Access Control in Extensible Systems*

Robert Grimm Brian N. Bershad
{rgrimm, bershad}@cs.washington.edu
Dept. of Computer Science and Engineering
University of Washington
Seattle, WA 98195, U.S.A.

UW-CSE-97-11-01

Abstract

The recent trend towards dynamically extensible systems holds the promise of more powerful and flexible systems. At the same time, the impact of extensibility on overall system security and, specifically, access control is still ill understood, and protection mechanisms in these extensible systems are rudimentary at best. In this paper, we identify the structure of extensible systems as it relates to system security, and present an access control mechanism that is user-friendly and complete. The mechanism, by using ideas first introduced by the security community, offers mandatory access control which can be used to enforce a given security policy. Additional discretionary access control allows users to express their own fine-grained access constraints.

We introduce a new access mode, called the *extend* access mode, in addition to the familiar *execute* access mode, to reflect how extensions interact. Furthermore, in a departure from work in the security community, we treat both extensions and threads of control as subjects, i.e., as active entities, in order to correctly capture their interaction in an extensible system. We present the design of the access control mechanism and define a formal model. We describe an implementation of the access control mechanism in the SPIN extensible operating system, which allows us to evaluate its performance and to explore optimizations that reduce the overhead of access control. The measured end-to-end overhead of access control in our system is less than 2%.

*This research was sponsored by the Advanced Research Projects Agency, the National Science Foundation and by an equipment grant from Digital Equipment Corporation. Grimm was partially supported by a fellowship from the Microsoft Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship and an Office of Naval Research Young Investigator Award.

1 Introduction

Dynamically extensible systems can provide applications, and ultimately users, with new and better functionality as well as better performance. Motivated by this promise of more power and more flexibility, several projects are pursuing system designs that are extensible by their very design: SPIN [6] and VINO [37] address extensibility in the context of operating systems. Inferno [26] focuses on extensibility for distributed services. The Java system [22, 19, 24] provides an infrastructure for executable content on the world-wide web. In addition, it is being proposed as the substrate for extensible operating systems on network computers. Finally, Juice [16], which utilizes "slim binary" technology [17, 21] originally developed for the Oberon system [41], provides a faster and leaner alternative to Java. At the same time, the impact of extensibility on overall system security and specifically on access control is still ill understood. And, the protection mechanisms in these extensible systems are rudimentary at best, as illustrated by the continuous string of security breaches in the Java system [12, 29].

Exactly what constitutes a secure system is defined by a security policy. Security policies usually reflect the requirements of an organization to control unauthorized access to and dissemination of data as well as the integrity of data, and are thus external to the system. At the core of any security policy are restrictions on *who* can access *what* system resources in *what way*. Thus, to mechanically enforce a security policy, an access control mechanism that can express and enforce these restrictions is necessary. Such an access control mechanism needs to address four criteria in order to meet the requirements of users, administrators and system programmers.

First, the access control mechanism must be complete; i.e., it must be possible to enforce a system wide security policy while also allowing users to introduce their own, fine-grained access restrictions. Second, it must be user-friendly, so that users understand how it

impacts their work and use it to its potential. Third, a formal model of the access control mechanism is required. Such a model enables the formal reasoning about security in a given system and is necessary to verify a given implementation, thus providing assurance that it works correctly. Fourth, the access control mechanism must allow for an efficient implementation. An efficient implementation is especially important for extensible systems which support fine-grained composition primarily for reasons of performance.

The access control mechanism presented in this paper draws on ideas explored by the security community and provides non-discretionary (mandatory) access control based on *domain and type enforcement* (DTE) [3, 4, 2]. DTE associates all entities in a system with a label that represents an entity's privileges and access constraints. The label for subjects (i.e., the active entities in a system) is called a *domain*, and the label for objects (i.e., the passive entities) is called a *type*. DTE defines a valid access mode for all practical pairs of domains and types, which results in an access matrix. For example, all trusted users in a given system could be labeled with the `TrustedUser` domain and all transaction data stored in the system with the `Transaction` type. The access matrix could then be set to only let users in the `TrustedUser` domain access data of the `Transaction` type, which ensures that no unauthorized users can access the (presumably) mission-critical transaction data.

Mandatory access control policies allow an organization to define its security terms, and provide no path by which an individual can circumvent security. As security is generally first an organizational, and then an individual, concern, mandatory access control policies have become increasingly important [11, 9, 4]. Domain and type enforcement is mandatory in the sense that it is imposed on all relevant system operations and can only be changed by the security administrator. It is thus an appropriate means to mechanically enforce the security policy of a given environment

DTE's constraints, however, can be expected to be relatively coarse-grained, and users may want to express their own, additional access constraints. Consequently, our access control mechanism also supports discretionary access control based on users, groups and access control lists which is a proven feature of mainstream file and operating systems (both local and distributed) and familiar to users.

1.1 Contributions

We present an access control mechanism for extensible systems that is both user-friendly—the DTE access matrix is a concise and explicit representation of access constraints in a given system—and complete—mandatory access control mechanically enforces a given security policy while discretionary access control allows users to express additional, fine-grained access constraints. Our particular contributions are as follows:

- We introduce the *extend* access mode in addition to the familiar execute access mode to model the interaction of services in an extensible system.
- We treat both extensions and threads in an extensible system as active entities, or subjects, to correctly capture access constraints between the two.
- We define a formal model for the access control mechanism.
- We present an implementation in the context of the SPIN extensible operating system. We introduce optimizations that minimize the performance overhead of access control and use the formal model to ensure that the optimizations preserve the security of the system.

1.2 Rest of This Paper

The remainder of this paper is structured as follows: Section 2 motivates our research by identifying the structure of extensible systems as it relates to access control, by describing the state of affairs in current extensible systems, and by reviewing related work. Section 3 describes the design of our access control mechanism. Section 4 discusses our implementation, and section 5 presents a performance evaluation of this implementation. Finally, section 6 concludes this paper. Appendix A contains the formal model of our access control mechanism.

The discussion in this paper focuses on just one aspect of overall security in extensible systems, namely access control. Other important issues, such as the authentication of extensions and users or the auditing of security relevant system events are only touched upon or not discussed at all. These issues are orthogonal to access control, and we believe that a complete and user-friendly 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.

2 Motivation

The basic innovation of extensible systems for the purposes of this paper is that units of code, which we call "extensions," can be dynamically loaded and linked into the base system and consequently become an *integral* part of the base system. Different systems use different terms for the extension mechanism and the

extensions themselves, such as "applets" in Java or "grafts" in VINO. But the basic functionality and structure of these extensible systems is sufficiently similar to rely on two basic mechanisms to tightly compose extensions.

First, extensions can call other parts of the system to build on already supported functionality. Second, extensions can extend the functionality of the base system by adding new services which are then invoked through already existing interfaces (which is sometimes referred to as specialization). Both mechanisms are usually provided by a central facility, by either building on programming language support (for example, the use of inheritance in Java or VINO), or a dynamic dispatch model (for example, the event-dispatch model in SPIN [32]).

While seemingly similar, the two mechanisms represent different semantics: In the first case, an extension invokes other services, while, in the second case, an extension is invoked by another service. The combination of both mechanisms provides extensions with considerable flexibility and power (which is one major argument for using extensible systems). For example, an extension can be used to provide a new data transfer protocol that is not supported by the original system. To provide this new network protocol, the implementing extension uses existing services (e.g., the datagram protocol) and builds upon them. At the same time, to utilize the new data transfer protocol, a user invokes the existing, general network interfaces that have been extended (or specialized) by the extension to also handle the new type of data transfer protocol.

In extensible systems, extensions are tightly integrated within the same address space and effectively form *one* system. The basic safety of the system is ensured through either programming language support (using type-safe programming languages such as the Java programming language in the Java system, Modula-3 in SPIN, or Oberon in Juice) or software fault isolation (e.g., in VINO). This tight integration of extensions and the low overhead of the safety mechanism are the fundamental reason for the good performance of extensible systems. The time to cross the boundaries between extensions thus approaches that of a procedure call (as opposed to closed systems which rely on "hard" boundaries, such as address spaces, that have a high crossing overhead), and the overhead of security checking becomes critical.

2.1 The State of Affairs

Access control in existing extensible systems is rudimentary at best. The current Java security model [18, 29] distinguishes between trusted extensions (code stored on the local file system), which have access to the full functionality of the Java system, and untrusted extensions (all remote code). Untrusted extensions are placed into a so-called "sandbox" that limits extensions from using some system services, such as accessing the local file system. Ideally, it would also isolate extensions from each other, but see [29] for a counter example.

Future versions of Java will provide authenticated extensions with a finer granularity of access control, such as allowing some extensions to access some files. However, no clear and flexible access control model, detailing how this finer grain of access control will be provided, has been presented. Furthermore, the security of the Java system, instead of relying on one central facility to enforce security (which is good design practice for secure systems [33]), relies on three facilities, or "prongs" [29]. This design makes it difficult to reason about the security of Java, and a design or implementation error in any one of the three prongs can break the security of the Java system, as has been repeatedly demonstrated [12, 29].

In SPIN, system services are partitioned into "domains" [38] (which are a separate concept from the domains used in domain and type enforcement), where each domain is a collection of Modula-3 interfaces. An extension is linked against one or more domains, and can only access and extend the system services that are in those domains. Other system services are inaccessible to an extension. Domains provide a useful mechanism to avoid a flat global name-space, to group several interfaces according to, for example, the functionality they provide, and to control the visibility of interfaces. However, in earlier implementations, an extension could either call on and extend all interfaces in all domains it had been linked against, or access control was ad hoc, with each extension responsible for implementing its own access checks whenever it was being extended.

Little or no information is available on system security for other extensible systems: VINO distinguishes between regular and privileged users, and uses dynamic privilege checks before accessing sensitive data [36]. Inferno uses encryption for the mutual authentication of communicating parties and their messages [27]. No information is available on security in Juice. While these systems ensure the basic safety of the system by relying on either programming language support or software fault isolation, no security model and specifically no access control model is discussed in the publicly available literature. In the absence of other information, we thus assume that access control is still an open issue that needs to be addressed in these systems as well.

2.2 Related Work

Discretionary access control based on users, groups, access modes and access control lists is a familiar and

flexible feature of mainstream file and operating systems such as Unix [30], the Andrew File System [34] and Windows NT [40]. However, it relies on *all* users to enforce a given security policy, since the owner or creator of a system resource also determines the access control list for that resource. Furthermore, it can be easily subverted and is thus not complete: an often cited example is the so-called "Trojan horse" attack [1, 11, 10] where an application appears legitimate but in fact also carries out some illicit action.

Mandatory access control as a method of enforcing a given security policy has been developed within the security establishment of the United States. Most work in this context is based on a lattice of subject and object labels that implicitly defines legal types of operations [5, 13, 7] and that has become part of the Department of Defense's standard for trusted computer systems [14]. For example, to enforce the Department of Defense's security classifications, one would create four labels `TopSecret`, `Secret`, `Confidential`, and `Unclassified` (ordered as given with `TopSecret` being the top-most label) and assign them to all subjects and objects. Subjects with a given clearance label can read all objects associated with the same or a lower label, and they can write all objects with the same or a higher label. The lattice model, while precise, is not very intuitive or user-friendly since access modes are implicit. It is not very flexible since the lattice structure dictates valid access modes. And, it only supports two different access modes, namely read and write [25, 11, 23, 9, 28].

The idea of domain and type enforcement as a more flexible and user-friendly alternative to the lattice model is first developed by Boebert and Kain [8]. They introduce the domain and type labels that represent an entity's privileges and access constraints and the access matrix that explicitly and concisely lists all legal access modes. They also introduce the notion of changing a subject's label (on invocation of a program) to provide for a controlled way of changing privilege. This model is later expanded by Badger et al. [3, 4, 2], who introduce a high-level language, called domain and type enforcement language, to express security policies, and who also apply DTE on the Unix operating system. Domain and type enforcement as a mandatory access control mechanism is both more flexible and more user-friendly than the lattice model, since access modes are explicit, subjects are grouped into domains, objects are grouped into types and a high-level language is used to express security policies.

3 Access Control Mechanism

Restrictions on who can access what resources in what way involve the subject that wants to carry out some operation (the "who" part), the subject or object on which the operation is to be carried out (the "what resources" part) and the operation itself (the "what way" part). Since, in any system, there can be an unbounded number of subjects, objects and operations, it becomes necessary to abstract over the subjects, objects and operations to mechanically enforce these restrictions. The key idea behind access control is thus to manage execution "contexts," where a given context determines what rights a particular subject has at a particular point of execution.

Domain and type enforcement is one way to manage such execution contexts. It associates all entities in a system (i.e., all subjects and objects) with a label representing an entity's privileges and access constraints. The label for subjects is called a domain, and the label for objects is called a type. DTE also defines a global access matrix that lists relevant access information (specifying the legal types of operations) for all pairs of subject labels and pairs of a subject and an object label. Given the label of the particular subject intending to carry out some operation and the label of the particular subject or object on which the operation is to be carried out, the DTE access matrix can now be used to mechanically determine whether the particular type of operation is legal or not.

DTE in Extensible Systems

In the original DTE model, subjects are processes executing in their own address space. However, since extensions execute within the same address space, with no clear separation between extensions, the notion of a process can not be maintained for extensible systems. In a clear departure from this model (and other work in the security community), we treat both extensions and threads as subjects. Extensions are written by a human programmer and some of their code may be executed by default (for example, to initialize the extension). Threads are active entities and, furthermore, are often independent of the extension whose code they execute (for example, when an extension calls another). Both extensions and threads are thus considered active entities, that is subjects, and are associated with a DTE domain.

Since a domain represents a subject's privileges, a particular subject can only be associated with one domain, where the choice of domain depends on the security policy and the user behind the subject (i.e., the programmer of the extension or the user for whom the thread is executing). We assume the authentication of subjects through some mechanism, such as digital signatures for extensions or passwords for users (who initiate threads through some invocation mechanism).

The DTE access matrix maps pairs of subject labels into an access mode, representing legal types of operations, and a so-called *target domain*, allowing for a

controlled change of privilege for a subject. The relevant access modes are *execute* and *extend*, and they represent the two ways extensions interact with each other. We introduce the extend access mode to capture the fact that, in an extensible system, subjects can extend existing interfaces; and we use the familiar execute access mode for a subject calling an interface. The access mode for pairs of DTE domains is used for access control at link-time when one extension wants to link against another extension (it needs to have either execute or extend or both permissions depending on the extension) and at run-time when a thread of control wants to execute code of an extension (it needs to have execute permission).

The target domain for pairs of DTE domains is used at run-time when a thread of control calls an extension, and the thread's domain is changed to the target domain for the duration of the call (after which the original DTE domain of the thread is restored). The intuition behind the use of target domains is that it allows for a controlled change of privilege for a given thread when executing a particular extension, comparable to `setgid` and `setuid` programs in Unix [30]. Since, in extensible systems, both the ability to execute code and to extend an interface are provided by a central mechanism, access control for extensions and threads is enforced by this central mechanism.

In DTE, objects are labeled with a type, and the access matrix maps pairs of a DTE domain and a DTE type into an access mode, representing legal types of operations. The extension that provides an object's abstraction needs to enforce access restrictions on its objects and explicitly uses the DTE access matrix, together with the subject's domain and the object's type, to determine whether a given type of operation is legal. Since, in an extensible system, not all objects and their semantics can be known a priori, the meaning of particular access modes (besides the extend and execute permissions used for extensions and threads) is left unspecified. Access control for objects is thus effectively delegated to the individual extensions. Certainly, a system-wide convention for common access modes should be established (e.g., defining read and write permissions). Furthermore, the fundamental abstractions of a system, such as threads, memory or the naming service, should be provided by extensions that can be trusted to always enforce the access constraints of domain and type enforcement (otherwise, a malicious extension could subvert the security of the system).

Fine-Grained Access Control

Domain and type enforcement is used to enforce a security policy on all relevant system operations. As such, its access constraints can be expected to be rather coarse-grained. Furthermore, its access constraints are imposed on all users and can only be changed by the security administrator. To give users the possibility of introducing their own, fine-grained access constraints, our access control mechanism also supports discretionary access control based on users, groups and access control lists (ACLs). Discretionary access control is a familiar feature of traditional file and operating systems [30, 34, 40] and is supported by associating subjects with a user and objects with an access control list. Given mandatory access control based on DTE and discretionary access control based on ACLs, an operation is legal if and only if *both* grant permission to carry it out.

3.1 An Example

This section illustrates the use of domain and type enforcement. It demonstrates the use of the DTE access matrix to mechanically enforce a given security policy and to express both data integrity and information flow constraints. Access control lists can be used to further restrict access but, since they are not relevant as far as the given security policy is concerned, have been omitted from this example.

Assume an extensible system that uses a storage manager to provide some form of file abstraction. The system also provides a transaction manager that extends the storage manager's interfaces and builds on top of the storage manager's file abstraction (so that, for example, the storage manager's file system maintenance utilities can be used for both regular files and transaction data). Both the storage manager and the transaction manager associate access control lists and DTE types with stored data and use *read* and *write* access modes to control who can read and modify data.

Furthermore, assume a security policy that separates the system's users into two classes, namely untrusted users that can use the storage manager, but not the transaction manager, and trusted users that are allowed to use the transaction manager. One possible motivation for this restriction could be that mission-critical data is stored in transactions. The transaction manager as a mission-critical resource in the system should thus only be usable by trusted users.

Both classes of users can extend the system with their own extensions, and the extensions of untrusted users are allowed to utilize the extensions of trusted users. Finally, to ensure the consistency of transactions, transaction data must always be accessed and modified through the transaction manager (the storage manager's file system maintenance utilities are trusted to not violate the consistency of transaction data and can thus access that data as well). The basic structure of this system and its security policy is illustrated in figure 1.

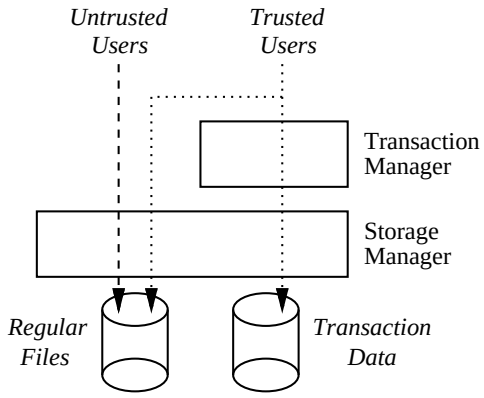

Figure 1: Overview of the example system and security policy. The storage manager provides a basic file abstraction, and the transaction manager builds on top of it to provide transactions. Threads executing for trusted users can access both regular files and transaction data, and threads executing for untrusted users can only access regular files. Transaction data can only be accessed through the transaction manager but not through the storage manager directly.

To translate this security policy into domain and type enforcement, we start by defining four DTE domains, `SM` for the storage manager, `TM` for the transaction manager, `TU` for trusted users, and `UU` for untrusted users. Note that DTE domains are associated with threads as well as with extensions. Consequently, a thread executing for a trusted user is associated with the `TU` domain and, if the user loads an extension into the system, that extension is also associated with the `TU` domain. Next, we define two DTE types representing the two different kinds of objects in the system, namely `S` for regular files and `T` for transaction data. Finally, we need to define the access modes and target domains for all pairs of domains and the access modes for all pairs of domains and types, resulting in a DTE access matrix.

To illustrate the definition of the DTE access matrix entries, we follow the right-most path in figure 1. The security policy requires that only trusted users can call on the transaction manager, and consequently the `TU` domain needs to have execute access to the `TM` domain. The transaction manager builds on and extends the storage manager, so the `TM` domain needs both extend and execute access to the `SM` domain. The transaction manager reads and writes transaction data for trusted users (thus ensuring its integrity), so the `TM` domain but not the `TU` domain has access to data of type `T`.

At the same time, threads executing for trusted users should have access to transaction data, as long as they access it through the transaction manager. But, the `TU` domain does *not* have direct access to data of type `T`. The solution is to provide for a controlled change of privilege for trusted user threads that call the transaction manager. Consequently, the target domain for calls from the `TU` domain into the `TM` domain is the `TM` domain. Note that this is the only situation where a change of privilege is necessary. All other target domain entries in the access matrix preserve the domain associated with a thread. This process is then repeated for all other paths in figure 1.

	SM	TM	TU	UU
SM	--ex SM	--ex TM	---x TU	---x UU
TM		--ex TM	---x TM	
TU			--ex TU	---x UU
UU				--ex UU
S	rw--		rw--	rw--
T	rw--	rw--		

Table 1: DTE access matrix for the original example, representing a policy that ensures data integrity. Column headings are domains, and row headings are domains as well as types. Matrix entries for pairs of domains are the access mode (top) and the target domain (bottom). Matrix entries for pairs of a domain and type are the access mode. Table entries representing no permissions are left blank, and table entries for a domain to itself always contain execute and extend permissions and the domain itself as target domain. Domains are `SM` for storage manager, `TM` for transaction manager, `TU` for trusted users and `UU` for untrusted users. Types are `S` for regular files and `T` for transaction data. Access modes are `r` for read, `w` for write, `e` for extend and `x` for execute.

Although the policy sounds complex, domain and type enforcement admits a concise representation as an access matrix. The complete DTE access matrix for our example is shown in table 1. The columns represent domains for a subject that wants to carry out some operation, and the rows represent the types for an object and the domains for a subject on which the operation is to be carried out. To find out what types of operations trusted users can carry out, for example, inspect the column labeled `TU`. The first row shows that trusted users can call the storage manager (the `TU` domain has execute rights to the `SM` domain). The second row shows that trusted users can call the transaction manager as well, and, when calling code in the `TM` domain, threads executing for trusted users are re-associated with the `TM` domain. The `TU` domain has execute and extend rights to itself (third row), and, finally, threads in the `TU` domain can read and write regular files (the `TU` domain has read and write rights to type `S`).

While the above security policy ensures the integrity

	SM	TM	TU	UU
SM	--ex SM	--ex TM	---x TU	---x UU
TM		--ex TM	---x TM	
TU			--ex TU	
UU				--ex UU
S	rw--		rw--	
T	rw--	rw--		
U	rw--		r---	rw--

Table 2: DTE access matrix for the stricter security policy which ensures data integrity and prevents unauthorized dissemination. It is structured as table 1 except that type `S` is used only for regular files accessible to trusted users, and type `U` is introduced for regular files accessible to untrusted users (but readable to trusted users).

of the transaction data, untrusted users, with the collaboration of a trusted user, can still access transaction data, for example through a regular file. A stricter security policy may restrict all information flow to only allow trusted users access to transaction data. This can be achieved by introducing a third DTE type, say `U` for untrusted data, that is associated with the regular files of untrusted users. Untrusted users can read and write such files, but trusted users can only read them and not write them. Furthermore, regular files of type `S` (which before were used by all users) must only be accessible by trusted users. The resulting DTE access matrix ensures the integrity of the transaction data and, at the same time, prevents unauthorized dissemination of data to untrusted users. It is shown in table 2, which has the same basic structure as table 1.

3.2 Summary

In this section, we have addressed the first two criteria for an access control mechanism (as introduced in section 1) and have described an access control mechanism that is both user-friendly and complete. To address the third criterion, we present the detailed and formal description of the full access control mechanism, which encompasses both mandatory and discretionary access control, in appendix A. The following two sections describe our implementation within the SPIN extensible operating system, thus addressing the fourth and final criterion.

4 Implementation

We have implemented our access control mechanism in the SPIN extensible operating system developed at the University of Washington. 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 dynamic loading and linking of extensions, for multiple concurrent threads of execution, and for overriding existing interfaces. Consequently, our access control mechanism could be implemented in other extensible systems that provide these three services, such as Java [19] or VINO [37].

Our implementation is guided by three constraints. First, it has to correctly enforce the security policy of a given environment. Second, it has to be simple, well-structured and break down into separate interfaces for the individual abstractions to allow for validation[†] and for easy transfer to other systems. Third, the implementation should be fast to impose as little performance overhead as possible.

4.1 Structure and Interfaces

In SPIN, a statically linked core provides the most basic services, including hardware support, the Modula-3 runtime [39, 20], the linker/loader [38], threads and the event dispatcher [32]. All other services, including networking and file system support, are provided by dynamically linked extensions. Services in the static core are trusted and, if they misbehave, can undermine the security of the system (and also crash the entire system). Dynamically linked extensions are not trusted and their access to other extensions must be carefully controlled. Thus, to correctly enforce a security policy on the system, the basic security services must be part of the trusted static core.

We added support for our access control mechanism to the static core. The implementation consists of 900 lines of well-documented Modula-3 interfaces and of 2200 lines of Modula-3 code. It includes access modes (the `AccessMode` interface), users and groups (the `UsersGroups` and `UsersGroupsPrivate` interfaces), access control lists (the `ACL` interface) and domain and type enforcement (the `DTE` and `DTEPrivate` interfaces). We also associated threads with a user identifier (for discretionary access control and auditing) and a stack of DTE domains (for mandatory access control) which can be accessed but not changed through the `SecurityContext` interface (the user identity and the original DTE domain are established at login-time, and the DTE domain may be changed by call-time access control). Furthermore, we changed the linker/loader to enforce access control on extensions at link-time and to set up call-time access control on threads and extensions.

[†] We have not validated the implementation. However, a critical characteristic of any security mechanism is that it be small and well-structured [33].

The `AccessMode` interface provides an immutable access mode abstraction. Each access mode consists of a set of simple, pre-defined permissions and a list of permission objects. The simple permissions include the execute and extend permissions required for the access control mechanism itself, but also common permissions such as read and write. The list of permission objects allows extensions to define their own additional permissions by subtyping from an abstract base class. The simple permissions are implemented as a bit-vector and thus add little performance overhead, while the list of permission objects gives extensions the flexibility to define their own permissions (at some performance cost). The `UsersGroups` interface supports user and group identifiers as well as an efficient mapping between users and groups; the `UsersGroupsPrivate` interface supports the definition of new users and groups as well as the addition of users to groups. The `ACL` interface provides an access control list abstraction that consists of both positive and negative rights.

The `DTE` interface provides the domain and type labels for domain and type enforcement and lets extensions query the DTE access matrix; the `DTEPrivate` interface supports the definition of new DTE domains and DTE types as well as the modification of the DTE access matrix. The DTE access matrix itself is implemented as a two-level hierarchy of hash tables. We expect matrix entries that represent access modes other than no access to be sparsely distributed (e.g., over 40% of all table entries in tables 1 and 2 represent no access rights), and the two-level hash table thus saves space over a two-dimensional array while still being reasonably efficient to access. The `SecurityContext` interface lets extensions discover for which user a thread is executing and with which DTE domain that thread is currently associated (which is the top of the domain stack maintained internally).

The `AccessMode`, `UsersGroups`, `ACL`, `DTE` and `SecurityContext` interfaces are executable, but not extendible, by all extensions. Extensions can use these interfaces to discover all security-relevant system state and to implement discretionary and mandatory access control on their own objects. The `UsersGroupsPrivate` and `DTEPrivate` interfaces allow for the modification of security-relevant state and should thus only be executable, but not extendible, by a few trusted extensions. Trusted extensions can then be used to implement the user interface to the basic access control mechanism and to support high-level abstractions such as DTEL, the domain and type enforcement language [3, 4].

4.2 Call-Time Access Control

Call-time access control determines whether a given thread can call the interfaces of a given extension. In other words, we need to execute a dynamic test that performs this access check before invoking the actual interface. Conceptually, this test determines the DTE domain of the thread (through the `SecurityContext` interface) and of the extension (which is passed to the test as a closure and is established in the linker/loader) and then does a lookup in the DTE access matrix for the pair of DTE domains. If the access mode contains the execute permission, the target domain is associated with the thread by pushing it onto the DTE domain stack, and the actual interface is invoked; if the access mode does not contain the execute permission, the interface is not invoked and an exception is raised. On completion of the call to the interface and before returning control to the call-site, the current DTE domain is popped off the domain stack and the old DTE domain is thus restored.

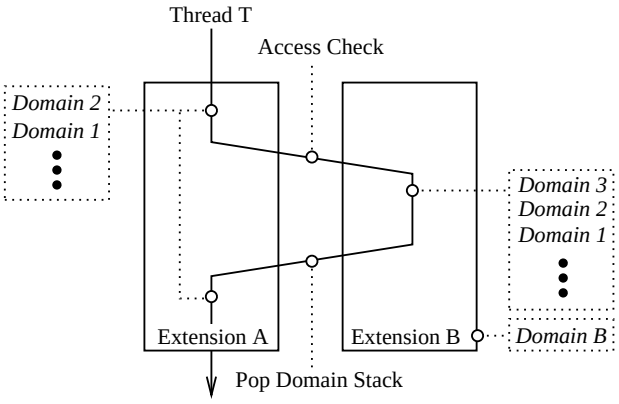

Figure 2: Illustration of call-time access control. Thread T calls an interface in extension B from within extension A. The access check uses T's current DTE domain, domain 2, and extension B's DTE domain, domain B, to find the relevant access mode and target domain in the DTE access matrix. If the access mode contains the execute permission, the target domain, domain 3, is pushed onto T's domain stack, and control transfers to extension B. On return, the top of T's domain stack is popped off.

Call-time access control is illustrated in figure 2. Thread T executes code in extension A and intends to call an interface provided by extension B. Before control transfers to extension B, a dynamic call-time access check is executed, which is based on the current DTE domain of thread T, domain 2, and the domain of extension B, domain B. If the corresponding access mode in the DTE access matrix includes the execute permission, the target domain, domain 3, is pushed onto T's domain stack, and control is transferred to extension B. Once control returns from extension B, the top of T's domain stack is popped.

In the actual implementation, we use an optimized version of the call-time access check. Since the DTE

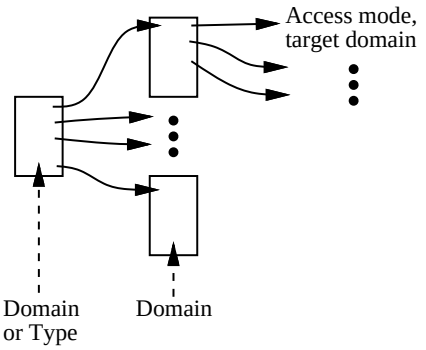

Figure 3: Implementation of the DTE access matrix as a two-level hierarchy of hash tables. The outer hash table maps the DTE domain or type of the subject or object on which the operation is to be carried out into another hash table. Each inner hash table maps the DTE domain of the subject that intends to carry out the operation into an access mode and target domain.

domain of the extension is known when the test is imposed on the interface (i.e., at link-time), one dynamic lookup in the two-level hierarchy of hash tables is unnecessary. We thus structure the hierarchy of hash tables so that the outer hash table maps DTE domains and types (associated with the subject or object on which an operation is to be carried out) into hash tables. These hash tables, in turn, map DTE domains (associated with the subject that intends to carry out the operation) into access modes and target domains. This structure of the DTE access matrix is illustrated in figure 3. So, instead of passing the DTE domain of the extension as a closure to the dynamic test, we execute the outer lookup at link-time and pass the inner hash table to the dynamic test.

Transparent Implementation

In SPIN, extensions interact through a central dispatcher [32] by raising events, which corresponds to calling an interface, and by handling events, which corresponds to extending an existing interface. The invocation mechanism for events is simply the procedure call. The dispatcher also support guards, which are imposed on a specific handler, and result handlers, which are associated with an event. The dispatcher first executes the guard for a handler, and only executes the handler if the guard evaluates to true. The result handler is executed after each regular handler for that event. We use guards to perform the call-time access test and target domain re-association, and result handlers to restore the old DTE domain. The necessary guards and result handlers are imposed at link-time and are thus transparent to extensions.

In our implementation, the linker performs link-time access control based on the DTE domains of extensions (we also support access control lists on extensions to allow for user-specified additional access constraints). If an extension passes link-time access control, i.e., it can be legally linked into the system, the linker determines from the DTE access matrix what form of call-time access control is necessary, and imposes necessary guards and result handlers.

Optimizations

Since call-time access checks and target domain re-associations need to be executed on the critical path, i.e., when control transfers from one extension to another, it is clearly desirable to avoid them whenever possible. Intuitively, if all threads that can legally attempt to call an interface are actually allowed to call that interface, no call-time access check is necessary. Similarly, if the target domain for all threads that can call an interface is the same domain as the domain currently associated with the thread, no target domain re-association is necessary.

For example, consider the storage manager discussed in section 3.1. The DTE access matrix in table 1 shows that all DTE domains have execute permission for the `SM` domain associated with the storage manager. Furthermore, the target domain for all DTE domains when calling the storage manager is the original domain. As a result, no call-time access checks or target domain re-associations are necessary for the storage manager.

As illustrated by this example, both conditions can be determined from the DTE access matrix. Consequently, when loading an extension into the system, we determine whether call-time access checks and target domain re-associations are actually necessary and only impose the necessary guards and result handlers on the interfaces of an extension. These optimizations are based on the formal model, guarantee the integrity of system, and are described in detail in appendix A.5.

5 Performance Evaluation

To determine the performance overhead of our implementation, we evaluate a set of micro-benchmarks that measure the performance of call-time access control. We also present end-to-end performance results for a transaction benchmark that uses the example setup and security policy described in section 3.1. 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 call-time access checks incur some latency on trivial operations, while the end-to-end experiment shows that the overall overhead of access control is minimal (less than 2%).

	Hot	Cold
Access test (Access)	2.8	6.0
Access test + domain push (Push)	3.1	7.2
Domain pop (Pop)	0.9	2.3
Dispatcher: Null procedure call	0.1	0.6
Dispatcher: Access + Null	2.8	6.2
Dispatcher: Push + Null + Pop	3.9	8.9

Table 3: Performance numbers for call-time access control. All numbers are the mean of 1000 trials in microseconds. *Hot* represents hot cache performance and *Cold* cold cache performance. The first three tests perform access control operations by themselves: *Access* performs an access control test, *Push* performs an access control test and a target domain re-association, and *Pop* pops the old domain off the domain stack. The last three tests perform a null procedure call through the dispatcher with additional access control operations as indicated, and represent the actual call path of the system.

5.1 Micro-Benchmarks

To evaluate the performance overhead of call-time access control, we execute six micro-benchmarks that break down the cost of the individual call-time access control operations. The first three micro-benchmarks perform the individual access control operations by themselves. The other three micro-benchmarks measure the total time for a null procedure call through the dispatcher with and without access control and represent the actual call path of the system.

The *Access* benchmark measures the performance of a call-time access check. The *Push* benchmark measures the performance of a target domain re-association in addition to the access check. And, the *Pop* benchmark measures the performance of restoring the original DTE domain of a thread. We do not measure the performance of a target domain re-association by itself, since the additional overhead of the access check compared with the partial DTE matrix lookup and the domain re-association is minimal (the access check is a simple bit-vector operation). The first dispatcher benchmark measures the the round-trip time for a null procedure call through the dispatcher. The second dispatcher benchmark adds an access control check over the first dispatcher benchmark. Finally, the third dispatcher benchmark further adds a target domain re-association and the restoration of the original DTE domain after completion of the null procedure call.

Table 3 shows the performance results for the six micro-benchmarks. All numbers are in microseconds and the average of 1000 trials. To determine hot cache performance, we execute one trial to pre-warm the 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 the cold cache performance, we measure the time before and after each trial separately, and flush both the instruction and data cache on each iteration.

The performance results show that call-time access control has noticeable overhead. Performing both access checks and DTE domain re-associations requires that a procedure is executed before the actual interface is invoked (for the check and to push the target domain) and after it has been invoked (to pop the target domain). Consequently, its overhead is higher than when performing the access check by itself, which only requires an additional procedure execution before the interface is invoked. The performance measurements underline the need for optimizations described in section 4.2, either to avoid target domain re-associations, or, preferably, to avoid call-time access control altogether.

5.2 End-to-End Performance

To evaluate the impact of the optimizations described in section 4.2, we present end-to-end performance results for the storage/transaction manager example in section 3.1. The storage manager used in our experiment provides an extent-based file system, and the structure of the transaction manager is similar to that of the Camelot system [15]. Both the storage and transaction manager are implemented as extensions while the benchmark runs as a user-space application. The benchmark itself is modeled after the TPC-A benchmark described in [35] and generates 100 transactions. For each transaction, it reads and then writes three 128 byte records and also writes a 64 byte record. The three 128 byte records represent a bank, a teller and an account, and the account for each transaction is chosen randomly out of 32768 possible accounts (and determines the teller and bank). The 64 byte record serves as an audit trail.

We run the benchmark without access control, as a baseline, and with access control, to measure the end-to-end overhead of access control. We use the DTE access matrix in table 1 for the experiments with access control. Since all DTE domains have execute access to the `SM` domain associated with the storage manager, no call-time access control is performed for the storage manager. The benchmark runs for a trusted user, and, when calling the transaction manager, we perform call-time access control and target domain re-association as required by the DTE access matrix.

The average latency for 10 trials of the benchmark without access control is 1.20 seconds, and for the benchmark with access control 1.18 seconds. Trials with access control incur 200 access checks and target domain re-associations, two for each transaction. Another four to seven access checks (depending on the number of page faults incurred while reading data)

would be necessary for each transaction, if the storage manager also required call-time access checks. The difference between trials without and with access control is in the noise for most trials. But, we consistently see one or two outliers for the benchmark with access control, which account for the 1.7% difference between the two versions of the benchmark.

Our call-time optimizations have eliminated most call-time access checks, and we see a minimal end-to-end overhead for access control. It is difficult to compare these results to those reported for DTE in Unix by Badger et al. [2] and for DTE in Mach by Minear [31], since their performance data is inconclusive. Badger et al. report a small performance improvement for some network operations (since DTE eliminates the need for re-authentication), but also a 13% worst-case overhead for FTP and a factor two slowdown for HTTP. The Mach version caches permissions since their lookup operation has high overhead, and performance greatly depends on the cache hit rate.

6 Conclusion

The access control mechanism for extensible systems described in this paper encompasses both mandatory access control, based on domain and type enforcement, and discretionary access control, based on access control lists. We have extended domain and type enforcement with the extend access mode, in addition to the familiar execute access mode, to correctly model the interaction of extensions. Furthermore, in a clear departure from previous work within the security community, we treat both extensions and threads in an extensible system as active entities, that is as subjects. The access control mechanism is user-friendly, complete and precisely specified in a formal model.

The implementation of our access control mechanism within the SPIN extensible operating system is simple, and, even though the latency of individual call-time access checks is 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

We thank Cynthia Irvine at the Naval Postgraduate School for sending us a "care package" of security papers; and we thank Timothy Redmond and Dennis Hollingworth at Trusted Information Systems for their input and discussion on domain and type enforcement. Marc Fiuczynski, Charles Garrett, Wilson Hsieh, Yasushi Saito, Stefan Savage, Emin Gün Sirer and especially Przemysław Pardyak at the University of Washington were most helpful with various implementation issues and the integration of the access control mechanism into SPIN. Przemysław Pardyak and Wilson Hsieh provided valuable feedback on earlier versions of this paper.

References

[1] Stanley R. Ames, Jr., Morrie Gasser, and Roger R. Schell. Security Kernel Design and Implementation: An Introduction. *Computer*, 16(7):14–22, July 1983.

[2] Lee Badger, Karen A. Oostendorp, Wayne G. Morrison, Kenneth M. Walker, Christopher D. Vance, David L. Sherman, and Daniel F. Sterne. DTE Firewalls—Initial Measurement and Evaluation Report. Technical Report 0632R, Trusted Information Systems, March 1997.

[3] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical Domain and Type Enforcement for UNIX. In *Proceedigns of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, Oakland, California, May 1995.

[4] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. 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.

[5] D. Elliott Bell and Leonard 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] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig 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] David F. C. Brewer and Michael 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] Center for Secure Information Systems, George Mason University. Security Glossary. World-Wide Web. `http://www.isse.gmu.edu/~csis/glossary/merged_glossary.html`.

[11] David D. Clark and David 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.

[12] Drew Dean, Edward W. Felten, and Dan 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.

[13] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[14] Department of Defense Computer Security Center. Department of Defense Trusted Computer System Evaluation Criteria, December 1985. Department of Defense Standard DoD 5200.28-STD.

[15] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, San Francisco, California, 1991.

[16] Michael Franz and Thomas Kistler. Introducing Juice. `http://www.ics.uci.edu/~juice/intro.html`, October 1996.

[17] Michael Franz and Thomas Kistler. Slim Binaries. Technical Report 96-24, Department of Information and Computer Science, University of California, Irvine, June 1996.

[18] J. Steven Fritzinger and Marianne Mueller. Java Security. Sun Microsystems, Inc., White Paper, `http://www.javasoft.com/security/whitepaper.ps`, 1996.

[19] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[20] Wilson C. Hsieh, Marc E. Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian N. Bershad. Language Support for Extensible Operating Systems. *Workshop on Compiler Support for System Software*, February 1996.

[21] Thomas Kistler and Michael Franz. A Tree-Based Alternative to Java Byte-Codes. Technical Report 96-58, Department of Information and Computer Science, University of California, Irvine, December 1996.

[22] Douglas Kramer, Bill Joy, and David Spenhoff. The Java Platform—A White Paper. JavaSoft White Paper, `ftp://ftp.javasoft.com/docs/JavaPlatform.ps`, May 1996.

[23] Theodore 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.

[24] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[25] Steven 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.

[26] Lucent Technologies Inc. Inferno: la Commedia Interattiva. `http://inferno.bell-labs.com/inferno/infernosum.html`, 1996.

[27] Lucent Technologies Inc. Security in Inferno. `http://inferno.bell-labs.com/inferno/security.html`, 1997.

[28] Catherine Jensen McCollum, Judith R. Messing, and LouAnna 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.

[29] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes and Antidotes*. Wiley Computer Publishing, John Wiley & Sons, Inc., New York, New York, 1997.

[30] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[31] Spencer E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, Salt Lake City, Utah, June 1995.

[32] Przemysław Pardyak and Brian 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.

[33] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[34] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 35–50, Orcas Island, Washington, December 1985.

[35] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 146–160, Asheville, North Carolina, December 1993.

[36] Margo I. Seltzer. Personal Communication, January 1997.

[37] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996.

[38] Emin Gün Sirer, Marc Fiuczynski, Przemysław Pardyak, and Brian Bershad. Safe Dynamic Linking in an Extensible Operating System. *Workshop on Compiler Support for System Software*, February 1996.

[39] Emin Gün Sirer, Stefan Savage, Przemysław Pardyak, Greg P. DeFouw, Mary Ann Alapat, and Brian N. Bershad. Writing an Operating System with Modula-3. *Workshop on Compiler Support for System Software*, February 1996.

[40] Karanjit S. Siyan. *Windows NT Server Professional Reference*. New Riders Publishing, Indianapolis, Indiana, 1995.

[41] Niklaus Wirth and Jürg Gutknecht. *Project Oberon—The Design of an Operating System and Compiler*. Addison Wesley Longman, Reading, Massachusetts, 1992.

A Formal Model

This appendix presents a formal model for access control in extensible systems. It is structured as follows: Section A.1 defines the terms used in the formal model. Section A.2 defines the access modes used for access control. Section A.3 defines the discretionary component of access control in extensible systems, and section A.4 defines the mandatory component. Finally, section A.5 defines safe optimizations for access control on extensions.

A.1 Terminology

Access Control List The mechanism used to enforce *discretionary access control*. Provides a mapping from *users* and *groups* to *access modes*. Acronym is ACL.

Access Mode Defines what type of operations a *subject* can carry out on an *object* or on another *subject*.

Discretionary Access Control A means of restricting access to *objects* and *subjects* based on *users* and *groups* which can be changed by the *users* themselves. Acronym is DAC.

Domain A label in *domain and type enforcement* that is associated with a *subject*.

Domain and Type Enforcement The mechanism used to enforce *mandatory access control*. Associates *subjects* with *domains* and *objects* with *types* and provides a mapping from *domains* and *types* to *access modes*. Acronym is DTE.

Extension A unit of code. Traditionally, a unit of code is an *object* (i.e., it contains the instructions and data of the extension). However, since extensions are written by a human programmer and their code is executed (thus turning extensions into an active entity), they are considered *subjects* as far as access control is concerned. Extensions are associated with both a *user* and an *access control list* for discretionary access control, and with a *domain* for mandatory access control.

Group A named collection of *users*.

Mandatory Access Control A means of restricting access to *subjects* and *objects* that is imposed on all system operations, that can only be changed by the security administrator, and that is used to enforce the *security policy*. Acronym is MAC.

Object A passive entity that contains or receives information. Associated with a *type*.

Security Policy The set of laws, rules, and practices that regulate how an organization manages, protects and distributes information within the computer system.

Subject An active entity. Here, either a thread of control which executes some code for some *user*, or an extension written by some *user*. Associated with a *domain*.

Thread of Control A single, sequential flow of control. Considered a *subject* as far as access control is concerned. Associated with a *user* for *discretionary access control* and with a *domain* for *mandatory access control*.

Type A label in *domain and type enforcement* that is associated with an *object*.

User Any person who interacts directly with a computer system and thus has threads of control executing some code on her behalf. Also the unit of accountability for auditing.

This glossary is inspired by [33], and some of the definitions are adapted from the security glossary in [10].

A.2 Access Modes

An access mode defines the legal types of operations a subject can carry out on an object or another subject. It is represented as a set of types of operations:

$m \subseteq \mathcal{M}$; The set of legal types of operations.

The following types of operations are necessary to properly model how extensions interact with each other. Other access modes are certainly necessary to model a complete system, but depend on the exact semantics of the objects a given system supports and of the operations that can be carried out on these objects. They are thus left unspecified.

`extend` Extend a given interface.

`execute` Execute code, or invoke a given interface.

Two rules are needed to capture the restrictions imposed on system security by access modes. The first rule simply formalizes the meaning of access modes, and the second rule further restricts when an extension is allowed to interact with another extension:

Rule 1 *A subject can only carry out those types of operations on an object or another subject that are described by the corresponding access mode.*

Rule 2 *An extension can only be linked against another extension if the corresponding access mode includes the* `execute` *type of operation or both the* `execute` *and the* `extend` *type of operation. Whether an extension can be linked against another extension when the corresponding access mode includes the* `extend` *but not the* `execute` *type of operation, depends on the semantics of the underlying extension model and is thus implementation dependent.*

A.3 Discretionary Access Control

Discretionary access control using access control lists is a familiar mechanism to limit access to resources. An access control list is a mapping from users and groups to access modes:

$$\begin{array}{lll} u & \in & \mathcal{U} \text{ ; The set of legal users.} \\ g & \in & \mathcal{G} \text{ ; The set of legal groups,} \\ & & \text{; where } \forall\, g \in \mathcal{G} \,.\, g \in \mathrm{P}(\mathcal{U}) \end{array}$$

$$\begin{aligned} \textsc{InGroups} &: \mathcal{U} \longrightarrow \{\mathcal{G}, \ldots\} \\ \textsc{Acl} &: \mathcal{U} \longrightarrow \{\mathcal{G}, \ldots\} \longrightarrow \mathcal{M} \end{aligned}$$

InGroups returns the set of groups a given user is a member in, and Acl is an access control list, which is a function local to each object and extension.

Access control lists are fully-featured and internally use two mappings, one for positive rights and one for negative rights. When trying to determine the access mode for a given user and the groups she is a member in, the access control list first collects all positive rights and then subtracts all negative rights from the first result, thus generating the final access mode. Since access control lists are a familiar mechanism, no pseudocode definition is given for Acl.

For the purposes of discretionary access control each subject needs to be associated with exactly one user, i.e., each thread of control and each extension are associated with a user. Extensions that are not clearly associated with a legal user can be mapped to a well-known *anonymous* user. Each object and each extension is associated with an access control list. Note that a thread of control associated with a user u can very well execute code in an extension associated with another user u', as long as the access control constraints are not violated.

All discretionary access control decisions involving extensions are done at link time. As a result a thread associated with user u can call the interface of an extension associated with user u', even though the access control list for that extension forbids `execute` access. While this obvious security loophole could be avoided by imposing dynamic discretionary access control checks on all calls into an extension, this would introduce unnecessary overhead while not solving the fundamental problems of discretionary access control (e.g., Trojan horse attacks would still be possible). Furthermore, this potential security loophole is properly addressed by mandatory access control which *does* dynamic access checks.

Discretionary access control is formalized by the following two rules:

Rule 3 *Each subject, that is, each thread of control and each extension, is associated with exactly one user u and each object as well as each extension is associated with an access control list* Acl.

Rule 4 *A subject associated with user u can* at most *carry out those types of operations on an object or another subject associated with an access control list* Acl *that are described by the access mode* Acl(u,InGroups(u)). *Discretionary access control decisions involving extensions are only done at link time.*

The use of the words "at most" in rule 4 reflects the fact that the final access mode for any access control decision is the intersection of the two access modes resulting from discretionary and mandatory access control.

A.4 Mandatory Access Control

Mandatory access control differs from discretionary access control in that it is imposed on all relevant system operations and can only be changed by a system's security administrator. Its constraints are the expression of some security policy external to the system. Furthermore, these constraints, to the user, appear to be dynamically enforced, that is, unlike discretionary access control, the constraints of the Rights and Target functions as defined below must always be adhered to.

The mechanism used for mandatory access control is domain and type enforcement. It uses domains and types in addition to access modes to express access restrictions:

$$\begin{array}{l} d \in \mathcal{D} \text{ ; The set of legal domains.} \\ t \in \mathcal{T} \text{ ; The set of legal types.} \end{array}$$

For the purposes of domain and type enforcement each subject needs to be associated with exactly one domain, i.e., each thread of control and each extension are associated with a domain. Extensions that are not clearly associated with a legal domain can be mapped into a dynamically created domain that is unique for this extension. The RIGHTS and TARGET functions (see below) are then updated to also accommodate this dynamically created domain according to a template. Each object is associated with exactly one type. Note that a thread of control associated with a domain d can very well execute code of an extension associated with another domain d', as long as the access control constraints are not violated. Furthermore, while the association of an extension with a domain is fixed for the lifetime of the extension within the extensible system, the domain associated with a thread of control may change according to rule 9 described below.

When a user logs in or an extension establishes its user identity, there may be a choice as to with which domain the initial thread of control or the extension should be associated. Domain and type enforcement includes a global mapping that defines which domains are valid domains for a given user and can thus be used to verify that the desired domain is legal. The global function INDOMAINS returns the set of legal domains for a given user:

$$\text{INDOMAINS} : \mathcal{U} \longrightarrow \{\mathcal{D}, \ldots\}$$

The association of subjects with domains and objects with types is captured by the following three rules:

Rule 5 *Each subject, that is, each thread of control and each extension, is associated with exactly one domain d, and each object is associated with exactly one type t.*

Rule 6 *An extension associated with a legal user u must be associated with a domain d, such that* $d \in$ INDOMAINS(u). *An anonymous extension is associated with a unique, dynamically created domain whose access rights are created according to a template. The domain an extension is associated with must not change for the lifetime of the extension within the extensible system.*

Rule 7 *A top-level thread of control, i.e., a thread of control that is created when a user u logs in, must be associated with a domain d, such that* $d \in$ INDOMAINS(u). *The domain associated with a thread of control may change according to the constraints expressed in rule 9.*

As an access control mechanism, DTE expresses restrictions on what types of operations a subject can carry out on objects and other subjects. These restrictions are expressed by two global functions, called RIGHTS and TARGET:

$$\begin{aligned} \text{RIGHTS} &: \mathcal{D} \longrightarrow (\mathcal{D} \cup \mathcal{T}) \longrightarrow \mathcal{M} \\ \text{TARGET} &: \mathcal{D} \longrightarrow \mathcal{D} \longrightarrow \mathcal{D} \end{aligned}$$

The semantics of the two functions follow, and, since both functions can be easily expressed as lookups in a two-dimensional array or a similar data structure, a pseudo-code definition is omitted:

- RIGHTS(d, d') : m — Given the domain d of a subject and the domain d' of another subject, return the legal access mode m representing the legal types of operations the first subject can carry out on the second subject. If $d = d'$, $m = \{\texttt{execute}, \texttt{extend}\}$ since a domain can always execute and extend itself.

- RIGHTS(d, t) : m — Given the domain d of a subject and the type t of an object, return the legal access mode m representing the legal types of operations the subject can carry out on the object.

- TARGET(d, d') : d_{target} — Given the domain d of a subject and the domain d' of another subject, return the target domain d_{target}. The target domain d_{target} represents the domain that will be associated with a thread of control *for the duration of a call* to an extension in domain d', given that the thread of control is currently associated with domain d. If $d = d'$, $d_{\text{target}} = d$.

The two rules relating subjects and objects, domains and types and the legal types of operations follow:

Rule 8 *A subject associated with domain d can* at most *carry out those types of operations on another subject associated with domain d′ or on an object associated with type t that are described by the access mode* RIGHTS(d, d') *or the access mode* RIGHTS(d, t) *respectively.*

Rule 9 *A thread of control associated with domain d that calls on code of some extension in domain d′ takes on the domain* TARGET(d, d') *for the duration of the call.*

Note that the restrictions imposed on a pair of subjects in rule 8 are relevant for three possible situations, namely for a thread of control executing code of an extension, for a thread of control that is about to call code of another extension, and for extension about to link against another extension.

A.5 Static Access Control for Extensions

Mandatory access control, in contrast to discretionary access control, must appear to the user as if it was dynamically enforced for all system operations. This implies dynamic access control checks for every call into an extension and the possible re-association of a thread of control with a new target domain (and the dis-association with that target domain on return). As it is clearly desirable to impose as little dynamic access control checks and target domain re-associations as possible, rules for possible optimizations are needed.

The rules discussed in this section depend on a new set operation, called DomainSet : $\mathcal{D} \longrightarrow \{\mathcal{D}, \ldots\}$:

$$\begin{aligned} \textsc{DomainSet}(d) := \ & \{d' \mid \\ & \{\texttt{execute}\} \subseteq \textsc{Rights}(d'', d) \wedge \\ & d' = \textsc{Target}(d'', d)\} \end{aligned}$$

The intuition behind DomainSet(d) is that it returns the set of all domains that are legal domains for a thread of control that is executing code of an extension in domain d. Using this new set operation, the following two rules capture when dynamic access control checks and target domain re-associations can be avoided:

Rule 10 *No dynamic access control checks are necessary for an extension in domain d if the following is true:*

$$\begin{aligned} & \forall\, d' \in \{d'' \mid \{\texttt{execute}\} \subseteq \textsc{Rights}(d'', d)\} \,. \\ & \quad \forall\, d''' \in \textsc{DomainSet}(d') \,. \\ & \qquad \{\texttt{execute}\} \subseteq \textsc{Rights}(d''', d) \end{aligned}$$

Rule 11 *No dynamic target domain re-associations for threads of control calling code of an extension in domain d are necessary if the following is true:*

$$\begin{aligned} & \forall\, d' \in \{d'' \mid \{\texttt{execute}\} \subseteq \textsc{Rights}(d'', d)\} \,. \\ & \quad \forall\, d''' \in \textsc{DomainSet}(d') \,. \\ & \qquad d''' = \textsc{Target}(d''', d) \end{aligned}$$

Both rules can result in further optimizations if the set of domains d'' is further limited to only consist of domains associated with extensions that are currently linked into the system. Another optimization uses two entry points per interface, one unchecked (thus, both rules must be true for all extensions that have access to this entry point) and one using dynamic access checks (for all other extensions).