Part Number 00-0930895A001 Rev B

Version Date 6 April 1999

Assurance in the Fluke Microkernel
Final Report

CONTRACT NO. MDA904-97-C-3047
CDRL SEQUENCE NO. A002

Prepared for:
Maryland Procurement Office

Prepared by:

Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113

Authenticated by ____________________ Approved by ____________________

(Contracting Agency) (Contractor)

Date ____________________ Date ____________________

Assurance in the Fluke Microkernel
Final Report

Secure Computing Corporation

Abstract

This document describes the results of the Assurance in the Fluke Microkernel program, and suggestions for further work.

Part Number	00-0930895A001 Rev B
Created	2 July 1997
Revised	6 April 1999
Done for	Maryland Procurement Office
Distribution	Secure Computing and U.S. Government
CM	`/projects/cmt/rcs/fluke/docs/final/RCS/final.vdd,v 1.2 6 April 1999`

This document was produced using the TEX document formatting system and the LATEX style macros.

Contents

1 Scope 1
1.1 Identification . . . 1
1.2 Program Overview . . . 1
1.3 Document Overview . . . 1

2 Program Summary 3
2.1 Background . . . 3
2.2 Objective . . . 4
2.3 Approach . . . 4
2.4 Significant Accomplishments . . . 5
2.4.1 Security Architecture . . . 5
2.4.2 Fluke Implementation . . . 6
2.4.3 Composition and Refinement Analysis . . . 7
2.5 Outputs . . . 7
2.5.1 Conference Papers . . . 7
2.5.2 Technical Reports . . . 8

3 Security Architecture 9

4 Dynamic Security Policy Study 12
4.1 A Model of Security Enforcement for Dynamic Policies . . . 12
4.2 Transactional Consistency . . . 14
4.3 Timeliness . . . 14

5 Fluke Implementation 15
5.1 Microkernel . . . 15
5.1.1 Object Labeling . . . 15
5.1.2 IPC Client and Server Identification . . . 15
5.1.3 Control Policy . . . 15
5.1.4 Access Vector Cache . . . 16
5.1.5 Support for Permission Revocation . . . 16
5.2 Operating System Servers . . . 17

6 Security Server 19
6.1 Policy Extensions . . . 19
6.2 Level Ranges for Subjects and Object . . . 20
6.3 Decoupling of Multilevel Security and Type Enforcement . . . 20
6.4 Role Based Access Control . . . 21

7 Composition Study 23

8 Future Work 25
8.1 Expanding the Flask Security Architecture . . . 25
8.1.1 Policy Layering . . . 25
8.1.2 Extending the Architecture to a Networked/Distributed System . . . 27
8.1.3 Other Architecture Enhancements . . . 28

8.2 Further Implementation . . . 28
8.3 Assurance . . . 29

A Bibliography 32

B FSPM Updates 34
B.1 Evolution of the Security Architecture . . . 34
B.1.1 Object Labeling . . . 34
B.1.2 Client and Object Manager Identification . . . 36
B.1.3 Control Policy . . . 36
B.1.4 Retrieving and Caching Security Decisions . . . 37
B.1.5 Changes to Security Decisions . . . 38
B.1.6 Polyinstantiation Support . . . 38
B.1.7 Audit . . . 38
B.2 Microkernel Control Policy Lessons Learned . . . 39
B.2.1 Microkernel State Model . . . 39
B.2.2 Service Definitions . . . 40
B.2.3 Requirements . . . 40

C FTLS Updates 42

List of Figures

1 DTOS Security Architecture Interfaces 34
2 Flask Security Architecture Interfaces 35
3 Fluke Security Architecture Interfaces as Documented in the FTLS 42

Section 1
Scope

1.1 Identification

This document contains the Final Report, CDRL A002, for the Assurance in the Fluke Microkernel program (hereafter referred to as the "Fluke Assurance" program), contract MDA904-97-C-3047.

1.2 Program Overview

The goal of the Fluke Assurance program is to support security in the Fluke operating system project at the University of Utah by applying known assurance techniques to assess the security and assurability of key elements of the Fluke technology, and by performing research to extend assurance techniques to address unique aspects of Fluke. The program is especially concerned with the implementation of Fluke known as Flask[1], in which the University of Utah and the National Security Agency have cooperated to augment Fluke with flexible access control mechanisms.

Flask is a successor to the DTOS program (described in the DTOS Lessons Learned Report [18]), under which Secure Computing and the NSA cooperated to prototype security enhanced versions of the Mach 3 microkernel and the Lites Unix server. That prototype has been used as a research platform for secure applications projects and as a basis for assurance research under the DTOS program. Flask is a comparable enhancement to the Fluke operating system, providing support for mandatory access control policies with a flexible design that separates access control decisions and enforcement into separate components.

1.3 Document Overview

This report begins with a summary of the objectives, accomplishments and outputs from the program, and then considers in turn each of the major technical areas of the program. The main body of the report concludes by providing suggestions for future efforts which build upon the successes of the program or address deficiencies. Two appendices provide updates to previously completed program documents.

More specifically:

- Section 1, **Scope**, defines the scope and this overview of the document.
- Section 2, **Program Summary**, provides a summary of the programs objectives, accomplishments and outputs.
- Section 3, **Security Architecture**, provides an overview of the security architecture developed under the program.

[1] Flask is sometimes considered to be an acronym of Fluke Advanced Security Kernel, though the effort is not limited to the Fluke microkernel.

- Section 4, **Dynamic Security Policy Study**, describes a study undertaken to determine the requirements necessary to support dynamic security policies. This study led to the most significant improvements in the security architecture as it evolved from the DTOS program.
- Section 5, **Fluke Implementation**, describes the implementation of the security architecture within the Fluke operating system.
- Section 6, **Security Server**, describes the innovations in the prototype security server implemented on the program.
- Section 7, **Composition Study**, discusses the results of the Composition Study.
- Section 8, **Future Work**, provides suggestions for future efforts which build upon the successes of the program or address deficiencies.
- Appendix A, **Bibliography**, provides citations for all referenced documents.
- Appendix B, **FSPM Updates**, provides two updates to the previously completed Formal Security Policy Model [20]. The first update describes the evolution of the security architecture from DTOS to Flask, including rationale for the changes. The second update provides a comparison between the experience of developing the control policy for the Fluke microkernel and the similar effort for DTOS.
- Appendix C, **FTLS Updates**, provides updates to the previously completed Formal Top Level Specification [22] to reflect the current implementation.

Section 2
Program Summary

This section provides a summary of the Fluke Assurance program. It is independent of the rest of this document[2] so that it can be distributed separately if so desired.

This summary begins with a brief overview of the larger Fluke project of which the Fluke Assurance program is one portion, followed by a description of the objectives specific to the Fluke Assurance program and the approach used to pursue these objectives. Significant results of the program are then identified, and we conclude with short descriptions of all papers and reports written during the program.

2.1 Background

Under the DTOS program, Secure Computing and NSA developed a security architecture intended to satisfy several goals:

- Flexibility to support a wide range of security policies
- Transparency to applications, except those specifically designed to be security aware
- Minimal performance impact by the security mechanisms
- Assurability through modular design

This architecture was prototyped on top of the Mach microkernel (from Carnegie Mellon University) and the Lites Unix server. Unfortunately, support for this version of Mach was discontinued, so a new platform for the prototype was needed. A cooperative effort was initiated between NSA and the University of Utah to improve the architecture and incorporate it into the Fluke operating system being developed at Utah, and SCC then became involved through the Fluke Assurance program. While the prototype is built on the Fluke operating system, the intent is to avoid relying upon specific features of Fluke, so that the results will be easily transferable to other operating systems and architectures.

Hence, the overall project combines the strengths of three organizations. The University of Utah brings the skills needed to develop and maintain the Fluke operating system, to rapidly adapt it when needed to address security or assurability concerns, and to assess and minimize the performance impacts of the security architecture. NSA has a thorough understanding of the implementation of the DTOS system and the rationale for its design, and led the effort to incorporate this design into the Fluke operating system. Secure Computing also brings knowledge of the DTOS system implementation but primarily contributes knowledge of the requirements for policy flexibility and experience in operating system assurance.

[2] Except for the bibliography, and the fact that it includes forward references identifying where to find more information.

2.2 Objective

The objective of the Fluke Assurance program, as stated in the program's proposal, was the following:

> The objective of the [Fluke Assurance] program is to support the security and assurability goals of the Fluke project by applying known techniques to help determine the assurance possible for key elements of the Fluke technology and by performing research to extend assurance techniques to address some of the unique aspects of Fluke. A key element of this effort is providing feedback on the assurance and security impact of Fluke design decisions.

This objective refers to the "security and assurability goals of the Fluke project", which are essentially the first and fourth goals described above for the DTOS program. To elaborate:

Security — The primary security goal is to prototype a policy flexible access control model based upon the DTOS security architecture, with a particular emphasis on dynamic security policy support. Other security features, such as polyinstantiation of shared resources and covert channel elimination, are generally given lower priority.

Assurability — The assurability goal is that critical security functions should be verifiable through the use of formal specifications and reasoning.

2.3 Approach

The program objective was pursued in two ways. Most of the tasks on the program were some form of analysis targeted directly at the Fluke prototype implementation, though we performed these tasks whenever possible with the intent that the results would be applicable to other systems as well. On the other hand, the System Composition Study task addressed the objective of extending assurance techniques for modular systems such as Fluke but did not attempt to analyze the Fluke implementation at all.

The kinds of analysis that were targeted at the Fluke implementation ranged from a simple informal assessment of design details proposed by the University of Utah or NSA to formal requirements and design analysis driving the implementation of a new feature. Analysis of basic security and assurability properties of the system were largely based upon the criteria described in Section 3 of the DTOS General System Security and Assurability Assessment Report [15]. Analysis of policy flexibility was largely based upon the policy characteristics described in Section 6 of the DTOS Generalized Security Policy Model [16], except that a new framework was developed to address dynamic security policies.

For the components of the system which were added to the Fluke system, formal methods (using PVS [9]) were applied in the analysis to increase confidence in the correctness and completeness of the analysis. In particular, formal methods were used to analyze the relationships between security policy decision making and enforcement. Formal methods were generally not used to analyze Fluke specific elements of the system to avoid the extra cost for analysis that is not transferrable to other systems.

The System Composition Study Task was directed specifically at the goal of extending assurance techniques, especially to address modular design practices. The task used formal methods and an example of a modular design for maintaining process integrity to explore the use of the DTOS Composition framework [14] (as expanded for the Composability for Secure Systems program [19]) for requirements analysis as part of the design refinement process.

2.4 Significant Accomplishments

The significant accomplishments achieved on this program are identified according to whether they apply generally to the security architecture used on the program, to the specifics of the Fluke implementation of the security architecture, or to the analysis of modular systems using composition and refinenement.

2.4.1 Security Architecture

There are two components in the DTOS security architecture, a central security server and an access vector cache (AVC) within each object manager. The central security server makes all access control decisions, and the AVC communicates with the security server to retrieve its decisions and stores those decisions for use within the object manager. The program improved and expanded upon this security architecture in several ways.

- We evaluated the interfaces between the AVC and the security server with respect to their ability to support the goal of policy flexibility, and found ways to simplify these interfaces (and the internal state maintained by the AVC) without loss of flexibility. We wrote formal specifications for these updated interfaces and the internal processing that supports them.

  This is discussed further in Section 3 and Appendix B.1.

- The most significant shortcoming of the DTOS security architecture was the inability to generally support revocation of permissions. On the Fluke Assurance program we addressed this shortcoming through a Dynamic Security Policy Study which included the following:
  - A high level formal model of a system consisting of a security server and an object manager with an AVC, including a statement of the property that the system correctly enforces a dynamic security policy.
  - Refinement of this property into requirements on the security server and object manager, including formal verification that the refinement is correct.
  - Formal description of a system design and formal verification that this design meets the requirements. This design was subsequently implemented in the Fluke system by the NSA.
  - Informal description of an alternate design intended to meet the high level property and other related properties not considered as part of the formal verification efforts.

  This is discussed further in Section 4.

- Prior to the start of the Fluke Assurance program, NSA and the University of Utah considered designs for incorporating a hierarchy of security policies and security servers into a single system, with the expected advantage of simplifying security administration and expanding flexibility through decentralization. However, these designs had deficiencies and only a very simplified (and considerably less flexible) design was eventually implemented. As part of the Fluke Assurance program, we revisited these issues and identified how one of the proposed designs could be changed to meet the needs of a policy hierarchy. Though this design has not been implemented, we also believe that it can be the basis for an expansion of the security architecture to a collection of systems cooperating to support multiple base security policies.

  This is discussed further in Section 8.1.1.

2.4.2 Fluke Implementation

Though the goal of the project is to develop a security architecture that is independent of Fluke and transferrable to other systems, the implementation of this security architecture in the Fluke system is important to verify that the architecture is sound. The Fluke Assurance program focused primarily upon the changes to the Fluke microkernel and Security Server, but also provided some review of proposed changes in other Fluke object managers.

- The base Fluke system provides a recursive virtual machine architecture [7] which is intended to support isolation of processes for security and other reasons. The first task on the Fluke Assurance program was to investigate this architecture in order to identify the requirements that this architecture places on the various system components. In doing so, we found that the DTOS security architecture could enhance the Fluke recursive virtual machine architecture by simplifying these requirements and directly supporting security policies other than isolation.

  This is discussed further in Sections 3 and 5.

- Each object manager has a control policy which defines the services which are controlled by the security policy and the permissions which must be held or other requirements that must be met for the service to be provided. Though an initial control policy for the microkernel was already defined when the Fluke Assurance program began, we defined a conceptual framework for determining the requirements on each service which was consulted as the control policy underwent modifications throughout the system development.

  This is discussed further in Section 5.1.3.

- As provider of IPC services, one of the key requirements of the Fluke microkernel is to provide each thread involved in an IPC with the security attributes of the other thread. Moreover, each thread must be provided with the ability to hide some of its attributes from the other thread. As part of the Fluke Assurance program we helped to design the mechanisms for meeting these requirements, which turned out to be surprisingly difficult in the case of long-lived IPC connections.

  This is discussed further in Section 5.1.2.

- A key element to supporting dynamic security policies is the ability to revoke previously granted permissions within the object managers where these permissions are used. We helped to develop an approach for performing revocation within the Fluke microkernel, which was subsequently implemented by NSA.

  This is discussed further in Section 5.1.5.

- Though object managers (other than the microkernel) were not generally a focus of the program, we reviewed proposed control policies for the file server, network server and process manager. Other features of these managers, such as security union directories (SUDs), were also reviewed. In all cases, changes were suggested when security or assurability goals could be furthered.

  This is discussed further in Section 5.2.

- The security server used in the Fluke prototype enforces a combination of type enforcement and multilevel security (MLS) policies. The policy structure used by this security server was initially inherited from the DTOS program, however as one task on the Fluke Assurance program we developed a new logical structure which was able to decouple the type enforcement and MLS policies. NSA then changed the security server implementation to incorporate this.

This is discussed further in Section 6.

- The security server also enforces a role based access control policy which allows for dynamic changes to the set of roles associated with a user. This policy demonstrates the effectiveness of the dynamic policy support in the microkernel.

  This is discussed further in Section 6.4.

2.4.3 Composition and Refinement Analysis

Through the use of composition and refinement analysis, properties of a modular system can be analyzed by formally combining lower level properties of individual components or dividing a higher level system analysis into properties on each component. For instance, this allows an individual component to be replaced by another satisfying the same properties without affecting the higher system level analysis.

An extended example of composition and refinement was performed on the program, with the following accomplishments. These are discussed further in Section 7.

- Analysis of complex systems like Fluke must accommodate the system's development process, which may be a combination of bottom-up and top-down construction techniques. Composition theory supports bottom-up analysis, while refinement theory supports top-down analysis. We constructed a demonstration to illustrate how composition and refinement techniques, specifically the composition and refinement framework developed under the CSS program, support effective and efficient reasoning about Fluke-like systems through the reuse of specifications and analyses.
- When combined with requirements-based specifications, formal composition and refinement analysis can expose subtle requirements on a system's components. Such requirements reflect component interactions that are often missed using less rigorous analysis techniques. The demonstration noted above exposed several examples.
- Composition and refinement analysis can help a security architect compare candidate implementations and assess the impact of change to existing implementations. We offered a simple strategy for performing these tasks.

2.5 Outputs

Two conference papers and three technical reports (in addition to this Final Report) were written as a part of the program. These papers and reports are summarized in the following sections.

2.5.1 Conference Papers

The Flask Security Architecture : System Support for Diverse Security Policies [10]
This paper provides an overview of the security architecture with an emphasis on policy flexibility and the support for dynamic security policies. It is currently in submission to the 1999 Usenix Security Conference.

This paper was coauthored by Ray Spencer from Secure Computing along with Stephen Smalley and Pete Loscocco from NSA and Jay Lepreau, David Andersen and Mike Hibler from the University of Utah.

Using Composition and Refinement to Support Security Architecture Trade-off Analysis [4] This paper describes the results of the System Composition Study with an emphasis on the use of the composition framework to perform a tradeoff study between competing designs. It is currently in submission to the 1999 National Information Systems Security Conference.

This paper was authored by Charlie Payne from Secure Computing.

2.5.2 Technical Reports

Formal Security Policy Model (FSPM) [20] This report studies the security architecture of the base Fluke system and identifies how the DTOS security architecture can complement and expand on the base Fluke security architecture. It also describes security enhancements to the Fluke microkernel.

The microkernel control policy, including a model of the state, of the controlled services, and of the requirements which must be met for each controlled service to be performed. is described in English language descriptions and tables. Formal specifications using PVS accompany these descriptions, in order to decrease any ambiguity that may inadvertantly appear in the informal language. Tables in the appendix of this report provide a mapping between the control policy model and the system implementation.

For those who are uninterested in reading the formal PVS specifications, this report is also available in a format which does not contain the PVS [21].

Formal Top Level Specification (FTLS) [22] This report focuses on the policy independent interactions between an object manager and a security server. It begins by laying out the general responsibility of each component, and then provides a formal specification of the interfaces provided by each component. However, these specifications are only a snapshot, as the interfaces have since changed primarily as a result of the dynamic security policy.

The dynamic security policy study, which is also documented in this report, provides a formal framework for stating requirements on the interactions between the object managers and security server and for analyzing designs against these requirements.

Finally, the report describes the internals of the prototype security server which enforces a combination of type enforcement and multilevel security.

For those who are uninterested in reading the formal PVS specifications, this report is also available in a format which does not contain the PVS [23].

System Composition Study Report [24] This report provides a complete description of the effort which is reported upon in the second conference paper mentioned above [4].

It is a demonstration of how composition and refinement techniques can be used to analyze Fluke-like systems. First, a simple model consisting of two object managers, a file server and process manager, is defined and shown to verify certain critical properties. Then two candidate implementations, each consisting of the above object managers plus a memory manager, are proposed. The demonstration emphasizes reuse of specification and analysis, separation of security and non-security concerns and the simplicity of requirements-based specifications. The power of composition and refinement for performing security architecture trade-off analysis is also discussed.

Section 3

Security Architecture

One of the major efforts of the Fluke Assurance program was to reevaluate the DTOS security architecture based upon the level of policy flexibility which it provides and to expand upon the architecture as necessary. The DTOS Generalized Security Policy Study [16] identified several weaknesses in policy support in DTOS, the most significant relating to dynamic security policies (see Section 10.4 of the DTOS Lessons Learned Report [18] for a brief summary of these weaknesses).

These efforts took place in two phases. In the first phase, the internal data structures within the access vector cache and the interfaces provided by the access vector cache and the security server were redesigned to provide the same or better policy flexibility as DTOS but through simpler mechanisms. Formal specifications for the AVC and security server interfaces were written during this phase and are contained in the FTLS [22].

In the second phase, we addressed the most significant shortcoming of DTOS, its limited ability to support dynamic security policies, with a study to identify the requirements for a system to support dynamic security policies and a design which would satisfy these requirements. This study is documented in Section 4.

The remainder of this section provides an overview of the final version of the Flask security architecture. More details can be found in the following documents:

- Section 2.2 of the FSPM [20] describes the motivation and functional elements of the Flask security architecture in more detail.
- Sections 2 through 4 of the FTLS [22] provide specifications of the interfaces which support this architecture. However, as mentioned above, the FTLS was written after the first phase of the development of the security architecture. Updates to these specifications are included in Appendix C of this Final Report.

The Flask security architecture describes the interactions between a subsystem which enforces 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.

The architecture currently provides for three kinds of policy decisions:

- The most fundamental are yes/no security decisions which specify whether a particular permission is granted from one labeled entity to another, based entirely upon the labels. The prototype for this is a yes/no decision controlling a particular access that some subject may request to an object.
- Labeling decisions specify the assignment of a label to an object (typically assigned upon its creation), computed from labels of related subjects or objects.

- Polyinstantiation decisions specify which member of a polyinstantiated set of resources should be accessed for a particular request, typically computed from the label of the requesting client and possibly other related objects.

Though the primary goal of the architecture is policy flexibility, we also have three important secondary goals for the architecture:

- Applications outside of the enforcement and decision making subsystems should be able to execute independent of those subsystems. Their awareness of the architecture must be limited to the potential that some service which they request may be denied because of a security decision.

  Another way to state this goal is that it should be possible to alter the enforcement mechanisms without affecting outside applications.

  For example, some security architectures require applications to be aware of all permissions which they require to execute correctly and to specifically request that these permissions be granted, while we believe that it is impractical to place this responsibility upon applications.

- The architecture should anticipate the performance impact of all operations that it requires and take steps to minimize those impacts whenever possible.

- The architecture should anticipate the need to verify the correctness of mechanisms which implement it, and should allow for implementation through relatively simple, isolated mechanisms whenever possible.

The architecture specifies interfaces and requirements on two components:

- Components which enforce the security policy decisions are referred to as *object managers*, based upon the origins of the architecture in a client/server model.

- 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 are policy dependent and are therefore not addressed by the architecture.

The Flask security architecture provides the following for object managers:

- Interfaces for retrieving security, labeling and polyinstantiation decisions from a security server.

- An *access vector cache* (AVC) module that allows the object manager to cache security decisions in order to avoid any performance costs associated with security server communications.

- The ability to register to receive notifications of changes to the security policy.

To take full advantage of these services, the object manager must do the following:

- All objects involved in security decisions must be labeled. The mechanism for assigning labels to new objects must be clearly defined, which may involve the use of the labeling interfaces provided by a security server.

- The object manager must define and implement a control policy which defines how security decisions are used to control the services which the object manager provides.
- The object manager must determine which policy changes are relevant and define handling routines which are called in response to policy changes.
- For all uses of polyinstantiation, the object manager must define the mechanism by which the proper instantiation of a resource is chosen, possibly using the interfaces provided by a security server.

Finally, the architecture requires that some form of source and destination authentication be provided by the underlying system for inter-process communication.

Section 4
Dynamic Security Policy Study

The Dynamic Security Policy Study determined a collection of requirements on an object manager and security server that ensure the security policy being enforced by the object manager is consistent with the policy internal to the security server, even as the policy changes to revoke previously granted permissions. The requirements are determined by derivation from a property which correlates all object manager actions with policy decisions in the security server.

The Dynamic Security Policy Study report [22, Section 6] begins by presenting a formal model of the policy relevant state of a security server and an object manager. It then defines the Fundamental Property which captures the consistency objective, and then systematically refines this property into a collection of individual requirements on transitions made by the object manager and security server. The derivation of these requirements was verified by use of the PVS specification and proof environment [9]. The study also considers an alternative approach to consistency based upon database replication protocols and briefly discusses requirements which ensure that changes to the policy can be made in a timely manner.

We discuss the formal model in Section 4.1. The two approaches to deriving a complete set of requirements on the object manager and security server are described in Sections 4.1 and 4.2. Section 4.3 discusses timeliness of policy changes.

4.1 A Model of Security Enforcement for Dynamic Policies

The formal model of a security server and object manager is an abstraction of the Flask architecture intended to apply to virtually all systems with some sort of decider/enforcer separation. There are three key elements to the model:

- The model allows objects to be labeled with a policy independent SID. The SID is translated to a security context within the security server. Though this is obviously based upon the Flask architecture, a system with direct labeling of objects is consistent with this model by considering SIDs and security contexts to be equivalent.
- The model defines three kinds of retained permissions within the object manager:
  - Permissions which are maintained within a data structure accessed by object SIDs. This is an abstraction of the Flask AVC.
  - Permissions which are maintained within a data structure accessed by the objects themselves. A typical example is a page table that defines allowed accesses between a process and a page of memory.
  - Permissions which are maintained within the state of a request currently being processed. This may be implicit in the state, for instance, the request processing may have successfully completed a permission check but not the operation controlled by that permission check.
- The model provides the security server with two views of the security policy:

- *current_perms* defines the current security policy as a function from pairs of security contexts to the set of permissions granted between each pair of contexts.
- *provided_perms* identifies the maximal set of permissions retained by the object manager, as a function from pairs of SIDs to the set of permissions granted between each pair of SIDs.

Within this model we can more precisely specify our consistency requirement, which we refer to as the Fundamental Property.

> For any service performed by the object manager, all permissions required for the service must be in the set *current_perms*.

There are three steps in the derivation of a final set of requirements from this property. In the first step, we state a single requirement on transitions of the object manager state and four state invariant properties. The transition requirement requires that the object manager only performs a service if all permissions required for the service are retained by the object manager. The invariant properties ensure that all retained permissions are in *provided_perms* and that all permissions in *provided_perms* are also in *current_perms*.

These state invariant properties are then refined in a straightforward manner into state transition requirements by placing requirements on any transition which can invalidate a state invariant. For instance, there is a requirement that for a permission to be removed from *current_perms*, it must not be in *provided_perms*.

At this point in the analysis there are essentially two requirements remaining which span both components:

- A permission cannot be removed from *provided_perms* unless the permission is not retained by the object manager in any form.
- A permission cannot be added to any form of retained permissions unless it is in *provided_perms*.

Satisfying these requirements requires coordination between the security server and the object manager. For the third (final) step in the derivation of requirements, we introduce a protocol between the security server and object manager to provide this coordination.

When the security server determines that a permission must be removed from *current_perms*, it checks if the permission is in *provided_perms*. If so, then the object manager may have retained that permission so the security server must first send a request to the object manager to revoke the permission. Similarly, the object manager can only add permissions to retained permissions if the security server notifies the object manager that the permission is in *provided_perms*.

However, a consistency problem arises because we do not assume that communications between the security server and object manager are absolutely serialized, as we believe this is too functionally limiting. Instead, we assume that sequence numbers are added to all communications in order to provide the necessary serialization. It is not necessary to provide a continuously increasing counter on each message, and in fact this is undesirable since it may lead to inefficiencies since any communications received or processed out of order would need to be discarded. Instead, we have attempted to derive the minimal requirements on the use of sequence numbers in order to satisfy the two requirements above. In particular, the model allows for distinct sequence numbers on each pair of SIDs, incremented only when the policy changes to revoke a previously granted permission between that pair of SIDs.

The model and all requirements derivation has been formalized and proven in PVS.

4.2 Transactional Consistency

In this approach we view the security policy maintained by the security server and object managers as a replicated database and attempt to adapt database concurrency control techniques for maintaining consistency between the portions of the security policy replicated at the security server and the portions replicated at each object manager. This allows us to generalize our approach to support multiple object managers fairly easily. In this model we represent a security policy as a set of mappings from a pair of security contexts and a permission to an abstract value representing an access decision. This allows us to retain information about negative access decisions as well as positive ones. The security server maintains a complete set of such mappings, which we refer to as *current_copies*.[3] We refer to the set of mappings that each object manager maintains as *retained_copies*. We insist that for any mapping from (*context1*, *context2*, *permission*) to a decision value v in *retained_copies*, there is a mapping from (*context1*, *context2*, *permission*) to v in *current_copies* as well. Thus, any access decision retained by an object manager is identical to the access decision maintained by the security server.

To ensure consistency between the two policies, when the security server wishes to modify a mapping for (*context1*, *context2*, *permission*) in *current_copies*, it locks the entry and instructs the object manager to discard its mapping for (*context1*, *context2*, *permission*) in *retained_copies*. A mapping for this triple is acquired the next time it is needed by querying the security server. After the object manager acknowledges removing the entry, the security server updates *current_copies* and removes its lock on the entry. This lock is used to delay queries posed by the object manager until the value has been updated.

In the FTLS we provide a detailed description of this protocol. We prove that the protocol ensures one-copy serializability, a synchronization property commonly used for replicated database systems. This ensures that the replication and concurrent accesses to the security policy are transparent to the security server and object manager. In addition, the protocol ensures that both the security server and the object manager experience the changes to the policy in the same order. One-copy serializability, by itself, is not sufficient to ensure the Fundamental Property described in Section 4.1. We believe that the proposed protocol does in fact satisfy this requirement, but have not proven it.

4.3 Timeliness

The protocols described above are intended to ensure consistency between the security server and object manager. They do not address the availability of the features which provide revocation of permissions, and in fact, a system in which requests for policy changes are ignored would satisfy the Fundamental Property. While it was not a major topic of the Dynamic Security Policy Study, we did briefly consider requirements on the security server, object manager and the underlying system which would ensure that policy changes are made in a "timely" manner. This is discussed in Section 6.7 of the FTLS.

[3] This is a simplification of the actual model.

Section 5

Fluke Implementation

This section contains an overview of the design and implementation of the Flask security architecture as an enhancement to the Fluke microkernel and operating system servers. More details can be found in Sections 2.2.2 and 2.2.3 of the FSPM [20].

Note that Fluke provides a security architecture based upon the concept of a recursive virtual machine and implemented using the basic mechanisms of a capability system. Section 2.1 of the FSPM describes this architecture and identifies two shortcomings. The first shortcoming is the difficulty of verifying the correctness of the design because of the difficulty of analyzing the propagation of capabilities. The second shortcoming is that the virtual machine architecture essentially presents an all-or-nothing ability to interact. The implementation of the Flask security architecture as an enhancement to Fluke is intended to address both of these shortcomings.

5.1 Microkernel

Flask enhancements to the Fluke microkernel address five aspects of the security architecture described in Section 3. We discuss each of these in a separate subsection.

5.1.1 Object Labeling

The microkernel controls subject access to memory and to nine classes of kernel objects. Memory is labeled through segment objects which are under the control of the kernel server though segment labels are visible and used within the microkernel to control direct memory accesses.

Segment labels are also used to define the labels on kernel objects. Each kernel object is associated with a location in user-space memory, and the label on each kernel object is equal to the label of the memory segment to which this location is mapped. Though this location can be changed, the label of a kernel object is not allowed to change so the object's new segment must have the same label as the original segment.

5.1.2 IPC Client and Server Identification

Flask enhances the Fluke IPC mechanisms to allow the client and server engaged in IPC to identify the SID of each other. They are also allowed to specify an effective SID by which they will be identified in an IPC operation.

5.1.3 Control Policy

The Fluke microkernel controls access to memory and kernel objects with what is effectively a capability based mechanism for propagating object and memory access rights. The most important aspect of the Flask control policy is to simply control which tasks can hold capabilities to objects. These controls are based upon the SID of the task, the SID of the object (or memory segment), and the flavor of the capability (read, write, execute and create).

Fluke also allows some operations involving an object to proceed with an indirect capability to an object, known as a *reference*. A reference is an object in itself, and the internal structure of the microkernel which makes it easy to control which task can hold capabilities to an object does not extend to references. Hence, the Flask control policy instead directly controls the use of references. For instance, one of the most important uses of references is in creating links from one object to another (e.g., to map memory). For some kinds of links (e.g., between a thread and its task) the control policy requires that the two linked objects have the identical SID. In the more general case, a permission is required to link two objects based upon their SIDs.

One other interesting area of the control policy is IPC, especially the extensions to allow for effective SIDs to be specified. Flask assigns effective SIDs to the client and server in an IPC operation, defaulting to the actual SIDs if no effective SID is specifically identified. It controls the assignment of an effective SID by requiring a permission between the thread and its effective SID. In total, Flask requires as many as three permissions for an IPC operation to be successful:

- The client must be granted *Specify_Client* permission to its effective SID, unless the effective SID is the same as the client's own SID.
- The policy must grant *Send*, *Call* or *Connect* permission between the client's effective SID and the server's effective SID, for one-way, idempotent or reliable IPC, respectively (the three distinct forms of Fluke IPC).
- The server must be granted *Specify_Server* permission to its effective SID, unless the effective SID is the same as the server's own SID.

5.1.4 Access Vector Cache

The Flask microkernel incorporates an access vector cache (AVC) providing the function described in Section 3. In particular, the AVC is responsible for interacting with the security server to retrieve security decisions and for caching these decisions locally within the microkernel. It also receives requests from the security server when a policy change occurs, and forwards them as necessary to the microkernel enforcement mechanisms through previously registered callbacks.

5.1.5 Support for Permission Revocation

The Flask microkernel supports the revocation and recomputation of all Flask permissions. As described in Section 5.1.3, there are three models for control, memory permissions, reference capabilities for linking and IPC controls, and the mechanisms for addressing revocation of each are discussed here.

If a memory permission from an address space (task) SID to a segment SID is being revoked, the Flask microkernel locates all memory translations that map memory from a segment with the given SID to an address space with the given SID, and updates the translation protections accordingly. The microkernel can rely upon the atomic properties of kernel requests [6] to ensure that requests in progress at the time of the revocation either finish in a timely manner or are restarted after the revocation is complete.

If one of the link permissions is revoked, the kernel locates all objects containing links requiring the revoked permission, and inserts a null link.

If an IPC permission is revoked, the microkernel locates all threads in established IPC connections requiring that permission, and breaks the connection. From the perspective of each thread in a reliable IPC connection, it will appear that the connection was dropped by its peer.

5.2 Operating System Servers

The Fluke operating system provides services through four servers, a virtual memory manager, a file server, a network server and a process manager. Each of these servers can be treated as an object manager in the Flask architecture, and designs for Flask enhancements to each have been proposed, and in some cases, implemented.

Many of the Flask enhancements to each server are specific to that server. However, a single AVC code module has been incorporated into each server. This module provides the same function as the microkernel AVC, though the implementation for user-space servers is slightly different.

Currently, all of the servers provide object labeling and access control over those objects. None of these servers support the requirements of the Dynamic Security Policy Study for revocation. Some of the unique aspects of the server enhancements include:

- While kernel objects inherit their labels directly from another object (memory segments), most objects provided by the user space servers allow the client creating the object to choose a label for the object. This includes memory segments, so that the label of a kernel object can be chosen indirectly by first choosing the label on the memory segment in which the kernel object is created.
- Like kernel objects, the labels on most server objects cannot be changed once the object is created. There are three exceptions: files, directories and processes. Changing the label of an object has potentially many of the same ramifications as changing the security policy, since permissions can be retained implicitly through the object itself. This is addressed as follows:
  - The label of a process is only allowed to change when the process calls the *exec* routine to execute a file. Unless a specific label is requested as a parameter to the routine, the process' new label is determined by combining its original label with the label of the file, through a call to the security server. In either case, the process manager controls which elements of the state of the process can be inherited across the *exec* call, with sanitization if necessary.
  - A file or directory can be relabeled through an explicit request to the file manager. Assuming that the request is allowed by the security policy, there are still a number of steps which must be performed to ensure that there are no explicitly or implicitly retained permissions. For example, permissions to the file may be stored within a file description object, or another operation on the file or directory may currently be in progress, and may need to be aborted.[4]
- Labeling of file systems, files and directories is complicated by the fact that they are persistent objects, and the interpretation of SIDs is persistent. Instead each file system

[4]Currently, the ability to change labels on files and directories has been implemented but without support for aborting operations in progress.

must record a mapping from every file and directory to its security context without the use of SIDs. The current file server stores this mapping in a special subdirectory of the root directory of each file system.

- Both the file server and network server have been designed to use polyinstantiation of resources.

  In the file server, certain directories (such as `/tmp`) may be designated as *Security Union Directories* (SUDs) to support polyinstantiation of files and subdirectories within that directory. To a typical client, a SUD looks like any other directory, and the correct instantiation of a file or subdirectory to be accessed by the client is determined through the use of the security server's interface for polyinstantiation. A security-aware client may be able to request a particular instantiation of a file or directory by specifying its label.[5]

  In the network server, the port name space is a single shared resource, and names in this space may be polyinstantiated like a SUD. For instance, this would allow a system to host multiple FTP servers, each handling clients with different security contexts, without placing any requirements on the clients to identify the correct server.[6]

[5] This discussion represents the design of Security Union Directories, though they have not yet been implemented.
[6] Polyinstantiation of port spaces was completed in an earlier implementation but is not part of the current release.

Section 6
Security Server

The Flask security server was developed as an evolution of the DTOS security server. Many of the changes to the security server were made to support new features of the security architecture described elsewhere in this report. This section describes four enhancements which were made independent of the security architecture:

- The Flask security server provides a policy independent interface for defining extensions to the security policy at run-time.
- The Flask security server defines a multilevel security (MLS) policy which allows a range of levels to be assigned to each object rather than just a single level as was supported under DTOS. This allows a finer grained control over the trust granted to subjects under the MLS policy.
- The Flask security server provides completely decoupled MLS and type enforcement (TE) policies, simplifying policy configuration.
- The Flask security server supports a role based access control policy (RBAC) with dynamic association of roles to users. The security server makes use of the interfaces added in Flask to support dynamic security policies.

The DTOS security server is documented in the DTOS Software Design Document [13] and Kernel Interface Document [17]. Policy extensions for Flask are described in Section 4 of the FTLS [22]. Changes to the MLS and TE policies are discussed in Section 5 of the FTLS. The addition of an RBAC policy was done after the FTLS was completed and there is no written source providing more details than this report.

6.1 Policy Extensions

The Flask security server defines a root security policy implementing multilevel security, type enforcement, role based access control and identity based access control. It also provides policy independent mechanisms for defining extensions to the root security policy, allowing clients to define finer-grained policies within the restrictions of the root policy.

These mechanisms include:

- The Flask security server distinguishes between two kinds of security contexts (and their corresponding SIDs). A root security context identifies attributes relative to the root security policy. An extension context contains a name by which it is known and a parent SID, which may be a root SID or an extension SID.
- Sets of allowed permissions can be defined between a pair of extension contexts or between an extension context and a root context. Default rules (described in Section 4.2.2.3.2 of the FTLS) are applied for computing such permissions when they are not explicitly defined. The fine-grained control made possible by extension policies cannot create a more permissive security policy; all permissions granted explicitly by an extension policy must be granted according to the default rules.

- SID transitions (as returned by the `transition_sid` request) can be defined between a pair of extension SIDs or between an extension SID and a root SID. Default rules (described in Section 4.2.5.3 of the FTLS) are applied when a transition is not explicitly defined. There are currently no default-related restrictions on explicitly defined transitions.

Section 8.1.1 discusses ways in which these mechanisms can be expanded to provide more flexibility in the use of extension policies.

6.2 Level Ranges for Subjects and Object

The Bell and LaPadula (BLP) model [2] for multilevel security associates a security level with every object and subject within a system. The security level of an object represents the maximum classification of data stored in the object. The security level of a subject defines the highest level of data which the subject is allowed to process. All data output by the subject is assumed to be at that level.

This model places no trust upon subjects, though Bell and LaPadula recognize the need to assign trust to some subjects in a practical system. In [2, SectionIV], they effectively define a trusted subject as one which is trusted to properly label all data which it handles. A trusted subject is still assigned a level, which represents the maximum level of data which it is trusted to process and label.

The security policy enforced by the DTOS security server followed this basic model, with the identification of trusted subjects determined by the type enforcement attribute of a subject. For Flask, we wanted to provide a more flexible definition of trust, one which allowed trust to be limited not only by a maximum level of data but also by a minimum level.

This limitation on trust has been addressed in at least three previous efforts: Bell [1] describes a model for network architectures which assigns a range of levels to each subject, while the GEMSOS [11] and Trusted Mach [25] operating systems actually implement such a feature.

For Flask, we extend these models in two ways:

- We allow objects as well as subjects to be assigned a range of levels. This is necessary when a subject is represented as an object but is also useful for objects shared between trusted subjects to allow data labeling to be performed internal to the object.
- We identify policy rules on operations not considered by previous models. In addition to the familiar read and write operations between a subject and an object, we address three subject-to-subject accesses provided by the Fluke microkernel.

The rules assigned to each operation were informally derived, like the original BLP model. However, we would prefer to have a more direct verification of the rules. For instance, the theory of noninterference can be used to justify the BLP rules. However, when partially trusted subjects (i.e., those with a lower limit on the level range) are incorporated into the model, the theory of noninterference is not sufficient to provide such verification. This is discussed as a topic for future efforts in Section 8.3.

6.3 Decoupling of Multilevel Security and Type Enforcement

The MLS and TE policies were tightly coupled in the DTOS security server and the original Flask security server (as documented in the FTLS). As mentioned in Section 6.3, the most significant reason for this coupling was the use of TE attributes to identify those subjects which are trusted with respect to the MLS policy. However, with the ability to assign a range of levels to each subject, the trust granted to a subject with respect to the MLS policy can be recorded in the MLS attributes of the subject and a decoupling between the MLS and TE policies is possible.

The principal advantage of this decoupling is a dramatic decrease in the complexity of the policy configuration, since the MLS and TE policies can now be configured and computed independently. The policies are only combined by verifying that any permissions granted by the security server are allowed under both policies.

The decoupling is not without some price however, as it leads to a potential loss of least privilege.[7] Though not implemented, we believe that a least privilege policy can be defined separately if desired, in such a way that only those permissions further restricted by the least privilege policy must be defined. This is similar to the manner in which the RBAC policy (described in Section 6.4) is currently implemented.

6.4 Role Based Access Control

An RBAC policy was added to the security server to provide a demonstration of Flask's support for dynamic security policies. The RBAC policy assigns a role set to each user and a role to each subject. The security server provides a `change_role_set` request which provides a dynamic element to the policy by allowing an administrator to change the set of roles associated with a user.

The RBAC policy defines a set of constraints on individual permissions based upon a combination of users and roles. These constraints must be satisfied in addition to all other policy requirements in order for a permission to be granted. Constraints are defined only on microkernel permissions since only the microkernel currently supports permission revocation.

To support policy changes, the security server also adds the infrastructure necessary to coordinate policy changes with object managers. The two most significant changes are

- The addition of a *latest_granting* value for each registered AVC to use in the sequence number protocol, and
- The addition of a collection of *provided_perms* which records every *allowed* vector which has been provided to an object manager and which may be affected by a change in a user's role set.

The Dynamic Policy Study Report (Section 6 of the FTLS [22]) describes the purpose of these values.

When an administrator requests the removal of a role from a user's role set, previously granted permissions may need to be revoked. To do this, the security server looks through its set of *provided_perms* to find all *allowed* vectors which grant a permission not allowed under the change in policy. For each one found, it does the following:

- The security server calls `avc_revoke` on each registered object manager which may have been given the *allowed* vector.

[7] Mathematically the decoupling can be viewed as replacing a policy function of two variables, e.g., $f(x, y)$, with a product of two single variable functions, e.g., $g(x)h(y)$.

- Upon receipt of a successful response from `avc_revoke`, the appropriate *allowed* vector in *provided_perms* is updated to indicate revocation of the requested permission.
- If any errors are returned from `avc_revoke`, then a permission may not have been revoked so `change_role_set` also returns an error.

When an administrator requests the addition of a role from a user's role set, previously denied permissions may now be allowed. This initiates similar processing, except that the security server calls `avc_remove` instead of `avc_revoke` and *provided_perms* is updated to indicate any additional permissions granted.

Section 7

Composition Study

The Composition Study explored the application of composition and refinement theory to the construction of Fluke-like operating systems, specifically to perform cost-effective, security architecture trade-off analysis. We used a composition and refinement framework developed here at Secure Computing to support the Study.

Security architects routinely face the following questions:

- Given two or more candidate security architectures, which candidate will be the easiest to build or the easiest to assure?
- How will modifying subsystem x affect my system's security posture?

Answering both questions requires some trade-off analysis. In the first case, the architect determines how each candidate satisfies its security obligations and identifies vulnerabilities in the candidate's approach. For the second case, the architect effectively performs a trade-off analysis between the existing implementation and its proposed modification. While the final decision for either case is subjective, the analysis that supports that decision should be rigorous. Composition and refinement techniques let us perform that analysis at lower cost.

Composition and refinement are complementary approaches for demonstrating that a system satisfies its critical properties. Composition supports bottom-up reasoning about the system in the same way that modular software design supports its bottom-up construction. Composition techniques let us model the system by combining the models of its components. The technique supports "plug-and-play": each component can be replaced as long as its replacement satisfies the component's properties. In addition, since the component analysis depends only on that component, the analysis is reusable when the component is used in a new context. Refinement, on the other hand, supports top-down reasoning. It is usually easier to prove desired system properties of the abstract model than for a more detailed one; on the other hand, it is easier to relate the more detailed model to the implementation since it includes more design details. Refinement techniques let us show that properties demonstrated for the abstract model are preserved in its refinements. Thus, we only need to analyze the property at the abstract level where the analysis is easiest. Combining composition and refinement, then, provides a powerful reasoning tool for complex systems.

The approach for applying composition and refinement techniques to security architecture trade-off analysis is straightforward. The first task is to prove that each candidate satisfies the critical requirements imposed on all candidates. Proving this of each candidate would be time-consuming and possibly redundant, since the candidates may be similar. Also, depending on the level of detail, the proof may be very difficult. Instead, an abstract model representing all candidates is constructed, and we prove that the abstract model satisfies the requirements. Then we prove that each candidate is a refinement of the abstract model. If we are unable to complete a refinement proof, it may be that the abstract model does not reflect that candidate, or it may be that the candidate will not satisfy the security requirements. Only further analysis will tell. However, for each candidate successfully refined, refinement theory lets us claim that the candidate therefore satisfies the critical requirements. The differences between candidates, then, exist in the obligations on each candidate that are necessary to demonstrate refinement.

At this point, the architect can weigh the obligations of successful refinements against other criteria.

The approach has several benefits. Refinement proofs are usually easier than satisfaction proofs, and if the candidates are similar, part of the refinement proof may be reused. If a new candidate is later proposed, we need only prove that it is a refinement of the abstract model. If a candidate is later modified, only the refinement proof must be redone. If a candidate is complex, we may use composition tools to build it up from its parts. Similarly, we can use decomposition tools to break a candidate into smaller pieces that reflect its intended implementation. The decomposition proof will distribute the requirements on the candidate to requirements on each component of the candidate.

We applied the approach to a simple example consisting of a file server, process manager and memory manager. The file server manages the file system and handles file requests from the process manager and the memory manager. The process manager maintains all running processes. The memory manager provides memory segments to the process manager for storing process executables. Any implementation must satisfy the critical property that all trusted processes run trustworthy executables. We considered two candidate implementations. Both candidates trust the file server to provide trustworthy executables upon request. The difference lies in whether the process manager or the memory manager is more highly trusted to assign executables to processes correctly. We first constructed an abstract model of both candidates and proved that it satisfies the critical property. Then we attempted to demonstrate that each candidate refines the abstract model. Unfortunately, we ran out of time before we could complete the entire refinement proof for both candidates; however, we did reduce each refinement proof to a set of simpler obligations that, if satisfied, can be used to compare the candidate architectures.

The Study demonstrated that composition and refinement techniques can simplify and reduce the cost of security architecture trade-off analysis. Of course as with any modeling effort, there are some risks, such as the risk that the abstract model or its refinements ignore critical behavior in the real system. However, we use formal methods as a best defense against larger unknowns, and while any modeling effort may miss details here and there, in the end the greater understanding that we achieve usually compensates for any perceived risk.

Section 8

Future Work

The primary accomplishments of the Flask program have been the development of a security architecture to support dynamic security policies and the implementation of this architecture as part of the Flask system. In this section we describe future work which addresses the need to expand the architecture itself and to further test and validate the architecture through implementation.

The two primary needs for expansion of the security architecture are to provide coordination within a networked/distributed system and to provide coordination among different policy environments. These and other architectural issues are discussed in Section 8.1. The primary implementation needs are to expand support of dynamic policies outside of the microkernel, to implement expanded features of the architecture and to apply the architecture to other systems. These are discussed in Section 8.2. Finally, Section 8.3 describes some efforts that could be undertaken to further the assurance of systems such as Flask.

8.1 Expanding the Flask Security Architecture

8.1.1 Policy Layering

The Flask security architecture as described in Section 3 specifies the interactions between a policy-enforcing object manager and a policy-defining security server. The architecture is immediately extensible to include multiple object managers, but there is no such extension to multiple security servers (unless they are identical copies of each other). Yet there are at least two reasons for allowing multiple security servers within a single system:

- Security policies within an organization are typically defined hierarchically, with certain rules applying across the organization but others defined within individual divisions or departments. Policy administration would likely be simplified through a security architecture which reflected such a hierarchy.
- Security policy administration can also be simplified by separating the policy based upon functional elements of a system. For instance, in a dedicated database server, administration of the policy over the individual records in the database may be separated from administration of the policy over low level functions such as kernel services or network services.

The Fluke recursive virtual machine architecture was originally defined with a vision of a security architecture to support a policy hierarchy. Therefore it was expected that the Flask security architecture could be extended within Fluke to provide a parallel hierarchy of security servers. However, some serious difficulties arose and this possibility has only recently been gaining renewed attention.

Recall that the Flask security server, as described in Section 6, provides some support for an arbitrary hierarchy of policy extensions to be loaded within a single security server. While these mechanisms are potentially useful, there are some shortcomings which significantly limit their applicability:

- All security contexts which are potentially needed as part of the extension policy must be explicitly loaded into the security server, rather than only creating those contexts which are specifically used within the system. This makes the mechanism impractical for policies with potentially large numbers of contexts, few of which are actually used. All sets of permissions between all possible extension contexts (not using default rules) must also be explicitly loaded.

  For example, suppose the Flask root policy supports 4 MLS sensitivity levels, 10 MLS categories, 10 TE types, 10 RBAC roles and 10 users. This policy has the potential for over 4 million distinct security contexts (note that 10 MLS categories means 1024 possible category sets) and hence 16 trillion pairs for which permissions must be defined. Obviously, it is completely impractical to include all such information within preloaded tables, but such a policy can be implemented using a combination of small tables and policy logic.[8]

- Permission computations involving extension contexts are made according to a combination of explicitly defined permissions and default rules. However, as discussed in Section 4.2.2.3.2 of the FTLS, these rules can lead to unexpected results and there is no clear solution.

- As described in Section 2.2.1 of the FSPM [20], security decisions provided by the security server to the AVC can be deferred. But there is no security server mechanism for deferring decisions involving extension contexts.

- The mechanisms provide no means for extending the list of permissions. Hence extension policies cannot extend the set of services controlled by the security server.

We believe that these problems can be overcome, and the advantages cited earlier realized, by extending the security architecture to provide for interactions among a hierarchy of security servers. This architecture will have the following properties:

- Each object manager will be associated with a single security server and the interactions between an object manager and its security server will be exactly as in the current security architecture. While it may be possible to define an architecture in which an object manager queries different security servers based upon the SIDs involved in a security decision, we prefer to assign the responsibility for correctly routing security decisions to the security server.

- There will be a tree of security servers, allowing security servers to parallel the organizational structure. It appears that a tree structure will also suffice to separate policy across functional elements of the system, though this is less clear.

- Each SID is owned by a single security server responsible for policy decisions involving the SID. The architecture must provide a way for a security server to identify whether a particular SID is owned by one of its descendents, and if so, through which child.

- If a security server is asked for a security decision involving a SID other than its own, it generally queries its parent. However, if the other SID belongs to a descendent then the security server queries the appropriate child.

[8] In this example, some of the contexts are likely to be invalid, for instance, every user is not going to be allowed to operate in every level and all categories will not be relevant at each level. Nonetheless, it is easy to see how the number of valid contexts can grow similarly.

These principles are independent of implementation in Fluke. However, if a Fluke implementation is pursued, we would expect the architecture to satisfy the following additional properties:

- The security server interface should be considered part of the Common Protocols interface.
- SIDs and security contexts within a virtual machines should appear relative to that virtual machine to support the location transparency property of the Fluke recursive virtual machine model.

  Since translation of SIDs to security contexts is done by a security server, there is no difficulty in expressing security contexts relative to the security server's virtual machine. The current implementation of SIDs as integers eliminates the possibility of completely relative SIDs. At best, we can assure that SIDs are allocated "randomly" so that the value of a SID does not provide any hint to its location within the hierarchy. An alternative would be to implement SIDs as a new type of kernel object.

Any effort to further pursue a general architecture for policy extensions should also be accompanied by a study, analogous to the dynamic security policy study, to derive requirements for the architecture. The fundamental property from which the requirements are derived should be an extension of the property that all services provided by the object manager are allowed by the current policy as defined by the security server. While it is unclear precisely how to extend this fundamental property, the property must ensure that the realm of influence of a single security server only includes its branch of the tree.

To define such a property requires that the model address aspects of the system not addressed in the dynamic security policy study, such as IPC, authentication and process integrity. For instance, it is not sufficient to show that a security server does not directly affect security decisions made by its parent, we must also show that it cannot affect the state of its parent indirectly through the decisions it makes for a particular object manager.

8.1.2 Extending the Architecture to a Networked/Distributed System

The Flask security architecture has so far been described and implemented within a single workstation. It is essential that the architecture be extended to a network of individual nodes. To accomplish this, there are two specific issues that must be addressed:

- For reasons of performance and reliability, we do not want to require a single global security server shared among all nodes.
- The dynamic security policy study required timely communications to complete revocation of permissions. A direct extension of this model to a network may require timely communication from a node that is currently down or isolated due to network partitioning.

The first issue appears to be a further extension of the architecture proposed in the previous section. However, in addition to allowing the refinement of policies through the tree, there is also a need to define an architecture for combining policies. Moreover, since SIDs are local (and non-persistent), policy decisions cannot be communicated across nodes in terms of SIDs.

The second issue is much more specific and will require a fundamental weakening in the requirements for revocation. While the timeliness requirements can still be applied within a node, across nodes it may be necessary to augment the single node model of permission granting and revocation with a mechanism for granting time-limited permissions which must be occasionally revalidated.

As in the previous section, attempts to develop such an architecture should be accompanied by a requirements derivation analogous to the dynamic security policy study.

8.1.3 Other Architecture Enhancements

In this section we list a few other features which could be incorporated into the security architecture.

Policy Coordination Among Object Managers The previous two sections discuss various needs for coordination among multiple security servers. There is also a possible need for coordination among object managers (through the security servers) when one is dependent upon another for a service. For instance, it accomplishes little to grant a user read access to a memory mapped file if the user isn't granted read access to memory with the same label. It would be useful to explore whether general mechanisms can be developed to address these kinds of dependencies or whether they appear to be handled best on a case-by-case basis.

Policy Configuration Tools Other than architectural support for extension policies, Flask provides no direct support for policy configuration. It is expected that high level policy configuration tools (as used by system administrators) will be policy specific and hence are not part of the general architecture. However, it may still be possible to define a collection of lower level tools which integrate a general policy language into the specific mechanisms and interfaces defined by the Flask architecture.

Integration with Audit and Intrusion Detection The Flask security architecture addresses the definition and enforcement of security policy. If the implementation of the entire system and all security administration is done perfectly, then this would be sufficient. However, accepting that any real system will have flaws, it is important to discover any actual or attempted security breaches and be able to analyze the history of the system to identify the consequences of any breach.

While commercial intrusion detection tools are generally based upon detection of known flaws independent of any particular security policy, a security architecture such as Flask invites the possibility of intrusion detection based upon recording and analysis of denied access requests. The main difficulty that must be overcome to facilitate this possibility is that security decisions are made in terms of SIDs so the actual objects involved in a denied access request can only be identified by their SIDs.

Evaluation in Comparison with other Security Architectures While informal comparisons of Flask with other security architectures (e.g., CORBA) have been made over the course of its development, a significant study comparing Flask to other security architectures has not been undertaken. While we would hope that the main outcome of such a study would be further validation of the architecture and a better appreciation for its features, it is reasonable to expect that such a study would identify shortcomings of the architecture as well.

8.2 Further Implementation

Section 5 describes the progress made so far to implement the Flask architecture as part of the Fluke operating system. In this section we list possibilities for implementation to further assess and validate the architecture.

Revocation in Fluke Object Managers Currently, only the Fluke microkernel has been enhanced to provide the features of the Flask security architecture which support revocation of permissions. Since the microkernel implementation depends heavily on the atomic nature of the Fluke interface, it is important to gain experience with the implementation of these features in other object managers. Without such experience, it is difficult to claim that the object manager requirements for revocation are appropriate for more general object managers.

Complete Implementation of File Server Features Section 5.2 describes two features planned but not yet implemented within the file server. The ability to abort in progress operations is needed to support general permission revocation as well as changes to file labels. Security union directories which use the polyinstantiation interfaces provided by the security server are also not yet implemented.

Network Server Implementation Flask enhancements to the network server have been partially designed and implemented. Completing the network server is a complex task requiring integration with cryptographic and authentication services. This task will need to be performed in conjunction with the extension of the Flask security architecture to a network, since it is likely that some of the interactions with cryptographic and authentication services will be represented at the level of the security architecture.

In addition, the prior implementation of polyinstantiated port name spaces should be updated.

Implementation in other Operating Systems A longer term goal should be to develop implementations of the Flask security architecture in operating systems with different fundamental structures, such as a monolithic Unix operating system or a Java operating system. The purpose of such an effort would be to determine the feasibility of meeting the object manager requirements in such a system.

Additional Security Policies With the incorporation of the dynamic RBAC policy, the current security server exercises all of the major interfaces of the security architecture. However, it may still be valuable to further validate that the current set of policy interfaces are sufficient by implementing other policies with dynamic characteristics, such as a Chinese Wall [3] or ORCON [8] policy.

More Flexible Access Vector Cache Interfaces The current interface to the access vector cache allows the security server to perform the basic services of granting and revoking permissions. However, these interfaces also have a potentially limited flexibility in how they allow the security server to choose precisely which permissions are affected by each request. A more flexible interface has been suggested in which the AVC would support a few primitive operations which can be arbitrarily combined in a single request through the use of logical expressions. This effort is important only if further experience implementing different security policies indicates that the current interfaces are clumsy or inefficient to use.

8.3 Assurance

While assurance that the Flask implementation meets its security requirements has not been a significant emphasis of the current program, we identify three assurance tasks that should be considered as part of future efforts. Two of these tasks were previously identified in the DTOS Lessons Learned Report [18], while the third is more directly an outcome of the current program.

Integration of Formal Methods with Other Forms of Assurance Evidence Assurance can be provided through many forms of evidence, such as requirements tracing, process documentation, use of trusted personnel, code reviews, code assertions, safe programming languages, testing and mathematical proofs. On the DTOS program we focused primarily on mathematical proofs and while few proofs were performed on the Fluke Assurance program, the use of formal specifications is intended to lay the groundwork for the use of formal mathematical methods. This focus on formal methods is due to the fact that when formal methods are of use, they provide the strongest guarantee of completeness in the analysis.

However, this justification ignores the fact that even greater confidence may be gained by combining various techniques, especially in those areas where formal methods are particularly weak. The most obvious weakness of formal methods is the inability to analyze an actual implementation. To address this, formal methods should be incorporated with security testing, for instance, by generating test cases from formal specifications.

Both this interaction and security testing itself are poorly understood. Security testing is very different than functional testing for two reasons. First, security testing is generally concerned with demonstrating that bad things do not happen, while functional testing is concerned with demonstrating that good things do happen. Second, security testing requires completeness, and is particularly interested in atypical cases, while functional testing focuses on those cases which legitimate users are likely to encounter.

Security testing is not the only other form of assurance which can be integrated with formal methods. For instance, if a requirement is very specific, placing an assertion in the code can be more convincing than a complex formal argument.

Information Flow Analysis A considerable effort was put forth on the DTOS program to define a methodology for analyzing the ability of a complex system to satisfy general information flow policies. The result of this effort was the recognition that the existing theoretical basis for information flow policies is lacking in two significant areas: intransitive information flow and information flow through multiple subjects (including multilevel subjects). This is discussed further in Section 13.5 of the DTOS Lessons Learned Report [18] and in [12].

Two Level Control Policy A control policy is written for two audiences which have differing needs: a security administrator or analyst attempting to understand how the granting of permissions affects the higher level policy satisfied by the system and a programmer implementing the control policy.

The major difference between these two audiences is that the security administrator or analyst is interested in the behavior of the combination of the object manager and security server so that they can directly relate the state of the security server to the state of the overall system. The programmer doesn't care about the security server but needs to know how each operation is directly controlled, e.g., whether a permission is found in the AVC or as a memory protection.

Another difference may be the level of the requirements themselves. For example, the security administrator or analyst may be interested in how a permission affects a state invariant, while a programmer would rather see how particular state transitions are controlled to satisfy the invariant. Some implementation level requirements may have no direct higher level analogue, for instance, the programmer needs to know that the SID of a Fluke kernel object is the same as the SID of the memory location through which it is accessed, but a security administrator only cares that the permissions defined in the security server actually control access to the object.

With these different audiences in mind, it may be beneficial to define the control policy at both levels, with a mapping between them.

Appendix A
Bibliography

[1] D. Elliott Bell. Secure computer systems: A network interpretation. In *Proceedings of the Second Aerospace Computer Security Conference*, December 1986.

[2] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, May 1973.

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

[4] Charles Payne. Using composition and refinement to support security architecture tradeoff analysis. Submitted to the 1999 National Information Systems Security Conference.

[5] Bryan Ford and Mike Hibler. *Fluke Version 2.1 Application Programming Interface Reference (DRAFT).* University of Utah, Department of Computer Science, February 1997.

[6] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullman. Interface and execution models in the fluke kernel. In *Proceedings of the Symposium on Operating Systems Design and Implementations (OSDI)*, February 1999.

[7] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Symposium on Operating Systems Design and Implementations (OSDI)*, October 1996.

[8] Richard Graubart. On the need for a third form of access control. In *Proceedings 12th National Computer Security Conference*, pages 296–304, Baltimore, MD, October 1989.

[9] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

[10] Ray Spencer and Stephen Smalley and Peter Loscocco and Mike Hibler and David Andersen and Jay Lepreau. The Flask security architecture : System support for diverse security policies. Submitted to the 1999 Usenix Security Conference.

[11] Roger R. Schell, Tien F. Tao, and Mark Heckman. Designing the GEMSOS kernel for security and performance. In *Proceedings of the National Computer Security Conference*, October 1985.

[12] Secure Computing Corporation. Information Flow Policies, Composition and Refinement. Unpublished report provided to the customer December, 1998.

[13] Secure Computing Corporation. DTOS Kernel and Security Server Software Design Document. DTOS CDRL A002, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1996.

[14] Secure Computing Corporation. DTOS Composability Study. DTOS CDRL A020, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.

[15] Secure Computing Corporation. DTOS General System Security and Assurability Assessment Report. DTOS CDRL A011, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.

[16] Secure Computing Corporation. DTOS Generalized Security Policy Specification. DTOS CDRL A019, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.

[17] Secure Computing Corporation. DTOS Kernel Interfaces Document. DTOS CDRL A003, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1997.

[18] Secure Computing Corporation. DTOS Lessons Learned Report. DTOS CDRL A008, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.

[19] Secure Computing Corporation. Composability for Secure Systems: Final Report. CDRL A010, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, October 1998.

[20] Secure Computing Corporation. Assurance in the Fluke Microkernel: Formal Security Policy Model. CDRL A003, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, February 1999.

[21] Secure Computing Corporation. Assurance in the Fluke Microkernel: Formal Security Policy Model (Non-PVS Version). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, February 1999.

[22] Secure Computing Corporation. Assurance in the Fluke Microkernel: Formal Top-Level Specification. CDRL A004, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, February 1999.

[23] Secure Computing Corporation. Assurance in the Fluke Microkernel: Formal Top-Level Specification (Non-PVS Version). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, February 1999.

[24] Secure Computing Corporation. Assurance in the Fluke Microkernel: System Composition Study Report. CDRL A005, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, March 1999.

[25] Trusted Information Systems, Inc. *Trusted Mach Philosophy of Protection*, May 1993.

[26] Patrick Tullman, Jay Lepreau, Bryan Ford, and Mike Hibler. User-level checkpointing through exportable kernel state. In *Proceedings of the Fifth IEEE International Workshop on Object-Orientation in Operating Systems*, October 1996.

Appendix B
FSPM Updates

This section contains two discussions which are logically an addition to the FSPM document. In the first, we describe how the security architecture evolved from the DTOS program. In the second, we compare the experience of creating the microkernel control policy for the Flask microkernel with the DTOS experience.

B.1 Evolution of the Security Architecture

In this section we discuss the evolution of the security architecture from the DTOS program to the Flask architecture described in Section 2.2 of the FSPM. This section is organized analogous to that section. There is also an additional subsection discussing audit of security decisions, which was a feature provided by the DTOS security architecture but is not currently supported under Flask.

Figures 1 and 2 provide an overview of the DTOS and Flask architectures, identifying major components of the architecture and interfaces between those components.

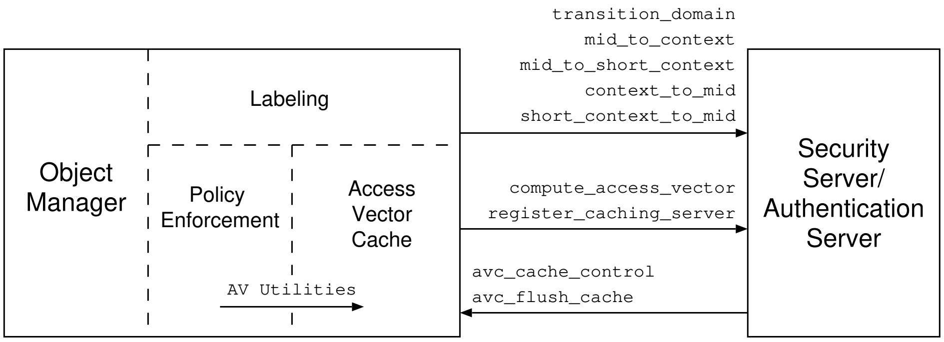

Figure 1: DTOS Security Architecture Interfaces

Further details of most of the DTOS interfaces between the object manager and security server can be found in the DTOS Kernel Interface Document [17]. The internal interfaces between the policy enfocement component of the object manager and the access vector cache were not documented outside of the DTOS code files.

Further details of some of the Flask interfaces can be found in the FTLS [22]. However, the FTLS was written prior to the dynamic security policy study, and consequently several new interfaces have been added and changes were made to existing interfaces. Appendix C provides a brief description of changes made since the FTLS was written.

B.1.1 Object Labeling

The basic role of object labeling has not changed since the conclusion of the DTOS program. However, some minor changes in the architecture have been made:

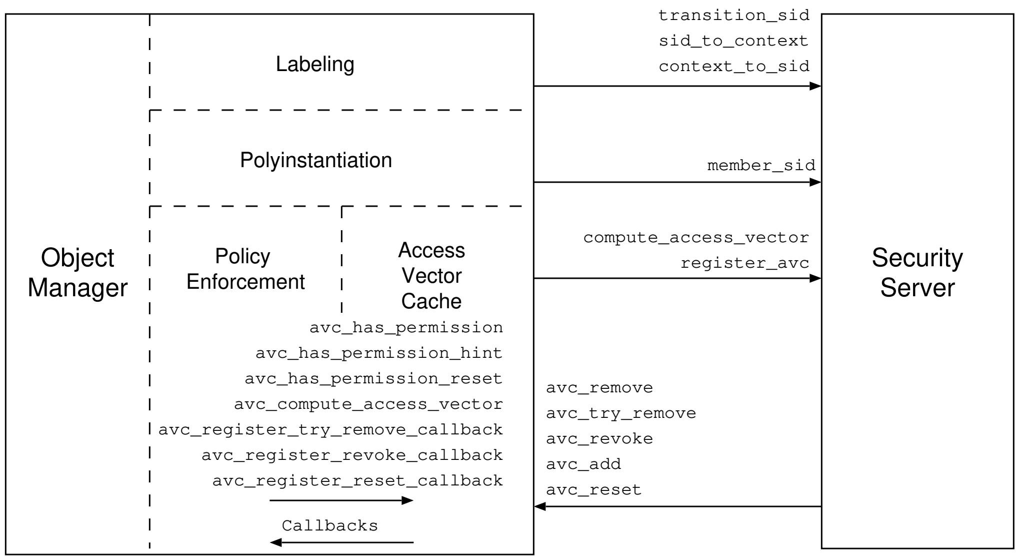

Figure 2: Flask Security Architecture Interfaces

- Under DTOS, the SID was a structured datatype. It was broken into two components, an *authentication identifier* (AID) and a *mandatory identifier* (MID). The MID was further broken into a *short MID* and a *classifier* equivalent to a Flask security class.

  DTOS also had an authentication server distinct from a security server. The authentication server mapped an AID into an authentication context (though this could be done through a request to the security server which in turn called the authentication server).

  The short MID was mapped by the security server to a mandatory context. The MID and AID were separated in DTOS to allow the AID and MID to be combined independent of the security policy and also to provide some optimization within the AVC (as discussed below in Appendix B.1.4). Though the distinction between authentication context and mandatory context has been retained by the Flask security server, they can now only be combined by the security server or other policy aware subjects through direct manipulation of security contexts.

  The classifier incorporated into the SID in the DTOS prototype has been completely separated in Flask. This provides for a more intuitive use of the classifier and explicitly reflects the fact that the classifier is independent of the security policy.

- Persistent SIDs were not explicitly called out in any description of the DTOS architecture. However, in both DTOS and Flask persistent SIDs have been used only in the file server implementations and neither implementation relies upon general architectural support for persistent SIDs.

- DTOS security server support for the labeling interface, known as `transition_domain`, was limited to the case of a process exec'ing a file. In Flask this interface was expanded to include an input parameter identifying a class of object being created so that the interface, known now as `transition_sid`, can be used during the creation of other classes of objects.

B.1.2 Client and Object Manager Identification

The basic requirements on client and object manager identification with the DTOS and Flask architectures are unchanged. However, there are some important conceptual and implementation differences to discuss.

- In Flask, a client preparing to send a message can identify the SID of the destination server prior to sending the message, since the server must have the same SID as the destination port. There was no such requirement in DTOS (as will be discussed further below). Instead, a client wishing to have some control over the SID of the destination had the option of specifying a recipient task SID when sending a message. If a task with a different SID attempted to receive the message, the task would not be allowed to receive the message unless the security policy granted the task explicit permission to override the client's request.

  The Flask model is preferable for two reasons. First of all, it allows a decision about the recipient of a message to be made prior to sending a message. Under DTOS, a message sent to a port belonging to a task with a SID other than the specified SID may be unreceivable with no notification to the sender. Second, the Flask model allows the client to make a yes/no decision about sending a message, which can be directed to the security server if desired. The DTOS model instead requires the client to identify a specific SID, which may be too restrictive if there are many acceptable recipient SIDs and which is more difficult to determine in a policy independent manner.

- The interpretation of the SID of a port has changed significantly since DTOS. DTOS assumed that the system would follow an object oriented model in which ports represented individual objects, and the SID of each port would be the same as the SID of the object. This allowed the IPC controls to be used to completely eliminate requests for access to an object from a task that had no permissions to the object.

  For Flask, the SID of a port is required to be the same as the SID of the server receiving messages on the port, while the SID of each object is internal to the server. This has the potential disadvantage of eliminating the use of IPC controls to restrict access to particular objects. However, the server is still able to restrict any access after a request is received, and a client which is unable to access any of the server's objects can still be prevented from sending any messages to the server.

  The main advantage to the Flask approach is that it is conceptually simpler. The microkernel need not ever see the SIDs of higher level objects, and doesn't need to enforce security decisions regarding these SIDs. This has the potential side effect of decreasing the required size of the access vector cache. Finally, as described above, it allows the client to directly determine the SID of the destination for any request.

B.1.3 Control Policy

The concept of a control policy (distinct from the security policy) was one of the major accomplishments of the DTOS program and under the Flask project we have found the need to only make one minor change to the concept. Under DTOS, a control policy is considered to include two kinds of requirements: initial state requirements (which were never presented under DTOS though their necessity was described) and state transition requirements. In the Flask description of control policy (see Section 2.2.1 of the FSPM), we explicitly add a third category, state invariant requirements.

This addition is more a change in presentation than in substance, since a state invariant requirement can always be refined into an initial state requirement and requirements on any state transition which can invalidate the invariant. Nonetheless, many requirements are most naturally understood as state invariant requirements so it is desirable to allow such requirements directly in the control policy. For example, the Flask requirement for SID equality between a thread and the task in which it executes is more naturally stated as a state invariant requirement than as a collection of state transition requirements.

B.1.4 Retrieving and Caching Security Decisions

The basic interfaces for retrieving security decisions from the security server and the mechanisms for caching them within the AVC have changed little since DTOS, other than those changes made to support dynamic security policies as described below in Appendix B.1.5. Perhaps the most significant change is that the AVC has been more cleanly separated into a distinct module from the object manager. The lack of such a distinction in DTOS is reflected by the "AV Utilities" label in Figure 1, which identified a collection of interfaces for accessing the AVC but with no distinction between those interfaces used by the object manager and those used only within the AVC itself.

Specific changes, other than those made explicitly as a result of the dynamic security policy study, include the following:

- The DTOS access vector cache made use of the division of AID and MID within the SID in an attempt to decrease the size of the AVC [18, Section 3.4.1.2], while this is not possible in Flask.
- As described in Section 2.2.1 of the FSPM, security decisions in Flask are represented by two vectors, the decided vector and the allowed vector. Under DTOS security decisions were also represented by two vectors, the allowed vector (which was then known as "the access vector") and the cache control vector. However, the interpretation of the cache control vector was quite different than the interpretation of the decided vector.

  While the details of these differences are somewhat involved because of differences in the security fault mechanisms under DTOS and Flask, the most significant difference is that the cache control vector allowed for a permission to be granted once but not be further retained. This feature was determined to be of little practical use and was dropped from Flask, as is discussed in Section 3.3.1 of the FTLS [22].
- Under DTOS, security decisions from the security server were accompanied with an expiration time. When the access vector cache was queried by the object manager for a decision, the expiration time was checked. If the decision had expired, the security server would be asked for an updated decision.

  However, since permissions are often retained within the object manager itself, expiration times did not always have the expected effect. As a result, this feature has been discontinued in Flask.[9]
- DTOS did not provide `avc_add` and `avc_remove` requests to explicitly add or remove nodes from the AVC. These requests are of limited, but essential, use. `avc_add` allows the addition of new wired nodes to the cache which may otherwise be unrequestable because of deadlocks. `avc_remove` allows the removal of wired nodes which are no longer necessary and of nodes which deny permissions which are now allowable.

[9] Note however, that this change was made after the FTLS [22] was completed, so that document includes descriptions of the use of expiration times. It also contains a discussion of the problems of their use in Section 3.1.

B.1.5 Changes to Security Decisions

The most significant changes to the security architecture made during the Flask program arose from the Dynamic Security Policy Study described in Section 4. This section lists those changes made to the architecture to support dynamic security policies by implementing the protocol and satisfying the requirements described in Section 4.1. Most of these changes were made after the FTLS was written and therefore more details can be found in Appendix C.

- DTOS included a policy ID on most communications between the security server and object manager, similar to the sequence number used in Flask to satisfy the requirements derived by the Dynamic Security Policy Study.
- DTOS provided a kernel request, `set_security_server`, which replaced the security server port. All future security server requests would then be sent to the new port which presumably would belong to a different security server. Given the amount of security policy information maintained in a running system outside of the security server (e.g., SIDs, persistent SIDs, cached permissions) it is unclear that there is any practical change in policy possible which can be handled outside of the existing security server. Moreover, the same effect can be had by simply having the existing security server pass the port to a new security server, without any intervention by the microkernel.

  Therefore this request has not been included in Flask.
- DTOS provided a request, `avc_cache_control`, which is an earlier version of the `avc_reset` request. The main difference between these requests is the support for callbacks to the object manager in Fluke.
- DTOS provided a request, `avc_flush_cache`, which is an earlier version of the `avc_revoke` request. Again, the main difference between these requests is the support for callbacks to the object manager in Fluke. DTOS did provide a single hardcoded callback for revoking page table permissions but did not address any other forms of retained permissions.
- The Flask request `avc_try_remove` had no DTOS counterpart.

B.1.6 Polyinstantiation Support

Under DTOS, polyinstantiation was not considered as a general feature of the architecture. Though the file server provided a polyinstantiation mechanism, determinations of the correct instantiation of a file or directory to be used by each client was made using an algorithm independent of the security policy. This algorithm had some particularly undesirable features, including the fact that the chosen instantiation could change from one access to another. This algorithm was not adopted for the Flask file server.

B.1.7 Audit

The Flask security architecture provides no direct support for any form of audit, while the DTOS security architecture did provide some rudimentary audit support.

In particular, under DTOS each security decision was accompanied by a notification vector. If a bit in the notification vector was set, the AVC would audit all requests from the object manager for the corresponding permission. However, since the only information available to the AVC are the SIDs involved in the request, not the identities of the actual objects themselves, the usefulness of such audit records is limited.

This is identified in Section 8.1.3 as a topic for future work.

B.2 Microkernel Control Policy Lessons Learned

The concept of a "control policy" as defined in the FSPM was formulated to a large extent under the DTOS program. The DTOS Lessons Learned Report [18, Section 6] discusses in some detail some of the accomplishments and difficulties encountered in developing the control policy for the DTOS microkernel. Since the Fluke Assurance program represents the first attempt to apply this concept to another system, we felt it would be appropriate to compare our experiences under the two programs.

The two key defining aspects of the DTOS control policy carried forth unchanged into the Flask control policy:

- The control policy is defined independent of the particular security policy embodied in the security server.
- The requirements within the control policy are defined whenever possible as requirements upon general state transformations which are not tied to particular requests or a particular implementation.

In the following subsections we compare the DTOS and Flask experiences in each of three major portions of a control policy: definition of the state model, services and requirements. An ongoing theme throughout these comparisons is the relative simplicity in developing the Flask microkernel control policy because of the relative simplicity of the underlying microkernel and the ability to leverage off of the security mechanisms of the Fluke microkernel.

B.2.1 Microkernel State Model

Since the Fluke microkernel provides for a fully exportable state [26], the state model for the Flask microkernel is almost entirely defined within the Fluke API [5]. There are only a few complications:

- The Flask microkernel added state not existing in the Fluke microkernel:
  - The access vector cache, which was straightforward to model.
  - The assignment of SIDs to kernel objects, which was incorporated into the fully exportable state of the Fluke model.
  - The assignment of SIDs to memory. This was more complex to address because of the indirection between user visible memory interfaces (e.g., reading and writing of virtual addresses) and the internal structures which map virtual addresses into labeled memory objects (known in Fluke as "flobs"). We settled on a model which was intended to capture this indirection without providing too much implementation detail.
- While the Fluke microkernel provides fully exportable state, some elements of the state are necessarily changed when it is exported. The most important example of this is that

IPC connections are broken[10], so the idea of a connection had to be added to the kernel model.

A perhaps surprising conclusion of this effort is that by allowing all essential kernel state to be externally visible and controllable, Fluke actually simplified the modeling exercise compared to Mach. The distinction between "essential" kernel state and implementation details in the Mach microkernel is unclear making it very difficult to determine an appropriate level of detail for the state model.

B.2.2 Service Definitions

Like the state model, the definition of the Flask microkernel services was initially made very simple due to the complete control allowed over microkernel state. While on DTOS there might be several different ways to change a particular state element, each possibly as a side effect of another operation, on Fluke each state element is typically changed as a result of a single request which allows it to be arbitrarily set. Therefore most of the service definitions simply identify a change to a particular element (or set of elements) of the state model.

A particularly striking example of this is the lack of side effects from destruction of an object. For instance, destruction of a message in Mach can lead to a chain reaction causing destruction of many other messages and ports. In Fluke, typically the only side effect of object destruction is that links to that object are no longer valid.

Another aspect of Fluke which simplifed the service definitions is the atomicity provided by kernel operations [6]. This eliminates the concern expressed on DTOS that the granularity of services in the implementation could be significantly different than the granularity of services at the control policy level.

B.2.3 Requirements

The primary differentiator between the development of control requirements under DTOS and under Flask is that for Flask we were able to leverage off of the underlying Fluke capability system. In fact, approximately three-fourths of all requirements recorded in the Flask control policy are requirements in the underlying Fluke microkernel as well. This was not possible under DTOS since while Mach also controls access to kernel objects through a capability system, a capability to a Mach kernel object grants full access to all methods on the object.

Perhaps the most notable outcome of the use of the Fluke security mechanisms is the relatively small number of distinct permissions enforced by the microkernel. While the DTOS microkernel enforced about 150 distinct permissions over approximately the same number of services, there are only 12 distinct permissions under Flask to control about 40 services.[11] Most Flask services are controlled by the underlying read and write permissions of the Fluke microkernel.

In the DTOS Lessons Learned Report [18], we suggested that the use of distinct permissions for each service was a desirable feature to allow for greater flexibility in policy configuration, especially for least privilege. In retrospect, we believe that while this is true in theory, it is unlikely that such flexibility could ever be used in practice. Security policies are defined in terms of much higher level services, such as reading files or modifying database records. Thus while controlling read and write operations through distinct permissions has obvious value,

[10] Actually, IPC connections are not completely broken, they are moved to a "pickled" state.

[11] Actually, since the Flask services have less granularity, it could be easily argued that a fair comparison would list about 70 services for Flask compared to 150 for DTOS.

there's little potential value in further distinguishing permissions on kernel objects, e.g., for distinguishing the permission to set a thread's task from permission to set it's scheduler.

Another difference between the development of the DTOS requirements and the Flask requirements was the ongoing development of the Fluke implementation. The interplay with the implementation team led to many changes in the policy[12] when compromises were required between implementation flexibility and the desired level of policy control. It also required us to elaborate more on the underlying principles for the control policy. The resulting compromises are reflected in the lack of any requirements over certain services which allow user space implementation of mutex, condition variable and (partially) reference objects, while potentially opening some small-bandwidth channels for information flow.

Two other differences in our approach were relatively independent of Fluke:

- DTOS allowed the security policy to differentiate in the policy between a thread operating on a thread in the same task from a thread in a different task with the same SID. As noted in the DTOS Lessons Learned Report [18], this led to some unexpected complications in the implementation.

  For Flask this feature was eliminated, we believe with little or no loss in flexibility. If different tasks are to be differentiated in the security policy, we believe this should be done by associating different SIDs with each task. This would also allow the distinction to be made at all functional levels in the system, not just for services performed on threads or tasks.

- For DTOS, automated tools were used to compile different versions of the requirements from a single source, in order to guarantee consistency of the presentation of the requirements within and among various documents. While we still believe this is a useful technique, no such tools were developed for Flask since the control policy is only described within a single document and it is short enough that such tools would not be cost effective for simply preserving consistency of presentation.

[12] Actually, the most significant impact of this interplay was the decision to use the Fluke mechanisms for Flask. The first informal version of the control policy assumed the DTOS approach of creating an entirely new security infrastructure.

Appendix C
FTLS Updates

This section describes the changes made to the AVC and Security Server since the FTLS was written. While all changes are described, our emphasis is on those changes which are independent of the policy enforced by the security server. In particular, we focus on the AVC and the interface between the AVC and the security server, with only passing mention of changes to internal processing of the security server.

Most of these changes are due to the incorporation of the sequence number protocol for coordinating policy changes, as proposed in the Dynamic Security Policy study and described earlier in Appendix B.1.5.

For reference, Figure 3 identifies the interfaces described in the FTLS. The current version of these interfaces, excepting the administrative interfaces provided by the security server, are depicted in Figure 2 on page 35.

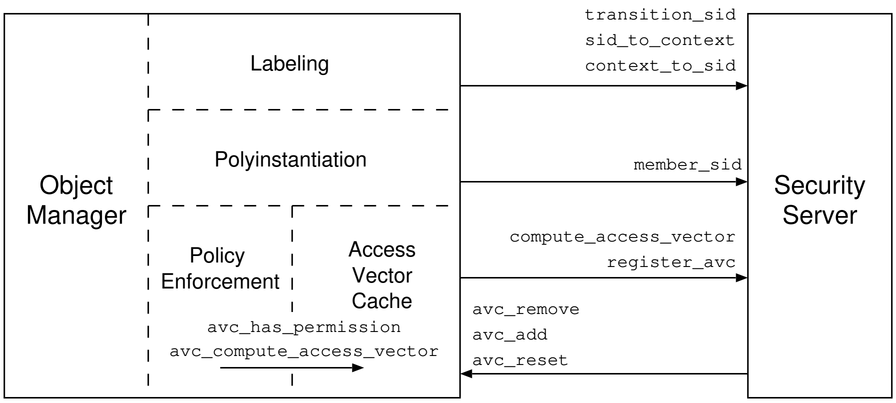

Figure 3: Fluke Security Architecture Interfaces as Documented in the FTLS

The following changes have been made to the AVC state model since the completion of Section 3 of the FTLS:

- The AVC model in the FTLS includes an expiration value in each node. As described in Appendix B.1.4, this is no longer being used.
- The AVC state has been expanded to include a *latest_notif* value, as required by the sequence number protocol. Though the protocol model provided in the FTLS allows a distinct value of *latest_notif* for each *(sclass, ssid, tclass, tsid)*, the requirements can be met by a simpler implementation in which a single value applies to all such 4-tuples.
- The AVC state has been expanded to include three varieties of callbacks, routines which the object manager registers with the AVC and which the AVC calls during processing of `avc_reset`, `avc_revoke` and `avc_try_remove` requests. The state maintained with each callback is as follows:
  - Each *reset_callback* identifies the address of the routine to call.

  - Each *revoke_callback* or *try_remove_callback* identifies the address of the routine to call, the (sclass, ssid, tclass, tsid) for which the routine should be called and the set of permissions for which the callback is relevant.

- The AVC state has been expanded to include a *version* number on each node. This is used to validate a node when accessing it directly in response to a `avc_has_permission_hint` request from the object manager, as described below.[13]

The following changes have been made to the interface which the AVC provides to the object manager since the completion of Section 3 of the FTLS:

- Two additional output parameters have been added to the request `avc_compute_access_vector`. In the case that a node is found to match the input parameters, *out_ae* returns a pointer to the node and *out_aever* returns the version number from that node.

  The object manager can store these parameters however it sees fit and use them to provide a hint to the AVC the next time a permission is requested which may be found in the same node (by calling `avc_has_permission_hint`). This is provided purely as a performance enhancement.

- A new request `avc_has_permission_reset` has been added as a slight variation to the `avc_has_permission` request. It has an additional output parameter, *revoke*, which returns true if any of the requested permissions are disallowed and false if they are all allowed. An error status return from this new request indicates a processing error. This differs from `avc_has_permission` which returns an error status in the case of a processing error or a denied permission.

  This request was added for use when recomputing a migrated permission. If the request returns *revoke* equal to true, then the migrated permission will be revoked, otherwise it is allowed to remain in the system state.

- A new request `avc_has_permission_hint` has been added as a slight variant to the `avc_has_permission` request.[14] This new request has two additional parameters, a pointer to an AVC node and a version number, allowing for optimized lookups. The AVC validates the node by ensuring that the SID values and version number provided in the request agree with the values in the reference node.

- Three requests `avc_register_{try_remove,revoke,reset}_callback` have been added to allow the object manager to register callback routines with the AVC. The parameters to these requests are the same as the data associated with each callback as described above. The callback routines are called in response to requests from the security server, as will be described below.

The following changes have been made to the interface which the AVC provides to the security server since the completion of Section 3 of the FTLS:

[13] The *version* number is an implementation detail that would ideally be avoided in any specifications. However, its appearance as a parameter in communications with the object manager compels us to discuss it.

[14] Actually, in the implementation there is no request by this name. Instead this description is based upon the request `avc_has_permission_ipc` which is used between the kernel and the kernel's AVC, and provides all of the function of `avc_has_permission_hint` while also processing permission denials (an error case) different than `avc_has_permission`. We do not describe this explicitly since the FTLS typically ignores error processing. Note also that `avc_has_permission_ipc` does not include *sclass* or *tclass* parameters, since there is only one class within the microkernel.

- An additional input parameter, *seqno*, has been added to the `avc_add` request for use in the sequence number protocol.

  In the processing of this request, and replies from the `compute_access_vector` request to the security server, the returned access vectors are discarded if *seqno* is less than *latest_notif* (though this is not considered to be an error case).

- An input parameter, *seqno*, has been added to the `avc_reset` request for use in the sequence number protocol.

  The processing of this request has been expanded. As a first step, the AVC is reset to its initial state as described in the FTLS. Then, all reset callback functions registered by the object manager are called. The expectation is that these functions will cause all migrated permissions to be recomputed. This may cause the AVC to become repopulated with nodes not in the initial state. Finally, if *seqno* is greater than *latest_notif*, then *latest_notif* is set to *seqno*.

- A new request, `avc_revoke`, has been added to the interface from the security server to the AVC. It has two additional parameters, *revoke* and *seqno*. *revoke* is an access vector identifying the permissions to be revoked and *seqno* is used for the sequence number protocol.

  This request does three things. First, if a node is found in the cache which matches the input parameters (unlike `avc_remove`, wildcarding is not supported for this request), then all permissions in *revoke* are removed from the node's *allowed* vector unless the node is wired and the *revokewired* flag is unset. Then, all revoke callback functions which match the input parameters are called. The expectation is that these functions will remove all instances of migrated permissions which match the input parameters. Finally, if *seqno* is greater than *latest_notif*, then *latest_notif* is set to *seqno*.

  While the expected primary use of `avc_remove` is to remove nodes which deny permissions that may now be allowed by the security policy, `avc_revoke` is used to revoke permissions which the security policy no longer allows.

- A new request, `avc_try_remove`, has been added to the interface from the security server to the AVC. This request is essentially a query of the state of migrated permissions with the possibility of removing a node from the AVC.

  Input parameters to the request include a *(sclass, ssid, tclass, tsid)* 4-tuple (with no wildcarding allowed), a set of *permissions*, a *removewired* flag and a *seqno*. The request returns a set of *retained* permissions.

  The request does three things. First, all try_remove callback functions which match the input parameters are called. These functions query the state of the object manager and return a *retained* vector indicating which of the identified *permissions* are retained. Then, the AVC itself is queried to see if any of the *permissions* are retained in the AVC but not in the object manager itself. If so, these permissions are removed from the AVC (as in the `avc_revoke` request), unless the node is wired and the *removewired* flag is unset. Finally, if *seqno* is greater than *latest_notif*, then *latest_notif* is set to *seqno*.

  The return value, *retained*, identifies all of the input *permissions* which are still retained within the object manager or the AVC.

The AVC model also needs to be expanded to describe processing that occurs when it receives responses from requests it makes to the security server or object manager. When the FTLS was written, the only such processing was as a result of a *compute_access_vector* call to the security server, but the enhancements provided to support revocation have added several more instances of AVC processing triggered by responses.

Many internal changes to the structure of the security policy have been made since the completion of Section 4 of the FTLS. Perhaps the most significant is the adoption of the alternate structure for the multilevel security/type enforcement policy which was proposed in Section 5 of the FTLS and briefly described in Sections 6.2 and 6.3 of this report. In addition, a role based policy framework has been added as described in Section 6.4.

However, our focus here is on changes in the general architecture itself. Only two minor changes have been made to the interfaces from the AVC to the security server since completion of Section 4 of the FTLS:

- An additional output parameter, *seqno*, has been added to the `compute_access_vector` request for use in the sequence number protocol.
- An additional input parameter, *pairs*, has been added to the `register_avc` request. This parameter identifies a sequence of security class pairs for which the registered AVC will be notified of policy changes.