The Flask Security Architecture: System Support for Diverse Security Policies

Ray Spencer *Secure Computing Corporation*
Stephen Smalley, Peter Loscocco *National Security Agency*
Mike Hibler, David Andersen, Jay Lepreau *University of Utah*

http://www.cs.utah.edu/flux/flask/

Abstract

Operating systems must be flexible in their support for security policies, providing sufficient mechanisms for supporting the wide variety of real-world security policies. Such flexibility requires controlling the propagation of access rights, enforcing fine-grained access rights and supporting the revocation of previously granted access rights. Previous systems are lacking in at least one of these areas. In this paper we present an operating system security architecture that solves these problems. Control over propagation is provided by ensuring that the security policy is consulted for every security decision. This control is achieved without significant performance degradation through the use of a security decision caching mechanism that ensures a consistent view of policy decisions. Both fine-grained access rights and revocation support are provided by mechanisms that are directly integrated into the service-providing components of the system. The architecture is described through its prototype implementation in the Flask microkernel-based operating system, and the policy flexibility of the prototype is evaluated. We present initial evidence that the architecture's impact on both performance and code complexity is modest. Moreover, our architecture is applicable to many other types of operating systems and environments.

1 Introduction

A phenomenal growth in connectivity through the Internet has made computer security a paramount concern, but no single definition of security suffices. Different computing environments, and the applications that run in them, have different security requirements. Because any notion of security is captured in the expression of a security policy, there is a need for many different policies and even many types of policies [1, 43, 48]. To be generally acceptable, any computer security solution must be flexible enough to support this wide range of security policies. Even in the distributed environments of today, this policy flexibility must be supported by the security mechanisms of the operating system [32].

Supporting policy flexibility in the operating system is a hard problem that goes beyond just supporting multiple policies. The system must be capable of supporting fine-grained access controls on low-level objects used to perform higher-level functions controlled by the security policy. Additionally, the system must ensure that the propagation of access rights is in accordance with the security policy. Lastly, policies are not, in general, static. To cope with policy changes or dynamic policies, the system must have a mechanism for revoking previously granted access rights. Earlier systems have provided mechanisms that allow several security policies to be supported, but they are inadequate to generally support policy flexibility because they fail to address at least one of these three areas.

This paper describes an operating system security architecture that demonstrates the feasibility of policy flexibility. This is done by presenting its prototype implementation, the Flask microkernel-based operating system, that successfully overcomes these obstacles to policy flexibility. The cleaner separation of mechanism and policy specified in the security architecture enables a richer set of security policies to be supported with less policy-specific customization than has previously been possible. Flask includes a security policy server to make access control decisions and a framework in the microkernel and other object managers in the system to enforce those access control decisions. Although the prototype system is microkernel-based, the security mechanisms do not depend on a microkernel architecture and will easily generalize beyond it.

The resulting system provides policy flexibility. It supports a wide variety of policies. It controls the propagation of access rights by ensuring that the security policy is consulted for every access decision. Enforcement mechanisms, directly integrated into the service-providing components of the system, enable fine-grained

This research was supported in part by the Defense Advanced Research Projects Agency in conjunction with the Department of the Army under contract DABT63–94–C–0058 and with the Air Force Research Laboratory, Rome Research Site, USAF, under agreement F30602–96–2–0269. It was also supported in part by the Maryland Procurement Office, contract MDA904-97-C-3047.

Authors: {sds,pal}@epoch.ncsc.mil, {mike,danderse,lepreau}-@cs.utah.edu, saltalk@lakenet.com (Spencer).

access controls and dynamic policy support that allows the revocation of previously granted access rights. Initial performance results, as well as statistics on the scale and invasiveness of the code changes, indicate that the impact of policy flexible security on the system can be kept to a minimum.

The remainder of the paper begins by elaborating on the meaning of policy flexibility. After a discussion of why two popular mechanisms employed in systems to provide security are limiting to policy flexibility, some related work is described. The Flask architecture is then presented through a discussion of its prototype design and implementation. The paper concludes with an evaluation of the policy flexibility of the system, an assessment of the performance impact, and a discussion of the scale and invasiveness of the Flask changes.

2 Policy Flexibility

When first attempting to define security policy flexibility, it is tempting to generate a list of all known security policies and define flexibility through that list. This ensures that the definition will reflect a real-world view of the degree of flexibility. Unfortunately, this simplistic definition is unrealistic. Real-world security polices in computer systems are limited by the mechanisms currently provided in such systems, and it is not always clear how security policies enforced in the "pencil-and-paper" world translate to computer systems, if at all [3, 48]. As such, a better definition is needed.

It is more useful to define security policy flexibility by viewing a computer system abstractly as a state machine performing atomic operations to transition from one state to the next. Within such a model, a system could be considered to provide total security policy flexibility if the security policy can interpose atomically on any operation performed by the system, allowing the operation to proceed, denying the operation, or even injecting operations of its own. In such a system, the security policy can make its decisions using knowledge of the entire current system state, where the current system state can be considered to encompass the history of the system. Because it is possible to interpose on all access requests, it is possible to modify the existing security policy and to revoke any previously granted access.

This second definition more correctly captures the essence of policy flexibility, but practical considerations force a slightly more limited point of view. It is unlikely that a real system could base security policy decisions for all possible operations on the entire current system state. Instead, a more realistic approach is to identify that portion of the system state that is potentially security relevant and to control operations that affect or are affected by that portion of the state. The degree of flexibility in such a system will naturally depend upon the completeness of both the set of controlled operations and the portion of the current system state that is available to the security policy. Furthermore, the granularity of the controlled operations affects the degree of flexibility because it impacts the granularity at which sharing can be controlled.

This description of policy flexibility seems limiting in three ways. It allows some operations to proceed outside of the control of the security policy, restricts the operations that may be injected by the security policy, and permits some system state to exist beyond the scope of the security policy. In actuality, each of these apparent limitations is a desirable property since many of the internal operations and state of any system are of no apparent use or concern to any security policy. Section 6.1 will discuss how these limitations were interpreted for the Flask system.

A system that is policy flexible must be capable of supporting a wide variety of security policies. Security policies may be classified according to certain characteristics, including such things as: the need to revoke previously granted accesses, the type of input required to make access decisions, the sensitivity of policy decisions to external factors like history or environment, and the transitivity of access decisions [43, Sec. 6]. The remainder of this section focuses on revocation, which is the most difficult of these characteristics to support.

Since even the simplest security policies undergo change (*e.g.*, as user authorizations change), a policy flexible system must be capable of supporting policy changes. Since policy changes may be interleaved with the execution of controlled operations, there is the risk that the system will enforce access rights according to an obsolete policy. Thus, there must be effective atomicity in the interleaving of policy changes and controlled operations.

The fundamental difficulty in achieving this atomicity is ensuring that previously granted permissions can be revoked as required by a policy change. When a permission is to be revoked, the system must ensure that any service controlled by the permission will no longer be provided unless the permission is later granted again. Revocation can be a very difficult property to satisfy because permissions, once granted, have a tendency to migrate throughout the system. The revocation mechanism must guarantee that all of these migrated permissions are indeed revoked.

A basic example of a migrated permission surfaces in Unix. The access decision for writing to a file is performed when that file is opened, and the granted permission is cached in the file description for efficient validation of write access during write operations. Revoking write access to that file in Unix only prevents future attempts to open the file with write access and has no effect on the migrated permissions in existing file descriptions. This revocation support may be insufficient to meet the needs of a security policy. This type of situation is not uncommon, and migrated permissions can be found in other places throughout a system including: capabilities, access rights in page tables, open IPC connections, and operations currently in progress. More complicated systems are likely to yield more places to which permissions can migrate.

In most cases, revocation can be accomplished simply by altering a data structure. However, it is more complicated to revoke a permission when there is an operation in progress that has checked the permission already. The revocation mechanism must be able to identify all in-progress operations affected by such revocation requests and deal with each of them in one of three possible ways. The first is to abort the in-progress operation, returning an error status. Alternately, it could be restarted, allowing another check for the retracted permission. The third option is just to wait for the operation to complete on its own. In general, only the first two are safe. Only when the system can guarantee that the operation can complete without causing the revocation request to block indefinitely (*e.g.*, if all appropriate data structures have already been locked and there are no external dependencies) may the third option be taken. This is critical because blocking the revocation effectively denies the revocation request and causes a security violation.

3 Insufficiency of Popular Mechanisms

This section discusses two popular mechanisms that are often employed to provide security to systems and the reasons why both are limiting to policy flexibility in normal usage. However, each has benefits despite its limitations, and both can be used within Flask in restricted ways that allow some of their benefits without incurring their limitations.

3.1 Capability-Based Systems

The goal of a single operating system mechanism capable of supporting a wide range of security policies is not a new goal. The Hydra operating system developed in the 1970's separated its access control mechanisms from the definition of its security policy [29, 52]. Hydra was a capability-based system, although the developers of the system recognized the limitations of a simple capability model and introduced several enhancements to the basic capability mechanisms. The Hydra approach was taken even further by the KeyKOS [40] and EROS [47] systems. Though popular, capability mechanisms are poorly suited to providing policy flexibility, because they allow the holder of a capability to control the direct propagation of that capability, whereas a critical requirement for supporting security policies is the ability to control the propagation of access rights in accordance with the policy. The enhancements introduced by Hydra and KeyKOS are intended to limit such propagation, but the resulting systems still generally only support the specific policies they were designed to satisfy, at the cost of significant complexity that diminishes the attraction of the capability model in the first place.

Primarily with an interest in solving the problem of supporting a multilevel security policy within a capability-based system, a few capability-based systems (*e.g.*, SCAP [25], ICAP [18], Trusted Mach [4]) introduced mechanisms that validated every propagation or use of a capability against the security policy. Kain and Landwehr [23] developed a taxonomy to characterize such systems. In these systems, the simplicity of the capability mechanism is retained, but capabilities serve only as a least privilege mechanism rather than a mechanism for recording and propagating the security policy. This is a potentially valuable use of capabilities. However, the designs for these systems do not define the mechanisms by which the security policy is queried to validate capabilities, and those mechanisms are essential to providing policy flexibility. The Flask architecture described in this paper could be employed to provide the security decisions needed to validate the capabilities in these systems. In the Flask prototype, the architecture is used in exactly this way.

3.2 Intercepting Requests

A common approach used to add security to a system is to intercept service requests or to otherwise interpose a layer of security code between all applications and the operating system (*e.g.*, Kernel Hypervisors [37], SPIN [20]), or between particular applications or sets of applications (*e.g.*, L3/L4 [30], Lava [22], KeySAFE [28]). This may be done in capability systems or non-capability systems, and when applied to an operating system the security layer may lie within the operating system itself (as in Spring [36]) or in a component outside of the operating system to which all requests are redirected (as in Janus [17]).

However, this approach has some serious limitations. In order to add security by intercepting requests, the existing functional interface must expose all abstractions and information flows that the security policy wishes to control. To avoid maintaining redundant state in the access control layer, the functional interface must ensure that all security-relevant attributes are either directly available as parameters or easily derived from parameters. A policy that requires the use of some internal state of the object manager as an input to the decision can not be implemented without either changing the manager to export the state or, if possible, replicating the state management in the enforcer itself. The level of abstraction provided by the interface may be inappropriate or may cause difficulties in guaranteeing uniqueness or atomicity. For example, typical name-based calls suffer from issues of aliasing, multi-component lookups, and preserving the tranquility of the name-to-object mapping from the time-of-check to the time-of-use. Finally, this approach is limited in that the security layer can only affect the operation of the system as requests pass through it. Hence, it is often impossible for the system to reflect subsequent changes to the security policy, in particular, the revocation of migrated permissions.

As was the case with capabilities, implementing access control within a security layer is a good approach when these disadvantages can be avoided through the use of other mechanisms. However, it is important to recognize that other mechanisms are necessary, often mechanisms that are more invasive than intercepting requests, in order to provide any degree of flexibility in supporting security policies.

4 Related Work

The previous section described the relationship between Flask and a variety of efforts that involved capability-based systems or the interception of requests. This section describes the relationship between Flask and other efforts not previously mentioned. We focus on the research most directly related to Flask, although there are many other efforts with some relation to our work.

The security architecture of the Flask system is derived from the architecture of our previous prototype system DTOS [35], which had similar goals. However, while the DTOS security mechanisms were independent of any particular security policy, the mechanisms were not sufficiently rich to support some policies [43], especially dynamic security policies.

At the highest level of abstraction, the flexible security model for Flask is consistent with the Generalized Framework for Access Control (GFAC) [2]. However, the GFAC model assumes that all controlled operations in the system are performed in the same atomic operation in which the policy is consulted, which is very difficult to achieve in a practical system and is the primary obstacle that the Flask system has had to overcome.

The specific issue of revocation is not a new issue in operating system design, although it has received surprisingly little recognition. Multics [39] effectively provided immediate revocation of all memory permissions by invalidating segment descriptors. Redell and Fabry [42], Karger [24] and Gong [18] all describe approaches for revoking previously granted capabilities, though none were actually implemented. Spring [49] implemented a capability revocation technique, though only the capabilities were revoked, not migrated permissions. Revocation of memory permissions is naturally provided by microkernel-based systems with external paging support, such as Mach [31], though revocation is not extended to other permissions. DTOS provided the security server with the ability to remove permissions previously granted and stored in the microkernel's permission cache. However, except for memory permissions where Mach's mechanisms could be used, DTOS did not provide for revocation of migrated permissions [38].

The Flask prototype is implemented within a microkernel-based operating system with hardware-enforced address space separation between processes. Several recent efforts (*e.g.*, SPIN [5], VINO [46] and the Java protection models in [50]) have presented software-enforced process separation. The distinction is essentially irrelevant for the Flask architecture. It is essential that some form of separation between processes be provided, but the particular mechanism is not mandated by the Flask architecture. The general applicability of key aspects of the Flask architecture to other systems was concretely demonstrated by the adoption of the DTOS architecture in the security framework of SPIN [20]. Indeed, we believe the abstract Flask architecture, and the lessons it teaches, can be applied to software other than operating systems, such as middleware or distributed systems, although of course vulnerability to insecurities in the underlying operating systems would remain.

5 Flask Design and Implementation

This section defines the components of the Flask security architecture and identifies the requirements on each component necessary to meet the goals of the system. The Flask security architecture is described here in the context of its implementation within a microkernel-based multiserver operating system. However, the security architecture only requires that the operating system include a reference monitor [16, Ch. 10]. In particular, the architecture requires the completeness and isolation properties, although verifiability is also ultimately necessary for confidence in any implementation of the architecture.

The Flask prototype was derived from the Fluke microkernel-based operating system [14]. The Fluke microkernel is especially well-suited for implementing the Flask architecture due to its lack of global resources [14] and the atomic properties of its API [13]. However, the original Fluke system was capability-based and was not in itself adequate to meet the requirements of the Flask architecture.

The remainder of this section starts by providing an

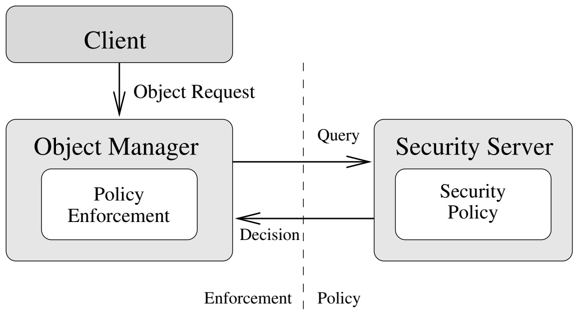

Figure 1: The Flask architecture. Components which enforce security policy decisions are referred to as *object managers*. Components which provide security decisions to the object managers are referred to as *security servers*. The decision making subsystem may include other components such as administrative interfaces and policy databases, but the interfaces among these components are policy-dependent and are therefore not addressed by the architecture.

overview of the Flask architecture. Then, it describes general support mechanisms required for the basic Flask architecture. It discusses the specific changes required for the microkernel. It explains how the complications caused by the need for revocation were overcome. This section ends by describing the prototype security server.

5.1 Architecture Overview

The Flask security architecture [44], as shown in Figure 1, describes the interactions between subsystems that enforce security policy decisions and a subsystem which makes those decisions, and the requirements on the components within each subsystem. The primary goal of the architecture is to provide for flexibility in the security policy by ensuring that these subsystems always have a consistent view of policy decisions regardless of how those decisions are made or how they may change over time. Secondary goals for the architecture include application transparency, defense-in-depth, ease of assurance, and minimal performance impact.

The Flask security architecture provides three primary elements for object managers. First, the architecture provides interfaces for retrieving access, labeling and polyinstantiation decisions from a security server. Access decisions specify whether a particular permission is granted between two entities, typically between a subject and an object. Labeling decisions specify the security attributes to be assigned to an object. Polyinstantiation decisions specify which member of a polyinstantiated set of resources should be accessed for a particular request. Second, the architecture provides an *access vector cache* (AVC) module that allows the object manager to cache access decisions to minimize the performance overhead. Third, the architecture provides object managers the ability to register to receive notifications of changes to the security policy.

Object managers are responsible for defining a mechanism for assigning labels to their objects. A control policy, which specifies how security decisions are used to control the services provided by the object manager, must be defined and implemented by each object manager. This control policy addresses threats in the most general fashion by providing the security policy with control over all services provided by the object manager and by permitting these controls to be configurable based on threat. Each object manager must define handling routines which are called in response to policy changes. For all uses of polyinstantiation, each object manager must define the mechanism by which the proper instantiation of a resource is chosen.

5.2 General Support Mechanisms

This section describes general support mechanisms that were introduced for all of the object managers in order to support policy flexibility. Despite the simplicity of the Flask architecture, some subtleties arise in the implementation, as will be discussed below.

5.2.1 Object Labeling All objects that are controlled by the security policy are also labeled by the security policy with a set of security attributes, referred to as a security context. A fundamental issue in the architecture is how the association between objects and security contexts is maintained. The simplest solution would be to define a single policy-independent data type which is part of the data associated with each object. However, no single data type is well-suited to all of the differing ways in which labels are used in a system. The Flask architecture addresses these conflicting needs by providing two policy-independent data types for labeling.

A *security context*, the first policy-independent data type, is a variable-length string which can be interpreted by any application or user with an understanding of the security policy. A security context might consist of several attributes, such as a user identity, a classification level, a role and a type enforcement [6] domain, but this depends on the particular security policy. As long as it is treated as an opaque string, a security context can be handled by an object manager without compromising the policy flexibility of the object manager. However, using security contexts for labeling and policy decision lookups would be inefficient and would increase the likelihood of policy-specific logic being introduced into the object managers.

The second policy-independent data type, the *security identifier* (SID), is defined by Flask to be a fixed-size value which can be interpreted only by the security server and is mapped by the security server to a particu-

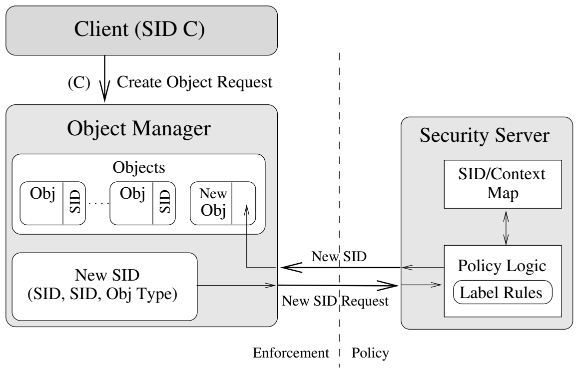

Figure 2: Object labeling in Flask. A client requests the creation of a new object from an object manager, and the microkernel supplies the object manager with the SID of the client. The object manager sends a request for a SID for the new object to the security server, with the SID of the client, the SID of a related object and the object type as parameters. The security server consults the labeling rules in the policy logic, determines a security context for the new object, and returns a SID that corresponds to that security context. Finally, the object manager binds the returned SID to the new object.

lar security context. Possession or knowledge of a SID for a given security context does not grant any authorization for that security context. The SID mapping cannot be assumed to be consistent across executions (reboots) of the security server nor across security servers on different nodes. Consequently, SIDs may be lightweight; in the implementation, SIDs are simply 32-bit integers. There is no specified internal structure to a SID; any internal structure is known only by the security server. The SID allows most object manager interactions to be independent of not just the content but even the format of a security context, simplifying object labeling and the interfaces that coordinate the security policy between the security server and object managers. However, in some cases, such as labeling persistent objects or labeling objects which are exported to other nodes, object managers must handle security contexts. This is described further in the discussion of the file server and network server in Section A.1 and Section A.2.

When an object is created, it is assigned a SID that represents the security context in which the object is created. This context typically depends upon the client requesting the object creation and upon the environment in which it is created. For example, the security context of a newly created file is dependent upon the security context of the directory in which it is created and the security context of the client that requested its creation. Since the computation of a security context for a new or transformed object may involve policy-specific logic, it cannot be performed by the object manager itself. The labeling of a new object is depicted in Figure 2. For some security policies, such as an ORCON policy [19, 34], the security policy may need to uniquely distinguish subjects and objects of certain classes even if they are created in the same security context. For such policies, the SID must be computed from the security context and a unique identifier chosen by the security server.

5.2.2 Client and Server Identification

Object managers must be able to identify the SID of a client making a request when this SID is part of a security decision. It is also useful for clients to be able to identify the SID of a server to ensure that a service is requested from an appropriate server. Hence, the Flask architecture requires that the underlying system provide some form of client and server identification for inter-process communication (IPC). However, this feature is not complete without providing the client and server a means of overriding their identification. For instance, the need of a subject to limit its privileges when making a request on behalf of another subject is one justification for capability-based mechanisms [21]. In addition to limiting privileges, overriding the actual identification can be used to provide anonymity in communications or to allow for transparent interposition, such as through a network IPC server connecting the client and server in a distributed system [11].

The Flask microkernel provides this service directly as part of IPC processing, rather than relying upon complicated and potentially expensive external authentication protocols such as those in Spring and the Hurd [7]. The microkernel provides the SID of the client to the server along with the client's request. The client can identify the SID of the server by making a kernel call on the capability to be used for communication. When making an IPC request, the client can specify a different SID as its effective SID to override its identification to the server. The server can also specify an effective SID when preparing to receive requests. In both cases, permission to specify a particular effective SID is decided by the security server and enforced by the microkernel. Thus, the Flask microkernel supports the basic access control and labeling operations required for the architecture and it provides the flexibility needed for least privilege, anonymity or transparent interposition.

5.2.3 Requesting and Caching Security Decisions

In the simplest implementation, the object manager can make a request to the security server every time a security decision is needed. However, to alleviate the performance impact of communicating with the security server for each decision and of the computation of the decision within the security server, the Flask architecture provides caching of security decisions within the object manager.

The caching mechanisms in Flask provide much more than simply caching individual security decisions. The access vector cache (AVC) module, which is a common

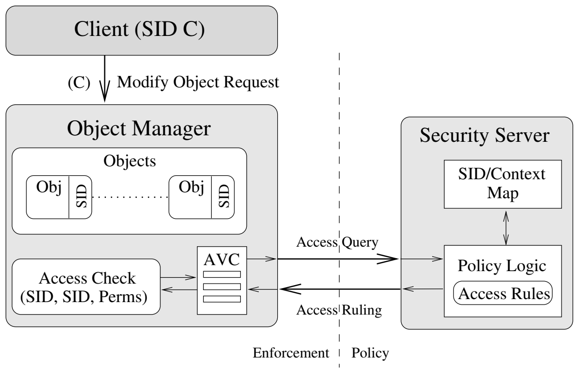

Figure 3: Requesting and caching security decisions in Flask. A client requests the modification of an existing object from an object manager. The object manager queries its access vector cache (AVC) module for an access ruling for the (client SID, object SID, requested permissions) triple. If no valid entry exists, then the AVC module sends an access query to the security server. The security server consults the access rules in the policy logic, determines an access ruling, and returns the access ruling to the AVC module.

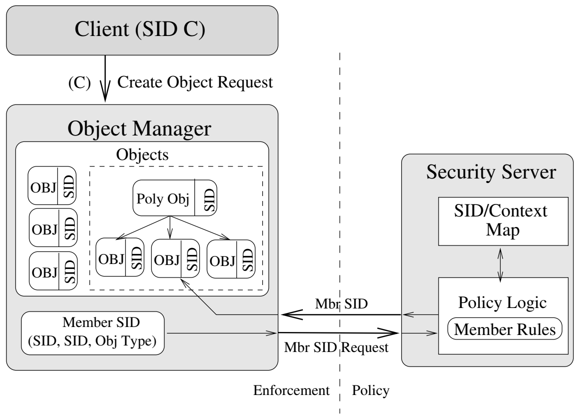

Figure 4: Polyinstantiation in Flask. A client requests the creation of a new object from an object manager, and the microkernel supplies the object manager with the SID of the client. The object manager sends a request for a SID for the member object to the security server, with the SID of the client, the SID of the polyinstantiated object and the object type as parameters. The security server consults the polyinstantiation rules in the policy logic, determines a security context for the member, and returns a SID that corresponds to that security context. Finally, the object manager selects a member based on the returned SID, and creates the object as a child of the member.

library shared by the object managers, provides for the coordination of the policy between the object manager and the security server. This coordination addresses both requests from the object manager for policy decisions and requests from the security server for policy changes. The first of these is discussed in this section, while the second is discussed in Section 5.4.

For a typical controlled operation in Flask, an object manager must determine whether a subject is allowed to access a object with some permission or set of permissions. The sequence of requesting and caching security decisions is depicted in Figure 3. To minimize the overhead of security computations and requests, the security server can provide more decisions than requested, and the AVC module will store these decisions for future use. When a request for a security decision is received by the security server, it will return the current state of the security policy for a set of permissions with an *access vector*. An access vector is a collection of related permissions for the pair of SIDs provided to the security server. For instance, all file access permissions are grouped into a single access vector.

5.2.4 Polyinstantiation Support A security policy may need to restrict the sharing of a fixed resource among clients by polyinstantiating the resource and partitioning the clients into sets which can share the same instantiation of the resource. For example, multi-level secure Unix systems frequently partition the /tmp directory, maintaining separate subdirectories for each security level [51]; the corresponding solution for Flask is discussed in Section A.1. A similar issue arises with the TCP or UDP port spaces, as discussed in Section A.2. The Flask architecture supports polyinstantiation by providing an interface by which the security server may identify which instantiation can be accessed by a particular client. Both the client and the instance are identified by SIDs. The instantiations are referred to as *members*. The general sequence of selecting a member is depicted in Figure 4.

5.3 Microkernel-specific Features

The previous sections described the security functions that are common to all of the Flask object managers. In this section, we discuss the specific features that have been added to the microkernel. Support for revocation, however, will be discussed separately in Section 5.4. The specific features that were added to some of the other Flask object managers are described in Appendix A.

Due to the requirements of Fluke's architecture, each active kernel object is associated with a small chunk of physical memory [14]. Though "memory" is not itself an object within the microkernel, the microkernel provides the base service for memory management and binds a SID to each memory segment. The SID of each kernel object is identical to the SID of the memory segment with which it is associated. This relationship between the label of memory and the label of kernel objects associated with that memory permits the Flask microkernel controls to leverage the existing protection model of Fluke, rather than introducing an orthogonal protection

SOURCE	TARGET	PERMISSION
Client SID	Effective Client SID	SpecifyClient
Server SID	Effective Server SID	SpecifyServer
Effective Client SID	Effective Server SID	Connect

Table 1: Permission requirements for an IPC connection to exist. The specify permissions are only required when a subject specifies an effective SID. If a subject does not specify an effective SID, then its effective SID is equal to its actual SID.

model as in DTOS. However, it also creates a potential loss of labeling flexibility, since the memory allocation granularity is much coarser than the allocation granularity for kernel objects.

Flask provides direct security policy control over the propagation of memory access modes by associating a Flask permission with each mode, based on the SID of the address space and the SID of the memory segment. These memory access modes also act as capabilities to kernel objects associated with the memory. During the initial attempt to access mapped memory, the microkernel verifies that the security policy explicitly grants permission for each requested access mode. Memory permissions cannot be computed at the level of any interface in Fluke, and are computed instead during page faults; hence, these controls provide an example where merely intercepting requests would be insufficient. Since the SID of a memory segment is not allowed to change, the Flask permissions need only be revalidated if a policy change occurs, as discussed in Section 5.4.

In Fluke, a port reference serves as a capability for performing an IPC to a server thread waiting on the corresponding port set. Control over propagation in Fluke may be performed through typical interposition techniques. In contrast, Flask provides direct control over the use of such port references by only allowing an IPC connection between two subjects if the appropriate permissions shown in Table 1 are satisfied. These direct controls permit the policy to regulate the use of capabilities, addressing the concerns of Section 3.1.

An interesting aspect of the Flask microkernel is the controls that are imposed on relationships between objects. In Fluke, these relationships are defined through the use of object references (*e.g.* the state of a thread contains an address space reference). Unfortunately, these references are used in many different ways, in contrast to the way in which read and write access modes are used to control access to kernel objects. For example, a reference to an address space may be used to map memory into the space or to export memory from the space. Hence, Flask introduces separate controls over these relationships and provides finer-grained control than Fluke. Some of the controls simply require the two objects to have equal SIDs, while others involve explicit permissions, as described in detail in [44, Sec. 3].

5.4 Revocation Support Mechanisms

The most difficult complication in the Flask architecture is that the object managers effectively keep a local copy of certain security decisions, both explicitly in an access vector cache and implicitly in the form of migrated permissions. Therefore a change to the security policy requires coordination between the security server and the object managers to ensure that their representations of the policy are consistent. This section is devoted to a more detailed discussion of the requirements on the components of the architecture during a change in security policy.

The need for effective atomicity stated in Section 2 is achieved by imposing two requirements on the system. The first is that after completion of a policy change, the behavior of the object manager must reflect that change. No further controlled operations requiring a revoked permission can be performed without a subsequent policy change. The second requirement is that object managers must complete policy changes in a timely manner.

This first requirement is only a requirement on the object managers, but it results in effective atomicity of system-wide policy when coupled with a well-defined protocol between the security server and the object managers. This protocol involves three steps. First, the security server notifies all object managers that may have been previously provided any portion of the policy that has changed. Second, each object manager updates its internal state to reflect the change. Finally, each object manager notifies the security server that the change is complete. Sequence numbers are used to address the interleaving of messages providing policy decisions to the object managers and messages requesting changes to the policy. Both the synchronization protocol, which has been implemented, and an alternative approach based on theories of database consistency are described in [45, Sec. 6]. The latter solution was drawn from a model of transactional consistency, but solutions related to distributed shared memory consistency may also serve as useful models.

The last step of the protocol is essential to support policies that require policy changes to occur in a particular order. For instance, a policy may require that certain permissions be revoked prior to granting new permissions. The security server cannot consider a policy change to be completed until it is completed by all affected object managers. This allows effective atomicity of system-wide policy changes since the security server can determine when the policy change is effective for all relevant object managers.

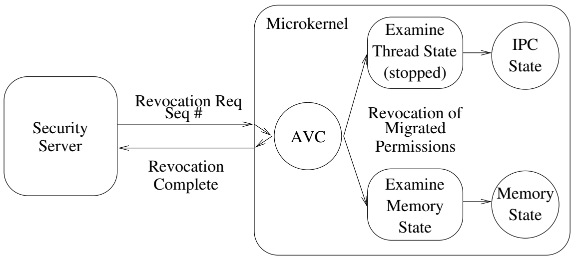

Figure 5: A revocation of microkernel permissions. Upon receipt of a revocation request from the security server, the microkernel first updates its access vector cache, and then proceeds to examine thread and memory state and perform revocations as necessary. The atomic properties of Fluke were leveraged to ease implementation of the revocation mechanism.

This protocol does not impose an undue burden in state management on the security server. The number of object managers in many systems is relatively small and the only transactions which require additional state are those where an object manager initially issues an access query for a permission that is granted. Furthermore, the security server may track permission grantings at various granularities to reduce the amount of state recorded by the security server.

The form of atomicity provided by the protocol is reasonable because of the timeliness requirement imposed on the object managers. It must not be possible for the revocation request to be arbitrarily delayed by actions of untrusted software. Each object manager must be capable of updating its own state without being indefinitely blocked by its clients. When this timeliness requirement is generalized for system-wide policy changes, it also involves two other elements of the system: the microkernel, which must provide timely communication between the security server and object managers, and the scheduler, which must provide the object manager with CPU resources.

The general AVC module handles the initial processing of all policy change requests and updates the cache appropriately. The only other operation that must be performed is revocation of migrated permissions. After updating the cache, the AVC module invokes any callbacks which have been registered by the object manager for revoking migrated permissions. The file server supports revocation of permissions which have migrated into file description objects, but currently lacks support for interrupting in-progress operations. Complete callbacks for revoking migrated permissions have currently been implemented only within the Flask microkernel, as shown in Figure 5.

Two properties of the Fluke API simplify revocation in the microkernel: it provides prompt and complete exportability of thread state and guarantees that all kernel operations are either atomic or cleanly subdivided into user-visible atomic stages [13]. The first property permits the kernel revocation mechanism to assess the kernel's state, including operations currently in progress. The revocation mechanism may safely wait for operations currently in progress to complete or restart due to the promptness guarantee. The second property permits Flask permission checks to be encapsulated in the same atomic operation as the service that they control, thereby avoiding any occurrences of the service after a revocation request has completed.

5.5 The Security Server

As stated earlier, the security server is required to provide security policy decisions, to maintain the mapping between SIDs and security contexts, to provide SIDs for newly created objects, to provide SIDs of member objects, and to manage object manager access vector caches. Additionally, most security policy server implementations will provide functionality for loading and changing policies. A security server might also benefit from providing its own caching mechanism, in addition to those contained in the object managers, to hold the results of access computations. This may prove advantageous because the security server can improve its response time by using cached results from previous, potentially expensive, access computations requested by any client.

The security server also is typically a policy enforcer over its own services. First of all, if the security server provides interfaces for changing the policy, it must enforce the policy over which subjects can access this interface. Second, it may limit the subjects that can request policy information. This is especially important in a policy where permission requests alter the policy, such as a dynamic conflict of interest policy. If the confidentiality of the policy information is important, then object managers that cache policy information must also be responsible for its protection.

In a distributed or networked environment, it is tempting to suggest that the security server of each node merely act as a local cache of the environment's policy. However, to support heterogeneous policy environments, it is desirable for each node to have its own security server with a locally defined policy component, with some degree of coordination at a higher level. Even in a homogeneous policy environment, a core portion of the security policy must be locally defined for the node in order to securely bootstrap the system into a state where it may consult the environment's policy. The development of a distributed security server for coordinating the per-node security servers within an environment remains as future work. For many policies, the security server should easily be scalable and replicable, since most policies will require little interaction among the individual nodes' security servers. However, some security policies, such as history-based policies, may require greater coordination among the security servers.

The security policy encapsulated by the Flask security server is defined through a combination of its code and a policy database. Any security policy that can be expressed through the prototype's policy database language may be implemented simply by altering the policy database. Supporting additional security policies requires changes to the security server's internal policy framework through code changes or by completely replacing the security server. It is important to note that even security policies that require altering the code of the security server do not require any changes to the object managers.

The current Flask security server prototype implements a security policy that is a combination of four subpolicies: multi-level security (MLS) [3], type enforcement [6], identity-based access control and dynamic role-based access control (RBAC) [10]. The access decisions provided by the security server must meet the requirements of each of these four subpolicies. The policy logic for the multi-level security policy is largely defined through the security server code, aside from the labels themselves. The policy logic for the other subpolicies is primarily defined through the policy database language. These four subpolicies are not all the policies supported by the architecture or its implementation in Flask. They were chosen for implementation in the security server prototype in order to exercise the major features of the architecture.

Because the Flask effort has focused on policy enforcement mechanisms and the coordination between these mechanisms and the security policy, the set of additional security policies that can be implemented solely through changes to this policy database is currently limited. This is simply a shortcoming of the current prototype rather than a characteristic of the architecture. We have yet to explore the development of a more expressive policy specification language or policy configuration tool for Flask. Such a tool would facilitate the definition of new security policies in the current prototype. There have been several recent projects that do consider flexible tools for configuring the security policies (e.g., Adage [53], ASP [8], Dynamic DTE [15], ARBAC [41]) that nicely complement the Flask effort by potentially providing ways to manage the mechanisms provided by Flask.

6 Results

This section describes the results of the effort in three areas: policy flexibility, performance impact, and the scale and invasiveness of the code changes.

6.1 Flexibility in the Flask Implementation

We evaluate the policy flexibility that the system provides based upon the description of policy flexibility in Section 2. The most important criterion discussed in that section was "atomicity," i.e., the ability of the system to ensure that all operations in the system are controlled with respect to the current security policy. Section 5.4 described how the Flask architecture provides an effective atomicity for policy changes and how the microkernel in particular achieves atomicity for policy changes relating to its objects. Achieving this atomicity for the other object managers remains to be done.

Section 2 also identifies three other potential weaknesses in policy flexibility. The first is the range of operations that the system can control. As described in Section 5.3 and Appendix A, each Flask object manager defines permissions for all services which observe or modify the state of its objects and provides fine-grained distinctions among its services. The advantages of the Flask controls over merely intercepting requests were clearly illustrated.

The second potential source of inflexibility is the limitation on the operations that may be invoked by the security policy. In Flask, the security server may use any of the interfaces provided by the object managers. Furthermore, the Flask architecture provides the security server with the additional interfaces provided by the AVC module in each object manager. However, this is obviously not the same as having access to any arbitrary operation. For example, if the security policy requires the ability to invoke an operation which is strictly internal to some object manager, the object manager would have to be changed to support that policy.

The third potential source of inflexibility is the amount of state information available to the security policy for making security decisions. Based upon our previous analysis of policies for DTOS, the provision of a pair of SIDs is sufficient for most policies [43, Sec. 6.3]. However, the limitation to two SIDs is a potential weakness in the current Flask design. The description of the Flask file server in Section A.1 identifies one case where a permission ultimately depends upon three SIDs and must be reduced to a collection of permissions among pairs of SIDs. An even worse situation is if the security decision should depend upon a parameter to a request that is not represented as a SID. Consider a request to change the scheduling priority of a thread. Here the security policy must certainly be able to make a decision based in part on the requested priority. This parameter can be considered within the current implementation by defining separate permissions for some classes of changes, for instance, increasing the priority can be a different permission than decreasing the priority. But it is not practical to define a separate permission for every possible change to the priority.

This is not a weakness in the architecture itself, and the design could easily be changed to allow for a security decision to be represented as a function of arbitrary parameters. However, the performance of the system would certainly be impacted by such a change, because an access vector cache supporting arbitrary parameters would be much more complicated than the current cache. A better solution may be to expand the interface only for those specific operations that require decisions based upon more complex parameters, and to provide separate caching mechanisms for those decisions. The Flask prototype provides a research platform for exploring the need for a richer interface to better support policy flexibility.

6.2 Performance

All measurements in this section were taken using the time-stamp counter register on a 200MHz Pentium Pro processor with a 256KB L2 cache and 64MB of RAM. While a complete assessment of performance requires analysis of all object managers, we limit ourselves to the microkernel, and primarily to IPC since it is a critical path which must be factored into all higher level measurements.

6.2.1 Object Labeling The segment SID for any piece of mapped physical memory is readily available, since it is computed when a virtual-to-physical address translation is created and is stored along with that translation. As the address translation must be obtained at object creation time anyway, the additional cost of labeling is minimal. We verified this by measuring the cost to create the simplest kernel object in both Fluke and Flask, showing the worst case overhead. Flask added 1% to the operation (3.62 versus 3.66 μs).

6.2.2 IPC Operations This section presents performance measurements for IPC operations under various message sizes and also measures the impact of caching within the microkernel. Table 2 presents timings for a variety of client-server IPC microbenchmarks for the base Fluke microkernel and under different scenarios in the Flask system. The tests measure cross-domain transfer of varying amounts of data, from client to server and back again.

For all of the tests performed on Flask in Table 2, the required permissions are available in the access vector cache at the location identified by a "hint" within the port reference structure. While we have provided the data structures to allow for fast queries of previously computed security decisions, we have not done any specific code optimization to speed up the execution. Therefore it was encouraging to find that the addition of these data structures alone is sufficient to almost completely eliminate any measurable impact of the permission checks.

		Flask		
message size	Fluke (μs)	naive	client identification	client impersonation
`''Null''`	13.5	+2%	+9%	+6%
`16-byte`	15.0	+2%	+4%	+6%
`128-byte`	15.8	+1%	+2%	+5%
`1k-byte`	21.9	+2%	+2%	+4%
`4k-byte`	42.9	+1%	+1%	+2%
`8k-byte`	78.5	+1%	+5%	+1%
`64k-byte`	503	+0%	+6%	+0%

Table 2: Performance of IPC in Flask relative to the base Fluke system. A "Null" IPC actually transfers a minimal message, 8 bytes in the current implementation. In *Fluke*, the tests use the standard Fluke IPC interfaces in a system configured with no Flask enforcement mechanisms. Absolute times are shown in this column as a basis for comparison. *Naive* runs the same tests on the Flask microkernel. In *client identification*, the tests have been modified to use the Flask-specific server-side IPC interface to obtain the SID of the client on every call. *Client impersonation* uses the client-side IPC interface to specify an effective SID for every call.

The most interesting case in Table 2 is the *naive* column, because it represents the most common form of IPC in the Flask system. Along this path there is only a single *Connect* permission check. The results show a worst-case 2% ($\sim$50 machine cycle) performance hit. As would be expected, the relative effect of the single access check diminishes as the size of the data transfer increases and memory copy costs become the dominating factor. The *client identification* column has a larger than expected impact due to the fact that, in the current implementation, the client SID is passed across the interface to the server in a register normally used for data transfer. This forces an extra memory copy (particularly obvious in the Null IPC test). The significant effect on large data transfers is unexpected and needs to be investigated. The *client impersonation* column shows the impact of checking both the *Connect* and *SpecifyClient* permissions.

The effect of not finding the permission through the hint is shown in Table 3, which presents the relative costs of retrieving a security decision from the cache and from the security server. The operation being performed is the most sensitive of the IPC operations, round trip of transfer of a "null" message between a client and a server and is consequently representative of the worst case.

The *cache* column shows that the use of the hint is significant in that it reduces the overhead from 7% to 2%.

		Flask			
	Fluke	using hint	using cache	calling trivSS	calling realSS
``Null''	13.5 μs	13.8 μs +2%	14.4 μs +7%	43.4 μs +221%	82.5 μs +511%

Table 3: Marginal cost of security decisions in Flask. The first two columns repeat data from Table 2, identifying the relative cost of Flask when the required permission is found in the access vector cache (AVC) using the hint. The third column is the time required when the hint was incorrect but the permission was still found in the AVC. The *trivSS* column is the time required when the permission is not found in the AVC, and a "trivial" security server, which immediately returns an access ruling with all permissions granted, is used. The *realSS* column is the time required when the permission is not found in the AVC and an access ruling is computed by our prototype security server.

The *trivSS* column shows a more than tripling of the time required in the base Fluke case. The IPC interaction between the microkernel and security server requires transfer of 20 bytes of data to the security server (along with the client SID) and return of 20 bytes. Since the permission for this IPC interaction is found using the hint, we see from Table 2 that over half of the additional overhead is due to the IPC. The remainder of the overhead is due to the identification of the request for a security decision, construction of the security server request in the kernel, and the unmarshaling and marshaling of parameters in the security server itself. The additional overhead in the *realSS* column compared to the previous case is the time required to compute a security decision within our prototype security server. Though no attempt has been made to optimize the security server computations, this result points out that the access vector cache can potentially be important regardless of whether interactions with the security server require an IPC interaction.

6.2.3 Revocation Operations The possible microkernel revocation operations are described in Section 5.4. For demonstration purposes we chose to evaluate the most expensive of those operations, IPC revocation. Table 4 shows the results with varying numbers of active connections. The large base case is due to the need to stop all threads in the system when an IPC revocation is processed. The Fluke kernel provides a mechanism to cancel a thread and wait for it to enter a stopped state when the kernel wishes to examine or modify the thread's state. The stop operation cannot be blocked indefinitely by the thread's activities nor by the activities of any other thread. Since a thread must be stopped prior to examination in order to ensure that it is in a well-defined state, the current Flask implementation must stop all threads when an IPC revocation is processed. Thus, the current implementation meets the completeness and timeliness requirements of the architecture but is quite costly. In contrast, the actual cost to examine and update the state

connections	revocation time
1	1.55 ms
2	1.56 ms
4	1.57 ms
8	1.60 ms
16	1.65 ms

Table 4: Measured cost of revoking IPC connections. A connection is established from a client to a server and then is immediately revoked. Increasing numbers of interposed threads are used to increase the work done for each revocation.

of the affected threads is small in relation, and as expected scales linearly with the number of connections. Changing the Fluke kernel to permit greater concurrency during the processing of a revocation request remains as future work.

The frequency of policy changes is obviously policy dependent, but the usual examples of policy changes are externally driven and therefore will be infrequent. Moreover, a performance loss in a system with frequent policy changes should not be unexpected as it is fundamentally a new feature provided by the system. Obviously, even these uncommon operations should be completed as fast as possible, but that has not been a major consideration in the current implementation.

6.2.4 Macrobenchmark A macrobenchmark evaluation of the Flask prototype is difficult to perform. Since Flask is a research prototype, it has only limited POSIX support and many of the servers are not robust or well tuned. As a result, it is difficult to run non-trivial benchmark applications. Nevertheless, we performed a simple comparison, running `make` to compile and link an application consisting of 20 `.c` and 4 `.h` files for a total of 8060 lines of code (including comments and white space), about 190KB total.

The test environment included three object managers (the kernel, BSD filesystem server and POSIX process manager) along with a shell and all the GNU utilities necessary to build the application (`make`, `gcc`, `ld`, etc.). The Flask configuration of the test includes the security server with the three object managers configured to include the security features described in Section 5.3 and Appendix A. For each configuration, we ran `make` five times, ignored the first run, and averaged the time of the final four runs (the initial run primed the data and metadata caches in the filesystem). To give a sense of the absolute performance of the base Fluke system, we also ran the test under FreeBSD 2.1.5 on the same machine and filesystem. Table 5 summarizes the experiment.

The slowdown for Flask over the base Fluke system is less than 5%. By running the Flask kernel with unmodified Fluke object managers (*Flask-FFS-PM*), we see that

OS Config	Time (sec)
BSD	18.6
Fluke	39.9
Flask	41.7 (4.5%)
Flask-FFS-PM	40.9 (2.5%)
Fluke-memfs	24.7
Flask-memfs	27.4 (11%)

Table 5: Results of running `make` to compile and link a simple application in various OS configurations. *BSD* is FreeBSD 2.1.5, *Flask-FFS-PM* is the Flask kernel with the unmodified Fluke filesystem server and process manager, and the *memfs* entries use a memory-based filesystem in place of the disk-based filesystem. Percentages are the slowdowns vs. the appropriate base Fluke configurations.

the overhead is pretty evenly divided between the kernel and the other object managers (primarily the filesystem server). However, this modest slowdown is against a Fluke system which is over twice as slow on the same test as a competitive Unix system (*BSD*). The bulk of this slowdown is due to the prototype filesystem server which does not do asynchronous or clustered I/O operations. To factor this out, we reran the tests using a memory-based filesystem which supports the same access checks as the disk-based filesystem. The last two lines of Table 5 show the results of these tests. Note that the Flask overhead has increased to 11%, as less is masked by the disk I/O latency.

Table 6 reports the number of security decisions that were requested by each object manager during testing of the *Flask* configuration and how those decisions were resolved. The numbers include all five runs of `make` as well as the intervening removal of the object files. These results reaffirm the effectiveness of caching security decisions, with well over 99% of the requests never reaching the security server.

6.2.5 Performance Conclusions

Initial microbenchmark numbers suggest that the overhead of the Flask microkernel mechanisms can be made negligible through the use of the access vector cache and local hints when appropriate. They also highlight the need for an access vector cache so that communications with the security server and security computations within the security server are minimized. They also point to several areas for potential optimization, such as the AVC implementation, the communications infrastructure and the prototype security server computations. A complete analysis of the effectiveness of the AVC remains as future work. Issues such as the optimal cache size and the sensitivity of the AVC hit ratios to policy changes remain to be explored.

Results of the simple macrobenchmark test are inconclusive. Although the performance impact numbers are encouraging (5–11% slowdown), the bad absolute performance of the prototype system cannot be ignored.

Object Manager	Total queries	Resolution using hint	Resolution using cache	Resolution calling SS
Kernel	603735	175585	428121	29
FFS	76708	N/A	76700	8
PM	892	N/A	890	2

Table 6: Resolution of requested security decisions during the compilation benchmark. Numbers are from the *Flask* configuration of Table 5 and includes all five runs of `make` and `make clean`.

More completely exploring the performance overhead of the Flask security architecture remains as future work, and will likely be done in the context of a Linux or OS-Kit implementation of the architecture. This will permit more realistic workloads to be measured.

6.3 Scale and Invasiveness of Flask Code

In Table 7 we present data that give a rough estimate of the scale and complexity of adding fine-grained security enforcement to the base Fluke components. Overall, the Fluke components increased in size less than 8%. Although the kernel increased the most at 19%, for large object managers the percentage is reassuringly small (4–6%). Of these modifications, we examined the magnitude of changes involved by classifying each changed location as “trivial” changes (*e.g.*, one-line changes, `#define` changes, name or parameter changes, etc.) or “non-trivial.” For the process manager, 57% of the changes fell into the trivial category. For the kernel, a similar percentage of the changes were trivial, 61%, despite the fact that the kernel is an order of magnitude larger and more complicated than the process manager.

The changes required to implement the Flask security architecture did not involve any modifications to the existing Fluke API. Extended calls were added to the existing API to permit security-aware applications to use the additional security functionality, such as the client and server identification support. All applications that run on the base Fluke system can be executed unchanged on Flask.

7 Summary

This paper describes an operating system security architecture capable of supporting a wide range of security policies, and the implementation of this architecture as part of the Flask microkernel-based operating system. It provides a usable definition of policy flexibility, identifies limitations of this definition and highlights the need for atomicity. It shows that capability systems and interposition techniques are inadequate for achieving policy flexibility. It presents the Flask architecture and describes how Flask overcomes the obstacles to achieving

Component	Fluke LOC	+Flask	%Incr.	#Locs.	%Locs.
Kernel	9271	1795	19.3	258	2.4
FFS	21802	1342	6.2	14	.06
Proc. Mgr	925	196	21.2	85	9.2
Net Server	24549	1071	4.4	224	9.1
Total	58435	4575	7.8	647	1.1

Table 7: "Filtered" source code size for various Flask components and the number of discrete locations in the base Fluke code that were modified. This count of source code lines filters out comments, blank lines, preprocessor directives, and punctuation-only lines, and typically is $1/4$ to $1/2$ the size of unfiltered code. The network server count includes the ISAKMP and IPSEC distributions, counting as modifications all Flask-specific changes to them and the base Fluke network component.

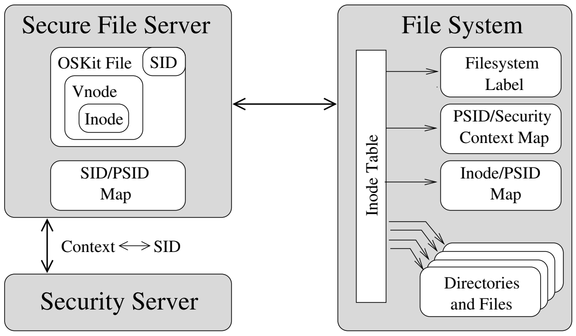

Figure 6: Labeling of persistent objects. The file server maintains a table within each file system which identifies the security context of the file system and every directory and file within the file system, thereby ensuring that the security attributes of these objects are preserved even if the file system is moved to another system. This table is partitioned into a mapping between each security context and an integer *persistent SID* (PSID) and a mapping between each object and its persistent SID. These persistent SIDs are purely an internal abstraction within the file system and have a distinct name space for each file system. Hence, persistent SIDs may be lightweight and the allocation of persistent SIDs may be optimized for each file system.

policy flexibility, including the need for atomicity. Although the performance evaluation of the Flask prototype is incomplete, this paper demonstrates that the architecture is practical to implement and flexible to use. Moreover, the architecture should be applicable to many other operating systems.

Availability

The Flask software and documentation are available at <http://www.cs.utah.edu/flux/flask/>.

A Other Flask object managers

This appendix describes the specific features that have been added to some of the Flask user-space object managers. Although the following subsections are not necessary for understanding the Flask architecture, they provide helpful insight into the details of providing policy flexibility in a complete system.

A.1 File Server

The Flask file server provides four types of controlled (labeled) objects: file systems, directories, files, and file description objects. Since file systems, directories and files are persistent objects, their labels must also be persistent. The binding of persistent labels to these objects is shown in Figure 6. The file server supports persistent labels without sacrificing policy flexibility or performance by treating security contexts as opaque strings and by mapping these labels to SIDs by a query to the security server for internal use in the file server. Control over file description objects is separated from control over the files themselves so that propagation of access to file description objects may be controlled by the policy. As noted in Section 3.1, the ability to control the propagation of access rights is critical to policy flexibility.

In contrast to the Unix file access controls, the Flask file server defines a permission for each service that observes or modifies the state of a file or directory. For example, whereas Unix permits a process to invoke *stat* or *unlink* on a file purely on the basis of the process' access to the file's parent directory, the Flask file server checks *Getattr* and *Unlink* permissions to control access to the file itself in addition to the directory-based permissions. Such controls are necessary to generally support nondiscretionary security policies. The Flask file server also supports fine-grained distinctions among services, such as separate *Write* and *Append* permissions for files and separate *Add_name* and *Remove_name* permissions for directories, which is important for supporting policy flexibility.

The file server provides operations to relabel files and directories, since the relabel operation has the potential of being much more efficient than merely copying such objects into new objects with different labels. There are a couple of complications of relabeling. First, migrated permissions pertaining to the file may need to be revoked. For instance, changing the SID of a file may affect the permission to write to a file that is stored in a file description object. Hence, all such permissions are recomputed and revoked if necessary. Second, a relabeling operation cannot be simply controlled through the SID of the client subject and the SID of the file, but must also involve the newly requested SID. This is addressed by requiring three permissions for a relabel to complete, as shown in Table 8. The provision of a single relabel operation is also helpful from a policy flexibility perspective, since the policy logic can be directly expressed in terms of any of these three possible SID pairs. In contrast, implementing the same policy logic in terms of the permis-

SOURCE	TARGET	PERMISSION
Subject SID	File SID	RelabelFrom
Subject SID	New SID	RelabelTo
File SID	New SID	Transition

Table 8: Permission requirements for relabeling a file. Additionally, the subject must possess *Search* permission to every directory in the path.

sions controlling operations involved in copying an object would be complicated by the much weaker coupling among the relevant SIDs.

The file server design proposes the use of the Flask architecture's polyinstantiation support for *security union directories* (SUDs); however, the design for SUDs has not yet been implemented. SUDs are a generalization of the partitioned directory approach taken by multi-level secure Unix systems for dealing with /tmp. The SUD mechanism is designed to use the polyinstantiation support to determine the preferred member directory for each client to access by default. However, unlike the simple partitioned directory approach, the SUD mechanism provides a unified view of all accessible members within the polyinstantiated directory to clients based upon access decisions between the client and the member directories.

As was noted in Section 3.2, file server operations provide a simple example of the problems with implementing security controls at the server's external interface. The Flask file server draws its file system implementation from the OSKit [12] whose exported COM interfaces are similar to the internal VFS interface [27] used by many Unix file systems. It was possible to implement the Flask security controls at that interface where these problems do not exist.

A.2 Network Server

Abstractly, the Flask network server ensures that every network IPC is authorized by the security policy. Of course, a network server cannot independently ensure that a network IPC is authorized by the policy of its node, since it does not have end-to-end control over data delivery to processes on peer nodes. Instead, a network server must extend some level of trust to its peer network servers to enforce its own security policy, in combination with their own security policies, over the peer processes. This requires a reconciliation of security policies, which would be handled by a separate negotiation server. The current negotiation server is limited to negotiating network security protocols and cryptographic mechanisms using the ISAKMP [33] protocol. The precise form of trust and the precise level of trust extended to peer network servers can vary widely and would be defined within the policy. Extending the concept of policy flexibility to a networked environment will require such support for complex trust relationships.

SOURCE	TARGET	LAYER
Process SID	Socket SID	Socket
Message SID	Socket SID	Transport
Message SID Node SID	Node SID Net Interface SID	Network

Table 9: Layered controls in the network protocol stack. Each layer applies controls based upon the SIDs of the abstractions directly accessible at that layer. Node SIDs are provided to the network server by a separate network security server, which may query distributed databases for security attributes, and network interface SIDs may be locally configured.

The principal controlled object type for the network server is the socket. For socket types that maintain message boundaries (*e.g.*, datagram), the network server also binds a separate SID to each message sent or received on a socket. For other socket types, each message is implicitly associated with the SID of its sending socket. Since messages cross the boundary of control of the network server, and may even cross a policy domain boundary, the network server may need to apply cryptographic protections to messages in order to preserve the security requirements of the policy and must bind the security attributes of the message to the message. Our prototype network server uses the IPSEC [26] protocols for this purpose, with security associations established by the negotiation server. The negotiation server may not pass SIDs across the network, since they are only local identifiers; instead, the negotiation server must pass the actual security attributes to its peer, which can then establish its own SID for the corresponding security context. Although the negotiation server must handle security contexts, it does not interpret them, and thus remains policy-flexible. Attribute translation and interpretation must be performed by the corresponding security servers in accordance with the policy reconciliation.

The network server controls are layered to match the network protocol layering architecture. Hence, the abstract control over the high-level network IPC services consists of a collection of controls over the abstractions at each layer, as shown in Table 9. The layered controls provide the policy with the ability to precisely regulate network operations, using all the information relevant to security decisions, and they allow the policy to take advantage of specific characteristics of the different protocols (*e.g.*, the client/server relationship in TCP). The network server provides another example of the problems with implementing security controls at the server's external interface. This is due to the need to control abstractions and interpose on operations which are not exported by the network server's external interface.

Since the TCP and UDP port spaces are fixed resources, the network server uses the Flask architecture's polyinstantiation support for *security union port spaces* (SUPs). SUPs are analogous to the SUDs discussed in Section A.1. The polyinstantiation support is used to determine the preferred member port space when a port number is associated with a socket and when an incoming packet has a destination port number which exists in multiple member port spaces. The SUP mechanism provides a unified view of all accessible port spaces within the polyinstantiated port space based on access decisions.

Many of the details of the Flask network server and other servers that support it are beyond the scope of this paper. A much more detailed description of an earlier version of the Flask network server can be found in [9].

A.3 Process Manager

The Flask process manager implements the POSIX process abstraction, providing support for functions such as *fork* and *execve*. These higher-level process abstractions are layered on top of Flask processes, which consist of an address space and its associated threads. The process manager provides one controlled object type, the POSIX process, and binds a SID to each POSIX process. Unlike the SID of a Flask process, the SID of a POSIX process may change through an *execve*. Such SID transitions are controlled by the process *Transition* permission between the old and new SIDs. This control permits the policy to regulate a process' ability to transition to different security domains. Default transitions may be defined by the policy through the default object labeling mechanism described in Section 5.2.1.

In combination with the file server and the microkernel, the process manager is responsible for ensuring that each POSIX process is securely initialized. The file server ensures that the memory for the executable is labeled with the SID of the file. The microkernel ensures that the process may only execute memory to which it has *Execute* access. The process manager initializes the state of transformed POSIX processes, sanitizing their environment if the policy requires it.

Acknowledgments

We especially thank Jeff Turner for his many contributions to the Flask vision and architecture. Duane Olawsky contributed much to our understanding of the features required for policy flexibility. We also thank Dan Wallach, Grant Wagner, Andy Muckelbauer, Ruth Taylor, Charlie Payne, Tom Keefe and the anonymous reviewers for reviewing earlier drafts of this paper, Roland McGrath for recent Fluke implementation, Ajay Chitturi for implementing an earlier version of our secure network server design, and other members of the Flux group for help in numerous ways.

References

[1] M. D. Abrams. Renewed understanding of access control policies. In *Proceedings of the 16th National Computer Security Conference*, pages 87–96, Oct. 1993.

[2] M. D. Abrams, L. J. LaPadula, K. W. Eggers, and I. M. Olson. A generalized framework for access control: An informal description. In *Proceedings of the 13th National Computer Security Conference*, pages 135–143, Oct. 1990.

[3] D. E. Bell and L. J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA, May 1973.

[4] T. C. V. Benzel, E. J. Sebes, and H. Tajalli. Identification of subjects and objects in a trusted extensible client server architecture. In *Proceedings of the 18th National Information Systems Security Conference*, pages 83–99, 1995.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[6] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, 1985.

[7] M. I. Bushnell. Towards a new strategy of OS design. *GNU's Bulletin*, 1(16), Jan. 1994.

[8] M. Carney and B. Loe. A comparison of methods for implementing adaptive security policies. In *Proceedings of the Seventh USENIX Security Symposium*, pages 1–14, Jan. 1998.

[9] A. Chitturi. Implementing mandatory network security in a policy-flexible system. Master's thesis, University of Utah, 1998. pp. 70. http://www.cs.utah.edu/flux/flask/.

[10] D. F. Ferraiolo, J. A. Cugini, and D. R. Kuhn. Role-Based Access Control (RBAC): Features and motivations. In *Proceedings of the Eleventh Annual Computer Security Applications Conference*, Dec. 1995.

[11] T. Fine and S. E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, May 1993.

[12] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[13] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and Execution Models in the Fluke Kernel. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 101–116, Feb. 1999.

[14] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Symposium on Operating Systems Design and Implementations*, pages 137–151, Oct. 1996.

[15] T. Fraser and L. Badger. Ensuring continuity during dynamic security policy reconfiguration in dte. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 15–26, May 1998.

[16] M. Gasser. *Building a Secure Computer Systems*. Van Nostrand Reinhold Company, 1988.

[17] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, July 1996.

[18] L. Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, May 1989.

[19] R. Graubart. On the need for a third form of access control. In *Proceedings of the 12th National Computer Security Conference*, pages 296–304, Oct. 1989.

[20] R. Grimm and B. N. Bershad. Providing policy-neutral and transparent access control in extensible systems. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1999.

[21] N. Hardy. The confused deputy. *Operating Systems Review*, 22(4):36–38, Oct. 1988.

[22] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the Seventh USENIX Security Symposium*, pages 143–157, Jan. 1998.

[23] R. Kain and C. Landwehr. On access checking in capability-based systems. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 66–77, May 1986.

[24] P. A. Karger. New methods for immediate revocation. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 48–55, May 1989.

[25] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, May 1984.

[26] S. Kent and R. Atkinson. Security architecture for the Internet Protocol. RFC 2401, Internet Engineering Task Force, Nov. 1998. ftp://ftp.isi.edu/in-notes/rfc2401.txt.

[27] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. of the Summer 1986 USENIX Conf.*, pages 238–247, Atlanta, GA, June 1986.

[28] C. R. Landau. Security in a secure capability-based system. *Operating Systems Review*, pages 2–4, Oct. 1989.

[29] R. Levin, E. Cohen, W. Corwin, P. F., and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 132–140, Unversity of Texas at Austin, Nov. 1975. ACM/SIGOPS.

[30] J. Liedtke. Clans and chiefs. In *Architektur von Rechensystemen*. Springer-Verlag, Mar. 1992.

[31] K. Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, Nov. 1992.

[32] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, Oct. 1998. http://-csrc.nist.gov/nissc/1998/proceedings/paperF1.pdf.

[33] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet security association and key management protocol (ISAKMP). RFC 2408, Internet Engineering Task Force, Nov. 1998. ftp://-ftp.isi.edu/in-notes/rfc2408.txt.

[34] C. J. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the pale of mac and dac - defining new forms of access control. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 190–200, May 1990.

[35] S. E. Minear. Providing policy control over object operations in a Mach based system. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, June 1995.

[36] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the Spring system. In *A Spring Collection*. Sun Microsystems, Inc., 1994.

[37] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 1997.

[38] D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and using a "policy neutral" access control policy. In *Proceedings of the New Security Paradigms Workshop*. ACM, Sept. 1996.

[39] E. I. Organick. *The Multics System : An Examination of its Structure*. MIT Press, 1972.

[40] S. A. Rajunas, N. Hardy, A. C. Bomberger, W. S. Frantz, and C. R. Landau. Security in KeyKOS. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 78–85, Apr. 1986.

[41] S. G. Ravi Sandhu, Venkata Bhamidipati and C. Youman. The arbac97 model for role-based administration of roles: Preliminary description and outline. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*, pages 41–50, Nov. 1997.

[42] D. Redell and R. Fabry. Selective revocation of capabilities. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 192–209, Aug. 1974.

[43] Secure Computing Corp. DTOS Generalized Security Policy Specification. DTOS CDRL A019, 2675 Long Lake Rd, Roseville, MN 55113, June 1997. http://-www.securecomputing.com/randt/HTML/dtos.html.

[44] Secure Computing Corp. Assurance in the Fluke Microkernel: Formal Security Policy Model. CDRL A003, 2675 Long Lake Rd, Roseville, MN 55113, Feb. 1999. http://www.cs.utah.edu/-flux/flask/.

[45] Secure Computing Corp. Assurance in the Fluke Microkernel: Formal Top-Level Specification. CDRL A004, 2675 Long Lake Rd, Roseville, MN 55113, Feb. 1999. http://www.cs.utah.edu/-flux/flask/.

[46] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Assoc.

[47] J. S. Shapiro. EROS: A capability system. Technical Report Technical Report MS-CIS-97-04, University of Pennsylvania, Department of Computer and Information Science, 1997.

[48] D. F. Sterne, M. Branstad, B. Hubbard, and B. M. D. Wolcott. An analysis of application specific security policies. In *Proceedings of the 14th National Computer Security Conference*, pages 25–36, Oct. 1991.

[49] SunSoft, Inc. *Spring Programmer's Guide*, 1995. On-line documentation included in the Spring Research Distribution 1.0.

[50] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 116–128, Oct. 1997.

[51] R. M. Wong. A comparison of secure unix operating systems. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, pages 322–333, Dec. 1990.

[52] W. Wulf, R. Levin, and P. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[53] M. E. Zurko and R. Simon. User-centered security. In *Proceedings of the New Security Paradigms Workshop*, Sept. 1996.