Separating Access Control Policy, Enforcement, and Functionality in Extensible Systems

ROBERT GRIMM and BRIAN N. BERSHAD
University of Washington

Extensible systems, such as Java or the SPIN extensible operating system, allow for units of code, or extensions, to be added to a running system in almost arbitrary fashion. Extensions closely interact through low-latency but type-safe interfaces to form a tightly integrated system. As extensions can come from arbitrary sources, not all of whom can be trusted to conform to an organization's security policy, such structuring raises the question of how security constraints are enforced in an extensible system. In this paper, we present an access control mechanism for extensible systems to address this problem. Our access control mechanism decomposes access control into a policy-neutral enforcement manager and a security policy manager, and it is transparent to extensions in the absence of security violations. It structures the system into protection domains, enforces protection domains through access control checks, and performs auditing of system operations. The access control mechanism works by inspecting extensions for their types and operations to determine which abstractions require protection and by redirecting procedure or method invocations to inject access control operations into the system. We describe the design of this access control mechanism, present an implementation within the SPIN extensible operating system, and provide a qualitative as well as quantitative evaluation of the mechanism.

Categories and Subject Descriptors: D.4 [**Software**]: Operating Systems; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms: Security

Additional Key Words and Phrases: Extensible systems, SPIN, Java, access check, protection domain, protection domain transfer, auditing, security policy, policy-neutral enforcement

This research was sponsored by the Defense Advanced Research Projects Agency, the National Science Foundation, and by an equipment grant from Digital Equipment Corporation (now Compaq Computer Corporation). Grimm was partially supported by fellowships from the Microsoft Corporation and IBM Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship and an Office of Naval Research Young Investigator Award.
An earlier version of this paper appeared under the title *Providing policy-neutral and transparent access control in extensible systems* in J. Vitek and C. D. Jensen, editors, *Secure Internet Programming—Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pp. 317–338, Springer-Verlag, 1999.
Authors' address: Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195; email: {rgrimm; bershad}@cs.washington.edu.

1. INTRODUCTION

Extensible systems, such as SPIN [Bershad et al. 1995] or Java [Gosling et al. 1996; Lindholm and Yellin 1996], promise more power and flexibility than traditional systems and enable new applications such as smart clients [Yoshikawa et al. 1997] or active networks [Smith et al. 1999; Wetherall 1999]. They are best characterized by their support for dynamically composing units of code, called *extensions* in this paper. In these systems, extensions can be added to a running system in almost arbitrary fashion, and they interact through low-latency but type-safe interfaces with each other. We use the term "interface" in this paper to simply denote the types and operations exported by an extension; interfaces may declare types but are not, as in Java, type declarations themselves. Extensions and the core system services are typically collocated within the same address space and form a tightly integrated system. Consequently, extensible systems differ fundamentally from conventional systems, such as Unix [McKusick et al. 1996], which rely on processes executing under the control of a privileged kernel.

As a result of this structuring, system security becomes an important challenge, and access control becomes a fundamental requirement for the success of extensible systems. Since system security is customarily expressed through protection domains [Lampson 1971; Saltzer and Schroeder 1975], an access control mechanism should

—structure the system into protection domains (which are an orthogonal concept to conventional address spaces [Chase et al. 1994]),

—enforce these domains through access control checks, and

—support auditing of system operations.

Furthermore, an access control mechanism must address the fact that extensions often originate from other networked computers and are untrusted, yet execute as an integral part of an extensible system and interact closely with other extensions.

In this paper, we present an access control mechanism for extensible systems that meets the above requirements. We build on the idea of separating policy and enforcement first explored by the distributed trusted operating system (DTOS) effort [Minear 1995; Olawsky et al. 1996; Secure Computing Corporation 1997a; 1997b] and introduce a mechanism that not only separates policy from enforcement, but also separates access control from the actual functionality of the system. The access control mechanism is based on a simple yet powerful model for the interaction between a policy-neutral enforcement manager and a given security policy, and it is transparent to extensions and the core system services in the absence of security violations.

Our access control mechanism works by inspecting extensions for their types and operations to determine which abstractions require protection and by redirecting procedure or method invocations to inject access control operations into the system. The access control mechanism provides three types of access control operations, which are expressed in terms of security identifiers, representing privilege, and permissions, representing the right to perform an operation. The operations are (1) explicit protection domain transfers to allow for a controlled change of privilege, (2) access checks to limit which procedures or methods can be invoked and which objects can be passed, and (3) auditing to provide a trace of system operations. The access control mechanism works at the granularity of individual procedures or methods and provides precise control over extensions and the core system services alike.

Our access control mechanism is based on the following three assumptions. First, because it is a software-only mechanism, it assumes that the code in an extensible system is safe, that is, that all code respects the declared interfaces and preserves referential integrity. Second, because our access control mechanism imposes access control operations on procedures and methods, it assumes that resources that need to be protected rely on encapsulation to hide their internal state. Finally, because our access control mechanism injects access control operations into an extensible system, it assumes the existence of some mechanism for binary interposition, such as the ability to dynamically patch object jump tables.

The main contributions of this paper are twofold. First, based on the observation that extensible systems differ fundamentally from conventional systems, we identify the specific goals for providing effective access control in extensible systems. Second, we present an access control mechanism that meets these goals by separating access control policy, enforcement, and functionality and which relies on a simple yet powerful model for their interaction.

Access control and its enforcement is but one aspect of the overall security of an extensible system. Other important issues, such as the specification of security policies or the expression and transfer of credentials for extensions, are only touched upon or not discussed at all in this paper. Furthermore, we assume the existence of some means, such as digital signatures, for authenticating both extensions and users. These issues are orthogonal to access control, and we believe that our access control mechanism can serve as a solid foundation for future work on other aspects of security in extensible systems.

The remainder of this paper is structured as follows: Section 2 motivates our research and elaborates on the goals for access control in extensible systems. Section 3 describes the design of our access control mechanism, and Section 4 presents an implementation within the SPIN extensible operating system. Section 5 reflects on our experiences with designing and implementing the access control mechanism, and Section 6 presents a detailed performance analysis of the implementation. Section 7 reviews related work. Finally, Section 8 concludes.

2. MOTIVATION AND GOALS

Extensible systems differ fundamentally from conventional systems, such as Unix, in their overall structure. In conventional systems, processes execute program code under the control of a privileged kernel and are isolated from each other through the use of address spaces. The kernel protects its resources through access checks, which are performed before relevant operations, such as opening a file, and which are coded into the kernel by the kernel's developers. Users can determine the access control policy, for example, by setting appropriate file permissions. But, they cannot control which operations are protected, as the decision on which resources and which operations need to be protected is fixed in the kernel design and implementation. This approach to access control is practical for conventional systems, because they feature a limited number of services and a relatively narrow interface. Furthermore, it can be effective, as long as interactions between processes are limited to the use of resources protected by the kernel.

In contrast, extensible systems are best characterized by their ability to dynamically compose units of code. Extensions and the core system services are typically collocated within the same address space and form a tightly integrated system, easily leading to more complex interactions between the different components than those in conventional systems. Because extensions can interact in ways not foreseeable by the system designers, it is not sufficient to protect only the resources of the core system services. Rather, an access control mechanism for extensible systems needs to protect extensions and the core system services alike by imposing additional structure onto the system. At the same time, it should only impose as much structure as is *strictly* necessary to preserve the advantages of extensible systems.

Based on this realization, we identify four specific goals that inform the design of our access control mechanism:

(1) *Separate access control and functionality*. The access control mechanism should separate access control policy and enforcement from the actual code of the system and extensions. This separation of access control and functionality supports changes to security policies without requiring access to extension or system source code. This is especially important for large computer networks, such as the Internet, where the same extension executes on different systems with different security requirements and where source code is typically not available. Separating access control and functionality does *not* prevent a programmer who writes an extension from defining (part of) the security policy for that extension. However, it calls for a separate specification of such policy, similar to an interface specification, which offers a distinct and concise description of the abstractions found in a unit of code. This policy specification may then be loaded into an extensible system as the extension is loaded.

(2) *Separate policy and enforcement.* The mechanism should separate the security policy from its actual enforcement. This separation of policy and enforcement allows for changes to security policies without requiring intrinsic changes to the core services of the extensible system itself. Rather, the security policy is provided by a trusted extension, and, as a result, the access control mechanism leverages the advantages of an extensible system and becomes extensible itself.

(3) *Use a simple yet expressive model.* The mechanism should rely on a simple model of protection that covers a wide range of possible security policies, including policies that change over time or depend on the history of the system. Such a model ensures that the access control mechanism can strictly enforce a wide range of security policies and that the security policy has control over all relevant aspects of access control. At the same time, it favors simplicity over complex interactions between policy and enforcement.

(4) *Enforce transparently.* The mechanism should be transparent to extensions and the core system services in that they should not need to interact with it as long as no violations of the security policy occur. Transparency ensures that the mechanism actually provides a clean separation of security policy, enforcement, and functionality. Furthermore, it provides support for legacy code (to a degree) and enables aggressive, policy-specific optimizations that reduce the performance overhead of access control. At the same time, extensions need to be notified of security faults so that they can implement their own failure model. In other words, transparency reduces access control, as seen by extensions, to handling a program fault such as division by zero or dereferencing a `NIL` reference.

These four goals, taken together, call for a design that separates functionality, security policy, and enforcement in an extensible system and that provides a clear specification for their interaction. In other words, the goals call for an access control mechanism that combines the extension itself, the security constraints for the extension as specified by the extension's programmer, and a site's security policy to produce a *secure* extension. At the same time, the mechanism is not limited to changing only the extension as a result of this combination process, but can impose security constraints on other parts of the extensible system as well. This process of combining functionality and security to provide access control in an extensible system is illustrated in Figure 1.

A design that addresses the four goals effectively defines the *protocol* by which the security policy and the access control mechanism interact and by which, if necessary, extensions are notified of security-relevant events. As such, this protocol is *internal* to the extensible system and the overall security mechanism. In other words, the abstractions used for expressing protection domains and access checks need not be, and probably should not be, the same abstractions presented by the security policy to users and

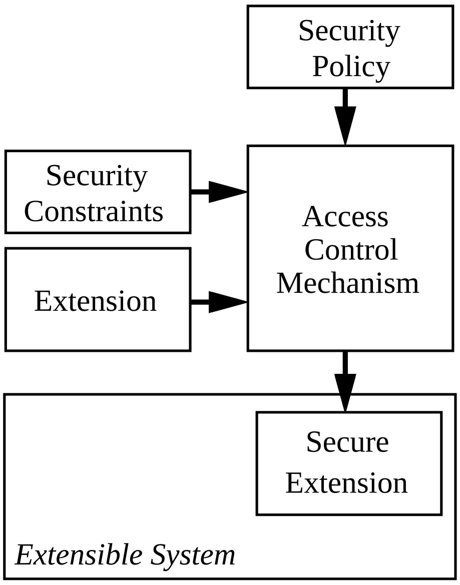

Fig. 1. Overview of access control in an extensible system. The access control mechanism combines the extension itself, the security constraints for the extension as specified by the programmer, and a site's security policy to place a secure version of the extension into the extensible system.

administrators. Rather, it is the responsibility of a security policy management tool to provide users and administrators with a high-level and user-friendly view of system security.

2.1 Examples

As long as extensions, such as Java applets in the sandbox model, use only a few, selected core services, providing protection in an extensible system reduces to isolating extensions from each other and performing access control checks in the core services. However, for many real-world applications of extensibility, such a protection scheme is clearly insufficient, as extensions use some parts of the system and, in turn, are used by other parts. For example, an extension may provide a new file system implementation, such as a log-structured file system, offer additional functionality for existing file systems, such as compression or encryption, or support higher-level abstractions, such as transactions, on top of the system's storage services. An extension may also implement new networking protocols, such as multicast, or higher-level communication services, such as a remote procedure call package or an object request broker (ORB), on top of the existing networking stack.

From a security viewpoint, the programmer who writes such an extension will want to protect the resources used by that extension. So, for a transaction manager, she would like to ensure that the files or disk extents used for storing transaction data can only be accessed through the transaction manager. And, for an ORB, she would like to ensure that the network port used for communicating with other nodes cannot be accessed by other extensions. A simple way to implement these security constraints is to place the transaction manager or ORB into its own protection domain and to use access checks in the storage services or networking stack to protect the resources used by the transaction manager or ORB. The security constraints in these examples thus not only affect the service provided by the extension itself, but also cover other services of an extensible system. At the same time, overall security in an extensible system requires coalescing the constraints for several extensions. Consequently, separating the specification of security constraints and functionality would clearly aid in providing security for an extensible system.

In addition to the programmer, the administrator of an extensible system may want to impose additional restrictions on an extension. For example, she may want to restrict how other extensions can call the transaction manager in order to ensure that only a transaction's initiator can commit it. Or, she may require auditing of the transaction manager's operations to ensure that a log record is generated if the commit operation is not performed by a transaction's initiator. Alternatively, the administrator may want to impose a security policy that conflicts with the security constraints expressed by the programmer. For example, she may want to integrate the ORB into the same protection domain as the networking stack, as the ORB is the only means for remote communication in an installation (such as a corporate intranet) and since providing access control on the ORB is adequate for security. As illustrated by these examples, the security policy for an extensible system varies according to the requirements of a specific installation, even if the functionality does not change. It is thus not sufficient to only separate access control from functionality, but also necessary to separate the security policy from its enforcement to allow security policies to change without requiring intrinsic modifications to the core system.

So far, we have illustrated the need for a clean separation of security policy, enforcement, and functionality, which suggests that the access control mechanism be transparent to extensions. However, extensions need to be notified of failures so that they can implement their own failure model. For example, the transaction manager might decide to abort the offending transaction, or the ORB may need to clean up the internal state of the corresponding connection. It is thus important that access control is only transparent in the absence of failures and that extensions are notified of security violations.

3. DESIGN

The design of our access control mechanism divides access control in an extensible system into an enforcement manager and a security policy manager. The enforcement manager is part of the core services of the extensible system. It provides information on the types and operations of

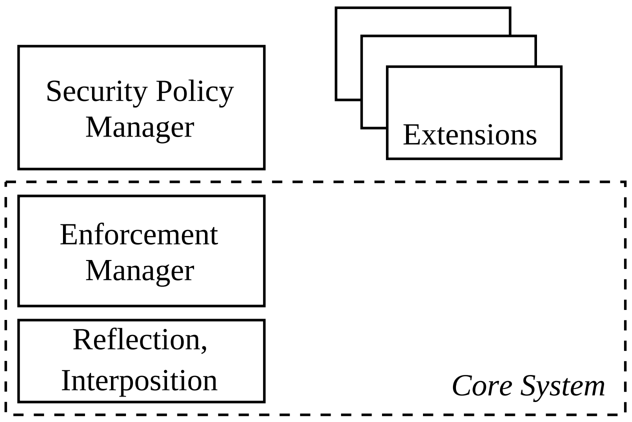

Fig. 2. Structure of the access control mechanism. The enforcement manager is part of the core system services, provides information on the types and operations of an extension (reflection), and redirects procedure or method invocations (interposition) to ensure that a given security policy is actually enforced onto the system. The security policy manager is a trusted extension, determines which abstractions require which access control operations, and performs the actual mediation.

an extension and redirects procedure or method invocations to perform access control operations. The security policy manager is provided by a trusted extension, so that it can be easily replaced, and determines the security policy for the system. It decides which procedures and methods require which access control operations when an extension is linked into an extensible system. It also performs the actual *mediation*, i.e., it makes the dynamic access decisions during execution of an extension's code. This structure is illustrated in Figure 2.

The protocol that determines the interaction between the enforcement and the security policy manager relies on two abstractions, namely security identifiers and sets of permissions, or access modes. Security identifiers are associated with both subjects and objects and represent privilege. Permissions are associated with operations and represent the right to perform an operation. Security identifiers and permissions were chosen as the basic abstractions in our design to rephrase the fundamental question of access control "May subject S perform operation OP on object O?" more abstractly as "Does the subject with security identifier SID_1 have permission P for the object with security identifier SID_2?"

As a result, security identifiers and permissions form an access matrix [Lampson 1971]. A subject's security identifier denotes the protection domain for that subject and determines the row of the access matrix. An object's security identifier determines the column of the access matrix. The corresponding permissions are the entry at the intersection of that row and column. This choice of abstractions enables a straightforward mapping of access control lists as well as capability lists onto the protocol, since access control lists express security constraints by the columns of an access matrix and capability lists by the rows.

The enforcement manager maintains the association of subjects and objects with security identifiers and performs access control checks based on access modes. But, it does not interpret security identifiers and access modes, as their meaning is determined by the security policy manager, which performs the actual mediation. The enforcement manager thus treats security identifiers and individual permissions as abstract tokens, onto which the security policy manager maps the security policy.

As extensible systems feature a considerably different structuring from traditional systems, such as Unix, it is necessary to define the exact meaning of subjects and objects. We treat threads in an extensible system as subjects, as they are the only active entities, and all other entities, including extensions, as objects. This is not to say that subjects only represent the principal that created a thread. Rather, the current rights of a subject depend on the current protection domain, i.e., the extension whose code the thread is currently executing, and, possibly, on previous protection domains, i.e., the history of extensions whose code the thread has executed before entering the current extension. Furthermore, while we treat extensions as objects, they are subject to a somewhat different form of access control than other objects in an extensible system.

3.1 Access Control on Extensions

Conceptually, access control determines whether a subject can legally execute some operation on some object. Access control on extensions differs from this concept in that it is sometimes necessary to control how extensions interact with each other. Specifically, this is the case at link-time: the extension to be loaded into the system needs to be linked against other extensions, whose interfaces it will execute and extend (where interface simply denotes the types and operations exported by an extension). It is thus necessary, at link-time, to provide access control over which interfaces a given extension can link against for execution and extending [Pardyak and Bershad 1996; Sirer et al. 1996a]. Enforcing this link-time control over extensions is important, since it presents a first line of defense against unauthorized access (after all, if an extension cannot link against an interface, it cannot directly use it), and since it may result in opportunities for optimizing away dynamic access control operations.

Link-time control over extensions *can*, however, be expressed through regular access control checks by imposing checks on linkage operations and by executing these linkage operations within a protection domain appropriate for the extension to be linked. To impose access control checks on linkage operations, the enforcement manager injects the appropriate checks into the linking service during system start-up. These checks verify that the caller has the EXECUTE permission when linking against another extension and the EXTEND permission when extending another extension. To execute these operations within a protection domain that is appropriate for an extension, the loader performs the linkage operations as well as other necessary initialization with a thread that is associated with the security identifier representing the extension's protection domain. For each extension to be loaded into the system, this security identifier is determined by the security policy manager based on the extension's signature and is associated with the thread as well as the extension's code by the enforcement manager.

When loading an extension, the enforcement manager also determines the types and operations exported by that extension and passes this information to the security policy manager. The security policy manager then decides which types and operations require access control operations and instructs the enforcement manager to maintain security identifiers for the extension's objects as well as to inject access control operations into the extension and, if necessary, other parts of the system. Once the actual linking of an extension is complete and the appropriate access control operations have been injected into the system, the extension is fully and securely integrated into the system and its code can now be executed.

3.2 Access Control Operations

The enforcement manager supports three types of access control operations. The operations are (1) protection domain transfers to allow for a controlled change of privilege, (2) access control checks to limit which procedures or methods can be invoked and which objects can be passed, and (3) auditing to provide a trace of system operations. Protection domain transfers change the protection domain associated with a thread, based on the current protection domain and on the code that is about to be executed. Access checks determine whether the current subject is allowed to execute the code of an extension at all. Access checks also control the passing of arguments and results; for each argument that is passed into a procedure or method and for each result that is returned from the procedure or method, they determine whether the subject has sufficient rights for that object. Finally, auditing generates a log-entry for each procedure or method invocation, thus producing an execution trace of the system.

Access checks on the object of a method invocation, i.e., the implicit pointer to `this` that is passed into the method, are customarily used to control which operations can be performed on a specific instance of an object. Our access control mechanism supports access checks on *all* objects passed to and from procedure and method invocations, because object references in an extensible system are a form of type-safe capability [Bershad et al. 1995]. Such checks can be used to protect both caller and callee and provide precise control over how object references can be propagated between protection domains, making it possible to prevent the leaking of capabilities [von Eicken et al. 1999]. They also enable policy-specific optimizations: if a subject has only access to references that it is authorized to use, operations on those references need not be protected by explicit access checks. For example, if access checks are used to control how file references are passed between protection domains and, as a result, all subjects have only access to references for files they are allowed to read, the file read operation does not require any access checks.

When instructing the enforcement manager to perform access control operations on a given procedure or method, the security policy manager specifies the types of access control operations, i.e., any combination of protection domain transfer, access checks, and auditing. For access checks, it also specifies the required access modes, one for the code of the extension, one for each argument, and one for each result. Any of these access modes can be void, which instructs the enforcement manager to not perform an access check for that particular object.

The access control operations are ordered as follows. Before a given procedure or method is executed, the enforcement manager first performs access checks, then a protection domain transfer, and, finally, auditing, which also records failed access checks. On return from the procedure or method, the enforcement manager first performs the reverse protection domain transfer, then access checks on the results, and, finally, auditing, which, again, also records failed access checks.

3.3 The Protocol between Security Policy and Enforcement Manager

To execute the access control operations, the enforcement manager needs to know the new security identifier for a thread when performing a protection domain transfer and the permissions a subject has for a given object when performing an access check. It also needs to know which security identifier to associate with a newly created object. The protocol between the security policy manager and the enforcement manager uses three mappings between security identifiers, access modes, and types to communicate this information. Using SID for security identifiers, ACCESSMODE for access modes, and TYPE for types as defined by the extensible system, the three mappings are:

$$\begin{array}{llcl} \text{(Domain Transfer)} & \textsc{Sid}_{\text{Thread}} \times \textsc{Sid}_{\text{Code}} & \rightarrow & \textsc{Sid}_{\text{Thread}} \\ \text{(Access Check)} & \textsc{Sid}_{\text{Thread}} \times \textsc{Sid}_{\text{Object}} & \rightarrow & \textsc{AccessMode}_{\text{Max}} \\ \text{(Object Creation)} & \textsc{Sid}_{\text{Thread}} \times \textsc{Type}_{\text{Object}} & \rightarrow & \textsc{Sid}_{\text{Object}} \end{array}$$

The domain transfer mapping is used for protection domain transfers. It maps the current security identifier of a thread and the security identifier of the code that is about to be called into the new security identifier of the thread. The enforcement manager associates the thread with the new security identifier before control passes into the procedure or method, and it restores the original security identifier upon completion of the procedure or method.

The access check mapping is used for access checks. It maps the security identifier of a thread and the security identifier of an object into an access mode representing the maximum rights the subject has on the object. The enforcement manager verifies that the maximal access mode contains all permissions of the required access mode, as specified by the security policy manager when originally requesting the access check.

The object creation mapping is used for the creation of objects. It maps the security identifier of a thread that is about to create an object and the type of that object into the security identifier for that object. The enforcement manager associates the newly created object with the resulting security identifier. A simplification of this mapping may omit the object type from the mapping and simply map all objects created by a thread into the same security identifier, thus providing a default security identifier for objects that are newly created within a protection domain.

Both variations of the object creation mapping provide relatively coarse-grained control over the security identifiers associated with objects and are clearly insufficient for some services, notably services that manage persistent storage. For example, a file system typically executes threads within its own protection domain but needs to associate different files with different security identifiers, because not all files share the same access constraints. To support trusted extensions that provide finer-grained control, the enforcement manager includes an interface through which a trusted extension can override the security identifier supplied by the object creation mapping. In the example of a file system, this interface could be used to map files to security identifiers based on files' names, similar to the name-based security attributes in domain and type enforcement [Badger et al. 1995a; 1995b].

New subjects, that is, freshly spawned threads, are associated with the same security identifier as the spawning thread so that they possess the same privileges. An exception to this rule occurs for threads that are created when a user logs into the system. In this case, an appropriate form of authentication (such as a password) establishes the identity of the user to the security policy manager, and the enforcement manager associates the thread with the corresponding security identifier.

3.4 Mapping Security Policies

To effectively manage a given security policy, we expect that the security policy manager not only imposes access control operations and performs mediation, but that it also provides users and administrators with a comprehensive and user-friendly view of system security. In particular, we expect it to support some high-level representation of its policy, which may range in expressive power from the relatively simple policy descriptions used for domain and type enforcement or Java platform security [Gong 1999] to the more powerful assertion languages used for automatic trust management [Blaze et al. 1999].

As a result, the expressive power of our access control mechanism depends on how well security policies map onto security identifiers, access modes, and access control operations. We believe that common security policies map well. For example, for multilevel policies [Bell and LaPadula 1976; Biba 1977; Denning 1976], the security labels for subjects and objects, representing, for example, the classification level and category, can be directly encoded as security identifiers, and a policy's permissions map directly onto the access modes of our access control mechanism. As for any policy, access checks can be used to ensure that subjects perform only authorized operations on objects. However, the security policy manager cannot use protection domain transfers outside the trusted computing base, because a subject's security label must not change for multilevel policies.

For domain and type enforcement, the domains and types of the policy can be mapped onto security identifiers by partitioning the space of security identifiers into two sets: the first set represents the domains associated with subjects, and the second represents the types associated with objects. Permissions, again, map directly onto the access modes of our access control mechanism. Furthermore, access checks and protection domain transfers specified by the policy map directly onto the corresponding access control operations in our access control mechanism.

For role-based access control [Ferraiolo and Kuhn 1992; Ferraiolo et al. 1995; Sandhu et al. 1996], policies can also be mapped onto security identifiers by partitioning the space of security identifiers: the first set represents the individual roles of a policy, and the second represents objects that share the same access restrictions. Protection domain transfers can be used when the role of a subject changes, and access checks can be used to prevent unauthorized access to objects.

Finally, Java's extended stack inspection [Wallach et al. 1997] can be mapped onto our access control mechanism by relying on the formalization of extended stack inspection described in Wallach and Felten [1998]. In that model, the current security state is represented by an additional argument to each method invocation; access checks only access the additional argument but need not walk the stack. By mapping this additional argument onto a thread's security identifier and changing it through a protection domain transfer when the security state changes, our access control mechanism can be used to implement Java-style security policies.

3.5 The Mediation Cache

Since the security policy manager needs to map the policy onto security identifiers, access modes, and types, lookup operations for the three mappings may incur a noticeable performance overhead. To minimize this overhead, the enforcement manager caches individual entries in the three mappings, which reduces the frequency with which the security policy manager needs to resolve entries and therefore the overall performance overhead of access control operations. The security policy manager has full control over this *mediation cache*. It sets the overall size of the cache, can remove any entry from the cache at any time, and flush the entire cache. Furthermore, for any lookup operation on any of the mappings, it specifies whether that particular entry can be cached and, if so, for how long.

For example, for multilevel policies, mappings can be cached indefinitely, since the mappings for multilevel policies never change. For domain and type enforcement as well as other policies that are directly based on an access matrix, mappings generally can be cached. If permissions are changed, however, the security policy manager must remove the corresponding entries from the mediation cache. For policies that may depend on time, such as role-based access control, mappings can also be cached, but only for as long as they are valid. Overall, we expect, that for a very large class of real-world security policies, lookup operations can generally be satisfied by the mediation cache. Only the initial lookup for an entry and a lookup for an entry that has been removed due to a permission change or expiration should require mediation by the security policy manager.

3.6 Policy Examples

We generally expect that access control operations need to be injected into an extensible system at the granularity of major functional units. Consider a transaction manager running on top of a storage manager, as discussed in Section 2.1. The public interface of the transaction manager might provide calls to start, commit, and abort transactions, as well as to read and write data within a transaction. The public interface of the storage manager might provide calls to read and write disk extents. Furthermore, consider a security policy that lets users access stored data only through the transaction manager but allows power users to access the raw disk extents directly, for example, to perform maintenance work. A straightforward implementation of this policy needs to impose access control operations on all calls in the public interfaces of both services. But, it does not need to impose access control operations on the lock manager that is internally used by the transaction manager to provide isolation between transactions and that has no public interface.

More specifically, a straightforward implementation of this policy imposes protection domain transfers on all operations of the transaction manager and access checks on all operations of the storage manager. The protection domain transfers on the transaction manager change the protection domain of the calling thread, which represents either a user or a power user, to the protection domain of the transaction manager. The access checks on the storage manager verify that the calling thread represents either the transaction manager or a power user by checking that the calling thread has the right to execute the code of the storage manager.

Using USER, POWERUSER, TRANSACTION, and STORAGE as security identifiers associated with users, power users, the transaction manager, and the storage manager, respectively, and EXECUTE as the permission representing the right to execute code, the domain transfer mapping is

$$\begin{array}{llll} \{ \text{USER} & \times \ \text{TRANSACTION} & \rightarrow & \text{TRANSACTION}, \\ \ \ \text{POWERUSER} & \times \ \text{TRANSACTION} & \rightarrow & \text{TRANSACTION} \ \} \end{array}$$

And, the access check mapping is

$$\begin{array}{llll} \{ \text{USER} & \times \ \text{TRANSACTION} & \rightarrow & \text{EXECUTE}, \\ \ \ \text{POWERUSER} & \times \ \text{TRANSACTION} & \rightarrow & \text{EXECUTE}, \\ \ \ \text{POWERUSER} & \times \ \text{STORAGE} & \rightarrow & \text{EXECUTE}, \\ \ \ \text{TRANSACTION} & \times \ \text{STORAGE} & \rightarrow & \text{EXECUTE} \ \} \end{array}$$

The two entries in the domain transfer mapping cause the security identifier of a thread associated with a user or power user to be changed to the security identifier of the transaction manager on calls to the transaction manager. The first two entries in the access check mapping allow users and power users to link against the transaction manager and execute its code, and the second two entries allow power users and the transaction manager to link against the storage manager and to execute its code.

If it can be proven that only authorized threads can call the storage manager, it is possible to optimize the implementation of this policy. In particular, if only threads associated with power users and threads going through the transaction manager can call the storage manager, both the protection domain transfers on the transaction manager and the access checks on the storage manager can be omitted. This is trivially the case under the mappings specified above, because user extensions cannot link against the storage manager. However, in a real system under a real policy, this analysis is considerably more complex, as, for example, other extensions may be shared between users and power users, but also be allowed to link against the storage manager. We thus expect that a real-world policy specification is based on a worst-case assumption and requires the interposition of access control operations on all security-relevant resources and operations. An optimization component of the security policy manager can then automatically analyze this policy specification and, based on the current state of an extensible system, optimize away access control operations that are not strictly necessary.

Obviously, the above example is a fairly simple one. The intent is to illustrate how to implement security policies in our access control mechanism by using protection domain transfers and access checks and by mapping the policy onto security identifiers and permissions. It also provides a flavor of how to optimize away dynamic access checks. At the same time, not all policies can avoid dynamic access checks. Consider, for example, the problem of the confused deputy [Hardy 1988]. In this example, a compiler produces two sets of outputs, the actual object files and an accounting file that contains billing information. If a user specifies the same name as the hard-coded name of the accounting file for an optional file containing debugging information, the compiler overwrites the accounting information with the debugging information, and the user can thus avoid proper billing. This problem can be solved by using our access control mechanism. An appropriate security policy imposes protection domain transfers on all calls to the compiler's accounting component and places the accounting component into its own protection domain. It also imposes dynamic access checks on the file system to ensure that only the protection domain of the accounting component can write billing data.

4. IMPLEMENTATION

We have implemented our access control mechanism in the SPIN extensible operating system [Bershad et al. 1995]. Our access control mechanism does not depend on features that are unique to SPIN and could be implemented in other systems. It requires support for dynamically loading and linking extensions, for multiple concurrent threads of execution, for determining an extension's types and operations, and for redirecting procedure or method invocations (for example, by dynamically patching object jump tables). Consequently, our access control mechanism can be implemented in other extensible systems that provide these features, such as Java.

Our implementation is guided by three constraints. First, it has to correctly enforce a given security policy as defined by the security policy manager. Second, it has to be simple and well-structured to allow for validation[1] and for easy transfer to other systems. Third, the implementation should be fast to impose as little performance overhead as possible.

4.1 SPIN Background

The SPIN extensible operating system lets applications safely change the operating system's interface and implementation by dynamically linking extensions into its kernel. A statically linked core provides most basic services, including device support, the Modula-3 runtime [Hsieh et al. 1996; Sirer et al. 1996b], the linker/loader [Sirer et al. 1996a], threads, and the event dispatcher [Pardyak and Bershad 1996]. All other services, including networking and file system support, are provided by dynamically linked extensions. Extensions must be written in Modula-3 [Nelson 1991], a type-safe and garbage-collected programming language, and rely on the Modula-3 module system: an extension is implemented by a module and exports its types and operations through one or more interfaces. User-space applications need not be written in Modula-3, are not guaranteed to be type-safe, and are isolated from the kernel and from each other through address spaces.

The event dispatcher is the central extensibility mechanism in SPIN. It is based on an event model: extensions and the core services alike raise an event, and one or more event handlers process the event. An event is a procedure that is declared in an interface, and its handlers are procedures that have the same signature. Each event has a default handler, which is provided by the module that exports the declaring interface. The event dispatcher can dynamically modify the bindings between events and event handlers at any time by removing or adding individual event handlers. It also provides support for conditionally executing handlers depending on the actual arguments to an event or the state of the system, for enforcing ordering constraints between handlers, and for bracketing individual handlers between additional procedures.

The implementation of the event dispatcher, just like procedure invocations in Modula-3, simply uses an indirect procedure call for the common case when an event is processed only by a single handler. For events with several handlers, the implementation interposes a dispatch procedure that

[1]We have not validated the implementation. However, a critical characteristic for any security mechanism is that it be small and well-structured [Saltzer and Schroeder 1975].

invokes the individual handlers. It uses dynamic code generation to specialize the dispatch procedure by, for example, inlining small handlers, thus reducing the performance overhead of event handling. The design and implementation of the event dispatcher are described in detail in Pardyak and Bershad [1996].

4.2 The Implementation

We have implemented the basic abstractions of our access control mechanism, notably security identifiers and access modes, as well as the enforcement manager as part of SPIN's static core. Services in the static core are trusted in that, if they misbehave, the security of the system can be undermined and the system may even crash. At the same time, the static core must be protected against dynamically linked extensions, which usually are not trusted. Consequently, the enforcement manager imposes access control checks on the core services, including the linker/loader as described in Section 3.1, to protect itself and other core services and to ensure that only a trusted extension can define the security policy. User-space applications cannot access *any* kernel-level objects directly, but only through a narrowly defined system call interface, which automatically subjects them to our access control mechanism.

The implementation of our access control mechanism consists of 1000 lines of well-documented Modula-3 interfaces and 2400 lines of Modula-3 code, with an additional 50 lines of changes to other parts of the static SPIN core. It uses the Modula-3 runtime to determine the types and operations of an extension. It uses the event dispatcher to inject access control operations into the system by bracketing event handlers. Our access control mechanism thus provides precise control over an extensible system, as it imposes access control on individual event handlers and not on declared events, which, as discussed in Section 4.1, may be handled by several event handlers.

The implementation defines the abstractions for security identifiers and access modes. Security identifiers are simply integers. Access modes are immutable objects and are represented by a set of simple, predefined permissions in addition to a list of permission objects. The predefined permissions are implemented as a vector of 64 bits to provide common and frequently used permissions, such as EXECUTE, EXTEND, READ, and WRITE, at a low overhead. The list of permission objects lets the security policy manager define additional permissions (where each permission object can represent several individual permissions) by subtyping from an abstract base class. The list of permission objects thus ensures that new permissions can be introduced into the system and that policies are not limited by the 64 predefined permissions, albeit at some performance cost when compared to the predefined permissions.

The functionality of the enforcement manager is visible through two separate interfaces. One interface is accessible by all extensions and lets them discover the state of system security, such as the current security identifier of a thread, in the presence of security faults. The other interface is trusted and, together with the interface of the security policy manager, defines the protocol between the enforcement manager and security policy manager. While the interface to the security policy manager is defined as part of the static core, an implementation of this interface needs to be provided by a trusted extension outside the static core, thus making it possible to simply "plug in" an implementation of the appropriate security policy. The enforcement manager operates as described in Section 3. It uses the simplified object creation mapping for assigning objects to security identifiers (see Section 3.3) and thus provides a default security identifier for all objects that are newly created within a protection domain. Security violations are signalled through a runtime exception and thus need not be explicitly declared anywhere.

4.3 Security Identifier Management

We have modified the Modula-3 runtime so that the security identifier associated with an object is stored in the object header. On object creation, the Modula-3 allocator, through a call into the enforcement manager, copies the default object security identifier for the current protection domain into the object header. However, only some types in an extensible system require access control. For example, the objects of the lock manager discussed in Section 3.6 are only used within the transaction manager and thus never require access control. To limit the memory overhead of allocating an additional word in each object header, the security policy manager can dynamically activate and deactivate object security for each Modula-3 type individually. Access checks on objects that *are* associated with a security identifier determine the specific privileges for that particular object instance. Access checks on objects that are *not* associated with a security identifier always fail.

Storing an object's security identifier in the object header considerably simplifies the mapping from objects to security identifiers, because the enforcement manager does not need to maintain a separate mapping. For example, when an unused object is freed by the garbage collector, the corresponding mapping is deleted with the object, and no additional operation needs to be performed by the enforcement manager. Furthermore, as security identifiers are stored in the same location as the object itself, the performance overhead for accessing an object's security identifier is minimized.

To maintain a thread's security identifier and the corresponding default object security identifier, the enforcement manager associates each thread with a separate security identifier stack. Each record on this stack contains the two security identifiers for the subject and its objects. On a protection domain transfer, the enforcement manager pushes a new record onto the stack before the thread enters the corresponding event handler, and it pops the record off the stack when the thread returns from the event handler. Records are preallocated in a global pool to avoid dynamic memory allocation overhead, and they are pushed and popped using atomic enqueue and dequeue operations to avoid the overhead of locking the global pool.

Using a separate stack of security identifier records is preferable over storing the security identifiers on the call stack, because we can reasonably assume that a thread only encounters a relatively small number of nested protection domain transfers. Consequently, a separate stack of security identifier records saves space, because the two security identifiers need to be only stored once for each protection domain and not once for each frame on the call stack. It also saves time, because the two security identifiers do not need to be copied between stack frames on procedure or method invocations that do not involve a protection domain transfer.

4.4 The Mediation Cache and Concurrency

In contrast to the global pool of stack records, the implementation of the mediation cache is protected by a single, global lock, which raises the question of whether this lock represents a limit on concurrency. Lookups in the mediation cache are the common-case operation and, as shown in Section 6.1, take 114 instructions. They are fast in absolute time, and they are fast relative to the work typically done by a protected resource as well as relative to the preemption quantum of 5 milliseconds. Consequently, we do not expect this lock to be a limiting factor on concurrency. However, should it turn out to be one, it can easily be removed, for example, by statically partitioning the three mappings based on the first security identifier and by protecting each partition with its own lock.

5. DISCUSSION

By using our access control mechanism, fine-grained security constraints can be imposed onto an extensible system. However, the expressiveness of our mechanism is limited in that it cannot supplant prudent interface design. In particular, three issues arise, namely the use of abstract data types, the granularity of interfaces, and the effect of calling conventions.

Our access control mechanism provides protection on objects in that it limits the code a subject can legally execute and the objects a subject can legally pass to and from a procedure or method invocation. To do so, it relies on abstract data types to hide the implementation of an object. In other words, if the type of an object does not hide its implementation, it is possible to directly access and modify an object without explicitly invoking any of the corresponding operations and thus without incurring access control.

The structure of an interface also influences the degree of control attainable over the operations on an object. In particular, the granularity of an interface, i.e., how an interface decomposes into individual operations on a type, determines the granularity of access control. So, an interface with only one operation, which, like `ioctl` in Unix, might use an integer argument to name the actual operation, allows for much less fine-grained control than an interface with several independent operations.

The calling convention used for passing arguments to a procedure or method affects whether argument passing can be fully controlled. Notably, call-by-reference grants both caller and callee access to the same variable. As caller and callee may be in different protection domains, call-by-reference effectively creates (type-safe) shared memory. In a multithreaded system, information can be passed through shared memory at any time, not just on procedure or method invocation and return. Consequently, caller and callee need to trust each other on the use of this shared memory, and access checks on call-by-reference arguments are not very meaningful. In contrast to conventional systems, the use of shared memory is inappropriate in extensible systems, because data, including shared data, should always be represented by abstract data types. Furthermore, in SPIN, call-by-reference is almost always used to return additional results from a procedure, as Modula-3 only supports one result value. This unnecessary use of shared memory could be avoided by supporting multiple results or thread-safe calling conventions such as call-by-value/result at the programming language level.

The three issues just discussed are directly related to our access control mechanism relying on an extension's interface, that is, on the types and operations exported by an extension, to impose access constraints. A more powerful model could be used to express finer-grained security constraints. And, more aggressive techniques, such as binary rewriting [Cohen et al. 1998; Graham et al. 1995; Lee and Zorn 1997; Romer et al. 1997; Srivastava and Eustace 1994; Wahbe et al. 1993], could be used to enforce these constraints in an extensible system. But such a system would also require a considerably more complex design and implementation. At the same time, an extension's interface is a "natural" basis for access control, as it provides a concise and well-understood specification of what an extension exports to other extensions and how it interacts with them. Consequently, we believe that our access control mechanism strikes a reasonable balance between expressiveness and complexity.

As our access control mechanism relies on extensions' interfaces to provide protection for an extensible system, it also requires some means to ensure that these interfaces are, in fact, respected by the actual code. SPIN uses a type-safe programming language, Modula-3, and a trusted compiler to provide this guarantee. As a result, the compiler becomes part of the trusted computing base. Clearly, it is preferable to establish this guarantee at load-time in the extensible system that actually executes the code, especially for large computer networks. Considerable work has been devoted to this issue, and viable alternatives include typed byte-codes [Lindholm and Yellin 1996], proof-carrying code [Necula and Lee 1996], as well as typed assembly language [Morrisett et al. 1998]. All of these efforts are complementary to our own.

To inject access control operations into an extensible system, our access control mechanism requires support for code interposition. Consequently, the code of an extension not only needs to respect an extension's interfaces, but exported operations must also be readily accessible by the interposition mechanism. As a result, common compiler optimizations, such as inlining, cannot be performed on these operations, as they would make it very hard to interpose on such a procedure or method. While the need for code interposition appears to severely restrict compile-time performance optimizations, these restrictions are already necessary to effectively support dynamic composition in extensible systems, including Java. To dynamically load and link an extension, extensible systems require a standard binary format that clearly identifies the interface of an extension. Furthermore, because the actual composition of extensions can only be reliably determined when they are linked into an extensible system, common intramodule as well as cross-module optimizations, such as inlining, generally cannot be performed at compile-time, but only in the extensible system itself.

6. PERFORMANCE EVALUATION

To determine the performance overhead of our implementation, we evaluate a set of microbenchmarks that measure the performance of access control operations. We also present end-to-end performance results for a Web server benchmark. We collected our measurements on DEC Alpha AXP 133MHz 3000/400 workstations, which are rated at 74 SPECint 92 and which were well-balanced machines when the measurements were taken. Each machine has 64MB of memory, a 512KB unified external cache, a HP C2247-300 1GB disk-drive, and a 10Mbps Lance Ethernet interface. In summary, the microbenchmarks show that access control operations incur some latency on trivial operations, while the end-to-end experiment shows that the overall overhead of access control is in the noise.

6.1 Microbenchmarks

To evaluate the performance overhead of access control operations in our access control mechanism, we execute seven microbenchmarks. All seven benchmarks measure the total time for a null procedure call (a procedure that returns immediately and does not perform any work), with and without access control operations. The first benchmark simply performs a null procedure call with no arguments. The other six benchmarks additionally perform a protection domain transfer, an access check on the procedure, and access checks on one, two, four, and eight arguments, respectively.

The performance of the security policy manager is determined by a given security policy and its implementation. For the microbenchmarks, we fix the necessary entries in the mediation cache of the enforcement manager (see Section 3.5). As a result, the benchmarks measure common-case performance, where the security policy manager is not consulted, because the necessary information is already available within the enforcement manager. Furthermore, benchmarks that perform access control checks use simple permissions instead of permission objects (see Section 4.2).

Table I. Performance Numbers for Access Control Operations. All numbers are the mean of 1000 trials in microseconds. *Hot* represents hot microprocessor cache performance and *Cold* cold microprocessor cache performance.

	Hot	Cold
Null procedure call	0.1	0.5
Protection domain transfer	4.4	7.8
Access check on procedure	2.8	6.4
Access check on 1 argument	4.0	9.7
Access check on 2 arguments	6.7	12.0
Access check on 4 arguments	12.1	17.7
Access check on 8 arguments	24.0	29.5

Table I shows the performance results for the seven microbenchmarks. All numbers are in microseconds and the average of 1000 trials. To determine hot microprocessor cache performance, we execute one trial to prewarm the processor's cache and then execute it 1000 times in a tight loop, measuring the time at the beginning and at the end of the loop. To determine cold microprocessor cache performance, we measure the time before and after each trial separately and flush both the instruction and data cache on each iteration.

Table II shows the instruction breakdown of the common path for protection domain transfers, excluding the overhead for the event dispatcher (which amounts to 31 or 48 instructions, depending on the optimizations used within the event dispatcher [Pardyak and Bershad 1996]). On a protection domain transfer, the enforcement manager establishes the new protection domain before control passes into the actual procedure and restores the original protection domain upon completion of the procedure. Before entering the procedure, the enforcement manager first determines the security identifiers of the thread and of the procedure. Then, based on these security identifiers, it looks up the security identifiers for the thread and for new objects created by the thread in the mediation cache, which requires obtaining a lock for the cache. Next, it sets up a new exception frame, so that the original protection domain can be restored on an exceptional procedure exit. Finally, it pushes a new record containing the security identifiers for the thread and its objects onto the thread's security identifier stack. After leaving the procedure, the enforcement manager pops the top record from the thread's security identifier stack and removes the exception frame.

Additional experiments show that performing a protection domain transfer in addition to access checks adds 3.9 microseconds to hot cache performance and 5.6 microseconds to cold cache performance for those of the above benchmarks that perform access checks. Furthermore, using permission objects instead of simple permissions for access checks, where the required permission object matches the tenth object in the list of legal permission objects (which represents a pessimistic scenario, as each permission object can stand for dozens of individual permissions), adds 6.8

Table II. Instruction Breakdown of the Common Path for Protection Domain Transfers, Excluding the Cost for the Event Dispatcher. "Overhead" is the overhead of performing both protection domain changes within their own procedure. The other operations are explained in the text.

Operation	Number of Instructions
Enter new protection domain	
Get thread's security ID	3
Get procedure's security ID	1
Lookup in mediation cache	52
Locking overhead	62
Set up exception frame	7
Push security ID record	26
Overhead	10
Total number of instructions	161
Restore old protection domain	
Pop security ID record	22
Remove exception frame	4
Overhead	4
Total number of instructions	30

microseconds for hot cache performance and 7.0 microseconds for cold cache performance per argument.

The performance results show that access control operations have noticeable overhead. They thus back our basic premise that access control for extensible systems should only impose as much structure as strictly necessary. Furthermore, they underline the need for a design that enables dynamic optimizations which avoid access control operations whenever possible.

6.2 End-to-End Performance

To evaluate the overall impact of access control on system performance, we present end-to-end results for a Web server benchmark. We believe that a Web server benchmark is relevant, because extensibility mechanisms for Web servers, such as Java servlets [Davidson and Coward 1999], are in wide-spread use and because their security is a primary concern. The Web server used for our experiments is implemented as an in-kernel extension. It uses an NFS client to read files from our group's file server and locally caches the file data in a dedicated cache, which is backed by a simple and fast extent-based file system. As spawning new threads in SPIN incurs very little overhead, the Web server forks a new thread for each incoming request. The thread first checks whether the requested file is available in the local cache and, if so, sends the file data directly from the cache.

Otherwise, it issues an NFS read request, stores the file in the local cache, and then sends the data.

Our security policy places the Web server into its own protection domain. It performs access control checks on all NFS and local cache operations. Files in the local cache are automatically associated with a security identifier as described in Section 4.2. Files in NFS are associated with a security identifier by using a mapping from the file system name-space to security identifiers similar to the one described in Badger et al. [1995a; 1995b] in order to provide fine-grained control over which files are associated with which security identifier. Since the security policy imposes access control checks on both the NFS client and the local cache, and since individual threads (spawned to serve requests) can only communicate through NFS and the local cache, the policy ensures that only authorized files are accessible through the Web server. Furthermore, it makes it possible to securely change privileges on a per-request basis, either based on a remote login or based on the machine from which the request originates.

Our performance benchmark sends HTTP requests from one machine that is running the benchmark script to another that is running the Web server. It reads the entire SPIN Web tree, to a total of 79 files or 5035KB of data. We run the benchmark without access control, as a baseline, and with access control, to measure the end-to-end overhead of our access control mechanism. For the measurements without access control, the access control mechanism, including all changes to the SPIN runtime, is disabled. For each measurement, we first perform 15 runs of the benchmark to prewarm the local cache and then measure the latency for 20 runs.

The average latency for one run of the benchmark both without and with access control is 16.9 seconds. Out of the 16.9 seconds, 5.4 seconds are idle time on the machine running the Web server, again both without and with access control. Trials with access control incur a total of 1573 access checks, on average 20 for each file, and all necessary lookups for the three mappings are satisfied by the mediation cache, which has been filled during the pretrial runs. While some time for each trial is spent reading data from disk and sending data over the network, the important characteristic for this benchmark is CPU utilization, as it determines the scalability of the Web server. The CPU utilization is 68% both without and with access control. The end-to-end performance experiment thus shows that the overhead of access control operations is negligible for a Web server workload running under a realistic security policy. We extrapolate from this result that other applications will see at most a small overhead under real-world security policies.

7. RELATED WORK

A considerable body of work focuses on system protection [Lampson 1971; Saltzer and Schroeder 1975] and appropriate security policies. Starting from multilevel security [Bell and LaPadula 1976; Biba 1977; Denning

1976], which has become part of the U.S. Department of Defense's standard for trusted computer systems [Department of Defense Computer Security Center 1985], much attention has been directed toward mapping nonmilitary policies onto multilevel security [Lee 1988; Lipner 1982], defining alternative policies more suitable for commercial applications [Badger et al. 1995a; 1995b; 1997; Brewer and Nash 1989; Boebert and Kain 1985; Clark and Wilson 1987; Ferraiolo and Kuhn 1992; Ferraiolo et al. 1995; Sandhu et al. 1996], and expanding multilevel security to be more flexible and powerful [McCollum et al. 1990; Myers and Liskov 1997].

7.1 Distributed Trusted Operating System

Based on the realization that no single security policy is appropriate for all environments, the distributed trusted operating system (DTOS) effort [Minear 1995; Olawsky et al. 1996; Secure Computing Corporation 1997a; 1997b] has the goal of providing a policy-neutral access control mechanism. As our mechanism builds on this work, we share with DTOS the same structuring of access control into a security policy manager and a policy-neutral enforcement manager as well as the same basic abstractions, namely security identifiers and permissions. However, as DTOS has been implemented on top of the Mach microkernel rather than within a type-safe system, it differs from our mechanism in that it relies on address spaces to isolate protection domains, which results in a relatively high overhead for changing protection domains. Furthermore, as the DTOS effort does not separate security from functionality, access checks have been integrated into the kernel sources and check for predefined permissions, making it impossible for users or administrators to change or remove access checks.

As reported in Secure Computing Corporation [1997a], embedding access checks within the kernel sources presented a considerable challenge as it fixed part of the security policy within the system. Furthermore, as noted in Secure Computing Corporation [1997b], their choice of checking whether a subject can perform an operation on an object, where the object is the primary argument to an operation, does not provide sufficient flexibility, since the security decision may depend on other arguments to the operation as well. Our access control mechanism avoids these limitations, because access control operations can be performed on any operation, are dynamically injected into the system, and are strictly more expressive.

7.2 Java Platform Security

Due to Java's [Gosling et al. 1996; Lindholm and Yellin 1996] popularity for providing executable content on the Internet and prompted by a string of security breaches [Dean et al. 1996; McGraw and Felten 1997] in early versions of the platform, research into protection for extensible systems has mostly focused on Java. In departure from the original sandbox model, which grants trusted code full access to the underlying system and untrusted code almost no access, the Java security architecture has been extended to allow for multiple protection domains, provide fine-grained access control primitives, and support cryptographic protocols [Gong 1997; 1999; Gong et al. 1997].

Dynamic access checks in Java are performed using a technique called extended stack inspection [Wallach et al. 1997]. With this technique, each extension is implicitly associated with a protection domain. Access checks verify that all callers, and therefore protection domains, represented on the current call stack have the required permission. The stack walk starts at the current stack frame and ends either at a stack frame that has explicitly asserted the necessary privilege or at the last frame. While the original specification of extended stack inspection is closely tied to Java's stack-based execution model, subsequent work has developed a formal model for this technique [Wallach and Felten 1998].

Compared to our access control mechanism, which requires explicit protection domain transfers and thus resolves them eagerly, extended stack inspection performs protection domain transfers lazily. As a result, access checks for extended stack inspection need to consider every stack frame during the stack walk, even though the number of distinct protection domains represented on the call stack is typically much smaller than the number of stack frames [Erlingsson and Schneider 2000]. Consequently, a straightforward implementation of extended stack inspection can incur a considerable performance overhead [Gong and Schemers 1998].

A more fundamental drawback of the Java security architecture is that it does not separate functionality from access control. Access control operations need to be embedded into an extension's source code by the programmer of that extension. As a result, users or administrators cannot express policies that require other access control operations than those coded into an extension. For example, Sun's Java platform release only includes access checks on file open operations, but not on file read or write operations, making it impossible to express security policies that also require checks on the latter operations.

7.3 Access Control by Limiting Effective Types

Hagimont and Ismail describe an alternative approach to access control for Java, which enforces access control by restricting the visibility of object methods [Hagimont and Ismail 1997]. In their access control mechanism, security constraints are expressed in an extended interface definition language by specifying so-called views. Views resemble Java interfaces with additional annotations to declare which methods cannot be invoked in that particular view and which object references passed to and from method invocations are restricted by other views. Views are implemented as proxy objects that encapsulate the original object. The methods of a proxy either throw an exception for disallowed methods or invoke the corresponding method on the original object while also wrapping passed references with the appropriate proxies.

In its use of limited effective types to enforce access control constraints, Hagimont and Ismail's protection scheme is similar to CACL, an access control mechanism for type-safe, object-oriented systems [Richardson et al. 1992]. In CACL, objects are protected by access control lists that specify who can perform which operations on an object. Each object instance has an owner, who determines that object's access control list and who may also transfer ownership of the object. Each object also has a so-called method principal, representing the subject on whose behalf an object's code is executed. The method principal defaults to an object's implementor, but an object's owner may take over as the method principal.

CACL's implementation uses different object jump tables for the same object to represent different access restrictions. Initially, all entries in a default jump table redirect method invocations to a protection manager, which makes the access control decisions. After mediation, this jump table is replaced with a different jump table that grants access only to the operations the subject is authorized to perform. If the current subject changes, because, for example, an object reference is passed to a method with a different method principal, the object's jump table is reset to the default jump table.

The proxy objects used in Hagimont and Ismail's protection scheme are statically generated by a preprocessor from the corresponding extended interface definitions and cannot be changed dynamically. As a result, their access control mechanism cannot accommodate dynamic security policy changes. Furthermore, the use of proxy objects to limit the effective type of an object is somewhat wasteful, both in time and space. CACL does not suffer from these limitations, because it is integrated with a system's runtime. Furthermore, CACL's use of limited effective types to effectively cache the results of access checks and to consequently avoid repeated dynamic access checks is complementary to our design. We believe that this technique could be used to provide an efficient implementation of the enforcement manager in a pure object-oriented system.

7.4 Separating Access Control and Functionality

Based on similar observations to ours (as discussed in Section 2), Ariel [Pandey and Hashii 1999], Naccio [Evans and Twyman 1999], and SASI [Erlingsson and Schneider 1999] have also explored the separation of access control and functionality for system protection. All three use binary rewriting to combine an external policy specification with the actual program code to produce a secure version of the code. At the same time, they differ significantly in how execution platforms are targeted and how security policies are specified.

Ariel targets Java, and policies are expressed as access constraints on individual methods in a declarative, constraint-based language. Naccio seeks to support the specification of platform-independent policies and has been implemented for Java and Win32 executables. Policies are expressed in terms of abstract resource manipulations in an imperative, Java-like language. Such a policy specification, together with the corresponding definition of the abstract resources and a mapping of these resources into the target platform, is then used to inject the policy into a particular program. Unlike Ariel and Naccio, which express policies at the level of individual procedures or methods, SASI supports the expression of policies at the instruction level. Two separate implementations target Java and x86 executables. Policies are specified as security automata which serve as execution monitors that reject policy-violating executions [Schneider 2000].

Two efforts have explored how to support Java's extended stack inspection, while also separating access control and functionality. Based on the formalization of Java's stack inspection model [Wallach and Felten 1998], Wallach has developed an implementation of the formal model that can be imposed on insecure code by rewriting it [Wallach 1999]. The implementation uses an additional argument representing the current security state for each method invocation, and this approach is hence called security-passing style. Erlingsson and Schneider have reimplemented security-passing style as well as an alternative approach, called IRM_{Lazy}, in the PoET/PSLang toolkit [Erlingsson and Schneider 2000]. PoET/PSLang is a successor to SASI and has been specifically designed to minimize the size of the trusted computing base by keeping the binary rewriter simple and small. Both Wallach's and Erlingsson and Schneider's implementations of security-passing style can introduce a noticeable performance overhead over a direct implementation of stack inspection. However, the implementation of IRM_{Lazy} performs comparably to the direct implementation and separates access control from functionality, which considerably simplifies changes to Java's stack inspection model and makes it possible to impose access control on insecure code.

The distributed virtual machine (DVM) architecture, which has been implemented for Java, aims at providing uniformity and manageability for all virtual machines within an organization [Sirer et al. 1999]. The DVM security service directly builds on our access control mechanism. The security policy manager is centralized and, by using binary rewriting, injects access control operations into extensions before they are loaded on a client machine. The enforcement manager is local to each client and remotely queries the security policy manager for making dynamic access decisions. As reported in Sirer et al. [1999], access checks under the DVM security service show performance generally comparable to Java's stack inspection for the common case, in which mappings are cached in the enforcement manager's mediation cache and in which the centralized security policy manager need not be queried.

Policies for the DVM security service are specified in a declarative language based on XML. A policy specification consists of an access matrix, which concisely specifies the domain transfer and access check mappings discussed in Section 3.3, name spaces, which map resource names into security identifiers, and the per-method mapping of access control operations onto the actual code. Since object constructors in Java, unlike the `NEW` operator in Modula-3, are as expressive as regular methods, the mapping of access control operations onto the actual code also specifies how newly created objects are associated with security identifiers and thus subsumes the object creation mapping discussed in Section 3.3.

The enforcement manager for DVMs differs from the enforcement manager described in this paper in three aspects. First, it supports additional access control operations, notably, to associate newly created objects with security identifiers based on an object's name and the name spaces declared by the policy. Second, it relies on symbolic security identifiers and permissions, which were chosen to simplify future integration with SPKI, a key-based authorization infrastructure [Ellison et al. 1999]. Third, in order to remain compatible with existing Java virtual machines, it explicitly maps objects to security identifiers, using a hash table and weak references for the objects. A subclass to Java's default thread implementation provides the per-thread security identifier stack and is injected into extensions by the binary rewriter to replace references to Java's default implementation.

The separation of access control and functionality in our access control mechanism as well as the systems discussed in this section raises the question of how the programmer who writes an extension can specify security constraints on the extension. One solution, which is used in Microsoft's .NET platform [Richter 2000], lets the programmer declare security constraints and embed them as attributes within an extension's executable file. Security constraints are expressed in terms of permissions at the granularity of entire classes or individual methods and specify, for example, the permissions that are required to link against a class or to dynamically call a method. An extension's security constraints, as specified by the extension's programmer, can then be read from the executable file when the extension is loaded and can be integrated with the local security policy.

7.5 Remarks

As the discussion in this section illustrates, several techniques can be used to inject access control operations into an extensible system. Binary rewriting is especially attractive for Java, because it preserves the Java virtual machine interface and does not require modifications to the virtual machine itself. However, since the binary rewriter is external to the execution platform, the access control mechanisms discussed in Section 7.4 cannot modify code dynamically as it is executing. As a result, these mechanisms cannot support dynamic policy changes that require reinstrumentation. The DVM security service mitigates this restriction somewhat, because it does not embed the entire policy in program code; rather, it relies on a separate security policy manager to perform mediation. The other mechanisms could obviously use a similar structuring to also lessen this restriction.

In addition to the SPIN event dispatcher and CACL, several other efforts have explored dynamic binary interposition, including interposition at the Unix system interface [Ghormley et al. 1998; Jones 1993; Krell and Krishnamurthy 1992], at arbitrary locations within the kernel [Tamches and Miller 1999], and at the function level for Win32 applications [Hunt and Brubacher 1999]. We are thus confident that binary interposition is a viable technique for imposing security constraints onto an extensible system.

We believe that imposing access control through binary interposition is preferable to traditional protection schemes directly based on access control lists, as used, for example, in the Unix file system, or capabilities, as used in Hydra [Cohen and Jefferson 1975]. As systems become more complex and are increasingly networked, it is especially important that users and administrators have the means to survey and change a system's security state. The separation of access control and functionality addresses this concern, because it centralizes control and thus makes it possible to concisely represent a system's security state. Furthermore, as illustrated by the DVM security service, the separation of policy and enforcement in our access control mechanism enables the uniform enforcement of the same policy across several nodes in a distributed system.

8. CONCLUSIONS

Extensible systems, such as Java or the SPIN extensible operating system, have a different overall structure from conventional systems, such as Unix, because of their ability to dynamically compose units of code, or extensions. Extensions and the core services in an extensible system are typically collocated within the same address space and form a tightly integrated system, easily leading to more complex interactions between the different components than those in conventional systems. Since extensions generally cannot be trusted to conform to an organization's security policy, access control becomes a fundamental requirement for the success of extensible systems.

To protect extensions and the core services alike, access control for extensible systems needs to impose additional structure onto an extensible system. At the same time, it should only impose as much structure as is strictly necessary to preserve the advantages of extensible systems. Based on this realization, we have identified four goals to guide the design of an access control mechanism for extensible systems: (1) separate access control and functionality, (2) separate policy and enforcement, (3) use a simple yet expressive model, and (4) enforce transparently.

We have presented an access control mechanism for extensible systems that directly addresses these goals. Our access control mechanism separates access control and functionality by inspecting extensions for their types and operations to determine which abstractions require protection and by redirecting individual procedure or method invocations to inject access control operations into the system. It separates policy and enforcement by breaking up access control into a security policy manager, which makes the actual access decisions, and a policy-neutral enforcement manager, which enforces these decisions in the extensible system. It uses a simple yet expressive model that supports protection domain transfers to allow for a controlled change of privilege, access checks to limit which procedures or methods can be invoked and which objects can be passed, and auditing to provide a trace of system operations. Finally, it enforces transparently, as long as no violations of the security policy occur; extensions are notified of security faults so that they can implement their own failure model.

The implementation of our access control mechanism within the SPIN extensible operating system is simple and, even though the latency of individual access control operations can be noticeable, shows good end-to-end performance for a Web server benchmark. Based on our results, we predict that most systems will see a very small overhead for access control and thus consider our access control mechanism an effective solution for access control in extensible systems.

ACKNOWLEDGMENTS

We thank Dennis Hollingworth and Timothy Redmond at Trusted Information Systems (now Network Associates) for their comments on our design. Marc Fiuczynski, Tian Lim, Yasushi Saito, Emin Gün Sirer, and especially Przemysław Pardyak at the University of Washington were most helpful with various implementation issues and the integration of our access control mechanism into SPIN. Larry Peterson, Stefan Savage, Stephen Smalley, Ray Spencer, Mike Swift, Amin Vahdat, and particularly the anonymous reviewers for this journal provided valuable feedback on earlier versions of this paper.

REFERENCES

Badger, L., Oostendorp, K. A., Morrison, W. G., Walker, K. M., Vance, C. D., Sherman, D. L., and Sterne, D. F. 1997. DTE firewalls—initial measurement and evaluation report. Tech. Rep. 0632R. Trusted Information Systems, Inc., Glenwood, MD.

Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. 1995a. Practical domain and type enforcement for UNIX. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May). IEEE Computer Society Press, Los Alamitos, CA, 66–77.

Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. 1995b. A domain and type enforcement UNIX prototype. In *Proceedings of the 5th USENIX UNIX Security Symposium* (Salt Lake City, UT, June). USENIX Assoc., Berkeley, CA, 127–140.

Bell, D. E. and LaPadula, L. J. 1976. Secure computer systems: Unified exposition and Multics interpretation. Tech Rep. MTR-2997 Rev. 1 (Mar.). MITRE Corp., Bedford, MA. Also ADA023588, National Technical Information Service.

Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C., and Eggers, S. 1995. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO, Dec.). ACM Press, New York, NY, 267–284.

Biba, K. J. 1977. Integrity considerations for secure computer systems. Tech. Rep. MTR-3153 Rev. 1 (Apr.). MITRE Corp., Bedford, MA. Also ADA039324, National Information Service.

Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A. D. 1999. The role of trust management in distributed systems security. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, J. Vitek and C. Jensen, Eds. Lecture Notes in Computer Science, vol. 1603. Springer-Verlag, New York, NY, 185–210.

BOEBERT, W. E. AND KAIN, R. Y. 1985. A practical alternative to hierarchical integrity policies. In *Proceedings of the 17th National Conference on Computer Security* (Gaithersburg, MD). 18–27.

BREWER, D. F. C. AND NASH, M. J. 1989. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May 1-3). IEEE Computer Society Press, Los Alamitos, CA, 206–214.

CHASE, J. S., LEVY, H. M., FEELEY, M. J., AND LAZOWSKA, E. D. 1994. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst. 12*, 4 (Nov.), 271–307.

CLARK, D. AND WILSON, D. 1987. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA). IEEE Computer Society Press, Los Alamitos, CA, 184–194.

COHEN, E., JEFFERSON, D., AND JEFFERSON, D. 1975. Protection in the Hydra operating system. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles* (Austin, TX, Nov.). 141–160.

COHEN, G., CHASE, J., AND KAMINSKY, D. 1998. Automatic program transformation with JOIE. In *Proceedings of the 1998 USENIX Annual Technical Conference* (New Orleans, LA, June). USENIX Assoc., Berkeley, CA, 167–178.

DAVIDSON, J. D. AND COWARD, D. 1999. Java servlet specification, v2.2. Sun Microsystems, Inc., Mountain View, CA.

DEAN, D., FELTEN, E., AND WALLACH, D. 1996. Java security: From HotJava to Netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA, May). IEEE Press, Piscataway, NJ, 190–200.

DENNING, D. E. 1976. A lattice model of secure information flow. *Commun. ACM 19*, 2 (May), 236–243.

DEPT. OF DEFENSE COMPUTER SECURITY CTR. 1985. Department of Defense trusted computer system evaluation criteria. Department of Defense Standard DoD 5200.28-STD.

ELLISON, C. M., FRANTZ, B., LAMPSON, B., RIVEST, R., THOMAS, B., AND YLONEN, T. 1999. SPKI certificate theory. RFC 2693. Internet Engineering Task Force.

ERLINGSSON, Ú. AND SCHNEIDER, F. B. 1999. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 ACM Workshop on New Security Paradigms* (Caledon Hills, Ontario, Canada, Sept.). ACM Press, New York, NY, 87–95.

ERLINGSSON, Ú. AND SCHNEIDER, F. B. 2000. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (Oakland, CA, May). 246–255.

EVANS, D. AND TWYMAN, A. 1999. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy* (Oakland, California, May). 32–45.

FERRAIOLO, D. AND KUHN, D. R. 1992. Role based access control. In *Proceedings of the 15th Annual Conference on National Computer Security*. National Institute of Standards and Technology, Gaithersburg, MD, 554–563.

FERRAIOLO, D., CUGINI, J., AND KUHN, D. R. 1995. Role based access control (RBAC): Features and motivations. In *Proceedings of the 11th Annual Conference on Computer Security Applications* (New Orleans, LA, Dec.). IEEE Computer Society Press, Los Alamitos, CA, 241–248.

GHORMLEY, D. P., PETROU, D., ANDERSON, T. E., AND RODRIGUES, S. H. 1998. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference* (New Orleans, LA, June). USENIX Assoc., Berkeley, CA, 39–52.

GONG, L. 1997. Java security: Present and near future. *IEEE Micro 17*, 3 (May/June), 14–19.

GONG, L. 1999. *Inside Java Platform Security—Architecture, API Design, and Implementation*. Addison-Wesley, Reading, MA.

GONG, L. AND SCHEMERS, R. 1998. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the 1998 Internet Society Symposium on Network and Distributed System Security* (San Diego, CA, Mar.). 125–134.

GONG, L., MUELLER, M., AND PRAFULLCHANDRA, H. 1997. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings* *of the USENIX Symposium on Internet Technologies and Systems* (Monterey, CA, Dec.). USENIX Assoc., Berkeley, CA, 103–112.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, MA.

GRAHAM, S. L., LUCCO, S., AND WAHBE, R. 1995. Adaptable binary programs. In *Proceedings of the 1995 USENIX Annual Technical Conference* (New Orleans, LA, Jan.). USENIX Assoc., Berkeley, CA, 315–325.

HAGIMONT, D. AND ISMAIL, L. 1997. A protection scheme for mobile agents on Java. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking* (MOBICOM '97, Budapest, Hungary, Sept. 26–30), L. Pap, K. Sohraby, D. B. Johnson, and C. Rose, Chairs. ACM Press, New York, NY, 215–222.

HARDY, N. 1988. The confused deputy. *ACM SIGOPS Oper. Syst. Rev. 22*, 4 (Oct.), 36–38. http://www.cis.upenn.edu/ KeyKOS/ConfusedDeputy.html

HSIEH, W. C., FIUCZYNSKI, M. E., GARRETT, C., SAVAGE, S., BECKER, D., AND BERSHAD, B. N. 1996. Language support for extensible operating systems. In *Proceedings of the ACM Workshop on Compiler Support for System Software* (Tucson, AZ, Feb.). 127–133.

HUNT, G. AND BRUBACHER, D. 1999. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium* (Seattle, WA, July). USENIX Assoc., Berkeley, CA, 135–143.

JONES, M. B. 1993. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, Dec. 5–8), A. P. Black and B. Liskov, Chairs. ACM Press, New York, NY, 80–83.

KRELL, E. AND KRISHNAMURTHY, B. 1992. COLA: Customized overlaying. In *Proceedings of the 1992 Winter USENIX Conference* (San Francisco, CA, Jan.). USENIX Assoc., Berkeley, CA, 3–7.

LAMPSON, B. 1971. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems* (Princeton, NJ, Mar.). 437–443.

LEE, H. B. AND ZORN, B. G. 1997. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (Monterey, CA, Dec.). 73–82.

LEE, T. 1988. Using mandatory integrity to enforce "commercial" security. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA, Apr.). 140–146.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA.

LIPNER, S. B. 1982. Non-discretionary controls for commercial applications. In *Proceedings of the 1982 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, CA, Apr.). IEEE Computer Society Press, Los Alamitos, CA, 2–10.

MCCOLLUM, C. J., MESSING, J. R., AND NOTARGIACOMO, L. 1990. Beyond the pale of MAC and DAC—Defining new forms of access control. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA). IEEE Computer Society Press, Los Alamitos, CA, 190–200.

MCGRAW, G. AND FELTEN, E. W. 1996. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, Inc., New York, NY.

MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley UNIX and open systems series. Addison-Wesley Publishing Co., Inc., Redwood City, CA.

MINEAR, S. E. 1995. Providing policy control over object operations in a Mach-based system. In *Proceedings of the 5th USENIX UNIX Security Symposium* (Salt Lake City, UT, June). USENIX Assoc., Berkeley, CA, 141–156.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1998. From System F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '98, San Diego, CA, Jan. 19–21), D. B. MacQueen and L. Cardelli, Chairs. ACM Press, New York, NY, 85–97.

MYERS, A. C. AND LISKOV, B. 1997. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (SOSP '97, Saint-Malo, France, Oct. 5–8), W. M. Waite, Ed. ACM Press, New York, NY, 129–142.

NECULA, G. C. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA, Oct.). 229–243.

NELSON, G., ED. 1991. *Systems Programming With Modula-3*. Prentice-Hall series in innovative technology. Prentice-Hall, Inc., Upper Saddle River, NJ.

OLAWSKY, D., FINE, T., SCHNEIDER, E., AND SPENCER, R. 1996. Developing and using a "policy neutral" access control policy. In *Proceedings of the 1996 ACM Workshop on New Security Paradigms* (Lake Arrowhead, CA, Sept.). ACM, New York, NY, 60–67.

PANDEY, R. AND HASHII, B. 1999. Providing fine-grained access control for Java programs. In *Proceedings of the 13th European Conference on Object-Oriented Programming* (Lisbon, Portugal, June). Springer-Verlag, New York, NY, 449–494.

PARDYAK, P. AND BERSHAD, B. N. 1996. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (OSDI '96, Seattle, WA, Oct. 28–31), K. Petersen and W. Zwaenepoel, Chairs. ACM Press, New York, NY, 201–212.

RICHARDSON, J., SCHWARZ, P., AND CABRERA, L. -F. 1992. CACL: Efficient fine-grained protection for objects. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '92, Vancouver, British Columbia, Canada, Oct. 18–22), J. Pugh, Chair. ACM Press, New York, NY, 263–275.

RICHTER, J. 2000. Microsoft .NET framework delivers the platform for an integrated, service-oriented web. *MSDN Mag. 15*, 10 (Oct.), 56–65.

ROMER, T., VOELKER, G., LEE, D., WOMAN, A., WONG, W., LEVY, H., BERSHAD, B. N., AND CHEN, B. 1997. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop* (Seattle, WA, Aug.). USENIX Assoc., Berkeley, CA, 1–8.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. *Proc. IEEE 63*, 9 (Sept.), 1278–1308.

SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. 1996. Role-based access control models. *IEEE Computer 29*, 2 (Feb.), 38–47.

SCHNEIDER, F. B. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur. 3*, 1 (Feb.), 30–50.

SECURE COMPUTING CORPORATION. 1997a. DTOS lessons learned report. Tech. Rep. DTOS CDRL A008. Secure Computing Corporation, Roseville, MN.

SECURE COMPUTING CORPORATION. 1997b. DTOS general system security and assurability assessment report. Tech. Rep. DTOS CDRL A011. Secure Computing Corporation, Roseville, MN.

SIRER, E. G., FIUCZYNSKI, M., PARDYAK, P., AND BERSHAD, B. N. 1996a. Safe dynamic linking in an extensible operating system. In *Proceedings of the ACM Workshop on Compiler Support for System Software* (Tucson, AZ, Feb.). 134–140.

SIRER, E. G., GRIMM, R., GREGORY, A. J., AND BERSHAD, B. N. 1999. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the 17th ACM Symposium on Operating System Principles* (Kiawah Island Resort, SC, Dec.). ACM Press, New York, NY, 202–216.

SIRER, E. G., SAVAGE, S., PARDYAK, P., DEFOUW, G. P., ALAPAT, M. A., AND BERSHAD, B. N. 1996b. Writing an operating system with Modula-3. In *Proceedings of the ACM Workshop on Compiler Support for System Software* (Tucson, AZ, Feb.). 141–148.

SMITH, J. M., CALVERT, K. L., MURPHY, S. L., ORMAN, H. K., AND PETERSON, L. L. 1999. Activating networks: A progress report. *IEEE Computer 32*, 4 (Apr.), 32–41.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language, Design and Implementation* (PLDI '94, Orlando, FL, June 20–24), V. Sarkar, B. Ryder, and M. L. Soffa, Chairs. ACM Press, New York, NY, 196–205.

TAMCHES, A. AND MILLER, B. P. 1999. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (OSDI '99, New Orleans, LA., Feb.). USENIX Assoc., Berkeley, CA, 117–130.

VON EICKEN, T., CHANG, C.-C., CZAJKOWSKI, G., HAWBLITZEL, C., HU, D., AND SPOONHOWER, D. 1999. J-Kernel: A capability-based operating system for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, J. Vitek and C. Jensen, Eds. Lecture Notes in Computer Science, vol. 1603. Springer-Verlag, New York, NY, 369–393.

WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, Dec. 5–8), A. P. Black and B. Liskov, Chairs. ACM Press, New York, NY, 203–216.

WALLACH, D. S. 1999. A new approach to mobile code security. Ph.D. Dissertation. Department of Computer Science, Princeton Univ., Princeton, NJ.

WALLACH, D. S. AND FELTEN, E. W. 1998. Understanding Java stack introspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (Oakland, CA, May). IEEE Computer Society Press, Los Alamitos, CA, 52–63.

WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. 1997. Extensible security architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating System Principles* (Saint-Malo, France, Oct.). ACM, New York, NY, 116–128.

WETHERALL, D. 1999. Active network vision and reality. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Kiawah Island Resort, SC, Dec.). 64–79.

YOSHIKAWA, C., CHUN, B., EASTHAM, P., VAHDAT, A., ANDERSON, T., AND CULLER, D. 1997. Using smart clients to build scalable services. In *Proceedings of the 1997 USENIX Annual Technical Conference* (Anaheim, CA, Jan.). USENIX Assoc., Berkeley, CA, 105–117.

Received: October 1998; revised: February 2000 and June 2000; accepted: December 2000