

# Providing Policy-Neutral and Transparent Access Control in Extensible Systems

Robert Grimm and Brian N. Bershad

{rgrimm, bershad}@cs.washington.edu  
Department of Computer Science and Engineering,  
University of Washington,  
Box 352350  
Seattle, WA 98195, U.S.A.

**Abstract.** 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.

## 1 Introduction

Extensible systems, such as Java [18, 25] or SPIN [6], promise more power and flexibility, and thus enable new applications such as smart clients [48] or active networks [44]. Extensible systems 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. Extensions and the core system services are typically co-located within the same address space, and form a tightly integrated system. Consequently, extensible systems differ fundamentally from conventional systems, such as Unix [29], 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. As system security is customarily expressed through protection domains [22, 38], an access control mechanism must:

- structure the system into protection domains (which are an orthogonal concept to conventional address spaces),
- enforce these domains through access control checks,
- 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 requisites. We build on the idea of separating policy and enforcement first explored by the DTOS effort [30, 34, 40, 39], and introduce a mechanism that not only separates policy from enforcement, but also 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 its policy-neutral enforcement manager and a given security policy, and is transparent to extensions and the core system services in the absence of security violations. It 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. The operations are (1) explicit protection domain transfers to delineate the protection domains of an extensible system, (2) access checks to control which code can be executed and which arguments can be passed between protection domains, and (3) auditing to provide a trace of system operations. The access control mechanism works at the granularity of individual procedures (or, object methods), and provides precise control over extensions and the core system services alike.

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 a simple, yet powerful access control mechanism, as presented in this paper, 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 elaborates on the goals of our access control mechanism, and Sect. 3 describes its design. Section 4 presents the implementation of our access control mechanism within the SPIN extensible operating system. Section 5 reflects on our experiences with designing and implementing our access control mechanism, and Sect. 6 presents

a detailed performance analysis of the implementation. Section 7 reviews related work, and Sect. 8 outlines future directions for our research into the security of extensible systems. Finally, Sect. 9 concludes this paper.

## 2 Goals

An access control mechanism for an extensible system must impose additional structure onto the system. But, at the same time, it should only impose as much structure as *strictly* necessary to preserve the advantages of an extensible system. Based on this realization, we identify four goals which inform the design of our system.

*Separate access control and functionality.* The access control mechanism should separate policy and enforcement from the actual code of the system and extensions. This separation of access control and functionality supports changing security policies without requiring access to source code. This is especially important for large computer networks, such as the Internet, where the same extension may execute on different systems with different security requirements, and where source code typically is not available. This goal does *not* prevent the 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, comparable 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.

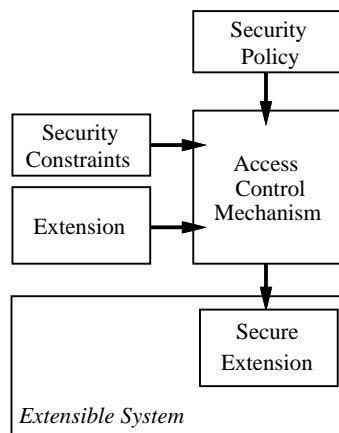
*Separate policy and enforcement.* The mechanism should separate the security policy from its actual enforcement. This separation of policy and enforcement allows for changing security policies without requiring intrinsic changes to the core services of the extensible system itself. Rather, the security 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.

*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. This goal 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 security policy and its enforcement.

*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. This goal 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, it guarantees that extensions *are* notified of security faults, and can implement their own failure model. Consequently, this goal attempts to reduce the access control interface,

as seen by extensions, to handling a program fault such as division by zero or dereferencing a NIL reference.

The above four goals, taken together, call for a design that isolates 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 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 Fig. 1.



**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.

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 an *internal* protocol. In other words, the abstractions used for expressing protection domains and access control checks need not be, and probably should not be, the same abstractions presented by the security policy. It is the responsibility of a security policy manager to provide users and system 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 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.

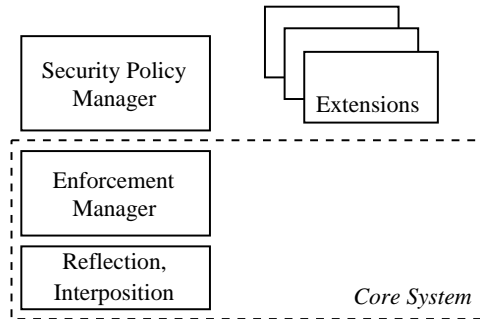
From a security viewpoint, the programmer who writes such an extension will want to protect the resources used by that extension. So, for a transaction manager, she would like to ensure that the files or disk extents used for storing transaction data can only be accessed through the transaction manager. And, for an ORB, she would like to ensure that the network port used for communicating with other nodes can not 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 they use. 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 on the transaction manager in order to ensure that only a transaction's initiator can actually 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, since the ORB is the only means for remote communication in an installation (such as a corporate intranet), and 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.

So far, we have argued for a clean separation of security policy, functionality, and enforcement, and for a system that transparently manages security. 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 an extension, and redirects procedure or method invocations to perform access control operations. The security policy manager is provided by a trusted extension, and determines the actual security policy for the system. It decides which procedures require which access control operations, and performs the actual mediation for access control. This structure is illustrated in Fig. 2.



**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.

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 and access constraints. Permissions are associated with operations, and represent the right to perform

an operation. 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.

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 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 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. It is thus necessary, at link-time, to provide access control over which interfaces a given extension can link against for execution and extending [41, 35]. 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 can not link against an interface, it can not 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 enforcing checks on linkage operations, and by executing these 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. To execute these operations within a protection domain appropriate for an extension, the loader spawns a new thread for each extension to be loaded into the system, which then performs the actual linkage operations as well as other necessary initialization. The initial security identifier of this thread represents the corresponding protection domain. It is determined by the security policy manager based on an extension's signature, and is associated with the thread and the procedures of the extension 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. Based on this information and an extension's signature, the security policy manager determines which operations and types require which access control operations. The security policy manager, in turn, instructs the enforcement manager to provide security identifiers for an extension's types, and to inject access control operations into the extension and other parts of the system, if necessary. 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 structure the system into protection domains, (2) access control checks to enforce these protection domains, 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 of a thread and on the procedure that is about to be invoked. Access checks determine whether the current subject is allowed to execute a procedure at all, and control the passing of arguments and results. For each argument that is passed into a procedure, and for each result that is passed back from the procedure, access checks determine whether the subject has sufficient rights for the corresponding object. Finally, auditing generates a log-entry for each procedure invocation, and serves as an execution trace of the system.

When instructing the enforcement manager to perform access control operations on a given procedure, 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 procedure itself, one for each argument, and one for each result.

The access control operations are ordered as follows. Before a given procedure 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, 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.

To perform the access control operations, the enforcement manager requires three mappings between security identifiers, types, and access modes. These mappings are used to communicate a site's security policy between security policy manager and enforcement manager. Using `SID` for security identifiers, `TYPE` for types as defined by the extensible system, and `ACCESSMODE` for access modes, the three mappings are:

- (1) `SID × SID`  $\longrightarrow$  `SID`
- (2) `SID × SID`  $\longrightarrow$  `ACCESSMODE`
- (3) `SID × TYPE`  $\longrightarrow$  `SID`

The first mapping is used for protection domain transfers. It maps the current security identifier of a thread and the security identifier of the procedure 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 actual procedure, and it restores the original security identifier upon completion of the procedure.

The second 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 requesting the access check.

The third 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 the newly created object. The enforcement manager associates newly created objects 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.

Both variations of the third mapping provide relatively coarse-grained control over the security identifiers associated with objects, and are clearly insufficient for some services. For example, a file server typically executes threads within its own protection domain but may need to associate different files with different security identifiers. Thus, to support trusted services that provide finer-grained control over the security identifiers associated with objects, the enforcement manager provides an interface through which a trusted security service can change the security identifier of an object. In the example of a file server, this interface could be used to map files to security identifiers similar to the name-based security attributes in domain and type enforcement [3, 2].

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 to link and initialize extensions (as discussed above), and for threads that are created when a user logs into the system. In the latter 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.

Depending on the complexity of the security policy implemented by the security policy manager, lookup operations for the three mappings may incur a relatively high performance overhead. Consequently, 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 also 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 multi-level policies [5, 12, 7], all mappings can be cached indefinitely since they never change. For access matrix based policies [22], mappings generally can be cached. But, if the access matrix is changed, the security policy manager must remove the corresponding entries from the mediation cache. And, for policies that depend on the mediation history, mappings may not be cached at all.

## 4 Implementation

We have implemented our access control mechanism in the SPIN extensible operating system [6]. 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 patching object jump tables either statically or dynamically). 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<sup>1</sup> and for easy transfer to other systems. Third, the implementation should be fast to impose as little performance overhead as possible.

In SPIN, a statically linked core provides most basic services, including hardware support, the Modula-3 runtime [42, 21], the linker/loader [41], threads, and the event dispatcher [35]. All other services, including networking and file system support, are provided by dynamically linked extensions. We have implemented the basic abstractions of our access control mechanism, such as security identifiers and access modes, as well as the enforcement manager as part of this 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 on the core services, including the linker/loader as described in Sect. 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 in SPIN need not be written in Modula-3, are not guaranteed to be type-safe, and thus are generally untrusted. They can not access *any* kernel-level objects directly, but only through a narrowly defined system call interface, which automatically subjects them to our access control mechanism.

---

<sup>1</sup> We have not validated the implementation. However, a critical characteristic for any security mechanism is that it be small and well-structured [38].

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. The implementation uses the Modula-3 runtime to determine the types and operations of an extension, and the event dispatcher [35] to inject access control operations into the system. It 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, pre-defined permissions in addition to a list of permission objects. The simple permissions provide 64 permissions at a low overhead. The list of permission objects lets the security policy manager define additional permissions (where each permission object can represent several permissions) by subtyping from an abstract base class, at some performance cost.

The functionality of the enforcement manager is visible through two separate interfaces. One interface lets extensions discover the state of system security in the presence of security faults. The other interface, together with the interface to the security policy manager, defines the trusted protocol between the security policy and the enforcement manager. The interface to the security policy manager is also defined as part of the static core, but an implementation of this interface must be provided by a trusted extension outside the static core. The enforcement manager operates as described in Sect. 3. It uses the simplified third mapping for assigning objects to security identifiers (see Sect. 3.2), and thus provides a default security identifier for all objects within a protection domain.

On object creation, the standard Modula-3 allocator, through a call-back into the enforcement manager, stores this default security identifier in the header of the newly allocated object. At the same time, only some types in an extensible system require access control. For example, an auxiliary object that is only used within an extension and never passed outside will never require access control. Thus, to limit the memory overhead of allocating an additional word in each object header, the security manager can dynamically activate and deactivate object security for each Modula-3 type individually. Access checks on objects that are not associated with a security identifier simply fail.

Storing an object's security identifier in the object header considerably simplifies the mapping from objects to security identifiers as 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 to access 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 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 procedure or method, and pops the record off the stack when the thread returns from the procedure or method. Records are pre-allocated in a global pool to avoid dynamic memory allocation overhead, and are pushed and popped using atomic enqueue and dequeue operations to avoid the overhead of locking the global pool.

## 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 can not 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 provides control over which operations a subject can legally execute on an object, including control over which objects can be passed to and returned from an operation. 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 object 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 multi-threaded system, information can be passed through shared memory at any time, not just on procedure 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 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 clearly 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 externally visible types and operations of 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 [45, 43, 19, 37], 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 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 [25], proof-carrying code [33], as well as typed assembly language [31]. All of these efforts are complementary to our own.

## 6 Performance Evaluation

To determine the performance overhead of our implementation, we evaluate a set of micro-benchmarks 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 a DEC Alpha AXP 133 MHz 3000/400 workstation, which is rated at 74 SPECint 92. The machine has 64 MByte of memory, a 512 KByte unified external cache, and an HP C2247-300 1 GByte disk-drive. In summary, the micro-benchmarks 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 Micro-Benchmarks

To evaluate the performance overhead of access control operations in our access control mechanism, we execute seven micro-benchmarks. 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. Consequently, for the micro-benchmarks, we fix the necessary entries in the mediation cache of the enforcement manager (see Sect. 3.2). The benchmarks thus 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 Sect. 4).

**Table 1.** 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 1 shows the performance results for the seven micro-benchmarks. All numbers are in microseconds and the average of 1000 trials. To determine hot microprocessor cache performance, we execute one trial to pre-warm 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 2 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 [35]). 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 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 new objects onto the thread’s security identifier stack. After leaving the procedure, the enforcement manager pops the top 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 bench-

**Table 2.** 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	# Instr.
<i>Enter new protection domain</i>	
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
<i>Total number of instructions</i>	161
<i>Restore old protection domain</i>	
Pop security ID record	22
Remove exception frame	4
Overhead	4
<i>Total number of instructions</i>	30

marks 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 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. 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, backed by a simple, 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 Sect. 3.2. 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 [3, 2]) 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 the 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 originated.

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 5035 KByte of data. We run the benchmark without access control, as a baseline, as well as with access control, to measure the end-to-end overhead of our access control mechanism. For each measurement, we first perform 15 runs of the benchmark to pre-warm 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 (including 5.4 seconds idle time on the machine running the web server), and the difference between the two is in the noise. Trials with access control incur a total of 1573 access checks, on average 20 for each file.

The end-to-end performance experiments show that the overhead of access control operations is negligible for a web server workload. We extrapolate from this result that other applications will see a very small overhead for other real-world applications. To better quantify this overhead, we plan to conduct further experiments in the future that use more complex security policies and require finer-grained access control operations.

## 7 Related Work

A considerable body of literature focuses on system protection [22, 38] and appropriate security policies. Starting from multi-level security [5, 12, 7], which has become part of the U.S. Department of Defense's standard for trusted computer systems [13], much attention has been directed towards mapping non-military policies onto multi-level security [26, 24], defining alternative policies more suitable for commercial applications [8, 10, 9, 3, 2, 1], and expanding multi-level security to be more flexible and powerful [27, 32].

Based on the realization that no single security policy is appropriate for all environments, the DTOS effort [30, 34, 40, 39] 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 (security identifiers and permissions). However, as DTOS has been implemented on top of the Mach micro-kernel, it differs from our mechanism in that it relies on address-spaces for protection domains, resulting in a relatively high overhead for changing protection domains. Furthermore, as the DTOS effort does not separate security from functionality, it uses explicit access checks on pre-defined permissions for enforcing protection domains, making it impossible to change or remove access checks.

As reported in [40], adding explicit access checks to the micro-kernel presented a considerable challenge as it fixed part of the security policy within the system. Furthermore, as noted in [39], 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 parameters to the operation as well. Our access control mechanism avoids these limitations, as access control operations are dynamically specified and injected into the system, and as they are strictly more expressive.

Due to Java's [18, 25] popularity for providing executable content on the Internet, and prompted by a string of security breaches [11, 28] in early versions of the system, 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 is currently being extended [15, 16] to allow for multiple protection domains, provide fine-grained access control primitives, and support cryptographic protocols.

The basic technique for performing dynamic access checks in Java, called extended stack introspection, is described in [46]. With this technique, each extension is implicitly associated with a protection domain, and access checks essentially take the intersection of all protection domains represented on the current call-stack to determine if an operation is legal. While extended stack introspection is sufficiently expressive to provide fine-grained access control [47], it is closely tied to the stack-based execution model of Java. Furthermore, it relies on explicit access checks, and thus fails to separate functionality from protection. Finally, as access checks need to walk the entire call-stack, they can incur considerable performance overhead [17].

Hagimont and Ismail [20] describe an alternative design for access control in Java which provides for a separate description of security constraints through an extended interface definition language. In their design, security constraints are expressed as part of the interface specification for each extension, and result in the creation of proxy objects which provide only limited functionality to their clients. The design essentially provides a form of type hiding [46] at the granularity of entire methods, as the visibility of object methods is controlled by the security constraints.

In its ability to provide access control at the granularity of object methods, Hagimont and Ismail’s design is similar to CACL [36] which presents a general protection model for objects. At the same time, CACL offers a more complete model (which includes explicit representations of the owner of an object and its implementor) and a more efficient implementation (through object jump tables instead of proxy objects). The idea of using limited effective types to avoid repeated dynamic access checks that determine whether a subject can call on an object is complimentary to our design. We thus believe that it could be used to provide an efficient implementation of the enforcement manager in a pure object-oriented system.

## 8 Future Work

The access control mechanism described in this paper provides us with an ideal test-bed for future research on the security of extensible systems. Specifically, the policy-neutral and transparent enforcement manager, with its ability to arbitrarily inject protection domains and access checks into an extensible system, offers us considerable power and flexibility. We are particularly interested in three areas for future work: First, programmers and security administrators need to be able to specify security constraints for the code they write and use. We thus plan to investigate appropriate specification languages that are both user-friendly (i.e., present a high-level of abstraction) and sufficiently powerful to conveniently express detailed security policies (i.e., provide enough flexibility). Second, as extensions often execute in networked environments, a protocol for the secure expression and transfer of credentials is required. We thus intend to examine distributed authentication and authorization protocols, such as those described in [23, 4, 14], in the context of extensible systems. Finally, as illustrated by the micro-benchmarks in Sect. 6, the access control operations show a relatively high overhead when compared to a simple procedure invocation. We thus plan to explore aggressive optimizations that avoid dynamic access control operations whenever possible.

## 9 Conclusions

The access control mechanism for extensible systems described in this paper breaks up access control into a policy-neutral enforcement manager and a security policy manager, and is transparent to extensions in the absence of security violations. It structures the system into protection domains through protection domain transfers, enforces these protection domains through access control checks, and provides a trace of system operations through auditing. It works by inspecting extensions for their types and operations to determine what abstractions require protection, and by redirecting procedure or method invocations to inject access control operations into the system. The access control mechanism is based on a simple, yet powerful protocol by which the security policy and the

enforcement manager interact, and by which, if necessary, extensions are notified of security-relevant events.

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. 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

The research presented in this paper was sponsored by the Defense Advanced Research Projects Agency, the National Science Foundation and by an equipment grant from Digital Equipment Corporation. Grimm was partially supported by fellowships from the Microsoft Corporation and the IBM Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship and an Office of Naval Research Young Investigator Award.

We thank Dennis Hollingworth and Timothy Redmond at Trusted Information Systems 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. Stefan Savage, Stephen Smalley, Ray Spencer, Amin Vahdat, and the anonymous reviewers for this volume provided valuable feedback on earlier versions of this paper.

## References

- [1] L. Badger, K. A. Oostendorp, W. G. Morrison, K. M. Walker, C. D. Vance, D. L. Sherman, and D. F. Sterne. DTE Firewalls—Initial Measurement and Evaluation Report. Technical Report 0632R, Trusted Information Systems, March 1997.
- [2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haight. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 127–140, Salt Lake City, Utah, June 1995.
- [3] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haight. Practical Domain and Type Enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, Oakland, California, May 1995.
- [4] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS Wide Area Security Architecture. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.
- [5] D. E. Bell and L. J. La Padula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford, Massachusetts, March 1976. Also ADA023588, National Technical Information Service.

- [6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.
- [7] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153 Rev. 1, The MITRE Corporation, Bedford, Massachusetts, April 1977. Also ADA039324, National Technical Information Service.
- [8] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 17th National Computer Security Conference*, pages 18–27, Gaithersburg, Maryland, 1985.
- [9] D. F. C. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, California, May 1989.
- [10] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, California, April 1987.
- [11] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, California, May 1996.
- [12] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.
- [13] Department of Defense Computer Security Center. Department of Defense Trusted Computer System Evaluation Criteria, December 1985. Department of Defense Standard DoD 5200.28-STD.
- [14] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI Certificate Theory. Technical Report draft-ietf-spki-cert-theory-04.txt, Internet Engineering Task Force, November 1998.
- [15] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, May/June 1997.
- [16] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. 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*, pages 103–112, Monterey, California, December 1997.
- [17] L. Gong and R. Schemers. Implementing Protection Domains the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998.
- [18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.
- [19] S. L. Graham, S. Lucco, and R. Wahbe. Adaptable Binary Programs. In *Proceedings of the 1995 USENIX Technical Conference*, pages 315–325, New Orleans, Louisiana, January 1995.
- [20] D. Hagimont and L. Ismail. A Protection Scheme for Mobile Agents on Java. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Budapest, Hungary, September 1997.
- [21] W. C. Hsieh, M. E. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B. N. Bershad. Language Support for Extensible Operating Systems. In *Proceedings of the Workshop on Compiler Support for System Software*, pages 127–133, Tucson, Arizona, February 1996.
- [22] B. W. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton, New Jersey, March 1971. Reprinted in *Operating Systems Review*, 8(1):18–24, January 1974.

- [23] B. W. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
- [24] T. M. P. Lee. Using Mandatory Integrity to Enforce “Commercial” Security. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 140–146, Oakland, California, April 1988.
- [25] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.
- [26] S. B. Lipner. Non-Discretionary Controls for Commercial Applications. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 2–10, Oakland, California, April 1982.
- [27] C. J. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the Pale of MAC and DAC—Defining New Forms of Access Control. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 190–200, Oakland, California, May 1990.
- [28] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes and Antidotes*. Wiley Computer Publishing, John Wiley & Sons, Inc., New York, New York, 1997.
- [29] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.
- [30] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, Salt Lake City, Utah, June 1995.
- [31] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, San Diego, California, January 1998.
- [32] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 129–142, Saint-Malo, France, October 1997.
- [33] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Washington, October 1996.
- [34] D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and Using a “Policy Neutral” Access Control Policy. In *Proceedings of the New Security Paradigms Workshop*, September 1996.
- [35] P. Pardyak and B. N. Bershad. Dynamic Binding for an Extensible System. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 201–212, Seattle, Washington, October 1996.
- [36] J. Richardson, P. Schwarz, and L.-F. Cabrera. CACL: Efficient Fine-Grained Protection for Objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '92*, pages 263–275, Vancouver, Canada, October 1992.
- [37] T. Romer, G. Voelker, D. Lee, A. Woman, W. Wong, H. Levy, B. N. Bershad, and B. Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–8, Seattle, Washington, August 1997.
- [38] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
- [39] Secure Computing Corporation. DTOS General System Security and Assurability Assessment Report. Technical Report DTOS CDRL A011, Secure Computing

- Corporation, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.
- [40] Secure Computing Corporation. DTOS Lessons Learned Report. Technical Report DTOS CDRL A008, Secure Computing Corporation, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.
  - [41] E. G. Sirer, M. Fiuczynski, P. Pardyak, and B. N. Bershad. Safe Dynamic Linking in an Extensible Operating System. In *Proceedings of the Workshop on Compiler Support for System Software*, pages 134–140, Tucson, Arizona, February 1996.
  - [42] E. G. Sirer, S. Savage, P. Pardyak, G. P. DeFouw, M. A. Alapat, and B. N. Bershad. Writing an Operating System with Modula-3. In *Proceedings of the Workshop on Compiler Support for System Software*, pages 141–148, Tucson, Arizona, February 1996.
  - [43] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, Florida, June 1994.
  - [44] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 25(1):80–86, January 1997.
  - [45] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, Ashville, North Carolina, December 1993.
  - [46] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, October 1997.
  - [47] D. S. Wallach and E. W. Felten. Understanding Java Stack Inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, California, May 1998.
  - [48] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, pages 105–117, Anaheim, California, January 1997.