as part of an existing name server mechanism (e.g., DNS) by including additional forms of *resource records*.

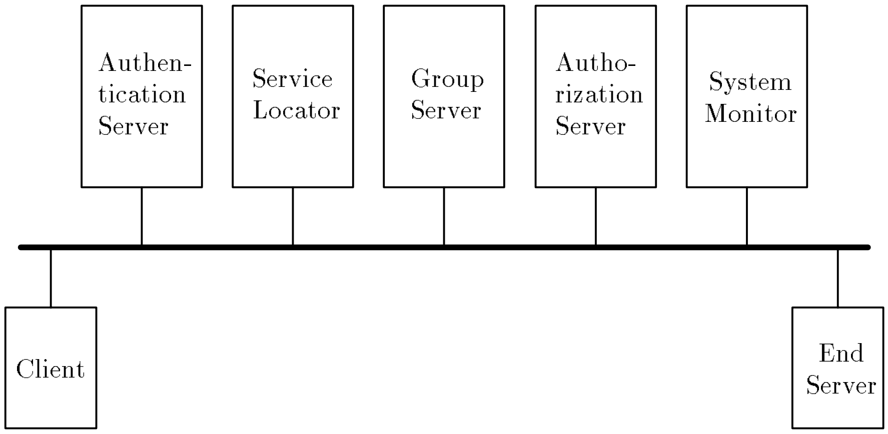

Figure 2: Distributed Authorization Framework

clients and authorization servers), and those between servers (e.g., end servers and authorization servers, system monitors and authorization servers).

- Authorization server — An authorization server performs authorization on behalf of an end server. Each end server can elect to offload its authorization to an authorization server. To do so, it needs to *contract* an available authorization server for this purpose. This requires the use of a *contracting protocol.* We will say more about this protocol in the next subsection. An authorization server hands out *authorization certificates* to authorized clients. These certificates are to be forwarded by clients to end servers along with their requests.

- Group server — A group server maintains and provides group membership information. From the perspective of authorization, its main function is to hand out two types of certificates: *membership* and *nonmembership* certificates. The former asserts that a client belongs to a particular group while the latter asserts the opposite. These certificates are requested by clients, and are to be forwarded to the authorization server together with their requests.

- System monitor — A system monitor tracks the values of system predicates. Typically, this is done by the monitor as well as a set of processes executing a distributed algorithm. Such a system monitor, however, cannot be expected to return the precise value of a system predicate at a particular time due to the asynchronous nature of distributed computation. Rather, if a system predicate is *stable*, then the monitor would eventually return its correct value.

We note that the above servers are only logically disjoint, they could easily be implemented as an integrated server or located on the same machine. To enhance efficiency, these servers can also be distributed[4] and/or replicated.[5] These servers are assumed to be *trusted.* For example, a group server is trusted to maintain and hand out correct membership information. A standard technique to ensure such trustworthiness is to implement these servers on dedicated machines that are physically secure (cf. Kerberos [2, 8]).

4.2 Operation and Protocols

In this section, we describe operational aspects of our framework, as well as the protocols needed in the framework. Due to length limitation, we will discuss just the the key ideas and omit details such as message format, file format and encryption/decryption issues.

When an end server E (who has elected to offload its authorization) starts up, it locates (possibly through a service locator) and contracts an authorization server A using a contracting protocol which performs several functions:

[4] This refers to the partitioning of a distributed system into subsystems and the assignment of distinct servers to handle the subsystems.

[5] This of course would bring in a number of standard distributed system problems (e.g., consistency) that need to be separately addressed.

- It mutually authenticates E and A, and distributes a new secret session key k for use between E and A.

- It establishes a *delegation key* k_d between E and A. The key k_d will be used by A to sign authorization certificates.

- It transfers an authorization specification *spec* from E to A. *spec* contains a specification of authorization requirements written in GACL, and will be used by A to determine authorization. The integrity of *spec* is protected by signing it with the session key k.[6]

Upon successful contracting, E notifies the service locator that A is its authorization server. This allows the service locator to direct clients of E to A first.[7]

There are two basic approaches to determine authorization using *spec*: *compilation* and *interpretation*. Compilation refers to the translation of *spec* into some form of executable specification that can be directly activated in making authorization decision. Interpretation refers to the use of a fixed algorithm to examine *spec* each time an authorization is to be determined. Compilation is preferred if *spec* is relatively static (e.g., for authorization of fixed system resources like printers) while interpretation is preferred otherwise. A hybrid of these alternatives is possible. For example, *spec* can first be translated into some intermediate form which can then be interpreted.[8]

Before contacting E, a client C contacts A to obtain the proper authorization. An authorization is typically in the form of an *authorization certificate* signed by A using k_d that contains, among other information, an *authorization key* k_a that is only known to C (and A of course). C can later submit this certificate to E to obtain the desired service. Knowledge of k_a is used by C to demonstrate to E that the authorization certificate was indeed obtained from A.

A only issues the appropriate authorization certificate to C after it has determined from *spec* that C can be granted access to E. The determination procedure may require C to submit certain group certificates to satisfy A, and can be iterative. That is, as A examines the entries in a particular gacl, it may request from C additional group certificates.[9] Indeed, C may not be aware of the group certificates that are required until instructed by A.[10] Hence, several message exchanges may be necessary before an authorization can be determined. We illustrate this with several examples in the next subsection.

Caching could be used to enhance efficiency. However, caching and the related issue of certificate expiration have correctness implications. For example, if cached group certificates are not invalidated when group membership changes, there may be incorrect grant or denial. Similarly, an unexpired authorization certificate should be invalidated when the particular authorization has been revoked. These issues are similar to those concerning the use of *capabilities*, and are beyond the scope of this paper.

4.3 Authorization Walkthrough

In this subsection, we present several authorization scenarios. We use the example requirements specified in Figure 1 as our authorization specification. Each scenario corresponds to a client request. We describe the messages exchanged in each scenario.

Let Charles be an individual who is a member of Dept, and Diane another individual who is a member of both Research and Dept. Also, let A denotes an authorization server and G a group server.

Consider a request to execute P.exe from Charles. We assume the request is accompanied by Charles's own identity credentials. Since gacl P.exe is an ordered list, A examines the entries in a sequential order. By checking the identity credentials of Charles,[11] A can easily determine that entry 1 does not apply. To dispose of entry 2, A requires knowledge of Charles's membership status regarding Dept. To this end, A sends Charles a *group membership status request* message regarding Dept and waits for a reply. Upon receiving this request, Charles retrieves his group membership certificate for Dept from his credential cache and forward a copy to A. However, if no such certificate can be found in the cache, Charles must request a fresh copy from G. This involves sending G a *certificate request* message, together with the appropriate identity credentials. Authorization completes when A receives Charles's group certificate for Dept.

[6] This is similar to a *zone transfer* in DNS, except that authorization data are involved here.

[7] Such redirection is similar to the use of MX records for *mail exchanges* in DNS. A major difference is that mail exchanges are responsible for forwarding mail to their final destinations, while authorization servers do not forward their decisions directly to end servers.

[8] Indeed, some form of pre-compilation of *spec* by E before transfer to A is also possible.

[9] This is commonly known as the *push* model. A *pull* model is one in which A itself gathers the relevant certificates from the group servers. However, it appears to be more desirable to reduce the load of A so that it does not become a bottleneck, even at the expense of the clients.

[10] This is typically the case when nonmembership certificates are needed by A.

[11] Individuals with different names are assumed to be distinct.

Alternately, the group membership status request message from A can be saved if Charles "remembers" to send along his group certificate for Dept together with his initial request. This of course requires prior knowledge on Charles's part.

We next look at an anonymous request. Suppose Charles desires to read P.doc anonymously,[12] and identifies himself only as a member of Dept (i.e., by sending only his group certificate for Dept along with his request, without his identity credentials). Since gacl P.doc is unordered, both entries 1 and 3 must be examined together to determine authorization. For entry 3, it is easy to see that no denial on read can be inherited by any member of Dept[13] Thus entry 3 does not apply. In disposing entry 1, A must be ascertained that the anonymous requestor is not Alice or Bob. To this end, A replies with a group membership status request message regarding the group {Alice, Bob}. To show that he is not Alice or Bob, Charles again goes to G for a group (nonmembership) certificate regarding the group {Alice, Bob}. With this, A completes the authorization.

Lastly, we consider a write request to P.src from Diane. Since gacl P.src is unordered, both entries 1 and 2 must be examined together. Similar to the above, A sends two group membership status request messages to Diane, one regarding Research and the other regarding Dept.[14] Again, Diane can simply forward the required group certificates to complete the authorization.

Alternately, if Research is always a subgroup of Dept, then Diane needs only return her group certificate for Research together with a *group relationship certificate* proving Research's subset relationship to Dept. A can easily deduce Diane's membership regarding Dept given her membership regarding Research and the subset relationship. These group relationship certificates should be cached by A for future use.

5 Conclusion

Distributed authorization is a relatively young area. Many issues still need to be explored and studied. The framework proposed in this paper is a first attempt at identifying and solving some of the problems.

Due to length limitation, we have omitted many of the details in this paper. We are in the process of implementing a prototype of our framework. Our current effort is focused mainly on implementing an authentication substrate upon which the authorization framework operates, and also on finding efficient evaluation strategies for GACL. We plan to report our implementation results in a future paper.

[12] Anonymous request to P.doc is allowed as indicated by the declaration in its gacl.

[13] Specifically, entry 1 in the ordered gacl Doc grants every member of Dept read access.

[14] This can be combined in a single message for an optimized implementation.

References

[1] M. Gasser, A. Goldstein, C. Kaufman, and B.W. Lampson. The Digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference*, pages 305–319, October 1989.

[2] J.T. Kohl and B.C. Neuman. The Kerberos network authentication service: Version 5 draft protocol specification. April 1993.

[3] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 165–182, October 13–16 1991.

[4] J. Linn. Practical authentication for distributed computing. In *Proceedings of the 11th IEEE Symposium on Research in Security and Privacy*, pages 31–40, May 7–9 1990.

[5] R. Molva, G. Tsudik, E. Van Herreweghen, and S. Zatti. *KryptoKnight* authentication and key distribution system. In *Proceedings of the European Symposium on Research in Computer Security*, November 23–25 1993.

[6] B.C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

[7] W. Rosenberry, D. Kenny, and G. Fisher. *Understanding DCE.* O'Reilley & Associates, Inc., 1992.

[8] J.G. Steiner, C. Neuman, and J.I. Schiller. *Kerberos*: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, February 1988.

[9] J.J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the 12th IEEE Symposium on Research in Security and Privacy*, pages 232–244, May 20–22 1991.

[10] T.Y.C. Woo and S.S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, January 1992.

[11] T.Y.C. Woo and S.S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the 13th IEEE Symposium on Research in Security and Privacy*, pages 33–50, May 4–6 1992.