Security Architecture for Component-based Operating Systems

Trent Jaeger Jochen Liedtke Vsevolod Panteleenko Yoonho Park
Nayeem Islam

IBM Thomas J. Watson Research Center, Hawthorne, NY 10532, U.S.A
(email: {jaegert,jochen,vvp,yoonho,nayeem} @ watson.ibm.com)

1 Introduction

We present a security architecture that system administrators, users, and application developers can use to compose secure systems from components. There are two major issues in the development of a security architecture for a component-based system: (1) that the architecture can enable the enforcement of effective security policies and (2) that the architecture can support the dynamics that systems composed from components will possess. Effective security policies (which we will define below) must be enforced by the system, but their implementation is complicated by the dynamic nature of system composition. The dynamics of system composition affect security enforcement by complicating: (1) the assignment of permissions to components because they can be run in more than one context; and (2) the authorization of object accesses because all servers cannot necessarily be trusted to enforce system security policy.

As has been known for quite some time [24, 15, 11], an effective security policy must be able to control *all* accesses to *all* objects exported by components (and protect itself from tampering [1]). Unfortunately, early component-based systems, such as Mach [18], Chorus [19], and Spring [16], only control access to component communication, not objects. In addition, despite the limited security flexibility provided by these systems, they still were shown to display less than effective performance [12]. Therefore, building a high-performance system that still provides the degree of security desired is difficult.

The dynamics of component-based system are that the set of components used by the system and objects defined and used by the components may not be known at boot time. However, the security architecture must be able to apply the system security policy effectively to components as they are loaded. We consider components to be analogous to dynamically-linked libraries (DLLs) (even when placed in separate address spaces) in that they can be loaded into different execution contexts based on the needs of the requestor of their services. Context-sensitivity in access control is being addressed by some Role-Based Access Control (RBAC) models [7, 13]. For example, principals can be parameterized, so that their permissions can be derived based on the runtime context. We examine the use of such models.

The issue that each component may develop its own object space is more troublesome to enforcing security. In a system with a fixed set of components (i.e., servers), the security administrators could specify the set of system objects and the security policy governing them. For example, DTOS [15] specifies the access rights of principals to nine system servers. The policy is passed to the servers who verify the access. We do not believe that it is possible for all servers to provide their own security architecture to effectively enforce each system's security policy. However, the security architecture cannot enforce its policy on server objects to which it has no knowledge. Our goal is to define the information that servers must provide to the security architecture to enable enforcement.

Language-based security, such as those architectures designed for Java [8], Tcl [17], Python (Grail [20]), and Perl (Penguin [6]), has gained popularity recently due to its portability and perceived performance advantages. We have shown that a security architecture that runs compiled code and has a moderate number of interdomain crossings can approximate the same performance (or perhaps provide better performance) than language-based systems (even running JIT-compiled code) [10]. In addition, address space protection has several security advantages: (1) a smaller TCB; (2) a simpler separation mechanism (to better protect the TCB); (3) complete system mediation; and (4) a history of proven security features.

Our approach is to implement security architecture at the systems level that enables enforcement of component-based system security policies. The security architecture is implemented outside of the system nucleus which enables different security mechanisms to be attached to different component processes. Thus, choices about security flexibility and performance can be made without impacting the the nucleus. The architecture supports the dynamic assignment of permissions to tasks and the communication of dynamically-developed object spaces. We have developed a prototype implementation of a security architecture on the Lava Nucleus. In subsequent sections, we describe the security problems that need to be solved, the system and security architectures, and solutions to the security problems using the architecture.

2 Problem Definition

A component-based operating system has a small, fixed, trusted computing base (TCB) and composes its system services from individual components. A component is an implementation with a publicly-available interface that other components use to access its services. Components may be loaded into the same address space or separate address spaces. In both cases, the code is the same; a component runtime converts inter-task method invocations into IPCs transparently. Any service can be a component, and we expect that multiple versions of services will be likely (e.g., to offer different functionality or performance optimizations). For example, our virtual memory system will have multiple memory object managers each implementing different memory management policies (e.g., pinned or not). Thus, we expect that component-based systems will have a greater number of "servers" than previous systems, such as Mach-based systems.

A long desired goal of security researchers has been to run system services in protection domains commensurate with their security requirements. A component-based operating system makes this goal achievable, in theory, but additional, practical issues must be addressed. In particular, all programs in a component-based system other than the TCB (both applications and other kernel programs) are analogous to dynamically-linked libraries. Such a library must execute in the context in which it is linked (recall that programs may be loaded into separate address spaces and "linked" via transparent IPC). The public interfaces of components and their ability to be dynamically linked enables other programs to load them for whatever purposes they deem necessary.

This model of component-based systems uncovers two major security implications. First, because a component may be used by a variety of principals, in a variety of situations, and may not always be completely trusted, a principal may wish to restrict the rights of the components it uses. Second, since the servers may not comprehend these restrictions, they may not be able to effectively interpret the restricted rights of such "dynamically created" principals.

Consider the following example. A principal (application or service) loads a component to perform a service on its behalf. Since the component is loaded for a specific purpose, its permissions should be restricted to those necessary for that purpose. Note that the enforcement of this restriction requires both proper authorization using the limited permissions and restricted delegation to this restricted task. Delegation is not restricted in early micro-kernels, such as Mach [18], Spring [16], and Chorus [19]. In addition, system security policy is enforced by the servers, so no restrictions external to the servers can be applied. DTOS [15] both enables the enforcement of system policy and restricts delegation. This is achieved by passing system policy with capabilities to the servers. Therefore, DTOS still depends on servers to understand and enforce the extensions specified in the capabilities. We do not believe that this will be feasible in a dynamic component environment because: (1) the number of servers and, hence, ad hoc security code will become unmanageable and (2) the servers may not be able to interpret the security requirements of the system. While system security libraries can be transferred to each server to enforce the system's policy, this does not solve the next problem.

In addition, since the system needs to authorize accesses upon objects, it needs to know the system object space. This is a particularly complex problem because each component may create security-sensitive objects. For example, a memory manager can create a memory object on behalf of one principal. The access rights of other principals to this object must be controlled based on the permissible sharing. For example, the creator may not be permitted to map this object into another principal's address space (i.e., delegate a capability to it). In general, each principal may dynamically create a set of objects and other principals may be able to perform operations upon those objects as well. The problem is to enable the security architecture to map security policy to these dynamically created objects.

Lastly, different component principals may not only have different permissions, but also different mechanisms for managing them. In a mandatory access control setting, a component may be associated with a principal with a fixed set of permissions. Delegations must occur within those rights. Another option is for multiple principals to be able to delegate rights to a component principal, each within different domains. This enables different servers to limit the rights that they can delegate independently (within their individual mandatory access control restrictions). Permissions closer to least privilege can be assigned if all the principals that understand the semantics of the use of the component can be involved in the assignment of its rights. Also, restrictions of a principal's permissions, both temporary and permanent, are possible. For example, a principal's rights may be intersected with those of another (e.g., Java security policy). Security mechanisms should enable any of the above mechanisms to be applied to component principals.

To summarize, we list the following requirements for a system security mechanism for component-based systems:

- Map abstract policy to actual components and objects
- Authorize all object accesses
- Limit capabilities delegated to components
- Contain a small number of unique permission management mechanisms
- Protect itself from compromise

Other notable systems have goals similar to these. Early capability-based systems, such as Hydra [24] and SCAP [11], endeavor to provide complete mediation to objects. Such systems were deemed insufficient performers for general computing, but we believe that such requirements are now becoming valid. Exokernel is an example of a recent system security architecture that enables restricted permissions to be assigned to tasks [14]. It uses hierarchical capabilities (actually principals) to create principals with a subset of their creators' permissions. The selection of the new principal's rights and the means to update all the effected ACLs is not specified. The OSKit is a component-based operating system environment [5]. The OSKit team is using the DTOS security architecture [15] with a Domain Type Enforcement access control model [2] although the details of its application are not yet available.

3 Architecture

The goal of the Lava system architecture is to enable the dynamic composition of systems from components while enforcing the system's security policy. The security requirements of the components are expressed in the access control model. First, we detail the system architecture of the Lava system, then we define the access control model that it uses.

3.1 System Architecture

The Lava system (see Figure 1) consists of a nucleus, security architecture interface, resource managers, reference monitors, and components. The nucleus provides basic system functionality, including address spaces, threads, and interprocess communication (IPC). Also, the nucleus provides IPC redirection which enables intertask messages to be forwarded to a specified task (in this case, the reference monitor).

The SAI provides an interface to load components into system tasks (i.e., processes) in such a way that the system's security policy can be enforced. The SAI can load components into new or existing tasks, but the permissions associated with an existing task may be changed when code is loaded into it. The SAI authenticates components and derives the permissions for the task in which the component is loaded.

In order to enforce a component's security requirements, the SAI associates a reference monitor with each component task. The monitor is trusted to enforce the system security policy. The monitor intercepts all IPC from or to the task in which it monitors. The monitors obtain object spaces from each server and authorize accesses using the object spaces. Once the security policy is derived, it is possible for system libraries linked into the servers to enforce the policy. However, protection of the authorization code in monitors provides a little more resilience to bugs (i.e., they are more likely in servers than the monitors). Also, we see no benefit since the object space information must be reported to the monitors anyway. Servers must use unique and immutable object names, restrict objects from be moved to another location in the hierarchy, and prevent operations from side-effecting client objects. The reasons for these restrictions will be demonstrated in the Implementation Section.

The architecture also includes a set of system resource managers that control access to fundamental system resources, such as an interrupt manager and a main memory manager. For example, in Lava any thread can bind to a free interrupt. The interrupt manager assigns threads to interrupts and replaces them with approved threads (e.g., device driver threads) as these are loaded. This prevents an obvious denial-of-service attack in which a component binds to an interrupt before the expected device driver is actually loaded.

The system TCB consists of a secure booter, the nucleus, SAI, reference monitors, and the resource managers described above.

A component invokes another component by calling a method defined in its interface. Component method invocations are either procedure calls (intra-task) or IPCs (inter-task). Procedure calls are not authorized, so the permissions of the task must effectively represent the trust in all components in the task. The reference monitors intercept and authorize all IPCs. The implementation of the component model enforces a specific format on all IPC method invocations. Therefore, any invocation that does not adhere to this format is rejected by the monitor. The monitors know the format of each component's methods by obtaining a *signature table* which describes the number of arguments and their data types. Components are trusted to define the interfaces that they export. Otherwise, other components would not be able to access them properly.

3.2 Access Control Model

To derive our access control model we start with a traditional model in which *principals* (e.g., users, services,

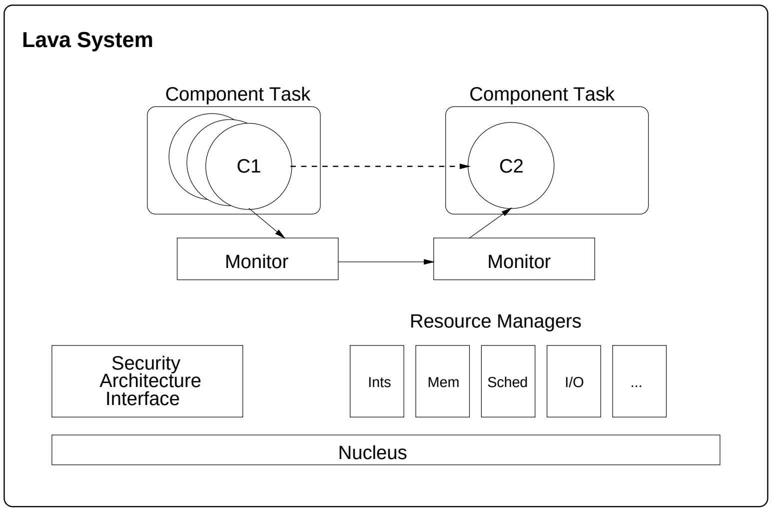

Figure 1: Lava system architecture

and components) perform *operations* (e.g., method invocations) on *objects* (e.g., files, communication channels, and memory). A principal's *access rights* or *permissions* specify the operations that it may perform on objects.

The security architecture's access control model consists of the following concepts:

- **Component**: A set of interfaces and implementations of those interfaces
- **Component Task**: A task (i.e., process) that is executing one or more components
- **Identity**: A reference to a unique executor of a component
- **Object**: Uniquely-named and strongly-typed entity that is a component or is served by a component
- **Operation**: Method invocation on an object
- **Capability**: The mapping of an object (including its type) to the specific operations that can be performed upon it via a component by the holder of the capability
- **Transform**: Specifies when and how a principal's capability set can change
- **Assignment Limit**: An association of a principal or set of principals with a capability set that describes the permissions that that principal can delegate (e.g., from transforms)
- **Principal**: The association of a set of identities with their capabilities and capability assignment limits

Components are static objects that define interfaces and implementations. They are executed in component tasks that may contain one or more components. Components are associated with an authenticated identity. A principal consists of the identities of the components in the task. Capabilities define the rights of the principals (associated with their specific identities) to perform operations on objects served by other components. Thus, capabilities store an association of: server component task, interface (type), object identifier, and operations. We must assume that server components enforce strong-typing upon their objects.

Changes in a principal's capabilities are implemented by transforms. Transforms may either grant (i.e., delegate) or revoke a capability. Control of delegation is enforced by assignment limits. Each principal may have a set of assignment limits that specify potential delegators and the rights that they may delegate to this principal. The architecture uses the assignment limits to authorize delegations (see Section 4.3).

Different flexibility in security policy can be achieved using different mechanisms for managing capability sets. We define three levels of flexibility: (1) fixed, (2) variable or (3) composable. Fixed capability sets enforce a standard mandatory access control policy. Variable capability sets enable multiple principals to specify mandatory access control limits. For example, the system administrator may specify one MAC domain, and permit the virtual memory system to specify another MAC domain within some limits. This enables the virtual memory system to restrict the permissions of the component more tightly than system administrator could without requiring that it provide its own security architecture to control such delegations. Composable capability sets are variable sets that enable transient composition of principal capabilities on method invocation. For example, temporary intersection or union of permissions on a method invocation, as is done in some Java security models [8] (intersection) or by protection domain extensions in Mungi [21] (union), can be supported by a composable capability set. We describe how the architecture supports permission composition in Section 4.4.

4 Implementation

We now describe the implementation of our security model on the Lava system. In this paper, we focus on demonstrating how the security requirements are satisfied by the implementation. These requirements are: (1) resolution of security policy to component tasks; (2) authorization of object operations; (3) control of delegation; and (4) enforcement of multiple security mechanisms.

4.1 Security Policy Resolution

When a component is loaded, the SAI derives a set of capabilities for it. The first problem is to find the policy relevant to a specific component. Since the component may be loaded into an arbitrary context, we specify formalize the notion of a context. A *context* is a combination of the component requestor, component author, application in which the component is loaded, and the role which the component plays in the application. The requestor determines the values of the application and role for the component. For example, if a component is being loaded as a device driver, the requestor specifies that fact (i.e., the application is device driver). If the component provides a specific memory object management policy on behalf of the virtual memory system, its rights are specific to that role in the virtual memory system. In a previous paper, we describe a model for managing such security policies [9].

Next, the policies themselves may specify that the assignment limits (i.e., the permissions that a principal can delegate to this component) themselves are context-dependent. For example, the permissions that the virtual memory system may delegate to the component may be dependent on the requestor application and virtual memory system's states. Parameterized access control models [7, 13] enable permissions to specified based on parameterized objects. For example, the memory objects to which a component can access may depend on those belonging to certain principals. However, changes in context can result in changes in assignment limits and, hence, changes in permissions. Therefore, the selection of context-sensitive rights should be made with care. Preferably, the contexts should specify the maximal permissions that can ever be delegated to the component.

The permissions are represented in two sets: (1) the set of resolved capabilities and (2) the set of unresolved policy entries. Since all components may not be present at load time, the policy must still be accessible to the reference monitors after the completion of the load. When a capability to a specific service is received (via delegation), it must be resolved against the policy before the component may use it. Components receive references to capabilities (indices in their capability table) rather than the actual capabilities, so the reference monitors can immediately revoke them. A capability consists of the following fields: (1) task and thread; (2) interface type; (3) component instance; (4) object; and (5) operations.

4.2 Authorization

In Lava, tasks define protection domains, so monitors are designed to authorize inter-task operations and responses. An inter-task operation specifies: (1) the destination task and thread; (2) the component instance; (3) the interface type; (4) the operation; and (5) the operation arguments. A response is differentiated from an operation because the component instance and interface are null.

When a component performs an object access on a server (actually another component, but we call it a server to differentiate between the two components), it specifies the identity of the object to the server. The format of this identity depends on the interface provided by the server. For example, files are specified by string path names. However, the object field in a capability uses immutable OIDs that are unique within the context of the object space. The use of object identifiers, rather than object names, in capabilities enables prevention of TOCTTOU attacks [3].

Therefore, when an operation in which an object name rather than an identifier is used, called a *bind* operation, the reference monitor must obtain the identifier for the object before performing the authorization. For example, consider opening a file in a file server. The monitor copies the file name argument to prevent modification by the caller and requests that the file server dereference the object name. The file server is trusted to dereference the object name to the proper OID. The monitor then authorizes the requested file operations using this OID. This object identifier must be used in the actual bind operation which, of course, requires a change to traditional file servers.

Object spaces for each server are a forest with roots corresponding to the identities of each object owner. Owning an object only implies that the object is in the owner's hierarchy, but does not imply control over access. The system security policy still restricts access and delegation. However, it is still beneficial to specify rights in terms of a hierarchy of objects rather than each object individually. Therefore, a component principal may possess a capability that refers to a set of objects (e.g., a file directory and its subdirectories). When an object name is presented the entire OID chain for that object is returns to the monitor on dereferencing, so such "higher-level" capabilities can be used to authorize the access. Hashing must be used to retrieve the capabilities quickly.

Authorization of operations involves verifying that the component has a capability that permits the operation to be invoked on the object specified. Components do not pass the OID of the object to the server, but rather an index to the capability in the monitor. The monitor returns these indices when the capability is delegated to the task. Since components cannot be trusted to present a legitimate capability and we desire immediate revocation, monitors must authorize each controlled operation. Using the reference, monitors can efficiently retrieve the capability and authorize the operation.

4.3 Controlled Delegation

An object reference may be passed as an argument in an operation, so it may be necessary to authorize a delegation within an operation. The fact that an object reference is being passed is determined by the argument's type. If this reference refers to a capability for a task other than the destination of the operation, then the monitor assumes that the capability is being delegated (regardless of whether the destination already has the right or not because multiple delegations of the same capability are separate). The destination task's monitor authorizes delegations using the assignment limits of the caller to the destination. Note that the lifetime of such a delegation is the duration of the operation (i.e., until the response is sent) unless an explicit transform is used. Performance may be affected because we need to retrieve: (1) the capability for this object in the assignment limits and (2) when no such capability is present, the capability for the ancestor objects in the assignment limits. To solve the first problem, capabilities are indexed by their OID, task/thread, component instance, and interface, so they can be found quickly (e.g., in a hash table). The second problem is addressed efficiently by having the object field refer to a chain of objects from the requested object through all its ancestors. This chain is set at bind time when the object names are dereferenced by the server.

In general, the lifetime of the delegations from transforms may vary in length, so the implementation must be able to revoke capabilities when their lifetimes expire. For example, a delegation may be permanent unless explicitly revoked (e.g., file access), for the life of the delegator (e.g., access to the delegator), for some specific duration (i.e., time-limited delegation), or for the duration of a method invocation. Delegations only for the duration of an invocation are handled upon the response by the delegatee's monitor. For other delegations, monitors cache mappings between events and delegations (delegatee tasks and capabilities). When the event occurs, the monitor requests that the delegatee's monitor decrement the delegation count of the capability. If the delegation count reaches 0, the capability is revoked and any further delegations of this capability are revoked. Typically, only the delegator can revoke a permission (either by explicitly executing a transform or triggering its revocation event). Special tasks may be allowed to revoke any privilege within their assignment limits. Revocation is immediate.

4.4 Multiple Enforcement Mechanisms

Since reference monitors are separate tasks and can be assigned per component task, different authorization and permission management policies can be provided for each task. This permits us to keep the simple monitors fast while still enabling some components to come under complex monitoring policies. Below, we describe a complex policy in which a component is invoked with either a union or intersection of the rights of itself and its caller (called a composable capability set).

For principals whose permissions are intersected or unioned upon an invocation, entry capabilities which note such a fact (via a bit set in the operations field as a few bits are reserved for such purposes) are used to invoke them. Upon receipt of an operation invoked using such a capability, the caller's reference monitor creates a task for the component dynamically (Lava tasks are cheap to create and the nucleus can support a large number of tasks). The new task (the callee) is assigned to a precreated monitor that implements the appropriate authorization mechanism (union or intersection of rights depending on another bit). Thus, it is important that the monitors are external to the nucleus, so different authorization mechanisms can be supported by different monitors. Also, in order to authorize operations using the capability sets of different principals, it is necessary for the monitors to share access to the each task's permissions, so a shared permission table is accessible (read-only) to all monitors. A monitor may only modify its own task's permissions (in which case locking is required). References to the principals involved are provided to the callee's monitor at load time.

Transforms are unchanged by the composition, but assignment limits may either be set to null or combined using the same mechanism as the capabilities. Also, like the assignment limit capability set described above, capabilities are hashed by OID, task/thread, component instance, and interface for efficient retrieval from multiple capability sets and object ancestor lists are stored with capabilities to enable retrieval of other capabilities that may authorize operations.

5 Conclusions

We have presented a security architecture for component-based systems. It is designed to support the dynamic composition of systems and applications from individual components. Components are analogous to dynamically-linked libraries in that they may be loaded into any context chosen by the requesting component. This leads us to two key conclusions: (1) a component's permissions may be restricted relative to the context in which it is loaded and (2) a component cannot be expected to understand such system contexts, so the system must be able to authorize object accesses. Components must be allowed to create their own object spaces, so this information must be transferred to the system security modules. Additionally, traditional problems with operating system security, such as control of delegation and object-level authorization must be supported by the system. Lastly, as a consequence of componentized model itself, the security architecture can support multiple authorization and permission management mechanisms easily. Since many different policies for permission management have been employed over the years, from various protected procedure calls of Dennis and Van Horn [4] and CAP [23] to protection domain extensions of Mungi [21] to Java policies such as stack introspection [22], it appears beneficial that different combinations of rights may apply for different components.

The security architecture consists of a security architecture interface (SAI) which loads components on behalf of requestors and derives permissions for the tasks in which they are loaded. Security policy describes services and the permissions associated with objects (including groups), so the security architecture maps components to services and the permissions that they may possess. Permissions are enforced and managed by reference monitors. Each task may be assigned a reference monitor which authorizes object operations and delegations of capabilities. The reference monitor must collaborate with servers to map object requests to locations in object spaces and their commensurate rights. Object hierarchies rooted by the owner of the object are currently used because they can directly indicate how objects are shared. However, experience with these hierarchies are limited, so some problems will likely occur or some policies may not be effectively handled.

References

[1] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, James P. Anderson and Co., Fort Washington, PA, USA, 1972.

[2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *IEEE Symposium on Security and Privacy*, pages 66–77, 1995.

[3] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[4] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[5] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[6] F. S. Gallo. Penguin: Java done right. *The Perl Journal*, 1(2):10–12, 1996.

[7] L. Giuri and P. Iglio. Role templates for content-based access control. In *Proceedings of the Second ACM Role-Based Access Control Workshop*, November 1997.

[8] L. Gong. Java security: present and near future. *IEEE Micro*, 17(3):14–19, 1997.

[9] T. Jaeger, F. Giraud, N. Islam, and J. Liedtke. A role-based access control model for protection domain derivation and management. In *Proceedings of the Second ACM Role-Based Access Control Workshop*, November 1997.

[10] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium*, pages 143–156, January 1998.

[11] P. A. Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, University of Cambridge, 1988.

[12] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 175–187, 1993.

[13] E. C. Lupu and M. Sloman. Reconciling role-based management and role-based access control. In *Proceedings of the Second ACM Role-Based Access Control Workshop*, November 1997.

[14] D. Mazieres and M. F. Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 56–61, May 1997.

[15] S. E. Minear. Providing policy control over object operations in a Mach-based system. In *Proceedings of the 5th USENIX UNIX Security Symposium*, 1995.

[16] J. G. Mitchell and *et al.* An overview of the Spring system. In *Proceedings of Compcon*, February 1994.

[17] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. The Safe-Tcl security model. In *Proceedings of the 23rd USENIX Annual Technical Conference*, 1998.

[18] R. Rashid, A. Tevanian Jr., M. Young, D. Golub, D. Baron, D. Black, W. J. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[19] M. Rozier and *et al.* Overview of the Chorus distributed operating system. In *USENIX Symposium on Micro-kernels and Other Kernel Architectures*, pages 39–69, 1992.

[20] G. van Rossum. Grail – The browser for the rest of us (draft), 1996. Available at http://monty.cnri.reston.va.us/grail/.

[21] J. Vochteloo, K. Elphinstone, S. Russell, and G. Heiser. Protection domain extensions in Mungi. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, pages 161–165, October 1996.

[22] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[23] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System*. North Holland, 1979.

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