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)

Univ. of Utah Technical Report UUCS-98-014
August 1998

Abstract

Operating systems must be flexible in their support for security policies, i.e., the operating system must provide sufficient mechanisms for supporting the wide variety of real-world security policies. Systems claiming to provide this support have failed to do so in two ways: they either fail to provide sufficient control over the propagation of access rights, or they fail to provide enforcement mechanisms to support fine-grained control and dynamic security policies. In this paper we present an operating systems security architecture that solves both of these problems. The first problem is solved by ensuring that the security policy (through a consistent replica) is consulted for every security decision. The second problem is solved through 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 OS, and the policy flexibility of the prototype is evaluated. We present initial evidence that the architecture's performance impact 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, 45, 49]. To be generally acceptable, any computer security solution must be flexible enough to support this wide range of security policies. This flexibility must be supported by the security mechanisms of the operating system.

Supporting flexibility in the operating system is a hard problem that goes beyond just the supporting of 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, there must be a facility for the revocation of previously granted access rights. Other systems that claim to support policy flexibility fail to adequately 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 OS, that successfully overcomes the hard problems of policy flexibility where other systems have failed. The cleaner separation of mechanism and policy specified in the security architecture facilitates policy flexibility. Flask includes a security policy server to make access control decisions and a policy-flexible enforcement framework in the microkernel and other object managers in the system. 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 policy types. It controls the propagation of access rights by ensuring that the security policy, through a consistent replica, is consulted for every access decision. Enforcement mechanisms directly integrated into the service-providing components of the system enable fine-grained access controls and dynamic policy support that allows the revocation of previously granted access rights. Initial performance results 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 that purport to be policy flexible are really limiting to policy flexibility, some

---

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement number F30602–96–2–0269.

Contact information: spencer@securecomputing.com, Secure Computing Corp., 2675 Long Lake Road, Roseville, MN 55113-2536;
{pal,sds}@epoch.ncsc.mil; {mike,danderse,lepreau}@cs.utah.edu,
Dept. of Computer Science, 50 S. Central Campus Drive, Rm. 3190, University of Utah, SLC, UT 84112-9205.
http://www.cs.utah.edu/projects/flux/fluke/html/flask.html

related work is described. The Flask security architecture is then presented. This is followed by a discussion on the design and implementation of the prototype system, including an evaluation of the policy flexibility of the system. The paper concludes with a section discussing the performance impact and what was done to mitigate it.

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 because this list cannot be generated. Real-world security polices in computer systems are limited by the facilities 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, 49]. As such, a more useful 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. 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 reasonable expectation is that a system be able to control certain operations based on that portion of the current system state that is relevant to that operation with respect to the security policy. This yields a new definition of security policy flexibility which is used throughout this paper: A system is considered to provide total security policy flexibility if the security policy can interpose atomically on any of a defined set of controlled operations, allowing the operation to proceed, denying the operation or even injecting certain operations of its own. In such a system, the security policy can make its decisions using knowledge of a defined portion of the current system state. The degree of flexibility in a system will naturally depend upon the completeness of defined set of controlled operations. Furthermore, the granularity of the controlled operation affects the degree of flexibility because it impacts the granularity at which sharing can be controlled.

This definition seems limiting in three areas. It allows some operations to proceed outside of the control of the security policy, places limitations on the operations available to be injected by the security policy, and permits some system state to exist beyond the scope of the security policy. In actuality, however, each of these apparent limitations is in fact 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 7.1 will discuss how these limitations were interpreted for the Flask system.

Security policies may be classified according to certain characteristics, including such things as: the ability to revoke previously granted accesses, the type of input required to make access decisions, the sensitivity of policy decisions to the external factors, and the transitivity of access decisions [45]. Supporting policy flexibility requires that a system be capable of supporting security policies with a variety of these characteristics. The remainder of this section focuses on the most difficult of these to provide, revocation.

An essential element of policy flexibility is that all controlled operations are controlled according to the current security policy, which means that there must be effective atomicity in the interleaving of policy changes and controlled operations. A system that enforces access rights that are not current can hardly be considered to be policy flexible, since even the simplest security policies undergo change (e.g., as user authorizations change). The fundamental difficulty in achieving this atomicity is ensuring that permissions can be revoked after they are granted.

When a permission is to be revoked, it is necessary for the system to ensure that any service controlled by the permission will be no longer provided by the system unless the permission is granted again later. 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 permission check to write to a file is done when that file is opened, but the granted permission is stored in a file description, and write access in that description is permission to write to the file. Revoking the right to write to that file by revoking the right to open it may not have the desired effect unless the migrated permission in the file description can also be revoked. 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.

When a revocation of a permission is deemed necessary by the security policy, the revocation mechanism must know how that permission has migrated through the system. For this to be possible, it is essential that the service a permission controls be explicitly defined. For instance, write permission to a file may control a request to open a memory-mapped file with write permission. As in the preceding example, the important operation for revoking that permission may be the actual writing to the file rather than the opening of the file. On the other hand, in a local caching file system the important operation for write permission may not actually be writes to the cache (as long as it is private) but flushes of the cache back to the file server.

In most cases, revocation can be accomplished simply by altering a data structure. However, revocation requests involving migrated permissions associated with an operation in progress are more complicated. 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 it immediately, returning an error status. Alternatively, it could be restarted allowing a permission check for the retracted permission to be generated. The third option is to just wait for it to complete on its own. In general, only the first two are safe. Only when the system can guarantee that the request 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.

As a final note, policy flexibility may not demand that all permissions be revocable. For instance, there may be permissions that are necessary for the system to be even minimally functional.

3 Insufficiency of Popular Mechanisms

This section discusses two popular mechanisms that are often employed within systems purporting to be policy-flexible, and the reasons why both are limiting to policy flexibility in normal usage. However, each has benefits despite their limitations, and both can be used within Flask in a restricted way that allows some of their benefits without the policy 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 introduced a security model explicitly based upon the separation of access control mechanisms from the definition of a security policy [31, 52]. Hydra was a capability based system, though the developers of the system recognized the limitations of a simple capability model and introduced several enhancements to the basic capability mechanisms in order to satisfy a specific collection of security policies. The result was that while Hydra could support several interesting security policies, there were no general mechanisms or principles that could be invoked to support policies outside of these specific policies. And for some of the policies that were supported, such as confinement, the solution was so limiting as to be impractical [52, Ch. 7].

The Hydra approach was taken even further by the KeyKOS system [41], which introduced the concept of a factory [21] to solve the mutual exclusion problem and the KeySAFE [30] system which was intended to support multilevel security. Much of the security philosophy and mechanisms of the KeyKOS system are being carried forward into the EROS [48] system.

Unfortunately, the ability of Hydra and KeyKOS to support several security policies has been interpreted as demonstrating that capability based systems are naturally suited to providing policy flexibility. Indeed, capability-based systems have been described as policy-flexible even though they provide few or none of the enhancements that were incorporated into Hydra and KeyKOS. This ignores the fact that these mechanisms were embedded into the basic function of the capability system of each system, and were absolutely necessary for those systems to support many of the security policies that they were capable of supporting.

Despite their popularity, capability mechanisms are poorly suited to providing policy flexibility. The basic function of a capability mechanism is to 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 mechanisms that are provided by Hydra and KeyKOS to support various policies are all intended to limit the propagation of access rights in various ways. The resulting systems still generally only satisfy the specific policies that they are designed to satisfy, and at the cost of significant complexity that diminishes the attraction of a 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 [27], ICAP [19], Trusted Mach [4]) introduced mechanisms that validated every propagation or use of a capability against the security policy. Kain and Landwehr [25] 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

Another mechanism commonly used in systems claiming to be policy flexible is to intercept requests or to otherwise interpose a layer of security code between all applications and the operating system (e.g., Kernel Hypervisors [38], SPIN [20]), or between particular applications or sets of applications (e.g., L3/L4 [32], Lava [24], KeySAFE [30]). 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 [37]) or in a component outside of the operating system to which all requests are redirected (as in Janus [18]).

There are several very attractive aspects of this approach. Security code is localized, making it easier to generate and verify. It may even be possible to generate much of the security code automatically from an interface language compiler. By interposing across an interface, it is also possible to change a request or the return values from a request. Best of all, this approach can be applied to existing systems with minimal modification.

But this approach also has some shortcomings that are not always acknowledged. The objects that should be involved in access control are often not directly accessible at the interface. For instance, the parameters to a request to access a file include a directory and a path name from that directory, while the access controls should involve all of the interim directories. Solutions to this problem can be cumbersome and require care to ensure that the actual objects being accessed are the same as the objects to which the access controls are applied prior to the request being forwarded. Another problem is the potential for inefficiencies, especially when the security code executes in a separate process, which is often necessary when using this approach on existing systems.

The biggest disadvantage of this approach is 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. But it is important to recognize that other mechanisms are necessary, often mechanisms that are more invasive, in order to provide any degree of flexibility to support dynamic security policies.

4 Related Work

This section describes the relationship between Flask and some other efforts not previously mentioned in Section 3. The specific issue of revocation is also not a new issue in operating system design, though it has received surprisingly little recognition. Multics [10] effectively provided immediate revocation of all memory permissions by invalidating segment descriptors. Redell and Fabry [43], Karger [26] and Gong [19] all describe approaches for revoking previously granted capabilities, though none were actually implemented. Spring [50] 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 [34], 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 [40].

While the Flask effort is focused on policy enforcement mechanisms and coordination between these mechanisms and the security policy, several recent projects consider policy-flexible tools for configuring the security policy itself (e.g., Adage [53], ASP [8], Dynamic DTE [17], ARBAC [42]). These projects nicely complement the Flask effort by potentially providing a way to manage the mechanisms provided by Flask.

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 [47] and the Java protection models in [51]) 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.

5 Flask Security Architecture

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 security architecture of the Flask system is derived from DTOS [36], which had a similar goal of policy flexibility in addition to the goals of application transparency, defense-in-depth, ease of assurance, and minimal policy-specific code changes. However, while the DTOS security mechanisms were independent of any particular security policy, it became clear as that project progressed that the mechanisms were not sufficiently rich to support some policies [45], especially dynamic security policies.

The Flask security architecture is described here using the language of a microkernel-based multiserver system, since it has been implemented in such a system. However, the security architecture only requires two properties in the underlying system. The first property is that the underlying system must provide separation between subjects and objects such that unbypassable access controls can be added to mediate all accesses to those objects. The second is that

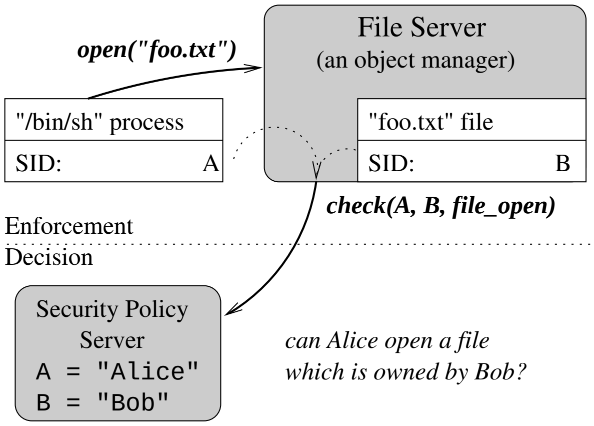

Figure 1: An example request and associated security decision. Object managers operate only on objects with opaque SIDs. The object manager obtains the client SID via the system's secure subject identification facilities.

the system must be capable of providing secure identification of the subject accessing the object for use in performing access decisions. Unix and safe extensible systems such as SPIN [5] meet these requirements, while any system with no enforced subject/object separation does not.

The basic Flask architecture is very simple. The system consists of object managers (such as a file server and process manager) which provide the system's controlled operations and a security server which provides security decisions for a particular security policy. The object managers are responsible for enforcing the security decisions. The object managers are policy independent, and only the security server may need to be altered when a different security policy is desired.

However, this very simple model begins to get complicated as we consider the interactions between the two kinds of components. One complication is that the decision whether to allow a controlled operation is usually dependent upon the identity of objects being accessed. Thus the security policy must have some knowledge about all objects in the system. This is accomplished by associating a security identifier (SID) with every object that may be part of an access decision. The object manager must maintain this mapping, while the security server defines the default SID that is associated with an object when it is created. A second complication is that the decision whether to allow a controlled operation is also usually dependent upon the identity of the client attempting to perform the operation. Since the microkernel is the basic provider of communication services, it provides mutual identification between clients and servers through their SIDs. Both complications were similarly addressed in DTOS. These components of the architecture are illustrated in Figure 1.

The most difficult complication in the architecture, however, is that the object managers effectively keep a local copy of certain security decisions, both explicitly in a cache used to minimize the need for security computations 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. Because this third complication was never fully addressed by the DTOS project, the remainder of this section will be devoted to a more detailed discussion of the requirements on the components of the architecture during a change in security policy.

We could state a requirement that when a policy change is triggered, the system enters into a non-interruptible state in which no other actions can occur until the change is complete and reflected in all object managers as well as the security server. This would provide the atomicity guarantee of Section 2 as atomicity between the request for a policy change within the security server and the actual policy change. However, we do not make such a strict interpretation, not just because it is impractical but also because it is unnecessary. A sufficient form of atomicity may be defined by imposing two requirements on the system. The first requirement is that after completion of the policy change, the behavior of the object manager must reflect that change. No further controlled operations requiring a revoked permission will be performed without a subsequent policy change. Policy changes require communication between the security server and object managers. The security server notifies each affected object manager that a policy change has been requested, and the object manager responds when the change is completed. 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.

This less stringent form of atomicity is reasonable because there is a second requirement imposed on the object managers. This requirement is that object managers must complete policy changes in a timely manner. It must not be possible for the revocation request to be arbitrarily delayed by actions of untrusted software. 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. Interdependencies among object managers may also interfere with meeting this requirement and must be carefully analyzed.

6 Flask Design and Implementation

The Flask prototype was derived from the Fluke microkernel-based operating system [16]. Although the

Flask architecture is not limited to a microkernel-based system, a microkernel-based system has certain advantages from a security, assurability and flexibility perspective. Despite the unresolved performance controversy of microkernels [24, 5, 47, 23, 33], these advantages warranted further experimentation. The enforcement mechanisms of the microkernel permit many threats to be transparently addressed directly by the microkernel and can provide defense-in-depth even in the event of a lapse of security in a server. The confinement of the operating system servers by the microkernel limits the trust that must be placed in each component of the operating system, simplifying the assurance analysis for each server. Furthermore, as composability theory [12, 44] advances, it should become possible to decompose the system into modules which are independently evaluatable. Since microkernel-based operating systems are already designed with the philosophy of separating policy from mechanism and permitting easy replacement of system components, the notion of a replaceable security policy server fits well within the existing model.

In addition to the general advantages of a microkernel-based operating system, the Fluke microkernel itself is especially well-suited for implementing the Flask architecture due to the lack of global resources [16] and the atomic properties of its API [15]. However, the original Fluke system was capability based and was not in itself adequate to meet the requirements of the Flask architecture. Each of the following subsections describes the changes required by the architecture.

6.1 Object Labeling

All objects that are controlled by the security policy are also labeled by the security policy with a *security context*. The interpretation of a security context is policy dependent but can indicate attributes such as usernames, roles, sensitivity labels (e.g., unclassified, secret, top secret) and data types. The mapping between an object and its security context is maintained in two steps. The security server assigns an integer value, the *security identifier* (SID), to each security context and the object manager maintains the mapping between each object and its SID. The SID allows all 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.

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. Similarly, for an *execve* call, the security context of the transformed process is dependent upon the security context of the executable file and the security context of the calling process. This SID is computed by the security server at the request of the object manager, which provides the type of the object and the SIDs of the related objects. For some security policies, such as a separation of duty policy [39], the security policy must 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.

There are two situations in which the security server may not be called directly to compute the SID of a new object. Objects that are local to a subject, such as newly allocated memory, are simply assigned the SID of the subject. Also, a policy-aware subject can request that a particular SID be assigned to a new object, though this request must still be approved by the security server.

6.2 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. However, this feature is not complete without providing the client and server with 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 [22]. 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 server connecting the client and server in a distributed system [13].

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 [50] 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 a 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 controlled by the security server.

6.3 Requesting and Caching Security Decisions

A typical control for an operation in Flask must determine whether a subject is allowed to access a object with some permission or set of permissions. An object manager poses this question to the security server by providing the SID of the subject, the SID of the object and the set of requested permissions. The security server responds with a yes/no decision for each of the requested permissions. To minimize the overhead of security computations and requests, the security server can provide more decisions than requested, and the object manager will cache 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 all 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.

An access vector does not provide simply a yes/no answer for each permission, as the traditional model of an access control matrix [29] is too limiting. There is an third possible value, undecided, for each permission. This is necessary for policies that include any kind of dynamic conflict of duty constraints. As a simple example, consider a policy that requires that a purchase order and payment for that order are authorized by different individuals. The policy cannot decide which of these operations to allow and which to deny for an individual until one of the operations is explicitly requested. Because there are three possible values for each permission, the return from the security server actually contains two access vectors. Each bit in the *decided* access vector indicates whether the corresponding permission has been decided. If so, then the *allowed* access vector indicates whether the permission is granted or denied.

Each object manager contains an access vector cache implemented as a common library. An entry in the access vector cache contains the two access vectors received from the security server, indexed by the two SIDs and the identity of the collection of permissions that the access vectors represent (e.g., file permissions).

6.4 Coordinating Policy Changes

A policy change is accomplished through three steps. The security server notifies all object managers that may have been previously provided any portion of the policy that has changed. Each object manager updates its internal state to reflect the change. Each object manager notifies the security server that the change is complete. This step is essential to support policies that require policy changes to occur in a particular order. For instance, permission to alter a purchase order must be removed prior to allowing the purchase order to be approved.

Since messages providing policy decisions to the object managers and messages requesting changes to the policy may not arrive in the same order in which they are sent, a sequence number is added to all messages from the security server to the object manager. An object manager will not accept a message providing policy decisions if it is received after a higher ordered message changing that portion of the policy. This does not require absolute sequencing of messages from the security server, but only that the sequence number be incremented for each policy change.

The general access vector cache library 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, which is currently only implemented within the microkernel and will be discussed in Section 6.6.

An alternative approach to coordinating policy changes is to use revocation lists. Though our approach requires additional activity by the object manager at the time of the policy change, it only requires a simple comparison of sequence numbers when a policy decision is received from the security server, rather than a search of a revocation list.

6.5 Policy Enforcement

The previous sections described the security functions that are common to all of the Flask object managers. In this section, we discuss the manager-specific features that have been added to the Flask object managers. Support for revocation, however, will be discussed separately in Section 6.6.

6.5.1 Microkernel All of the state of the Flask microkernel is encompassed in six primitive object types provided for three services: execution (threads), memory management (address spaces, mappings, regions) and IPC (ports, port sets). (The three other Flask object types are not security-relevant.) Due to the requirements of Fluke's architecture, each active primitive object is associated with a small chunk of physical memory [16]. 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 basic object is identical to the SID of the memory segment with which it is associated. Once assigned, the SID of an object may not be changed.

The microkernel provides several simple controls over its services. Creating microkernel objects or reading and writing to their state are controlled by *Create*, *Read* and *Write* permissions. Direct memory accesses are controlled through *Read*, *Write* and *Execute* permissions. To support containment of the program counter, *Read* permission can be granted without simultaneously granting *Execute* permission. Memory permissions cannot be computed at the level of any interface, and are computed instead during page faults. An IPC connection between two subjects is allowed only if three permissions are satisfied. Each subject that requests an effective SID must have *Specify* permission to the effective SID. Then *Connect* permission must be granted between the effective SIDs of the two subjects. To minimize the performance impact on an IPC, the state of a port reference (capability) includes a pointer to the access vector cache entry containing the permissions to the port from the subject that most recently accessed the reference. Of course, such hints are validated before use.

The most interesting aspect of the microkernel controls considers the relationships between objects. Setting the state of one microkernel object can have an indirect effect on the state of another. For instance, a thread is assigned to an address space by setting the state of the thread, but this has an obvious effect on the address space as well. Rather than requiring the subject that sets the state of the thread to have *Write* permission to the address space, which might be too permissive, the relationship between the thread and the address space is controlled explicitly. In the cases where one of the objects is essentially "owned" by another, like a thread and an address space (or port and port set), the microkernel verifies that the two objects have identical SIDs. In the other cases, the relationship between the two objects is controlled by an explicit permission. For example, there is a permission between two address spaces to control memory mapping from one address space to another. Therefore this operation requires two permissions: *Write* permission to the recipient address space and *Map* permission between the two address spaces.

6.5.2 Virtual Memory Manager The Flask memory manager interface provides two high-level abstractions of memory for use by applications. A *segment* is a contiguous piece of memory, either of fixed size or growable to a maximum limit. Address spaces are populated by mapping one or more segments. A *mempool* is a hierarchical resource control mechanism; segments and nested subpools are created in the context of a mempool and share the resources allocated to that pool. When a mempool is destroyed all contained mempools and segments are destroyed as well. The Flask Virtual Memory Manager (VMM) is a user-mode process that exports the memory manager interface, implementing demand-paged, virtual memory segments using the mempool resource limits as a basis for page replacement decisions.

The access checks controlling segments and mempools are straightforward. By default, mempools and segments are labeled with the SID of the creating subject though there are additional operations to create either with a particular SID. Individual permissions exist to control each exported mempool and segment operation, e.g. the ability to change the size of a segment. Since mempool and segment creation operations involve three potentially different SIDs (the SID of the creating subject, the SID of the parent mempool and the requested SID for the object), up to three distinct checks are performed.

The most significant complication posed by the Flask architecture involves the management of physical memory. The kernel controls the labeling of physical memory pages through segments it exports via the memory manager interface. When the VMM provides a segment with a particular SID, it must call the kernel to obtain appropriately labeled physical memory to resolve page faults for that segment. Thus the VMM's physical memory is fragmented into many different groups of physical pages, one per active SID. Because moving physical memory between those groups is relatively expensive (returning memory of one SID to the kernel and then allocating memory with a different SID) it is important to achieve a good balance between the competing groups and minimize inter-SID page replacement. Note that this would not be as great a problem in a system with a more conventional, kernel-based VM manager.

6.5.3 File Server The Flask file server provides four types of controlled (labeled) objects: file systems, directories, files, and open file objects. Entire file systems are labeled not only to control operations such as mounting and unmounting but to also represent the aggregate label of all files within the file system. Control over open file objects is separated from control over the files themselves so that propagation of access to open file objects may be controlled by the policy.

Since most file system operations are not as conceptually simple as a read or write operation, there are a large number of distinct permissions enforced by the file server, including 24 permissions on directory operations alone. This provides fine-grained access control with minimal additional performance impact since the security server can provide decisions on all of these permissions when just one is requested. This does, however, complicate the low-level specification of the security policy.

One unique aspect of the file server is that since files are persistent, the label on a file must also be persistent. This is accomplished by storing a mapping of between each security context used in that file system and an integer valued *persistent SID* within the file system. Although the file server maintains this mapping, it does not interpret the security contexts associated with the files.

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 [14] whose exported COM interfaces are similar to the internal VFS interface [28] used by many Unix file systems. It was possible to implement the Flask security controls at that interface where these problems do not exist.

6.5.4 Process Manager The Flask process manager is a user-mode process which implements the POSIX process abstraction, providing support for functions such as *fork* and *execve*. These higher-level process abstractions are layered on top of Fluke 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 Fluke process, the SID of a POSIX process may change through an *execve*. Such SID transitions are controlled by the *Transition* permission between the old and new SIDs. Default transitions may be defined by the policy through the default object labeling mechanism described in Section 6.1. In addition to ensuring that *Transition* permission is granted, the process manager must verify that the calling process has *Execute* permission to the executable file.

In combination with the virtual memory manager, 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.

6.5.5 Network Server The Flask network server [9] is a user-mode process that provides applications with access to the network through an interface based on the BSD socket system call interface. The network server draws its network protocol implementation from the OSKit [14]. Many of the details of the Flask network server and other servers that support it are beyond the scope of the paper.

Abstractly, the 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 is handled by a separate negotiation server using the ISAKMP [35] protocol. The precise form of trust and the precise level of trust extended to peer network servers can vary widely, and is defined within the policy.

The principal controlled object type for the network server is the socket. The network server binds a SID to each socket object. The default SID used for a socket object depends on the manner in which it is created. If it is created by a local process, then the SID of the creating process is used as the default. If it is created as a result of a new connection being accepted on an existing socket, then the SID of the existing socket is used as the default.

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 applies cryptographic protections to messages in order to preserve the security requirements of the policy and to bind the security attributes of the message to the message. Our prototype network server uses the IPSEC [2] protocols for this purpose, with security associations [2] 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. Attribute translation and interpretation must take place 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. For example, control over the transmission of a datagram from one process to another is provided through controls at the socket layer involving the process SID, the message SID and the socket SID and controls at the network layer involving the message SID, the destination node SID and the network interface SID. 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 security contexts may be locally configured. Due to the need to control abstractions that are not visible at the server interface or the OSKit network component interface, it was infeasible to add the security controls at these interfaces; consequently, modifications to the internals of the OSKit network component were necessary.

6.6 Revocation of Migrated Permissions

Revocation of migrated permissions has currently been implemented only within the Flask microkernel. Microkernel revocation is simplified by the fact that Fluke has a pure interrupt-model API [15]. This is useful in two ways. First, it means that the kernel provides prompt and complete exportability of thread state. Second, it means that kernel operations are either atomic or cleanly subdivided into user-visible atomic stages. 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.

If memory permissions are being revoked, Flask locates all memory translations that map memory from a segment with the specified segment SID into an address space with the specified address space SID, and updates the translation protection accordingly. If IPC *Connect* permission is being revoked, Flask locates all threads whose IPC state corresponds to the specified pair of effective SIDs and breaks their connections. If IPC *Specify* permission is being revoked, Flask locates all threads whose IPC state corresponds to the specified real SID and the specified effective SID and breaks their connections. When a connection is broken by a revocation, it will appear as though the connection was broken by the peer thread from the perspective of each of the formerly connected threads.

In the original Fluke implementation, the kernel provided a mechanism to cancel a target thread and wait for it to enter a stopped state when the kernel wished to examine or modify the thread's state. Flask likewise stops each thread prior to examining its IPC state during a revocation request. 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.

6.7 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, and to manage object manager access vector caches. In addition to the requirements listed above, most security policy servers will provide functionality for loading policies and changing policies. Because many operations depend on the results of a security server decision, in many cases it may be advantageous for the security server to provide its own caching of decisions in addition to the caching performed by the access vector cache in the object managers. The security server also is typically a policy enforcer over its own services.

In the remainder of this section we describe the security server implemented for the Flask prototype. The security policy encapsulated by this security server is defined through a combination of its code and a policy database. Changes to the security policy that can be expressed through the existing policy database language may be implemented simply by altering the policy database. More complex changes to the security policy may be implemented by altering the code of the security server or by completely replacing the security server.

The 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 [45] and dynamic role-based access control (RBAC) [11]. The access decisions provided by the security server must meet the requirements of each of these four subpolicies. The Flask security server differs from the DTOS security server in its approach to implementing MLS, its support for dynamic RBAC, and its enhanced support for identity-based access control. Changing the collection of subpolicies encapsulated within the security server is straightforward as long as the changes are consistent with the interdependencies among the subpolicies.

The *MLS policy* is a form of the Bell-LaPadula (BLP) model [3] which has been extended to support multi-level subjects and direct subject-to-subject interactions [46]. The policy database defines which users may be associated with which MLS ranges. An MLS access decision is computed by determining the relationship between the source MLS range and the target MLS range. The decision of whether or not to grant a permission is based upon this relationship and the information flow characteristics of the service that the permission controls.

The *type enforcement policy* explicitly defines an access matrix on the basis of types. Unlike typical representations of type enforcement, there is no distinction between domains and types in the type enforcement subpolicy of the security server. The policy database defines which users may be associated with which types. It also defines all allowed access vector permissions for each (source type, target type) pair. These type enforcement access rules encompass both the domain definition table and the domain transition table of traditional type enforcement [6]. A type enforcement access decision is computed by looking up and returning the allowed permissions for the (source type, target type) pair.

The *identity-based access control policy* is defined in the form of constraints that are imposed on particular access decisions based on the user identity attributes of the source and/or target. These constraints may be imposed on all access decisions for a given set of permissions or may be limited to access decisions involving certain (source type, target type) pairs for a given set of permissions. They may specify the relationship that must exist between the source user identity and the target user identity in order for a set of permissions to be granted or it may specify that a set of permissions is to be granted only if the source and/or target have a particular user identity.

The *dynamic RBAC policy* is defined by similar constraints, but these constraints are based upon the role attribute of the security context. The security server binds a role attribute to each SID, which reflects the active role associated with the SID. The policy database defines a partial order for the set of roles and a set of role-based constraints. Role-based constraints are similar to identity-based constraints except that role relationships are based on the partial order. One policy-specific interface was implemented to support the dynamic RBAC policy, the *change_role_set* function, which can be used to dynamically change a user's authorized role set. Since the authorized role set for a user is dynamic, the security server does not ensure that the user may be associated with the role when a SID is issued. Instead, it checks on each access decision where the role is relevant. If the user is not permitted to operate in the role, then any role-based constraint will fail, which leads to a denial of the constrained permissions. When a role is removed from a user's authorized role set, the security server notifies all relevant object managers of any formerly granted permissions that must now be denied. The security server waits for all relevant object managers to notify it that the changes have been completed and then notifies the caller of *change_role_set*.

7 Results

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

7.1 Flexibility in the Flask Implementation

We evaluate the policy flexibility which the system provides, based upon the description of policy flexibility in Section 2. The most important criteria 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 6.4 described the coordination of policy changes in Flask, and section 6.6 described how the Flask microkernel provides effective atomicity for the revocation of microkernel permissions. However, the implementation of revocation of migrated permissions in the other object managers has not yet been done.

Section 2 also identifies three other potential weaknesses in policy flexibility. The first is the range of operations that the system can control. In an object-based system such as Flask, there is generally not much confusion over the operations that must be controlled. The most difficult decisions regarding the operations to be controlled were within the network server because of differing needs at the different levels in the protocol stack. However, by generally allowing the access controls to be embedded within the object managers, Flask is able to provide more complete and fine-grained controls than systems that force all of the controls to be at the level of external interfaces.

The second potential source of inflexibility is the limitation on the operations available to the security policy. In Flask, the security server has access to all interfaces provided by the system, but this is obviously not the same as having access to any arbitrary operation. While a security server with complete control over the state of the object managers would clearly be more flexible, we know of no need for such a feature.

The third potential source of inflexibility is the amount of state information available to the security policy for making security decisions. This is a potential weakness in the current Flask design because all permissions are defined as a function of two SIDs. The description of the Flask virtual memory manager in Section 6.5.2 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.

Another potential source of inflexibility in the current Flask design stems from the assumptions encoded into the object managers with respect to object labeling. The potential for inflexibility is greater for object types where the SID of a new object is directly inherited from some related object rather than obtained from the security server. Furthermore, even in the case where the SID of a new object is computed by the security server, the degree of flexibility is limited by the fact that the object manager determines the set of related object SIDs provided as inputs to the computation, despite the fact that different policies may require different sets of related object SIDs. Finally, as with security decisions, the computation of a new object SID is a function of two SIDs, typically the SID of the creating process and the SID of a related object, which may likewise inhibit flexibility.

7.2 Performance

In implementing the system, our goal was to provide the basic mechanisms and data structures which would allow for fast queries of previously computed security decisions. This goal is primary because computation of a new security decision is only required when a permission is required between a new pair of SIDs or when a policy change occurs. Permissions between a new pair of SIDs are rarely required, because most subjects, even if they access many different objects or different pages of memory, access objects with only a few distinct SIDs, and new subjects are usually created with a SID of an existing subjects. 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.

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 particularly comforting to find that the addition of these data structures alone is sufficient to almost completely eliminate any measurable impact

message size	Fluke (μs)	Flask naive	Flask client identification	Flask 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 1: Performance of IPC in Flask relative to the base Fluke system.

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

Table 2: Marginal cost of security decisions in Flask.

of the permission checks.

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. Also, the microkernel is currently the only object manager with complete support for revocation, an important aspect of the security architecture.

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.

7.2.1 Object Labeling As with other managers, all objects within the microkernel are identified by SIDs. When a kernel object is created, it is labeled with the SID of the memory segment in which it resides. 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. Flask added 1% to the operation (3.62 versus 3.66 μs).

7.2.2 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 1 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. It should be noted that a "Null" IPC actually transfers a minimal message, 8 bytes in the current implementation. For all of the tests performed on Flask, the required permissions are available in the access vector cache at the location identified by the "hint" within the port structure. The effect of not finding the permission through the hint is investigated below.

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. This is the most interesting case because it represents the most common form of IPC in the Flask system, in which messages are sent using the unmodified Fluke IPC interfaces. 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.

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. The additional Flask processing in this case includes the *Connect* permission check and the passing of the SID (a 32-bit integer) across the server interface. The larger than expected impact in this case is 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.

Client impersonation uses the client-side IPC interface to specify an effective SID for every call. The additional Flask processing in this case includes the *Connect* and *Specify* permissions and the passing of the SID parameter across the client interface. For this request, the SID parameter is not passed through a data transfer register and the performance impact is predominately the result of the two access checks.

Table 2 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, transfer of a "null" message from client to server and back again.

The first two columns repeat data from Table 1, identifying the relative cost of Flask when the required permission is found in the access vector cache at the location identified by the hint in the port structure. The third column is the time required when the hint was incorrect but the permission was still found in the access vector cache. This shows that the use of the hint is significant in that it reduces the overhead from 7% to 2%.

The *trivSS* column is the time required when the permission is not found in the access vector cache, and a trivial

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

Table 3: Measured cost of revoking IPC connections.

security server is implemented to immediately respond by allowing all permissions. The result is a more than tripling of the time required in the base Fluke case. The IPC interaction between the microkernel and security server require transfer of 20 bytes of data to the security server and returning 18 bytes. Since the permission for this IPC interaction is found using the hint, we see from Table 1 that over half of the additional overhead is due to the IPC. The remainder of the overhead is due to the identification of the security fault, construction of the security server request in the kernel, and the unmarshaling and marshaling of parameters in the security server itself.

The *realSS* column is the time required when the permission is not found in the access vector cache and it is computed by the prototype security server. The additional overhead compared to the previous case is the time required to compute a security decision within this 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.

7.2.3 Revocation Operations The possible microkernel revocation operations are described in Section 6.6. For demonstration purposes we chose to evaluate the most expensive of those operations, IPC revocation. For the experiment we established a client to server connection, and have the server revoke the connection. We then add increasing numbers of interposed threads to increase the work done for each revocation.

Table 3 shows the results with 1 to 16 active connections. The large base case is due to the need to stop all threads in the system when an IPC revocation is processed, as explained in section 6.6. The actual cost to examine and update the state of the affected threads is small in relation, and as expected scales linearly with the number of connections.

7.2.4 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 identify the need for an access vector cache so that communications with the security server and security computations within the security server are minimized.

Extensive macrobenchmarking has not been performed since such measurements would be necessarily policy specific. However, we have measured compilation of a simple program using `gcc` with the existing security server and found that Flask increased the compilation time by 13%. This is encouraging since so few efforts have been made to optimize the Flask changes, but it should not be accepted as indicative of performance under other security policies.

Component	Fluke LOC	+ Flask	% Incr.	# Locs.	% Locs.
Kernel	9227	1758	19.0	229	2.4
FFS	21802	1342	6.2	14	.06
VMM	2006	159	7.9	100	4.9
Process Mgr	851	245	28.7	80	9.4
Net Server	24549	1071	4.4	224	9.1
Total	58435	4575	7.8	647	1.1

Table 4: "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.

7.3 Scale and Invasiveness of Flask Code

In Table 4 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%).

8 Summary

This paper describes an operating system security architecture for supporting a wide range of security policies with minimal, localized changes for each policy, and the implementation of this architecture as part of the Flask microkernel-based operating system. The most significant innovation in this work is the ability to support dynamic policies, and in particular, policies that require total revocation of previously granted permissions. Although performance evaluation of our particular prototype is incomplete, we have demonstrated that the architecture is practical to implement and flexible to use. Moreover, the architecture should be applicable to many other operating systems.

Availability We will be making a source distribution of the Flask OS by the date of the conference.

Acknowledgements

We especially thank Jeff Turner of RABA Technologies for his many contributions to the Flask vision and architecture. We thank Grant Wagner and Andy Muckelbauer for reviewing earlier drafts of this paper, Roland McGrath for recent Fluke implementation, Ajay Chitturi for implementing the secure network server, and other members of the Flux group for help in numerous ways.

References

[1] M. D. Abrams. Renewed Understanding of Access Control Policies. In *Proc. of the 16th National Computer Security Conference*, pp. 87–96, Oct. 1993.

[2] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, Internet Engineering Task Force, Aug. 1995. ftp://-ds.internic.net/rfc/rfc1825.txt.

[3] D. E. Bell and L. J. La Padula. Secure Computer Systems: Mathematical Foundations and Model. TR M74-244, The MITRE Corp., 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 *Proc. of the 1995 National Information Systems Security Conference*, pp. 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 SOSP*, pp. 267–284, Copper Mountain, CO, Dec. 1995.

[6] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proc. 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 *Proc. of the Seventh USENIX Security Symp.*, pp. 1–14, Jan. 1998.

[9] A. Chitturi. Implementing Mandatory Network Security in a Policy-flexible System. Master's thesis, Univ. of Utah, 1992. pp. 70.

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

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

[12] T. Fine. A Framework for Composition. In *Proc. of the Eleventh Annual Conference on Computer Assurance*, pp. 199–212, June 1996.

[13] T. Fine and S. E. Minear. Assuring Distributed Trusted Mach. In *Proc. IEEE Computer Society Symp. on Research in Security and Privacy*, pp. 206–218, May 1993.

[14] 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 SOSP*, pp. 38–51, St. Malo, France, Oct. 1997.

[15] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullman. Interface and Execution Models in the Fluke Kernel. Submitted to this conference., Aug. 1998.

[16] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. of the Symp. on Operating Systems Design and Implementations*, pp. 137–151, Oct. 1996.

[17] T. Fraser and L. Badger. Ensuring Continuity During Dynamic Security Policy Reconfiguration in DTE. In *Proc. of the 1998 IEEE Symp. on Security and Privacy*, pp. 15–26, May 1998.

[18] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proc. of the 6th Usenix Security Symp.*, July 1996.

[19] L. Gong. A Secure Identity-Based Capability System. In *Proc. of the 1989 IEEE Symp. on Security and Privacy*, pp. 56–63, May 1989.

[20] R. Grimm and B. N. Bershad. Providing Policy-Neutral and Transparent Access Control in Extensible Systems. TR UW-CSE-98-02-02, Univ. of Washington, Dept. of Computer Science and Engineering, Feb. 1998.

[21] N. Hardy. Computer Security System. U.S. Patent 4,584,639, Apr. 1986.

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

[23] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The Performance of u-Kernel-Based Systems. In *Proc. of the 16th ACM Symp. on Operating System Principles*, pp. 66–77, Oct. 1997.

[24] T. Jaeger, J. Liedtke, and N. Islam. Operating System Protection for Fine-Grained Programs. In *Proc. of the Seventh USENIX Security Symp.*, pp. 143–157, Jan. 1998.

[25] R. Kain and C. Landwehr. On Access Checking in Capability-Based Systems. In *Proc. of the 1986 IEEE Symp. on Security and Privacy*, pp. 66–77, May 1986.

[26] P. A. Karger. New Methods for Immediate Revocation. In *Proc. of the 1989 IEEE Symp. on Security and Privacy*, pp. 48–55, May 1989.

[27] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proc. of the 1984 IEEE Symp. on Security and Privacy*, pp. 2–12, May 1984.

[28] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proc. of Summer USENIX '86*, pp. 238–247, Atlanta, GA, June 1986.

[29] B. Lampson. Protection. In *Proc. Fifth Annual Princeton Conference on Information Sciences and Systems*, pp. 437–443. Princeton Univ., 1971.

[30] C. R. Landau. Security in a Secure Capability-Based System. *Operating Systems Review*, pp. 2–4, Oct. 1989.

[31] R. Levin, E. Cohen, W. Corwin, P. F., and W. Wulf. Policy/mechanism separation in Hydra. In *Proc. of the Fifth SOSP*, pp. 132–140, Unversity of Texas at Austin, Nov. 1975. ACM/SIGOPS.

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

[33] J. Liedtke. Achieved IPC Performance. In *Proc. of the 6th HotOS*, pp. 28–31, May 1997.

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

[35] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). Internet draft ipsec-isakmp-10, Internet Engineering Task Force, July 1998. ftp://ds.internic.net/internet-drafts/draft-ietf-ipsec-isakmp-10.txt.

[36] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proc. of the Fifth USENIX UNIX Security Symp.*, pp. 141–156, June 1995.

[37] 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.

[38] T. Mitchem, R. Lu, and R. O'Brien. Using Kernel Hypervisors to Secure Applications. In *Proc. of the Annual Computer Security Applications Conference*, Dec. 1997.

[39] M. J. Nash and K. R. Poland. Some Conundrums Concerning Separation of Duty. In *Proc. of the 1989 IEEE Symp. on Security and Privacy*, pp. 201–207, May 1990.

[40] D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and Using a "Policy Neutral" Access Control Policy. In *Proc. of the New Security Paradigms Workshop*. ACM, Sept. 1996.

[41] S. A. Rajunas, N. Hardy, A. C. Bomberger, W. S. Frantz, and C. R. Landau. Security in KeyKOS. In *Proc. of the 1986 IEEE Symp. on Security and Privacy*, pp. 78–85, Apr. 1986.

[42] S. G. Ravi Sandhu, Venkata Bhamidipati and C. Youman. The ARBAC97 Model for Role-Based Administration of Roles: Preliminary Description and Outline. In *Proc. of the Second ACM Workshop on Role-Based Access Control*, pp. 41–50, Nov. 1997.

[43] D. Redell and R. Fabry. Selective Revocation of Capabilities. In *Proc. of the International Workshop on Protection in Operating Systems*, pp. 192–209, Aug. 1974.

[44] Secure Computing Corp. DTOS Composability Study. DTOS CDRL A020, 2675 Long Lake Rd, Roseville, MN 55113, June 1997. http://-www.securecomputing.com/randt/HTML/dtos.html.

[45] 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.

[46] Secure Computing Corp. Assurance in the Fluke Microkernel: Formal Top-Level Specification. CDRL A004, 2675 Long Lake Rd, Roseville, MN 55113, May 1998.newblock http://www.cs.utah.edu/-s̃ds/synergy/microkernel/fluke/docs.html.

[47] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of the Second OSDI*, pp. 213–227, Seattle, WA, Oct. 1996.

[48] J. S. Shapiro. EROS: A Capability System. TR TR MS-CIS-97-04, Univ. of Pennsylvania, Dept. of Computer and Information Science, 1997.

[49] D. F. Sterne, M. Branstad, B. Hubbard, and B. M. D. Wolcott. An Analysis of Application Specific Security Policies. In *Proc. of the 14th National Computer Security Conference*, pp. 25–36, Oct. 1991.

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

[51] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proc. of the 16th SOSP*, pp. 116–128, Oct. 1997.

[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 *Proc. of the New Security Paradigms Workshop*, Sept. 1996.