Fine-Grained Access Control in a Transactional Object-Oriented System

Luis-Felipe Cabrera, Allen W. Luniewski
and James W. Stamos
IBM Almaden Research Center

ABSTRACT: We believe that access controls for object-oriented systems should be fine-grained and thus apply to individual methods of individual objects. The efficient support of fine-grained access control is challenging because a check is done on every method invocation. We present a design that uses access control lists (ACLs) and exploits virtual memory facilities to make these checks run fast. The costs include an extra level of indirection for method invocation and per-user storage for preprocessed access control information. Given a choice between immediacy of revocation and serializability of transactions, we selected a compromise that uses a nested top-level transaction for each invocation of an ACL method.

1. Introduction

In the Melampus vision [2], a single object-oriented system would efficiently store, retrieve, and process a diversity of on-line information for a large enterprise. Such a system would provide *navigational* access as well as nonprocedural, *associative* access. Navigational access selects objects by following pointers, while associative access selects objects that match a descriptive predicate. In general, such a system would be large, continually evolving, and geographically distributed. Atomic transactions would be an important foundation for applications.

We believe that access control should be based on logical operations on abstract objects (*i.e.,* methods) rather than physical operations on files or segments that contain objects. For example, read-write permissions cannot distinguish between the **enqueue** and **dequeue** methods of a request queue, because a user must either be authorized for both or for neither. Permissions based on individual methods, in contrast, allow more possibilities, since a user may be authorized for any subset of methods.

We also believe that access control should be based on individual objects rather than on all objects of a class. Different users will instantiate the same class for different reasons and thus will have different access control needs. A single user may also have a variety of access control needs for objects of a given class.

Fine-grained access control based on individual methods of individual objects has enough expressive power to emulate other designs, like Hydra [13], in which rights and methods are not in a one-to-one correspondence. Our one-to-one correspondence means that the designer of a type need not anticipate all the relevant rights for a type that may be used by diverse applications.

Existing access control mechanisms for object-oriented systems cannot be used for Melampus. For example, access control mechanisms for persistent object systems [8] do not support associative access. Access control mechanisms for object-oriented databases, such as Rabitti et al. [7], support associative access, but have not addressed fine-grained access control. This paper presents a design that supports efficient access control at the level of individual methods on individual objects.

Our design for access control has several goals. First, it should prevent unauthorized method calls on objects. Second, revocation of access rights should take effect immediately. Third, the cost of access control should not be excessive. Fourth, the system should support mutual suspicion so that a user can execute untrusted methods in a safe fashion.

We make the following assumptions:

- programs are written in a safe subset of a strongly-typed, high-level language;
- all executable code is produced by a trusted compiler;
- at each point in time, each thread in Melampus runs on behalf of some atomic transaction; and
- each distributed transaction uses an atomic commit protocol to ensure that all participants agree on the outcome.

The presence of serializable transactions conflicts with our goal of immediate revocation. Serializability makes a concurrent system easier to understand, prove correct, and use, but it may block a transaction for an arbitrarily long period of time. Revocation immediacy, however, cannot tolerate blocking. For Melampus, we advocate that immediacy take precedence over serializability.

The remainder of the paper is structured as follows. In Section 2 we discuss access control in Melampus. Section 3 defines the semantics of ACLs. In Section 4 we present our support for efficient access controls. Section 5 examines the support for immediate revocation. Section 6 discusses related work, and Section 7 presents our conclusions.

2. *Access Control in Melampus*

A *principal* is an entity to which authorizations are granted [10]. In Melampus, each method invocation is done on behalf of some principal called the *current principal* (CP). If the CP is not authorized, the method is not invoked and an exception is raised. Every CP is always authorized to apply methods to instances of the built-in types such as integer, array, and record.

In multi-user systems, suspicion may exist between any pair of principals. In Melampus, suspicion may exist between the CP and the owner of an object on which the CP wishes to invoke a method. If method execution takes place with the method caller as the CP, then the caller is vulnerable to the misdeeds of unknown code. If execution takes place with the object's owner as CP, the object's owner must ensure that his privileges are not used by the caller in unanticipated ways.

This mutual suspicion is exacerbated in Melampus by queries, because a single query may manipulate many objects. Consider the query "Show me all objects that have a method M that returns a string equal to 'felipe' when M is applied to the object." The invoker of such a query often has no knowledge of the objects the query manipulates or their owners and implementers. The power of such queries also increases the severity of a security breach when a principal's privileges are abused.

Melampus addresses these problems by automatically determining how and when the CP changes. Upon entry to a method, the CP is changed to the owner of the object whose method is being invoked. Upon exit, normal or abnormal, the CP is returned to its previous value. Thus, the CP follows a strict LIFO stack discipline which we call the *principal stack*. The program has no explicit control over the principal stack or the CP. In this approach, a method cannot use the privileges of its callers, nor can a caller abuse the privileges of object owners. The caller must ensure that the callee is authorized to manipulate the method arguments. Note that the owner of an object must trust the implementation of the object, because its methods execute with the full privileges of the owner.

An *access control list* (ACL) [9] is a set of authorized principals. We decompose the access-control matrix [5] into access control lists rather than capabilities because an ACL localizes access control information and thus facilitates revocation. In addition, ACLs seem preferable to capabilities in the presence of associative access to objects.

Our granularity of authorization protects each method of every object with an ACL. Each site has a *trusted protection manager* (PM) that performs access control checks and administers cached ACL information. ACLs may be shared, because several methods from the same or different objects may be protected by the same ACL. Based on data gathered from our AFS [11] installation at the IBM Almaden Research Center, we expect a high degree of sharing of ACLs. For example, in Section 4.4 we use extrapolation to show that in a system like Melampus the number of ACLs could be four orders of magnitude smaller than the number of objects.

3. ACL Semantics

In addition to traditional set operations such as **insert, delete,** and **lookup,** each ACL supports two methods dealing with revocation:

- **setRevocationPolicy(A,P)** sets the revocation policy of ACL A to P. P is either passive or aggressive.
- **getRevocationPolicy(A)** returns the revocation policy of ACL A.

The semantics of **delete** depend on the revocation policy in effect at the time of deletion. Suppose principal Alice is inserted into ACL A and later deleted from A. If A has a passive revocation policy when Alice is deleted, the **delete** method simply removes Alice from A. If A has an aggressive revocation policy, the **delete** method removes Alice from A and then aborts all ongoing transactions that noticed that Alice was a member of A while A had an aggressive revocation policy. Transactions that used A during this time but did not notice that Alice was in A are not aborted. This is implemented by having the ACL implementation inform the local transaction manager whenever a transaction refers to an ACL that has an aggressive revocation policy. When Alice is removed from A, the ACL implementation sends an abort message to the local transaction manager. The local transaction manager adds all transactions that noticed that Alice was in A while A had an aggressive revocation policy to a volatile list of ongoing transactions that should be aborted. The transaction manager consults this list during the voting phase of the commit protocol and votes to abort a transaction if it appears on the list.

The serializable nature of transactions requires that a transaction that deletes a principal from an ACL logically execute after every transaction that previously inserted the principal in the ACL or that noticed its presence in the ACL. This conflicts with immediacy of access control revocation, which requires that a **delete** method execute immediately.

Our solution to this dilemma involves *nested top-level transactions* [6]. Nested top-level transactions let one transaction (T1) synchronously invoke a separate transaction (T2) in such a manner that the commit/abort decisions of the two transactions remain independent. The application programmer (or end user) is responsible for any compensation; *i.e.,* undoing some or all of the effects of T2 if T1 ultimately aborts.

Each public method on an ACL executes as a nested top-level transaction. Thus, when an ACL method completes, any locks it set on the ACL may be released. This allows for the possibility of fast **lookups** and immediate revocation without unduly complicating the semantics or implementation of ACLs. Because nested top-level transactions may require explicit compensation, each method that modifies an ACL returns sufficient information for the user or application to do any later compensation.[1]

4. *Support for Efficient Access Control*

Since an access control check is made for every method invocation, a naive implementation of access controls (*e.g.,* compiler generated checks) could severely degrade performance. The dynamic nature of this protection scheme precludes the compiler from optimizing out the run-time protection checks. To counter these problems we cache ACL information (Section 4.1), we exploit special cases (Section 4.2), and

1. Note that when at least two users can modify the same ACL, ACL updates that involve the same principal require additional coordination in order for compensation to be correct.

we use paging hardware (Section 4.3). For simplicity, in this paper we consider only a single site implementation. Extensions needed to handle distribution are discussed in Stamos et al. [12].

4.1 Caching

The PM caches the results of ACL lookups as triples of the form <ACL, principal, boolean>, where the boolean indicates whether the principal is in the ACL. The protection manager consults an ACL only after a cache miss. ACL updates (*i.e.*, insertions and deletions) are propagated to the cache. This provides immediate revocation as well as immediate authorization.

4.2 System-Defined ACLs

A system-defined ACL is an authorization "common case" that is represented by a distinguished, reserved, object identifier. These object identifiers are known to the system and cannot be used for any other purpose. When a system-defined ACL is encountered, the protection manager performs a trivial, ACL-dependent computation rather than consulting its cache or the ACL. For example, the *owner-only* ACL lets only the owner of an object perform a method, whereas the *world* ACL lets every principal in the system perform a method, and the *system-administrator* ACL lets the local administrator of the system perform tasks such as backup. The object identifiers of these system-defined ACLs distinguish them from ACLs whose contents are determined by users.

4.3 VM-based Support for Access Control

Because of the extremely high frequency of access control checks, caching and special cases together will still not make the approach practical. In this section we outline an approach that uses a computer's virtual memory facilities to provide an efficient implementation of ACL-based protection.

The generated code for method invocation in conventional object-oriented systems first finds a method dispatch vector for the object whose method is being invoked. Then, the i^{th} entry from that vector is used to invoke the ith method. An efficient ACL implementation is obtained through two techniques:

1. the method dispatch vector is made dependent upon the CP; and
2. an extra level of indirection is introduced between the method dispatch vector and the actual methods.

The first technique allows principal-dependent changes to be made to the state of memory. The extra level of indirection isolates the effects of revocation to a single principal.

Figure 1 illustrates the data structures involved in this approach. When an object is mapped into the address space of a process, a surrogate for that object, the *local object,* is created in that address space. The local object has three components: a pointer to the real object, a local owner field, and a pointer to a local implementation. The pointer to the real object allows the method code to find the instance variables for that object. The local implementation is a principal-dependent method dispatch vector for this object. All data structures in Figure 1 other than the local object and the local implementation are shared by all principals.

All local implementations for a given principal are gathered into a single segment, called the *local object segment,* which is mapped at a well-known place in the address space of the process. The local implementation for a given object is at the same offset within the object segment for all principals. When the CP changes, the local object segment for the new CP is mapped into the address space. Observe that the local implementation pointer in any local object is always valid since the object segment is always mapped to the same location and the local implementations for the object are at the same location.[2]

Each entry in a local implementation is either **null** or it indirects through another segment, the *ACL segment.* A **null** pointer implies that either access to this method is not permitted or that the protection manager has not yet determined if access is permitted.

2. An alternative implementation for local object segments is to partition a portion of the address space into nonoverlapping ranges called partitions, assign each partition to a different principal, and use the contents of an index register to select the appropriate partition. In this scheme, local implementations for the same object are at the same offset in each partition, and changing the CP is as simple as updating the index register.

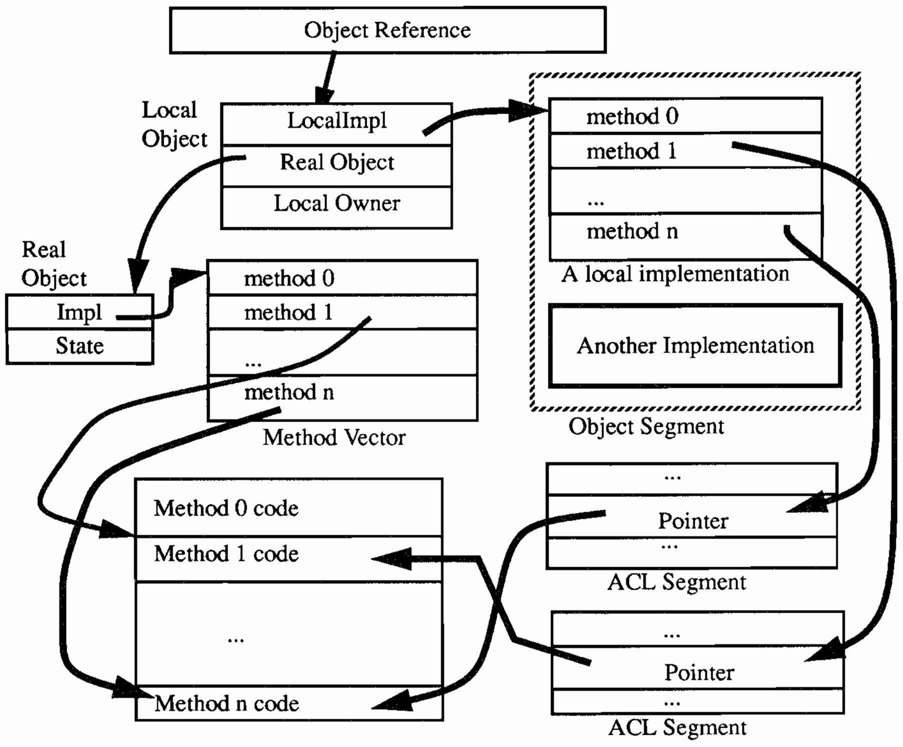

Figure 1 Data structures to implement efficiently ACLs.

There is one ACL segment for each pair of ACL and principal.[3] Each entry in an ACL segment is either **null** or a pointer to the appropriate method code. If **null,** then either access is denied to that method or access to the method needs to be validated. Note that an entry in an ACL segment may be referred to by many pointers in local implementations (implying that multiple objects of the same type are protected by the same ACL).

If a **null** pointer is encountered in a local implementation, the protection manager fields the fault and must determine, and interrogate, the ACL associated with that method. If access is denied, an exception

3. This implies many segments. Alternatively, the set of ACL segments for a given principal could be gathered into a single segment whose mapping is changed upon change of CP.

is raised. If access is permitted, an ACL segment must be found[4] for that ACL and the CP, and an entry made for this method in that segment. The process is then resumed and the method invocation proceeds.

If a **null** pointer is encountered in an ACL segment, the protection manager fields the fault and determines if access is permitted to that method by the current CP. If access is not permitted, an exception is raised. If access is permitted, the entry in the ACL segment is altered to refer to the correct method code and the invocation is resumed.

Revocation occurs when a principal is removed from an ACL. When this happens, the system unmaps the ACL segment associated with that ACL and principal. Observe that this is a constant time operation. The next time that the process attempts to indirect through that ACL segment, a fault to the protection manager will occur. The protection manager will then determine if the CP is allowed access to that method. If access is denied, an exception is raised. If access is permitted, a new ACL segment is mapped with all **null** pointers in it. The entry for the method currently being invoked is then set to refer to the appropriate method code, and the process is resumed.

Revocation may also occur when one ACL is replaced by a second ACL as the ACL for an object's method. All principals that are in the original ACL but not in the new ACL must have their access to the object's method revoked. This is achieved by updating the local implementation of the object for each affected principal: a **null** pointer is written in the entry that corresponds to the method. The next time one of the affected principals tries to invoke the object's method, a **null** pointer is encountered in the local implementation. The protection manager fields the fault and handles it as explained above.

Access rights are granted when a principal is added to an ACL; this authorization takes effect immediately. If we kept the mapping from each ACL to the set of objects that use the ACL for access control, the affected local implementations and ACL segments could be updated immediately. However, to reduce space consumption, we do not keep the mapping from ACLs to objects. We instead defer the up-

4. An ACL segment is created if necessary.

dates until the protection manager fields a null-pointer fault and establishes that the current principal is indeed authorized to invoke the current method.

Access rights may be granted when one ACL is substituted for another ACL. Again, authorization takes effect immediately. Because we know the specific object and the specific ACLs, the affected local implementations and ACL segments can easily be found and updated. Alternatively, these updates could be deferred until the protection manager fields a null-pointer fault.

Our approach permits method invocation to proceed with low time overhead in the normal case—one extra level of indirection. There is additional CPU overhead when a valid invocation involves a trip through the protection manager (the abnormal case). Our approach imposes two other costs: there is a space cost for the local object segments and the ACL segments, and changing the CP involves remapping the object segment.

4.4 ACL Statistics

The space and time efficiency of this implementation depends on the number and size of ACLs. Since Melampus envisions a world with a very large number of objects and principals, the Melampus protection implementation depends on moderate growth in the number and size of ACLs as the number of objects and principals grows.

To understand these issues, we examined the ACLs in our local AFS [11] fileservers to help us predict the size, nature, and shareability of ACLs that might be found in a system like Melampus. Although AFS protection is not as fine-grained as it would be in Melampus, AFS is a real system, and statistics on its ACLs could be obtained.

Each directory in AFS has its own ACL, and all files in the directory share the ACL. Each entry in the ACL lists the set of rights granted to a principal for that directory.

We considered 18,874 AFS directories and hence 18,874 ACLs. There were 243 distinct principals listed in the ACLs; most principals were individuals. The remaining principals were roles individuals would assume or groups of individuals such as projects or departments. Each ACL contained 2 to 12 entries. The average ACL size was 3.1 entries, and only 2.5% of ACLs contained more than 4 entries. There were 367 ACLs with different contents. If ACLs could be shared, the number of ACLs would be between 367 (maximum sharing) and 18,874 (no sharing).

Because there are 7 AFS rights (rliwdka), one can view each directory as having 7 fine-grained ACLs. Each fine-grained ACL would be a set of principals, since the single right would be understood from the context. We considered all $18{,}874 * 7 = 132{,}118$ fine-grained ACLs. There were 636 distinct, fine-grained ACLs, and their average size was 1.9 principals. Note that the number of ACLs increased by a factor of 7, but the number of distinct ACLs increased by only 1.7. These results suggest that the number of distinct ACLs scales well with the average number of methods per object. For example, if the growth is linear and each object has 20 methods, the number of fine-grained ACLs would grow by another factor of $20/7 = 2.9$ for a total increase factor of 4.9.

To test the sensitivity of our results to the number of ACLs, we selected every N^{th} directory ($1 <= N <= 20$) and repeated the above analysis. The number of distinct (fine-grained) ACLs grew sublinearly with the number of directories. Let A be the number of distinct ACLs, and let D be the number of directories. Using a least-squares fit on a log-log plot, we determined that $A = 30 * D^{0.25}$. For fine-grained ACLs, we determined that $A = 77 * D^{0.21}$. These results suggest that the number of distinct ACLs scales well with the number of objects.

To test the sensitivity of our results to the number of principals, we selected every M^{th} principal ($1 <= M <= 40$) as well as 3 frequently-used system principals of the form system:*. We completely ignored the other principals and directories administered only by ignored principals. We learned that the number of distinct ACLs per principal varied between 1.2 and 3.0, while the number of distinct, fine-grained ACLs per principal varied between 1.9 and 3.4. The results suggest that the number of distinct ACLs varies linearly with the number of principals. The average size of ACLs varied between 1.9 and 3.1, while the average size of fine-grained ACLs varied between 1.3 and 1.9. The average size tended to increase slightly with the number of principals.

Our analysis of AFS ACLs lets us estimate the number of distinct ACLs in Melampus. Consider a Melampus system in which every principal owns 1 million objects and the average number of methods per object is 20. The least-squares fit predicts less than 22,000 distinct, fine-grained ACLs for a Melampus system with 243 principals. If we scale the system so that there are 10,000 principals and 10 billion objects, we would expect to have less than 1 million distinct, fine-grained ACLs. Note that the number of ACLs is more than four orders of magnitude smaller than the number of objects.

In summary, AFS ACLs were small, and a high degree of ACL sharing was possible at various granularity levels. We believe that similar results would be obtained if AFS directories and files were replaced by Melampus objects in our research environment. Our sensitivity analysis suggests that scalability in terms of the number of objects, methods, and principals would not be a problem.

Our analysis raises some concerns, since it implies a large degree of sharing of ACLs. This means that changes to an ACL with an aggressive revocation policy may abort many transactions. There are four reasons why this should not be a serious problem in practice. First, we believe that ACL changes are likely to be infrequent. An average ACL size of 3.1 suggests that many ACLs are "owner+world" and thus rarely change. Second, at any point in time most objects, like most files in file systems, will not be accessed. Revocation affecting only idle objects will not abort any ongoing transactions. Third, adding a principal to an ACL does not abort ongoing transactions. Fourth, in many cases removing a principal from an ACL will not abort ongoing transactions. For example, in a fairly friendly environment **delete** operations generally reflect organizational changes and remove a principal from group-related ACLs. In this case it is likely that the affected user has no ongoing transactions that would be aborted.

5. *Support for Immediate Revocation*

The operational semantics we provided for the **delete** method suggests a strategy for implementing passive revocation and aggressive revocation. To implement aggressive revocation, protection managers track dependent transactions in volatile storage. For cached results (Section 4.1), for example, each protection manager associates each of its cache entries with the ongoing transactions that depend on the cache entry.

An indirect way of revoking access is by substituting one ACL for a second ACL as the ACL for an object's method. ACL substitution is a nested top-level transaction. All ACL methods that execute after ACL substitution are governed by the revocation policy of the new ACL. Transactions that depend on principals being in the old ACL are dealt with according to the revocation policy of the old ACL. Nothing needs to be done if the old ACL has a passive revocation policy. If the old ACL has an aggressive revocation policy, the system considers each principal that has transactions depending on it for the object and method in question. If such a principal is not present in the new ACL, the system votes to abort the corresponding transactions. Otherwise, the relevant transaction dependence information is reestablished if the new ACL has an aggressive revocation policy.

6. *Related Work*

A great deal of research has been done in the area of computer security. We discuss the design principles that we did not adopt as well as access control mechanisms in object-based systems.

Saltzer and Schroeder [10] proposed eight design principles for secure systems; we adopted all but two. *Separation of privilege,* which requires multiple keys to unlock a security mechanism, is not immediately applicable to our design, but may be applicable to enhancements implemented by users.

Least privilege requires that users and programs operate using the least set of privileges necessary to complete the job. This principle conflicts with the database policy of *maximized sharing* [3], which permits a user to make maximum use of the information in a database. In our design, which maximizes sharing, the privileges of a principal depend only on the mapping from object methods to ACLs and the contents of ACLs. We support mutual suspicion by changing the protection domain when the current principal (CP) is changed.

The BirliX Operating System [4] implements a least privilege policy for untrusted principals by using subject restriction lists (SRLs) in addition to ACLs. In BirliX, a trusted principal can access an object if the object's ACL permits the access. An untrusted principal can access an object only if both the object's ACL and the principal's SRL permit the access. We did not pursue this approach because it conflicted with our policy of maximized sharing.

Our design uses existing techniques such as ACLs [9], a run-time stack of principals, nested top-level transactions [5], and caching. Protected subsystems [10], for example, are constructed using the principal stack and access control checks at the level of methods.

The Melampus protection model changes the current principal on every method invocation. This has some similarity to the Unix *setuid* mechanism which has some known security flaws [1]. Melampus avoids these flaws through three techniques. First, in Melampus the privileges available to a principal during method execution are precisely the privileges granted to that principal. In contrast, with setuid the setuid principal gains the privileges of its caller. Second, in contrast to the setuid facility, there is no programmatic way to explicitly change the current principal. Third, for the purpose of keeping a security audit trail, the CP stack provides the complete current CP history of the process, thus allowing a faithful audit.

7. Conclusion

This paper presents a design for efficient fine-grained access control that supports immediate revocation even in the presence of transactions. We have developed these ideas in the context of Melampus, but we believe they apply to any object-oriented system.

Our key contribution is the identification of discretionary access control problems in a multi-user, distributed, transactional, object-oriented system that permits associative access. Our design addresses these problems by supporting mutual suspicion and by combining serializable transactions with immediate (and thus nonserializable) revocation.

Our design contains a fine-grained access control mechanism that lets the owner of an object specify the set of users that may invoke each of the object's methods. We provide a protection mechanism based on access control lists for granting access rights and a stack model for switching between protection domains. For reasons of autonomy and efficiency, the system caches results of lookup operations on access control lists. These caches are outside the transaction mechanism and are subject to immediate revocation. Owners of access control lists determine the trade-off between the semantics of revocation and the overhead of execution.

We plan to exploit virtual memory hardware to realize an efficient implementation of our design. The high frequency of (successful) access control checks and the low frequency of access right changes will let us skew the implementation appropriately.

Acknowledgements

We thank Joel Richardson and Peter Schwarz for their help in the design of the Melampus protection system. Their comments, together with those of Laura Haas, have also helped us improve the presentation.

References

[1] S. Bunch. The SETUID feature in Unix and security. In *Proceedings of the 10th National Computer Security Conference,* pages 245–253, September 1987.

[2] L.-F. Cabrera, L. Haas, J. Richardson, P. Schwarz, and J. Stamos. The Melampus project: Toward an omniscient computing system. Technical Report RJ 7515, IBM Almaden Research Center, San Jose, California, June 1990.

[3] E. B. Fernandez, R. C. Summers, and C. Wood. *Database Security and Integrity*. Addison-Wesley, Reading, Massachusetts, 1981.

[4] O. C. Kowalski and H. Härtig. Protection in the BirliX operating system. In *Proceedings of the 10th International Conference on Distributed Computing Systems,* pages 160–166, May 1990.

[5] B. W. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems,* pages 437–443. Princeton University, March 1971. Reprinted in Operating Systems Review, 8, 1, January 1974, pp. 18–24.

[6] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Prog. Lang. Syst.,* 5(3):381–404, July 1983.

[7] F. Rabitti, D. Woelk, and W. Kim. A model of authorization for object-oriented and semantic databases. In *Proceedings of the International Conference on Extending Database Technology,* pages 231–250, March 1988.

[8] J. Rosenberg and J. L. Keedy, editors. *Security and Persistence.* Springer-Verlag, May 1990.

[9] J. H. Saltzer. Protection and the control of information sharing in Multics. *Commun. ACM,* 17(7):388–402, July 1974.

[10] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE,* 63(9):1278–1308, September 1975.

[11] M. Satyanarayanan, J. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles,* pages 35–50. ACM Special Interest Group on Operating Systems, December 1985.

[12] J. W. Stamos, L.-F. Cabrera, and A. W. Luniewski. Mutual suspicion, immediate revocation, and serializability in Melampus. Research Report RJ 8161, IBM Almaden Research Center, San Jose, CA, June 1991.

[13] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Commun. ACM,* 17(6):337–345, June 1974.

[received April 3, 1992; revised June 17, 1992; accepted July 2, 1992]