Developing and Using a "Policy Neutral" Access Control Policy

Duane Olawsky, Todd Fine, Edward Schneider and Ray Spencer
Secure Computing Corporation
2675 Long Lake Road,
Roseville, Minnesota 55113-2536

Email: *olawsky@sctc.com, fine@sctc.com, spencer@sctc.com*

December 2, 1996

ABSTRACT

The foundation for security enforcement is *access control*. Resources must be protected against access by unauthorized entities. Furthermore, authorized entities must be prevented from accessing resources in inappropriate ways. A major challenge to the developer of an access control policy is to provide users the flexibility to protect their resources as they see fit; system policies that are inconsistent with user needs are inadequate. In particular, systems that enforce a single, hard-coded policy cannot satisfy the needs of all users.

As part of the Distributed Trusted Operating System (DTOS) program, we have developed and implemented a flexible security architecture using the Mach microkernel. In this architecture, the security rules enforced by the system are defined by a system component outside the microkernel. This reduces the problem of supporting other security policies to redefining this system component; the same microkernel can be used to support a wide range of policies.

Formal methods were used to provide a rigorous approach for the development of the policy. Recognizing that most people are uninterested in reading security requirements stated in formal specification languages, an approach was developed for representing and maintaining the policy in a tabular format. This paper describes the flexibility of the DTOS security architecture and the approach used in developing the access control policy for this flexible architecture. It also gives examples of how to define a component that makes security decisions for the microkernel.[1]

1 INTRODUCTION

One of the goals of the Distributed Trusted Operating System (DTOS) program is to investigate an approach for developing an operating system microkernel that supports a wide range of security policies. Rather than simply following the guidelines in the Trusted Computer Security Evaluation Criteria (TCSEC) [12] and implementing Discretionary Access Control (DAC) and Multilevel Security (MLS), the DTOS microkernel must provide a framework that encompasses these policies as well as others. The DTOS program is exploring this framework through prototyping and study efforts.

Given that secure system developments have traditionally focused on implementing a particular security policy,[2] a natural question to ask is why we think supporting a wide range of policies is important. One reason is that different sites need to protect against different threats. A site controlling a nuclear reactor needs to protect the integrity of the processes and data used to run the reactor. A site containing proprietary or confidential data needs to protect that data from unauthorized disclosure. A site managing medical records needs to protect the records both from unauthorized disclosure and inappropriate modification. While access control policies are appropriate for each of these examples, a different type of access control policy might be desired for each. Policies such as Type Enforcement [3] and Clark-Wilson [5] can be used to address integrity concerns. Other policies such as MLS, Chinese-Wall [4], and ORCON [10] can be used to address confidentiality concerns. However, no single policy is appropriate for all cases.

A second reason for supporting a wide range of policies is that the set of threats against which each site must protect is constantly evolving. Some threats that are of concern today might not be of concern next year. Furthermore, the system must protect against new threats that exploit previously unknown security flaws in existing applications and security flaws introduced through new applications. A system that is hard coded to enforce a single security policy will have much more trouble adapting to the evolving set of threats than a system supporting a flexible security architecture. This is especially true when high assurance is a goal. Then, time is required to model the system, state the policy, and perform the assurance analysis. By basing the assurance for a specific site on assurance performed on a policy neutral system, the time required to assure the final system can be greatly reduced.

Without policy flexibility, users must either make due with a system that does not provide exactly the type of protection they would like or must wait until someone develops a system that does. Given that the development of a secure system takes a significant amount of time, the threats against which a user needs protection typically will have changed between the time that development begins and the time a new system is completed. Thus, users are constantly forced to make due with the policies provided by existing systems.

In Section 2 we describe the DTOS architecture for a policy flexible system, and in Section 3 we describe the method used in DTOS to develop a flexible, policy-neutral access control policy. Section 4 presents two examples of the use of the architecture to implement a high-level policy (MLS and Clark-Wilson), and Section 5 discusses the range of policy flexibility supported by DTOS.

[1] This work was supported in part by the Maryland Procurement Office, contract MDA904-93-C-4209 and was performed in cooperation with researchers at the Information Security Computer Science Research Division of the Department of Defense.

[2] The Data Secure Unix system described in reference [17] is an exception.

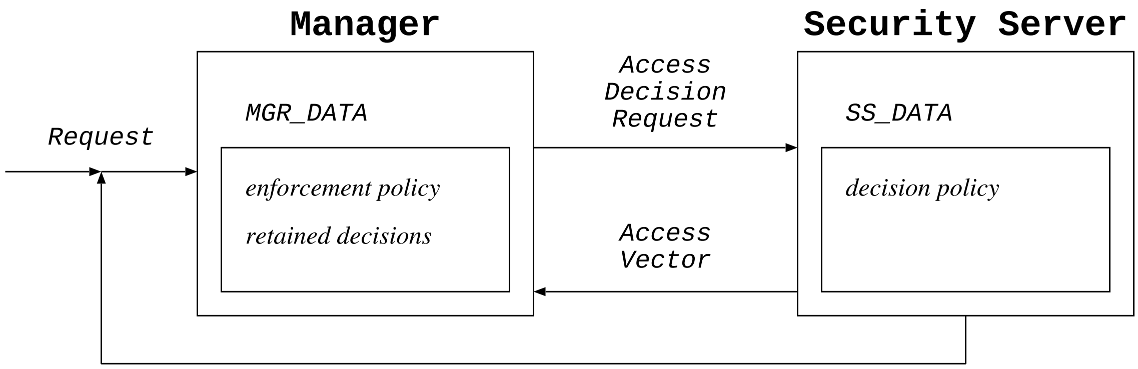

Figure 1: DTOS Architecture

Section 6 describes the use of composability analysis [1] to deduce properties of the system. Section 7 notes some related research. Finally, Section 8 summarizes results and open issues.

2 AN ARCHITECTURE FOR POLICY FLEXIBILITY

The DTOS security architecture [11], depicted in Figure 1, supports policy flexibility by separating the making of policy decisions from the enforcement of those decisions. The policy decisions are made by *security servers*. A security server[3] is simply a process executing in the system that makes decisions based on a set of security rules. The enforcement of these decisions is performed by each system component managing the objects protected by the policy. A manager is the only subject able to directly access some collection of objects that it manages. It receives a sequence of requests from various client subjects to perform actions on its objects and must decide, based on its current state (possibly augmented by new access decisions received from a security server), whether or not to carry out that request.

The manager receives requests from other subjects, including the Security Server, and it sends access decision requests to the Security Server. The Security Server sends responses to the access decision requests to the manager. The manager and Security Server each have internal data that records their processing state. The Security Server's data includes a *decision policy* which is the data and/or code governing the Security Server's policy decisions. The manager's data includes an *enforcement policy* specifying the required access decisions that the manager associates with each manager request and a set of *retained decisions* specifying access decisions that the Security Server has previously made which have been cached by the manager.

When requesting a security decision, the manager must provide information indicating the subject that is requesting the service and the object upon which the service is to operate. Thus, it suffices for an object manager to associate security information with each object that it manipulates as a result of client requests. The process manager[4] manages the subjects and therefore associates security information with each subject. In addition to providing the security information for the accessing subject and for the entity acted upon, the manager also provides the type of operation that is desired. The operation type is specified by a *permission* name. In response, the Security Server provides a set of decisions, called an access vector, indicating which operations the accessing subject may perform on the entity. Although the Security Server could simply respond with a yes/no answer as to whether the requested operation is permitted, we return an access vector for efficiency. By caching the returned access vector and consulting the cache before requesting decisions from the Security Server, the manager can avoid interactions with the Security Server when the necessary information is in the cache.[5]

The security information that a security server needs in order to make access decisions depends on the particular policy implemented by that security server. For example, a security server enforcing an MLS policy makes its decisions based on the security levels of the accessing subject and the accessed entity. However, having the manager provide security levels to the Security Server would be incorrect since it would hard code into the manager that each entity has a security level. To be truly policy flexible, the manager cannot contain any policy specific information. Thus, the manager associates a label called a *security identifier* (SID) with each manager object. The Security Server defines a mapping between SIDs and *security contexts*. This mapping defines the meaning of each SID. In the case of an MLS policy, a security context might consist of simply a security level. In the case of a Type Enforcement [3] policy, the security context associated with a subject SID might contain only a domain while the security context associated with an object SID might contain only a type.[6] The level of indirection provided by SIDs allows the same manager to be used regardless of how the Security Server interprets SIDs and makes access decisions. The Security Server provides an interface allowing managers and other tasks to map SIDs to their associated contexts and vice versa.[7] Of course, a security server may restrict access to this information if this is required by its policy goals.

There are several types of policy involved in a system using this architecture. The first is the *high-level system policy*. Some examples are MLS, Clark-Wilson and ORCON. This is the policy that would be hard-coded into a system using a traditional architecture not designed for policy flexibility. In the DTOS architecture this policy emerges from the interaction of the manager and the Security Server, each of which is implementing its own policy.

---

[3] We use "a security server" when referring to security servers in general and "the Security Server" when referring to the security server present in a given instance of DTOS.

[4] The process manager in DTOS is the Mach microkernel.

[5] The interaction between the manager and Security Server in DTOS is slightly more complicated than that described here. For example, there are also facilities for the Security Server to instruct the manager not to cache certain parts of the returned access vectors and to flush vectors from the cache. Such features are necessary to support policies in which accesses can be revoked.

[6] Type Enforcement controls subject-to-subject access on a domain-to-domain basis and subject-to-object access on a domain-to-type basis. Thus, the security information needed to make decisions consists of domains and types.

[7] The developer of any particular security server must decide whether it is important to the goals of the policy that SIDs be cryptographically protected (or even opaque) from interpretation by other tasks. Such protection is not required by the architecture. However, to maintain policy-neutrality, all managers should be written with the assumption that SIDs are opaque. If this guideline is violated, the manager will not work correctly with any security server that does not supply transparent SIDs with the same structure.

The manager's *enforcement policy* defines the security requirements governing when the manager provides service. In particular, this policy identifies the points in the manager processing at which a security decision needs to be obtained. It also indicates which security decision is needed at each point. This policy defines what it means for the manager to enforce policy decisions made by the Security Server. It will be the same no matter what decision policy is supplied by the security server. Each manager is trusted to correctly implement its enforcement policy.

The Security Server's *decision policy* defines the security requirements on how the security server makes access decisions. Since the intent is to allow different security servers to make security decisions differently, there is no single security server policy. However, there is a well-defined interface that the managers expect each security server to implement. The main requirement on the interface is simply that whenever the Security Server sends the results of a security decision, the results are consistent with the decision policy that the Security Server is implementing.[8]

Although a manager could be any of a variety of components including a file server enforcing access decisions made on files and an application enforcing access decisions on application specific data, the remainder of this paper considers only the enforcement by the DTOS microkernel of access decisions made on microkernel subjects and objects.

3 DEVELOPING THE DTOS MICROKERNEL ENFORCEMENT POLICY

To explore the use of the DTOS architecture, a primary focus of the DTOS program has been to modify the Mach microkernel to serve as a manager in that architecture. In doing so, we have added support to Mach for a wide range of access control policies. This has been accomplished by inserting control logic in the microkernel. The processing of each microkernel request has been modified to request a security decision by a security server before providing a service. We have also implemented a prototype user-space security server that makes these security decisions for the microkernel. The security-enhanced microkernel and the prototype Security Server have been released to a number of sites for use in research on secure systems. Some sites are developing their own security server while others are developing additional policy-flexible applications. Additional information on the implementation of both the microkernel and the Security Server can be found in [11].

Although the work described here deals with enhancing Mach to function as a policy-neutral object manager, this is merely an example. The architecture is general enough to be applied not only to other microkernels but to a wide variety of managers.

3.1 APPROACHES TO POLICY DEVELOPMENT

Traditionally, there have been two related but distinct approaches to developing security policies. The first approach, the *threat-based* approach, is to identify the system threats that are of concern and develop requirements that address the threats. The second approach, the *criteria-based* approach is to interpret a set of requirements specified by an evaluation criteria document (such as [12]) for the target system. The relation between the two approaches is that in the second approach it is assumed that the developers of the evaluation criteria have already identified all of the relevant threats.

The criteria-based approach is infeasible for DTOS due to the goal to support a wide range of policies. Regardless of whether an evaluation criteria document contains MLS, integrity, or availability requirements, there is always the possibility that the user of a DTOS system will want to enforce some other type of security. Consequently, the DTOS policy must provide a framework in which a variety of policies can be supported rather than simply interpreting requirements in an existing evaluation criteria.

Thus, the DTOS policy development is threat-based. However, the threats identified are of a different nature than those traditionally identified. When developing the policy for a system that is intended to enforce a single policy, the identified threats typically are specific to that policy. For example, while covert channels [12] are a threat with respect to MLS policies, they are typically not a threat with respect to integrity policies. Since the DTOS policy is intended to provide a framework that supports a wide variety of policies, the threats identified for DTOS must be policy independent.

The intent is for users to be able to counter threats to their systems by appropriately configuring DTOS. Furthermore, as the set of threats against which a site must protect evolves, administrators should be able to reconfigure DTOS to address the new set of threats. This requires controls to be placed on essentially all services. For example, DTOS must control the setting of the scheduling priority for a thread since some users will want to protect against service denial to user threads. Although the denial of service threat might be of little concern to most users, the possibility that some users might be concerned suggests viewing it as a real threat. Since providing protection against every conceivable threat is impossible, a judgement call must be made on the set of threats that are of concern.

The approach taken in defining the enforcement policy for the DTOS microkernel is to view any access of the microkernel state as being a potential threat. By viewing each access as a potential threat and providing appropriate control mechanisms, the goal of supporting multiple policies can be achieved.[9]

3.2 POLICY DEVELOPMENT

Although developing a policy for a system intended to be "policy neutral" seems paradoxical, the "paradox" is largely resolved by the separation of security enforcement from security decision making. In this section we describe a process for defining the hard-coded enforcement policy in the manager. We use the DTOS microkernel enforcement policy as an example. In Section 4 we give examples of how to define a decision policy in a security server to achieve a given high-level policy.

The process we have used for defining the enforcement policy of the DTOS microkernel consists of the following primary steps:

1. Identify the services that are provided by the microkernel,
2. Relate each microkernel service to one or more access decisions that must be obtained for the service to be performed.

3.2.1 IDENTIFYING SERVICES

To perform the first step we determine the following information about the system:

- the microkernel data structures, and
- the requests that clients may make to access those structures.

With this information in hand we proceed to identify the services provided by the microkernel that need to be controlled. We distinguish the following two classes of service: transformation and invocation. A *transformation service* is one that is defined in terms of a change to one or more of the data structures that comprise the system state. For example, one component of the DTOS system state is $existing\_tasks$, the set of existing tasks. Since any change to a set involves adding or removing elements (or

[8] A security server may also provide specialized interfaces for use by particular managers.

[9] See the DTOS Generalized Security Policy Specification [14] for more on supporting multiple policies.

both), these are two natural services to associate with this component. Consideration is then given to whether a threat is posed by the ability to add or remove elements from this set. The ability to remove an element poses a denial of service threat. Thus, we define a service, $TerminatesTask(task)$ to be any modification to the contents of the system state that results in $task$ being removed from $existing\_tasks$. Any system transition in which an element is removed from $existing\_tasks$ is an instance of this transformation service. As another example, each task has an associated priority that determines the initial priority of its threads. The service $SetsTaskPriority(task)$ is defined as a modification to the system state that results in the priority of $task$ changing. The $AllocatesReadRegion(task, page\_index)$ service is defined as a state change in which a new page is allocated at $page\_index$ for $task$ and the protections of that page include read access. The $AllocatesWriteRegion(task, page\_index)$ and $AllocatesExecuteRegion(task, page\_index)$ services denote the allocation of pages with write and execute access, respectively.

In all of these examples, a service is equated with a characterizing property of state transitions. Any state transition satisfying the characterizing property is considered to have provided the service. Conversely, a state transition that does not satisfy the characterizing property is considered not to have provided the service.[10]

Not all microkernel requests alter the modeled state of a microkernel entity. Some of them only observe the modeled state of some entity, and these requests cannot easily be characterized as performing transformation services. For example, consider the Mach **task_info** request which returns information on the state of a specified task. Since this operation simply observes data, no trace is left in the contents of the system state to indicate when the operation has been performed. For each such request we define an *invocation service*.[11] Any system transition in which one of these requests is invoked is an instance of the corresponding invocation service.

Unlike a transformation service, which may be performed by multiple requests, an invocation service is associated with exactly one request. Since transformation services address the ways in which subjects can modify the system state, they address primarily denial of service and integrity concerns. In contrast, invocation services address ways in which subjects can observe objects, thus focusing on confidentiality concerns.

3.2.2 STATING THE POLICY

The second step in the development of an enforcement policy is to define the relationship between the manager's services and the access decision computations that must be requested of the Security Server by the manager. The manager enforcement policy must indicate which access decisions need to be checked before providing each service.[12] Thus, the enforcement policy must map each service to a triple consisting of the SIDs of the subject and object involved together with the permission requested.

For example, the DTOS microkernel's enforcement policy maps the service $SetsTaskPriority(task)$ to the permission $Change\_task\_priority$ and the SIDs for the client[13] and $task$ (the target). In other words, the microkernel policy requires that the client have $Change\_task\_priority$ permission to $task$ before providing the service $SetsTaskPriority(task)$ to the client. Similarly, the DTOS enforcement policy maps the service $TerminatesTask(task)$ to the permission $Terminate\_task$ and the SIDs for the client and $task$. The services $AllocatesReadRegion(task, page\_index)$, $AllocatesWriteRegion(task, page\_index)$ and $AllocatesExecuteRegion(task, page\_index)$ are mapped to $Have\_read$, $Have\_write$ and $Have\_execute$ permission, respectively, and to the SIDs for $task$ and the indicated page.

Requirements on $client$ to $task$ Accesses

Transformation Service	Required Permission
$SetsTaskPriority(task)$	$Change\_task\_priority$
$TerminatesTask(task)$	$Terminate\_task$

Table 1: Tabular Policy Example

The DTOS enforcement policy is stated in two different forms. To provide a clear, precise statement, the policy is formalized in the Z specification language [16]. This requires formalizing the system state and the transformation services. Then the enforcement policy can be formalized as a relation between the services, the permissions and the SIDs. The expression $kernel\_allows(task\_sid(client), task\_port\_sid(task))$ denotes the set of permissions allowed from the SID of the client to the object SID of the target task. It thus models the access vector associated with the pair of SIDs. The formalization of the requirements on $SetsTaskPriority$ is as follows:

$$\begin{array}{l} \forall\ Transition;\ task : TASK \\ \bullet\ SetsTaskPriority \\ \qquad \Rightarrow Change\_task\_priority \\ \qquad\qquad \in kernel\_allows(task\_sid(client), \\ \qquad\qquad\qquad\qquad task\_port\_sid(task)) \end{array}$$

Experience has shown that most people are uncomfortable reading such mathematical statements. Consequently, formal security policies are to a great extent ignored by all but formal methods advocates. This is unfortunate since people such as system developers, evaluators, accrediters, and users need to understand the system policy. After all, the distinguishing characteristic of a secure system is that it has a policy that it is assured to satisfy.

The DTOS enforcement policy addresses this by providing a tabular representation of the policy as well as the formal Z statement. Tables 1 and 2 contain brief excerpts that illustrate the tabular representation of the policy.

The heading of a table indicates the SIDs that should be used for the permission checks specified in the table. Each row of a table identifies a binding between a service and a permission. One such table is defined for each pair of entity types for which there are associated permission checks. A similar approach is used for invocation services. The only difference is that the tables associate permissions with DTOS requests instead of transformation services. The system developers have found the tables to be a convenient representation of the policy. This has allowed the people coding the security checks to obtain a better understanding of the security checks than if the policy was documented only in the Z specification language.

[10] One issue that might be taken with the task priority example is that a request that sets a task's priority to the same value as the task's current priority will not be recognized as a $SetsTaskPriority$ service. As the request is effectively a no-op, we maintain that there is no need to view the request as providing a service. Of course, there are also covert channel issues that must be addressed when the permission checks are being implemented. Care must be taken that if the service is disallowed an "insufficient permission" status is returned even when the operation would be a no-op. Otherwise, if a client c does not have permission to obtain a service $SetsTaskPriority(t_1)$, then c could determine the priority of t_1 by attempting to set t_1's priority and observing whether the return status indicates "success" or "insufficient permission". (Permission checks in DTOS are implemented in a way that prevents this channel.)

[11] There are a few Mach requests (e.g., **task_get_special_port**) for which multiple invocation services are defined. This allows finer-grained control depending upon which system state information is requested as specified in the parameters of the request.

[12] Recall that we allow the possibility that the result of an access decision request is cached. If the result of a required access decision can be obtained from the cache, then the microkernel will not make a new request for that access decision.

[13] The client is the task that initiated a request for service.

Requirements on $task$ to $page\_sid(task, page\_index)$ Accesses

Transformation Service	Required Permission
$AllocatesReadRegion(task, page\_index)$	$Have\_read$
$AllocatesWriteRegion(task, page\_index)$	$Have\_write$
$AllocatesExecuteRegion(task, page\_index)$	$Have\_execute$

Table 2: A Second Example

To help maintain consistency between the enforcement policy, the design documents and the microkernel itself, we have written tools that automatically extract information on services and permissions from underlying data tables. These tools analyze the data tables to produce the following:

- The policy requirements (both the formal Z versions and the tables shown above) included in the enforcement policy document,
- The lists of permissions needed to invoke each request (there may be several),
- The list of permissions associated with each class of object, and
- C files defining the permissions used in the microkernel and the checks to be performed.

These tools have proved useful in maintaining consistency between the assurance and implementation efforts as the system evolved.

Although the tabular representation of the policy has been quite useful, it is incomplete without the definitions of each of the services. In the DTOS approach, the service definitions are given informally in English and formally in Z. Although the Z formalization could be omitted, some benefits have been achieved from the formalization. First, the formal definitions are much more precise than the informal ones. This additional precision is especially useful in capturing some of the more subtle aspects of the system such as the transfer of capabilities. The lack of precision in informal definitions can lead to inconsistencies between how the security requirements are interpreted by the people implementing the system versus the people analyzing the system. Second, the formalization of the policy has allowed other tools to be used in the development of the policy. For example, a parser can be used to check the syntax and typing of the requirements. In particular, referencing a service that has not been formally defined results in an undefined function being reported when the formal policy statement is generated and parsed. This has actually occurred on the DTOS program when system implementors have added new services to the tables. In these cases, parsing the formal policy identifies that the new services still need to be formally defined.

3.2.3 EVALUATION OF THE APPROACH

The two-step process described here was relatively straightforward to apply to DTOS. The microkernel documentation describes the system data structures and microkernel requests from which the service definitions are derived. The approach worked well for the initial development of the policy as well as for the incorporation of system components that were added later. Having a well-defined process for identifying the services is much more desirable than using an *ad hoc* approach. Since there is nothing Mach-specific to this approach, it is of use to other secure system developments, too. This includes operating system and application developments as well as other microkernels. The only assumption made by the approach is that the system uses the client-server paradigm.

We note that it would be possible to define one or more invocation services for each system request and not define any transformation services. This would eliminate the need to model the system state in the enforcement policy specification. However, we prefer the use of transformation services whenever possible because requirements based on them provide general enforcement statements. A transformation service defines a state transition that might be provided by multiple system requests. For these cases, defining the security requirements in terms of a common transformation service ensures a more coherent policy. Rather than a separate permission check being specified for each individual request providing the service, a single permission is globally associated with the service regardless of what requests are implemented in the system. This has the following advantages:

Robustness — If the system interface is modified, we only need analyze what services are performed by the modified/added system calls.

Support of High-Level Reasoning — Transformation services allow general high-level reasoning about permission checking without repeated analysis of all the requests that perform a given service.

In contrast, the invocation services control the invocation of requests rather than the providing of services. A requirement that a client have get_task_info permission to a task in order to invoke the **task_info** request on that task places no restrictions on other ways in which the client can obtain information about the task. To perform higher level reasoning about which tasks "know" a given piece of information for a particular task, one must first identify all of the requests that return that information. Then, the permissions associated with each of these requests must be analyzed to ensure that the policy is satisfied.

However, even for transformation services, it is still necessary at some point to determine which requests provide the transformation service. In particular, the system developers will need to determine which portions of the code provide a given service so that the access decision requests required by the enforcement policy can be included. Thus, while transformation services have advantages, they might complicate arguments that the implementation obeys the enforcement policy.

The number of permissions defined in DTOS is much greater than those defined for other systems. For example, most MLS systems reduce the set of permissions to $read$ and $write$. In DTOS, there are currently about 150 different microkernel permissions. Not coincidentally, there are approximately 150 microkernel calls in Mach. Thus, the large set of permissions is necessary to support fine-grained control. For example, there are different types of "read" accesses in Mach that a given policy might wish to differentiate. Two such read accesses are

- read a task's address space, and
- read a task's IPC name space by copying a port right from the task.

Since the goal of DTOS is to support a wide range of policies, a large set of permissions is necessary. Otherwise, DTOS will not be able to support system policies that require fine-grained control.

Fine-grained control is very closely linked to the concept of *least privilege*. An enabling design principle for secure systems is to limit the privileges held by each subject to the minimum required. Then, the system decision policy can be relied upon to prohibit the subject from performing unwanted operations. This allows the majority of the assurance analysis for the subject to focus on demonstrating that the subject correctly performs the operations that it is permitted to perform.

The large number of permissions raises two concerns:

- the complexity of inserting code to check so many permissions, and
- the effect on performance of checking so many permissions.

In DTOS, most microkernel calls require only a single permission check, and most of these permission checks can be done at the same point in the code before processing of the request is dispatched to the individual processing routines. This resolves the first concern to a large extent.

To address the second concern, we implemented an access vector cache in the microkernel. To reduce cache searching, pointers from key data structures to associated cache entries are maintained by the microkernel. Heavy use of Mach send-once rights reduces the effectiveness of this secondary caching mechanism (the pointers). A few preliminary timing studies have been performed, but they are not sufficient to draw solid conclusions. They suggest that the impact on performance is determined largely by the effectiveness of the caching scheme. That is, if access vectors are easily available, permission checking does not have a significant effect on performance. The data are probably obscured by other factors such as paging performance and page alignment of microkernel code as well as disk fragmentation and contention. See [11] for more information on the implementation and the performance tests.

4 EXAMPLE SECURITY SERVERS

A security server has complete freedom to make each security decision in whatever manner it wants. The particular high-level policy enforced by the system is a function of the decision policy implemented by the Security Server and the enforcement policy implemented by a manager. One possible decision policy grants all permissions. If we combine such a security server with the Mach microkernel, the resulting system would be essentially equivalent to vanilla Mach. This is, of course, not very interesting from a security standpoint. In this section we give a brief sketch of two decision policies for DTOS that are more interesting with regard to security. When combined with the DTOS microkernel, the first example implements a high-level policy consisting of MLS with Type Enforcement, and the second implements the Clark-Wilson integrity policy [5]. We have also investigated the ORCON policy [10].

4.1 MLS WITH TYPE ENFORCEMENT

The only security server currently included in Secure Computing's DTOS release is one that performs level-based and type enforcement security checks [7]. This security server

- maps each subject SID to a level-domain pair,
- maps each object SID to a level-type pair, and
- makes security decisions based on the levels, domains, and types associated with the SIDs provided by the microkernel according to the usual level dominance and type enforcement conventions.

4.2 CLARK-WILSON

The Clark-Wilson integrity policy [5] is concerned with the correctness of data and the prevention of fraud rather than the prevention of disclosure. The data items that are to be protected are called *constrained data items* (CDIs). The primary way in which CDI correctness is protected is by allowing CDIs to be modified only by certain programs, called *transformation procedures* (TPs), that have been certified to take the set of CDIs from one valid state to another. (Validity is defined in some application-specific way.) Each TP is certified to manipulate only certain sets of CDIs in a single execution.

Prevention of fraud is furthered by providing mechanisms for the separation of duty. A user u is allowed to modify a CDI, c, only if there exists a set of CDIs, S, and a TP, t, such that

- c is an element of S,
- u modifies c by executing t,
- u is certified to execute t to modify the CDIs S.

Consider a check-writing program that requires a purchase order to be entered into the system before a check will be printed. With the above requirement, we can prevent the person who can run the check-writing program from also running the equipment purchasing program. In this way no single person can produce a purchase order, discard it, and then write a check to pay for an item which is never ordered. Fraud then requires at least two people conspiring together.[14]

In defining the decision policy of a Clark-Wilson security server the primary consideration is the maintenance of a history for each TP execution. Each process is assigned a unique subject SID (and thus a unique subject context).[15] A subject context indicates the user in whose name the process is executing and the TP that the process is executing. Every time a process p, executing a TP t, is granted write access for the first time to any CDI c_1, this event is recorded in the Security Server. Let $CDI\_history(p)$ denote the set of all CDIs for which p has been granted write permission. When p requests write access to a CDI c_2, the Security Server checks the CDI history associated with p. Write permission for c_2 is granted only if $\{c_2\} \cup CDI\_history(p)$ is a subset of some set S_1 of CDIs that TP t is certified to manipulate and some set S_2 of CDIs that the user is certified to manipulate via t. In this way the Security Server ensures that granting p write access to c_2 will not allow p to manipulate a set of CDIs in violation of the Clark-Wilson constraints. This example decision policy shows that the architecture can support dynamic policies.

5 RANGE OF POLICY FLEXIBILITY

The example decision policies in Section 4 demonstrate some of the flexibility of the DTOS microkernel enforcement policy. In this section we discuss in more general terms the capabilities and limitations of the enforcement policy in supporting high-level policy flexibility.

We have already seen an example of a decision policy that provides a dynamic policy that is sensitive to the history of granted permissions. DTOS can also support dynamic policies that are environment-sensitive. For example, DTOS could be used to implement a time-of-day policy in a bank where different decision policies are used during banking and non-banking hours. This can be achieved by writing a security server that monitors the system

[14] For brevity, we have omitted some of the requirements of Clark-Wilson. These requirements are considered in [14].

[15] The current version of DTOS does not adequately support this one-to-one relationship between subjects and SIDs. It can be obtained but may require modifications to many programs, include some that are not security aware. Of course, the inadequate support is not a concern if it is acceptable to view all processes with the same SID as being the same logical "process".

clock and alters its method of making decisions at the appropriate times. DTOS also supports both transitive and intransitive decision policies. A transitive policy is one where if a subject A can modify an object d_{A} and if a subject B can detect the modifications made by A to d_{A} and can itself modify a data item d_{B}, then A can also modify d_{B}. Any policy that does not satisfy this constraint for all subjects and objects is intransitive.

As observed in Footnote 15, DTOS does not adequately support a one-to-one relationship between processes and SIDs. A second limitation is that the DTOS microkernel does not send the parameters of a request to the Security Server. This prevents the implementation of certain policies. For example, suppose someone wishes to implement a policy that allows each task t_1 to set the priority of a task t_2 to any value p such that $min\_pri(t_1, t_2) \leq p \leq max\_pri(t_1, t_2)$ where min_pri and max_pri are functions that map a pair of tasks to a priority. To support this type of high-level policy, the microkernel would have to send the desired priority to the Security Server as part of the access decision request. This effectively defines a unique permission (and service) for each possible value of a task's priority. The DTOS enforcement policy does not support this level of granularity.

Another limitation results from the fact that all access decision requests are in terms of a pair of SIDs. It would probably be useful to allow access decision requests with more than two SIDs. For example, we might want to control port requests in Mach based upon a SID-triple containing the client, the target port and the task receiving from the target port. As another example, a Clark-Wilson decision policy could probably be implemented with much less history information if the Security Server interface allowed a process to request access to an entire set of CDIs in one interaction.

DTOS allows the Security Server to specify that an access decision is non-cachable and to request that a decision be removed from the microkernel's cache. However, in the first several releases of DTOS, because of the way in which memory access is controlled in Mach, both of these abilities had no effect on read, write and execute permissions. This limited the ability of DTOS to support policies that must retract permissions that have already been granted.[16] We note that this does not make the system insecure, it only limits the policy flexibility supported by the DTOS microkernel.

Obviously, this permission retraction problem applies only to the DTOS microkernel and does not affect any other manager that might be used in a DTOS system. Furthermore, the other limitations discussed in this section really only apply to the microkernel and the *current* Security Server. A new security server could allow an arbitrary number of SIDs or additional parameter information to be sent in a decision request. If Clark-Wilson CDIs were managed by a file server rather than the microkernel, then the file server could request access to a set of CDIs in a single interaction. We also point out that each manager is responsible for defining and enforcing its own policy. A security server can be written or extended to make policy decisions for any such manager. Thus, an MLS DBMS acting as the manager for database objects can have its own enforcement policy dealing with tuples, attributes and relations. A security server could be defined to supply access vectors instructing the MLS DBMS on which operations are to be allowed and which rejected.

6 COMPOSABILITY

A question to be answered in any system with the DTOS architecture is whether the interaction of a manager and security server, each following its own policy, guarantees that the system as a whole enforces the high-level system policy. We are using composability theory [1, 15] to perform this analysis [6]. To do so we first specify for each component (i.e., the manager and the Security Server) the component's behavior and the assumptions the component makes about the actions of its environment including the other components of the system and the environment of the entire system. In both cases, we focus on safety properties. After showing that no component violates the environmental assumptions of any other component, we compose the two specifications by taking their conjunction. Using this method we can analyze access control policies such as simple security, the *-property and integrity.

The advantage of applying composability analysis to the system is that we need demonstrate the correct implementation of the enforcement policy in the manager only once. When a new Security Server is developed, its decision policy and the composition of this Security Server with the manager must be analyzed. However, any analysis that has already been performed on the manager can be reused. We expect the manager to normally be much larger and more complicated than the Security Server, so most of the analysis is in fact reused.

7 SOME RELATED WORK

Page et al. [13] proposes the use of rule-based policies to obtain policy flexibility. Like the DTOS separation of manager and security server, this allows the system policy to be altered without changing the manager. The way in which the rules in a rule-based policy are interpreted by an object manager is roughly equivalent to what we call an enforcement policy. Abrams et al. [2] presents a framework (GFAC) for studying and constructing access control policies. An access control policy is viewed as rules expressed by *authorities* in terms of *access control information* and *context*. Much of the information in Section 5 regarding the range of policy flexibility in DTOS came from an effort similar to the GFAC work to categorize policies according to what they require of the enforcement policy and the interface between the manager and security server. Hosmer [8, 9] considers a Decider-Enforcer architecture in which the Decider may incorporate multiple policies. These policies are related via *metapolicies* which capture the similarities between policies and the ways in which their decisions may be combined when they are being used in the same Decider.

8 CONCLUSIONS

This paper describes the approach used to develop a policy-neutral enforcement policy for the DTOS microkernel. The approach is clarified through small examples of its application to DTOS. This paper also provides examples of the combination of that enforcement policy with a decision policy to implement a system with a desired high-level system policy. Overall, the approach seems quite effective. The policy developed provides a fine degree of control which can be used for both confidentiality and integrity policies. The approach has also allowed the policy development to be more closely integrated with the system implementation by using a tabular representation of the policy. Tools have been developed to maintain consistency between the assurance and implementation efforts as the policy evolves.

Although we have presented a process for systematically developing an enforcement policy for a policy-neutral system, this process is not entirely objective. Choices must frequently be made regarding the level of granularity of the services. For example, others might choose to split the $SetsTaskPriority(task)$ service into two services: $IncreasesTaskPriority(task)$ and $DecreasesTaskPriority(task)$. This would provide finer control by allowing, for example, a task to have permission to increase a second task's priority but not decrease the priority. Some users might

[16] The problem is that Mach caches protections in the page table, and removing permissions from the access vector cache has no effect on the page table. This problem was remedied in the October 1996 DTOS release by having the microkernel walk the page tables updating the page protections as indicated by the security server.

want even finer-grained control such as that described in Section 5 with regard to specific ranges of allowed priorities.

At the other end of the spectrum is the question of whether other parts of the DTOS enforcement policy have an unnecessarily fine grain. That is, are there service distinctions in DTOS that no policy will ever need to use? Artificial examples can be created of policies that require each of the permissions that we have defined. However, the real question is what permissions will people actually need to support the policies they want to implement. Our current approach for selecting the granularity is still rather *ad hoc* and is based upon our perceptions of the likelihood that a policy will need to make different decisions with respect to the sub-services. Further analysis is required to determine which of the currently defined permissions are really necessary to support the policies of interest to users.

Finally, the DTOS architecture has the advantage that a system with a new high-level policy may be implemented merely by substituting a security server that implements a new decision policy. In assuring this new system policy we do not need to redo analysis that has already been performed upon the manager for the assurance of other policies.

REFERENCES

[1] Martín Abadi and Leslie Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] Marshall D. Abrams, Kenneth W. Eggers, Leonard J. La Padula, and Ingrid M. Olson. A Generalized Framework for Access Control: An Informal Description. In *13th National Computer Security Conference*, pages 135–143, Washington, D.C., October 1990.

[3] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th National Computer Security Conference*, pages 18–27, Gaithersburg, MD, October 1985.

[4] David F. C. Brewer and Michael J. Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989.

[5] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[6] Todd Fine. A Framework for Composition. In *Proceedings of the Eleventh Annual Conference on Computer Assurance*, pages 199–212, Gaithersburg, Maryland, June 1996.

[7] Todd Fine and Spencer E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, Oakland, CA, May 1993.

[8] Hilary H. Hosmer. Metapolicies II. In *15th National Computer Security Conference*, pages 369–378, Baltimore, MD, October 1992.

[9] Hilary H. Hosmer. The Multipolicy Paradigm. In *15th National Computer Security Conference*, pages 409–422, Baltimore, MD, October 1992.

[10] Catherine Jensen McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC – defining new forms of access control. In *IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, CA, May 1990.

[11] Spencer 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.

[12] NCSC. Trusted Computer Systems Evaluation Criteria. Standard, DOD 5200.28-STD, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, December 1985.

[13] John Page, Jody Heaney, Marc Adkins, and Gary Dolsen. Evaluation of Security Model Rule Bases. In *12th National Computer Security Conference*, pages 98–111, Baltimore, MD, October 1989.

[14] Secure Computing Corporation. DTOS Generalized Security Policy Specification. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1995. DTOS CDRL A019.

[15] N. Shankar. A lazy approach to compositional verification. Technical Report TSL-93-08, SRI International, December 1993.

[16] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1992.

[17] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and Verification of the UCLA Unix Security Kernel. *Communications of the ACM*, 23(2):118–131, February 1980.