Part Number 87-0902025A006
Version Date 27 June 1997

DTOS LESSONS LEARNED REPORT

CONTRACT NO. MDA904-93-C-4209
CDRL SEQUENCE NO. A008

Prepared for:
Maryland Procurement Office

Prepared by:

Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113

Authenticated by ____________________ (Contracting Agency)   Approved by ____________________ (Contractor)

Date ____________________   Date ____________________

DTOS LESSONS LEARNED REPORT

Secure Computing Corporation

Abstract

This report includes the final report for the DTOS program and provides an overview of all of the technical efforts on the program. It describes significant accomplishments and obstacles encountered during these efforts and lessons learned while carrying out the program. Finally, it provides suggestions for future efforts which build upon the successes of the program or address deficiencies.

Part Number	87-0902025A006
Created	22 December 1993
Revised	27 June 1997
Done for	Maryland Procurement Office
Distribution	Secure Computing and U.S. Government
CM	`/home/cmt/rev/dtos/docs/lessons/RCS/report.vdd,v 1.20 27 June 1997`

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

Contents

List of Figures

List of Tables

Section 1 Scope

1.1 Identification

This document contains the Lessons Learned Report, CDRL A008, for the Distributed Trusted Operating System (DTOS) program, contract MDA904-93-C-4209.

The DTOS program began in October 1993, as a direct successor to the Distributed Trusted Mach (DTMach) program [64]. It is scheduled to complete in June 1997.

1.2 Document Overview

This report includes the final report for the DTOS program and provides an overview of all of the technical efforts on the program. It describes significant accomplishments and obstacles encountered during these efforts and lessons learned while carrying out the program. Finally, it provides suggestions for future efforts which build upon the successes of the program or address deficiencies.

The report is structured as follows:

- Section 1, **Scope**, defines the scope and this overview of the document.
- Section 2, **Program Summary**, provides a summary of the program's goals and results. It contains a list of all documents created on the program and a brief description of each document. This section is intended to take the place of a more traditional final report, and can be read independent of the rest of the report.
- Section 3, **Prototype**, provides an overview of the results from the prototype development tasks on the program.
- Section 4, **Performance Testing**, provides an overview of the results of this task.
- Section 5, **Assurance Overview**, provides an overview of the results from the assurance tasks on the program.
- Section 6, **Formal Security Policy Model**, provides an overview of the results of this task.
- Section 7, **Formal Top Level Specification**, provides an overview of the results of this task.
- Section 8, **Specification to Code Correspondence**, provides an overview of the results of this task.
- Section 9, **Composability Study**, provides an overview of the results of this task.
- Section 10, **Generalized Security Policy Specification**, provides an overview of the results of this task. This overview incorporates significant portions of the report [84] generated under the task, especially those portions expected to be of interest to a more general audience than the report itself.

- Section 11, **Covert Channel Analysis**, provides an overview of the results of this task.
- Section 12, **General System Security and Assurability Assessment**, provides an overview of the results of this task.
- Section 13, **Future Work**, provides suggestions for future work.
- Section 14, **Notes**, contains an acronym list.
- Appendix A, **Bibliography**, provides citations for all referenced documents.

Section 2
Program Summary

2.1 DTOS Objectives and Approach

The DTOS program is part of a broad operating systems research program known as Synergy [60], the objective of which is to develop a flexible, microkernel-based architecture for secure distributed systems. The current Synergy efforts are part of a long term strategy to encourage operating system vendors to include strong security mechanisms in the next generation of commercially available operating systems.

A near term element of this strategy is to develop a prototype which can be used to demonstrate the feasibility of including strong security mechanisms without sacrificing other desirable features. The prototype is also intended to be used as a platform for further research into other components of the Synergy architecture.

The main purpose of the DTOS program was to begin to address two of the basic needs of the Synergy program.

- To develop a prototype of the lowest level software components in the Synergy architecture, the microkernel and the security server.
- To provide assurance documentation and evidence that these components meet their security requirements, and more generally, to perform research addressing how to best provide evidence for the overall security of a system conforming with the Synergy architecture.

Each of these is discussed in a separate subsection below.

2.1.1 Prototype Objectives and Approach

DTOS is a successor to the DTMach program, which produced a high-level design for a secure distributed operating system. The DTOS prototype effort was almost exclusively limited to two components in this design, the microkernel and the security server.

The microkernel was designed as a collection of enhancements to the existing Mach 3.0 design [41, 42]. One purpose of starting from an existing microkernel was to demonstrate that security need not be incompatible with other desired features. The Mach microkernel itself was chosen for a variety of reasons, including its acceptance within the research and commercial operating systems communities and because source code to a Mach implementation was available, from Carnegie Mellon University (CMU), without restrictive licensing agreements.

The security server embodies the system's security policy. Its main function is to provide security decisions which are enforced by other service-providing components such as the microkernel. These policy enforcing servers query the security server for decisions during processing at certain well-defined control points.

The separation of policy decision making, performed in the security server, from policy enforcement, performed in other object servers, is a fundamental property of the Synergy architecture.

It encourages flexibility so that those other servers can be reused in many different policy environments.

The design goals of the prototype effort were the following:

Policy Flexibility The DTOS kernel should be capable of supporting a range of security policies, including mandatory access control policies such as a DoD multilevel security policy. To increase the ability to change between policies, system changes required by a change in policy should be localized within a single system component, the security server.

Mach Compatibility The DTOS kernel should be capable of supporting all existing Mach applications, subject only to the restrictions of the security policy being enforced. In particular, if the security policy permits all operations, then existing Mach applications should run without change.

Performance The performance of the DTOS kernel should be similar to that of the Mach kernel from which it is derived.

Comparing these to the goals of the Synergy program, the first goal emphasizes one instance of the flexibility which is important to Synergy, while the second and third goals aim to demonstrate that security mechanisms need not be incompatible with other features of a commercial operating system.

In addition to these three design goals, the other important goal for the DTOS prototype was that it be made available to any interested researcher, in particular, researchers involved in other Synergy related projects.

2.1.2 Assurance Objectives and Approach

It is not enough that a system is declared to be "secure". An individual responsible for evaluating a system for a particular use requires evidence that the system will actually meet the security needs of the users. We use the term *assurance* to refer to this evidence and the process of developing the evidence. Assurance evidence consists of two elements; a statement of the security properties that a system is claimed to satisfy, and some kind of argument that the system actually does satisfy those properties.

The statement of a system's security properties is generally done at a high level, so that they are relevant to users of the system. For instance, a security property of an operating system used for classified information might be that the system prevents a user from accessing data for which the user is not cleared.

On the other hand, the assurance argument often involves specification of the system at multiple levels of abstraction between the high level statement of the security properties and the actual implementation. Verification evidence is presented to justify that the specification at each level meets the requirements of the next highest level. Common forms of verification evidence include the following:

- Requirements tracing
- Engineering process documentation
- Use of cleared personnel
- Code reviews
- Code assertions
- Type-safe programming languages

- Testing
- Formal mathematical proofs

For the DTOS program, the primary goal of the assurance efforts is to perform research for assurance of systems constructed following the Synergy architecture. The particular aspects of this architecture with which the efforts are most concerned are security policy independence, decider/enforcer separation and modularity of the multiserver architecture. This research considered techniques for developing assurance evidence and for documenting and presenting the evidence. In most cases, the research was performed independent of the DTOS prototype, however, the prototype was often used to demonstrate the effectiveness (or ineffectiveness) of assurance techniques.

The DTOS research was strongly influenced by the use of formal mathematical methods. Formal mathematical theories were used as a basis for verification evidence and formal mathematical languages are used throughout the documentation.[1]

One drawback of this reliance on formal methods is that formal languages are often a barrier to the use of assurance documentation. To avoid this, an emphasis was placed on minimizing the reliance upon formal languages to only those situations when the corresponding English statements contained possible ambiguities. In particular, formal languages are considered as an enhancement to the English language rather than a replacement for it.

The emphasis on formal methods on DTOS is not an indication that the other forms of verification evidence are perceived to have no value. On the contrary, all of the listed techniques are valuable components of an overall assurance argument. Formal methods do however provide the strongest guarantee of completeness in an assurance argument. Completeness is especially important when verifying security properties, when even a single failure can have potentially devastating consequences. This is reflected in the TCSEC [51], where the only distinction between the highest rating (A1) and the second highest rating (B3) is the extent to which formal methods are used for system verification.

2.2 Significant Results and Accomplishments

- The DTOS prototype was successful as a proof of concept. While performance measurements are incomplete (see Section 4), the three design goals of the prototype were met sufficiently to demonstrate that, at the microkernel level, policy flexible security mechanisms need not be incompatible with other desired features.

- While the prototype was successful as a proof of concept, it also became clear during the program that the code base from which the DTOS prototype was developed is not a suitable code base for a secure, assured microkernel. The code is simply too complex and poorly structured to provide any meaningful assurance that it satisfies all of the required security properties.

  This is discussed further in Section 5.2.

- The DTOS program provided for distribution of the DTOS prototype and a government developed secure operating system environment to several sites performing research under the Synergy program. These sites were able to use the operating system as a platform for developing other prototype elements of a flexible secure system.

  This is discussed further in Section 3.5.

[1] The only significant task which did not rely on formal methods was the Specification to Code analysis task, which is essentially a highly structured code review.

- A demo was created to demonstrate an application satisfying a security policy specific to the application. The application relies upon a combination of security controls in the application and the security controls and services of the underlying platform.

  This is discussed further in Section 3.6.

- The results of the DTOS program have directly or indirectly influenced several other programs being conducted at Secure Computing. While none of these programs are specifically part of the Synergy program, they all have goals which will contribute to the overall success of the Synergy program.

  These programs are listed in Section 2.4, with a brief description of each.

- Because of the goal of policy flexibility in the prototype, it is important that security properties of the prototype be stated independent of any particular security policy. This allows any assurance evidence for the properties to be reused in distinct policy environments. The DTOS Formal Security Policy Model (FSPM) was written so that the basic security requirements on the DTOS prototype microkernel were stated in terms of security decisions retrieved from the security server, rather than in terms of specific security attributes of microkernel entities.

  This is discussed further in Section 6.1.

- The Synergy architecture emphasizes the use of multiple, interchangeable components. Therefore to provide assurance for a system constructed in this way, it is very desirable to be able to analyze the components individually and then easily combine the analysis to provide assurance evidence for the entire system. Using prior work by others as a base, a framework for composing assurance evidence of different system components was developed on the program. Worked examples of such an analysis were also conducted.

  This is discussed further in Section 9.

- One of the design goals of the prototype was policy flexibility. To consider the extent to which this goal was met, a framework was developed for evaluating the flexibility of a system with distinct policy decision making and enforcement components. This framework is general enough to apply to any system with an architecture providing a distinction between a policy decider and policy enforcers. It was applied specifically to the DTOS prototype to determine the range of policies supportable by DTOS. Results from this analysis were provided to the prototype effort to suggest ways to increase the range of supportable policies.

  This is discussed further in Section 10.

- The goal of policy flexibility similarly affects the covert channel analysis of the system. Ideally, covert channel properties can be stated independent of any security policy. Another important issue in covert channel analysis for a system built with the Synergy architecture is the ability to analyze the various servers independently.

  The DTOS program conducted several areas of research into theories and techniques for performing a covert channel analysis independent of a policy and within a multiserver architecture such as Synergy. A technique for performing analysis of a single server independent of a security policy was developed and demonstrated with a simple example. Properties of this technique and of traditional techniques were considered and some weaknesses common to each have been identified. Many open issues remain in this area.

  This is discussed further in Section 11.

- While the basic DTOS design was developed earlier on the DTMach program and had been subjected to considerable review, there are still other possible ways to meet the three basic design goals within a secure system implementing the Synergy architecture. A study was performed during the program to compare the security mechanisms and models provided by four other microkernel based secure systems. This study validated the approach taken by DTOS as providing the greatest policy flexibility and assurability, but also suggests some ways in which other mechanisms could potentially be incorporated into DTOS to provide further enhanced security.

  This is discussed further in Section 12.

- In any program such as Synergy which implements a long term strategy, it is important to record the results of each stage of the program to limit the tendency to repeatedly address the same issues. This Lessons Learned Report provides a comprehensive discussion of the various efforts undertaken on the program, including the significant contributions, methodology and lessons learned by each effort. This is expected to be valuable input to future elements of the Synergy program as they are undertaken.

- The long term goals of Synergy require involvement and interest within the research and commercial vendor communities. One vehicle for generating this interest is the presentation of papers at conferences. As part of the DTOS program, several papers were contributed to security and operating systems conferences, and four of these papers were accepted and presented.

  The published papers are described in Section 2.3.2.

2.3 Technical Documentation

One of the most significant outputs of the program is the DTOS prototype itself. All other outputs of the program are provided as technical reports. This section presents brief descriptions of all technical reports written for the program. It is divided into sections for contract deliverables (CDRLs), published papers, and other documents.

2.3.1 CDRL Document Summary

This section describes all technical documents provided as contract deliverables (CDRLs). The missing CDRLs in this list are for the prototype itself and program management documents.

DTOS Software Requirements Specification (SRS), CDRL A001 [78] This document is an updated version of the DTMach SRS [63]. In addition to requirements on the kernel and security server, it contains requirements for several other components of a secure distributed operating system. For these other components, the requirements have not been updated from the DTMach SRS since those components are not part of the DTOS prototype effort.

DTOS Kernel and Security Server Software Design Document (SDD), CDRL A002 [76] This document provides a design for the DTOS enhancements to the CMU Mach microkernel. It is derived from the DTMach System Segment Design Document (SSDD) [66], but the level of detail is much more focused. To provide the detailed information required to support the prototype work, the format of the document was changed from the high-level design format of an SSDD to the more detailed level of design work called for by an SDD.

DTOS Kernel Interface Document (KID), CDRL A003 [85] This document defines the interfaces to the prototype kernel and security server. It is based upon the OSF KID [41] for the OSF MK14 version of the kernel. In addition to defining the data structures visible at the interface, the KID defines the privileges required to be held by users of each interface.

DTOS Formal Security Policy Model (FSPM), CDRL A004 [72] This document defines the security policy enforced by the DTOS prototype kernel. The policy is defined formally using the Z specification language, however, there is also a version available [73], written only in English for readers that want an overview but are not interested in the formalization. The FSPM begins with a brief description of the Mach kernel data structures and then describes new kernel data structures required by the DTOS design; these data structures are used to identify the services provided by the DTOS kernel. Security requirements are stated governing when each service may be provided, in terms of permissions granted by the security server.

DTOS Formal Top-Level Specification (FTLS), CDRL A005 [74] This document provides a partial formal specification of the DTOS microkernel. It describes the system behavior both in English and in the Z specification language.

DTOS Covert Channel Analysis Report, CDRL A007 [67] This classified document contains the informal description and analysis of each of the interference channels identified in the Journal Level Proofs (CDRL A016) [69]. The analysis includes descriptions of possible implementation scenarios, bandwidth estimation and techniques for slowing or closing the channel.

DTOS Lessons Learned Report, CDRL A008 [86] This report includes the final report for the DTOS program and provides an overview of all of the technical efforts on the program. It describes significant accomplishments and obstacles encountered during these efforts and lessons learned while carrying out the program. Finally, it provides suggestions for future efforts which build upon the successes of the program or address deficiencies.

DTOS General System Security and Assurability Assessment Report, CDRL A011 [83] This document contains the results of the DTOS Essential Requirements Study. It presents criteria for assessing microkernel based systems in their ability to be configured to meet a range of security policies and the feasibility of assuring that a system meets the security requirements of the policies. Five systems are then assessed against these criteria. The goal of this report is to provide guidance for future secure system development by providing the design criteria and examples of designs which meet and do not meet those criteria.

DTOS Software System Development Test and Evaluation Plan, CDRL A012 [70] This document defines the performance and functional tests which were performed on the development releases of the DTOS kernel.

DTOS Software Test Report, CDRL A013 [71] This document describes the results of the performance and functional tests defined in CDRL A012. It has two volumes, one documenting results of the performance tests and one documenting results of the functional tests. Note that performance testing was discontinued before the end of the program because of the difficulty of obtaining consistent results.

DTOS Risks and Mitigators Report, CDRL A015 [87] This document provides a biannual listing of a database recording risks associated with the development of a distributed, trusted operating system based upon the DTOS conceptual model. This database was developed on the program and associates with each risk a severity rating, the development phase of DTOS in which the risk was found, factors which indicate the risk is present, potential cost and schedule consequences of the risk, potential mitigators for the risk, and the mitigators effectiveness observed on DTOS. The most significant risks from this database have been incorporated in some form into the discussion within this Lessons Learned document.

DTOS Journal Level Proofs, CDRL A016 [69] This classified document contains the formal analysis of the noninterference properties of the DTOS system. The methods for performing the analysis are described and justified in the DTOS Covert Channel Analysis Plan (CDRL A017) [82]. The result of the analysis is the identification of design level interface flows according to the Formal Top Level Specification (FTLS) of the system. Due to the experimental nature of the analysis, it is only performed on a small part of the system for a proof-of-concept.

DTOS Covert Channel Analysis Plan, CDRL A017 [82] This document describes an approach for performing analysis of a system's ability to satisfy information flow security policies. Although the plan addresses source and object code level analysis, it focuses on design level analysis, considering development of both a theoretical framework and a specific methodology. The document contains discussion of deficiencies in current theory and practice for performing such analysis in general and for multiserver operating systems in particular.

DTOS Specification to Code Correspondence, CDRL A018 [79] This document contains an analysis of the correspondence between the DTOS FTLS and the implementation. It includes a data correspondence that maps components of the FTLS state to data structures in the implementation and a functional correspondence that correlates FTLS state transitions to the implementation of corresponding kernel requests. The report documents any discrepancies. Due to the experimental nature of the analysis and to the lack of tools to support the analysis, the functional correspondence considers only a small part of the system. It also discusses some serious shortcomings of current practices and the need for additional research and tool development.

DTOS Generalized Security Policy Specification, CDRL A019 [84] This document analyzes the policy flexibility of the DTOS architecture and records insights into the effect that the goal of policy flexibility has on the design of an object manager and its interface with the rest of the system. The report first surveys a variety of security policies from the computer security literature. It then gives a general framework for modeling a system in which security enforcement is separated from the making of security decisions. The DTOS kernel and security servers for several example policies are specified within this framework and the resulting systems analyzed. The report also identifies a list of policy characteristics and classifies a variety of policies with respect to these characteristics.

DTOS Composability Study, CDRL A020 [81] This document studies existing composability techniques and applies them to analyzing DTOS. The goal is to assess these techniques for specifying and verifying modular systems by composing specifications and proofs for the individual system components and to develop new techniques as necessary. The report includes a framework for composition and an example of the composition of several components.

2.3.2 Published Papers

This section describes all papers presented at conferences.

Providing Policy Control Over Object Operations in a Mach Based System [49] This paper provides an overview of the DTOS prototype and control mechanisms. It begins by describing the basic elements of the Mach kernel and the Mach access controls. It then describes the controls added to the kernel under the DTOS program and interactions with the security server. It concludes with analysis of some preliminary performance figures.

Developing and Using a Policy Neutral Access Control Policy [53] This paper describes the DTOS security policy flexibility. It begins with a discussion of the basic architecture and the control philosophy for DTOS. It describes the structure of the security policy document, and discusses the range of policies supportable by DTOS.

Defining Noninterference in the Temporal Logic of Actions [27] This paper is an output of the DTOS Composability Study. It describes an approach for using Lamport's Temporal Logic of Actions to specify noninterference properties. In addition to providing a more intuitive definition of noninterference than previous attempts, this approach also supports analysis of systems that do contain covert channels to demonstrate limitations on their exploitations. In relating the definition of noninterference given here to prior definitions of noninterference, this paper also discusses ways in which other definitions of noninterference can be formalized in TLA. Finally, this paper discusses how prior work on specification refinement and composition might be applied to the noninterference problem within the framework provided by TLA.

A Framework for Composition [28] This paper is also an output of the DTOS Composability Study. It describes an interim version of the composability framework developed as part of the study. It explains the advantages of the new framework, and illustrates the use of the framework through a simple example.

2.3.3 Other Technical Documents

This section describes technical documents which were developed on the program but not officially identified as contract deliverables.

DTOS Users Manual [80] This document is the user's manual for the DTOS prototype system. It provides background concepts, procedures, and reference information needed for installing and using the DTOS prototype. It should be consulted by anyone who plans to use the DTOS system.

DTOS Demonstration Software Design Document [68] This document defines the design of the DTOS demonstration software. This example application simulates a hospital database and demonstrates how the security policy can be used to control access to medical records.

DTOS Notebook of Technical Issues [77] This document records various technical discussions from the program. The purpose is to record rationale for decisions that have been made, track technical issues to closure, and record discussion on technical issues that have not yet been resolved. It is not always easy to follow because the discussions are generally included as they occur, with little editing or rewriting for clarity.

2.4 Dependent Programs

Several other programs currently underway or completed have evolved directly or indirectly from the work performed on the DTOS program. This section briefly describes these programs and their relationship to DTOS.

- Adaptive Security Policies (December 1994 to January 1996)

  This program used the DTOS platform for experimentation with some adaptive security policies which had recently been presented theoretically. Some areas of interest included coupling between policy and implementation, control over policy changes, effect of stale cached data, reassigning security attributes, and recovery from change. An adaptation of an MLS policy enhanced with Type Enforcement to a similar but more permissive policy was demonstrated, first using a single Security Server in which the policy table is replaced, and then handing off security decisions from one Security Server to another. Exercise of the relaxed permissions was audited, as specified by the Security Server.

  The program also studied dynamic security lattices and task-based access control, and the issue of trade-offs between security and fault tolerance. These studies were presented in the context of the Synergy architecture.

  The results of this program are documented in [61].

- Adding Security to Commercial Microkernel Based Systems (June 1995 to January 1996)

  As mentioned earlier, one result of the DTOS program was the determination that the CMU Mach code base used on DTOS was too complex and poorly structured to meet the requirements of an assured, secure system. Also recall that one of the goals of the Synergy program was to involve commercial operating system vendors.

  In an attempt to address both of these issues, this program investigated the possibility of using SunSoft's Spring microkernel based operating system as a code base for a secure system in the Synergy architecture. Unfortunately, due to the research nature of the Spring system the code is similar in structure and complexity to the CMU Mach code, and it therefore would also fail to meet the needs of an assured, secure system. Moreover, SunSoft's lack of interest in the Spring system itself eliminated the possibility of any cooperative effort to use Spring as a base for a secure system.

  The results of this program are documented in [91].

- Supporting a Secure DBMS on the DTOS Microkernel (August 1995 to August 1996)

  This program investigated the possibility of supporting a secure distributed database management system (DBMS) on the DTOS microkernel. The work made use of a preliminary design project done by MITRE Corporation, which in particular identified some desirable features of a microkernel based operating system hosting a secure distributed DBMS. Strengths and weaknesses of DTOS in supporting these features were identified.

  The results of this program are documented in [48].

- Adaptive Security Policy Experience (August 1996 to present)

  This program continues the experimentation with adaptive security policies which was begun on the Adaptive Security Policies program discussed above. Particular areas of interest include:

  – Assessing the impact on system assurance of switching between policies.
  – Assessing the usefulness of auditing the switch/recovery process.

– Developing tools to facilitate construction of security databases.
– Assessing tradeoffs between different mechanisms for implementing policy changes.

- Hypervisors for Security and Robustness (August 1996 to present)

  This program explores a concept similar to that of a hypervisor in the work of Bressoud and Schneider [16], but implements the hypervisor on top of an operating system kernel rather than on top of the hardware. These kernel hypervisors are constructed from wrappers placed around system calls for selected system components. They could be used to make a component more robust, as described by Bressoud and Schneider, or to perform various security functions such as fine-grained access control and auditing of events.

  This program will use the results of the DTOS composability study to analyze the security properties of "wrapped" components.

- Composability for Secure Systems (October 1996 to present)

  The objective of the CSS program is to develop and demonstrate a composable methodology for building highly-assured, secure, fault-tolerant distributed systems and networks, and to design an automated development environment to support the methodology. The methodology will be demonstrated via the development of a microkernel-based network server design that transparently extends IPC across a network.

  This program builds directly upon the efforts of the DTOS composability study.

2.5 Future Plans

The ultimate goal of the Synergy program is still unrealized, and there is considerable work to be done. Due to the determination that the Mach code base from which the DTOS prototype was developed is insufficient for providing high assurance, it is likely that further efforts with the DTOS prototype will not be pursued beyond 1997. Currently, the prototype is being maintained while other Synergy efforts remain dependent upon it, but no longer than required for that purpose.

It is expected that the next generation Synergy microkernel will be ready for distribution to Synergy research sites soon. This microkernel will be based on the Fluke [33] microkernel which is being developed at the University of Utah as the logical successor to Mach within the operating systems research community. Security enhancements to this kernel, similar to those made to Mach during the DTOS program, are being added by government researchers.[2]

Secure Computing will continue to participate in Synergy research and development after making the transition to Fluke. A program is in place to provide some of the basic assurance infrastructure for the Fluke kernel and to continue research in the areas of security policy modeling, specification and composability.

Other areas of interest for future work are described in Section 13.

[2] The resulting security enhanced microkernel has been termed Flask, for "Fluke advanced security kernel".

Section 3
Prototype

DTOS is part of a broader operating systems research program known as Synergy [60], the objective of which is to develop a flexible, microkernel-based architecture for secure systems. The main emphasis of the DTOS prototype effort was to develop a prototype security enhanced version of the CMU Mach microkernel as the base for the Synergy operating system project. The high-level design of the prototype was developed as part of the DTMach program.

The primary goals for the DTOS prototype were the following:

Policy Flexibility The DTOS kernel should be capable of supporting a range of mandatory access control security policies, such as a DoD multilevel security policy.[3] To increase the ability to change between policies, system changes required by a change in policy should be localized within a single system component.

An assessment of the range of policies ultimately supportable by DTOS was one of the purposes of the Generalized Security Policy Study (see Section 10).

Mach Compatibility The DTOS kernel should be capable of supporting all existing Mach applications, subject only to the restrictions of the security policy being enforced. In particular, if no security policy is being enforced, then all existing Mach applications should run without change.

Performance The performance of the DTOS kernel should be similar to that of the Mach kernel from which it is derived.

Testing to determine the performance impacts of the DTOS kernel enhancements is discussed in Section 4.

3.1 Architecture

A key distinction among microkernel-based secure operating systems is the amount of explicit security functionality within the microkernel itself. Two microkernel systems designed to meet DoD multilevel security policies, KeyKOS/KeySAFE [34, 40] and Trusted Mach [93, 5], have kernels which provide essentially the minimal amount of security function required to meet the policies for which they were designed, while still remaining free of any explicit dependencies on the particular policy. In particular, the security policy for these systems is largely defined in terms of server layer entities external to the kernel.

DTOS takes a more aggressive attitude towards providing security mechanisms in the microkernel. The approach taken on DTOS is to provide a general mechanism for control over all kernel operations, therefore allowing the security policy to be defined much more in terms of kernel entities. The main advantage of this approach is to provide another layer of defense if higher level security features are ever penetrated. For instance, in both KeyKOS/KeySAFE and Trusted Mach, a single security failure in the server layer may allow an attacker to perform

[3] The DTMach program also considered discretionary access control policies, however this was one of the elements of DTMach which was not pursued further on DTOS due to resource limitations.

many forbidden operations through the kernel. In both systems this can ultimately lead to a total breakdown of security.

While this defense-in-depth is the main reason for adopting kernel based security, there are other potential benefits as well:

- Providing additional features in the kernel can simplify development of security features in trusted servers and applications.
- The DTOS approach provides a single consistent model for protection of all kernel entities, unlike both KeyKOS/KeySAFE and Trusted Mach. This model simplifies understanding and analysis of the system's security properties.

In order to support the goal of flexibility and localization of the security policy, the DTOS architecture provides a strict separation between security policy enforcement and security policy decision making. The DTOS kernel provides policy enforcement by associating every kernel operation with a particular access right which the client performing the operation must have to the object on which the operation is performed. The kernel learns whether the access right is granted by consulting a distinct component, the *security server*, which embodies the system's security policy.

Therefore the DTOS prototype consists of two major components, a security enhanced Mach kernel and an example security server. These components are discussed in Sections 3.2 and 3.3, and their interface is discussed in Section 3.4.

3.2 Kernel Enhancements

The main purpose of security enhancements to the kernel is to provide enforcement of security server decisions over all kernel operations. As stated above, these decisions define whether a particular access right is granted from a client task to some object.

However, the kernel does not ask the security server for a decision based on the actual identity of the client or the object. Instead, the kernel associates a *security identifier* (SID) with each client and with each object, and provides the security server with the security identifiers. This relieves the security server of the responsibility of recognizing every system entity individually, and similarly relieves the kernel of the responsibility of informing the security server about every new entity as it comes into existence.

Therefore there are three primary enhancements to the DTOS kernel:

- The kernel manages the relationship between SIDs and kernel objects. This includes assignment of SIDs to kernel objects through default rules and providing new interfaces to allow clients to explicitly set or get the SIDs of kernel objects. This is discussed further in Section 3.2.1.
- The kernel provides access control over each kernel operation, and an interface to retrieve decisions from the security server. This is discussed further in Section 3.2.2.
- The kernel uses security decisions from the security server to define the hardware based memory access controls. This is discussed further in Section 3.2.3.

In addition, there are a few miscellaneous enhancements:

- A new set of interfaces for task creation was added, with the primary goal of allowing a task to create another task more trusted than the creator. These enhancements met with mixed success. This is discussed further in Section 3.2.4.
- Extensions were added to the messaging interface to allow clients to set and get the SIDs associated with the message sender and recipient. This is discussed further in Section 3.2.5.
- A new set of name service interfaces were added, to hold capabilities for accessing trusted servers. Ultimately, these interfaces do not belong in the kernel itself but rather in a trusted name server, but since there is no trusted name server, they were added to the kernel as a temporary solution. This is not discussed any further.

Finally, mechanisms for caching security server decisions within the kernel were added. Though not a part of the fundamental security architecture, this is a significant topic in itself and will be discussed much more in Section 3.4.

3.2.1 SID Management

A SID represents the security attributes of an object within a particular security policy. The DTOS kernel associates SIDs with all kernel objects which are visible through the kernel interface and relevant to the security model. Since the kernel is independent of the security policy being enforced, the kernel does not interpret or in any way make decisions based upon the content of any SIDs, other than the *classifier*.

The classifier is a standard field within the SID that provides a numerical representation of the class (or type) of an object. The classifier is needed to ensure the proper interpretation of access right values in communications between the kernel and security server.

SIDs are associated with kernel objects from the time of creation to their destruction. The original assignment of the SID is based upon certain class specific default rules. To provide an alternative to these rules, DTOS added object creation interfaces which are identical to the Mach interfaces but which also allow the client to specify a particular SID to associate with the new object.

The SID of objects can also be retrieved by a client using other interface extensions.

The SID of object is generally not allowed to change. The only exception is that an interface allows a client to request to change the SID of a task. As a consequence of a change to a task SID, the SID of all threads associated with the task are also changed to correspond to the new task SID.

3.2.2 Kernel Access Control

As mentioned above, the focus of the DTOS kernel enhancements is a consistent model for access control over all kernel operations. This was accomplished by insertion of control points into the kernel. At each control point, the security server is queried for a decision about whether an impending operation is allowed. The security server's decision at a control point is based upon two SIDs and a required access right.

There is at least one control point associated with every kernel request. However, for some requests there are multiple control points, because the control points are ultimately defined in terms of kernel *services*, and some requests perform multiple services. The definition of the kernel services to be controlled and the required access rights for each service was performed as part of the Formal Security Policy Model (FSPM) task and is discussed at length in Section 6.

There are two factors determining the location of the control points. First are the requirements of the FSPM, which define which services are controlled, and how. But this leaves open exactly where the control point is placed in the code. In general, we distinguish between two kinds of control points:

- *Initial permission checks* are performed in the kernel's request dispatch code or at the entry point to the main processing routine for a particular kernel request.
- *Deferred permission checks* are performed in lower layers of the code.

Since the standard case for the access control model is to perform a single permission check for each request between the client making the request and the object being accessed, initial permission checks are the typical case. They are also by far the simplest to insert into the code because they do not require detailed understanding of the code in order to locate the proper location for the control point and they do not require any clean-up if an access right is denied by the security server (see Section 3.2.2.3). However, initial permission checks require that both of the SIDs and the required access right be directly available at the kernel entry point.

Deferred permission checks are only required when it is necessary to perform some kernel processing before identifying all of the information required for a permission check. In these situations it is often the case that processing needs to be performed before it is even clear that a permission check needs to be performed. For instance, the DTOS kernel controls the transfer of port rights in messages, but a message must be parsed in order to find all of the port rights being transferred.

The remainder of this section discusses some problems encountered while adding control points to the kernel and some possible solutions.

3.2.2.1 Code Generation for Initial Permission Checks

On DTOS, initial permission checks were coded in two ways.

In the typical case, the security decision requires the SID of the client task, the SID of the target object on which the request is made, and an access right depending only upon the particular request being called. In this case, the permission check is performed as part of the common request dispatch code and the access right required for each request is defined as part of a large table.

In other cases, all information necessary to make a permission check is available at the request interface, but it is not exactly the information required in the standard case. Two typical examples are:

- One of the SIDs can depend upon a parameter to the request. Often the SID itself is a parameter to the request, such as when creating a new object with a client specified SID.
- The access right being checked can depend upon a parameter to the request. A typical example is a request which returns various kinds of information, and the specific information returned is dependent upon a parameter. Different access rights may be required for each kind of information, and thus the access right is dependent upon the parameter.

In these cases, the permission check is typically performed immediately upon entry to the particular routine being called.

In both of these situations, it would have been possible to extend the MIG interface generator to automatically generate the code performing the permission checks. This approach was not taken on the program because the initial emphasis was upon determining how to insert the permission checks, not how to insert them most efficiently. Moreover, gaining an understanding of the Mach microkernel was quite time consuming and attempting to simultaneously understand MIG in sufficient detail to modify it in this way did not appear to be a good use of time.

In retrospect, this was an unfortunate decision. Maintenance of the initial permission checks would have been greatly eased by integration with MIG. In particular, the large table used in the standard case was extremely prone to error because it was based upon the order in which interfaces were defined in the MIG datafiles. When this order was changed (because of addition or deletion of interfaces), the table was no longer accurate.

3.2.2.2 Monitoring of Permission Checks

The code initially added at kernel control points only included a query for a decision by the security server. However, it later became clear that it was necessary to provide a means for logging all permission checks, and code was added at the control points to provide this logging.

In retrospect, this code should have been inserted concurrent with the initial code additions, since logging made it easier to identify missing and redundant control points.

3.2.2.3 Problems with Deferred Permission Checks

Deferred permission checks can present serious difficulties when the control points are deeply buried in the kernel. One problem that can arise is a deadlock situation due to calling out of a routine while various structures are locked. This occurred occasionally but was not a significant source of difficulties.

Much more serious problems can arise in the case of failed permission checks. A failed permission check indicates that some imminent operation cannot be performed, and that some exit path must be followed out of the current body of code. This presents little problem for initial permission checks, because they are located where the existing code is prepared to handle error situations which result in aborting some operation.

But when a control point is located deep within the kernel there may be no existing error handling code which can be used to ensure that the operation is aborted while leaving the kernel in a consistent state. In a few cases, not only is there no existing provision for errors, but the kernel explicitly relies upon a routine to complete successfully and yet a failed permission check may prevent that from happening. Two examples of this situation are the following:

- When the kernel sends a message, it assumes that the message can actually be enqueued at the requested port. DTOS permission checking invalidates this assumption.
- Permission checking may actually prevent garbage collection by the kernel. For instance, the FSPM requires a permission check before a port right is destroyed. But port rights are destroyed when garbage collecting obsolete ports, and it is unacceptable to prevent garbage collection from occurring.

No conscious effort has been undertaken on DTOS to fix problems caused by deferred permission checks or even to understand the scope of the problem. This may be acceptable in a prototype, but obviously is not if a more robust system is required.

3.2.3 Memory Access Controls

In addition to controlling kernel operations based upon the decisions of the security server, the DTOS prototype controls all memory accesses based upon those decisions. Whenever a new region is added to a task's virtual memory space, the kernel queries the security server to determine the accesses which the task is allowed to that memory region, based upon the SID of the task and the SID of the memory object backing the region. The response from the security server defines the maximum accesses allowed. The protection bits in the page table are then set according to these accesses or they may be further limited based upon the Mach memory access controls.

The security server interface and the high level Mach code support any combination of read, write and execute permission to a memory region. However, the low level Mach code only supports three distinct modes: no access, read/execute and read/write/execute. The failure to provide read access without simultaneously supporting execute privilege is a characteristic of most operating systems though it is sometimes considered to be a significant shortcoming for supporting a secure system (see for instance Section 3.3.2 of [83]).

3.2.4 Task Creation Interfaces

In Mach, when a task creates a second task it is given total control over the state of the second task throughout its existence, and it similarly receives any privileges granted to the second task. The DTOS kernel includes enhancements to the task creation interface designed to weaken the link between a task and its creator. The enhancements allow a security policy to specify that the creator be allowed to initialize a newly created task while limiting further interactions between the two tasks.

One purpose of these enhancements was to provide for least privilege, in accordance with process models that distinguish the responsibility of a creator during initialization of a new process from subsequent responsibilities.

However, one other goal of these enhancements proved to be difficult to meet. It was hoped that the enhancements would make it possible for a task to create another task which could be granted more privileges than the creator. This ability had been of use on the LOCK program, for instance to allow an untrusted shell to initiate trusted processes.

The difficulty in accomplishing this goal is due to requirements which are placed upon the initial state of trusted tasks. If a task creates a more trusted (or privileged) task, then the creator cannot be responsible for verifying the initial state of the new task. But the Mach kernel is designed to be independent of any particular process model, and therefore it is unable to verify the initial state of the new task. Various potential techniques were identified for solving this problem, all of which required kernel enhancements so that the kernel could understand enough about binary formats to verify that the task's initial state was valid.[4]

In the end, none of these potential solutions were adopted. Instead, it was decided that the ability to create a task more trusted than the creator should instead be provided by a user level server. The creator makes task creation requests to this server, and it is then the server's responsibility to define the initial state of the new task. Since the initial state is dependent upon the higher level operating system environment, this approach is consistent with keeping all explicit operating system dependencies out of the kernel.

[4] The DTOS Notebook [77] contains a record of the various techniques considered.

3.2.5 IPC Extensions

The Mach IPC interface was extended to incorporate sender and recipient SIDs in the following ways:

- The recipient of a message can retrieve the SID of the task which sent the message. If the recipient is a server, it can perform access control for a request based upon this sender SID.
- The sender of a message can specify a SID by which it should be known during processing of a message and to the recipient. This service is controlled by the security policy. Some uses for this service include:
  - A task may specify a different SID to provide anonymity. For instance, a task which is transparently interposing between two tasks (e.g., a debugger or network server) can forward a message from one of the tasks to the other without changing the sender SID.
  - A task may specify a different SID to limit the privileges which may be associated with some message. For instance, a server task which makes a request on behalf of a client task can specify the client task's SID to prevent the server from making a request which the client is not allowed to make.
- The sender of a message can specify a SID which must match the SID of the task which receives the message. This can be useful when the message contains particularly sensitive data (e.g., a password) or when the sender wishes to ensure that a particular server perform some service requested in the message. If a task with a different SID attempts to receive the message, the message is destroyed.

  This feature can unfortunately cause messages to be destroyed if a task is interposing between the sender and the intended recipient. Therefore, in some cases the security policy may allow a task to receive a message even if the task's SID does not match the SID specified by the sender.

3.2.6 Obstacles

This section lists several general obstacles that were encountered while implementing the DTOS kernel enhancements.

3.2.6.1 Application Portability Whenever changes are made to a system, there is the risk that applications already in use may need to be rewritten to run on the new system. The DTOS prototype development process has been structured to try to ensure that we maintain backward compatibility with the UNIX emulator and applications provided with Lites. Prior to the DTOS transition to Lites, the goal was to maintain backward compatibility with the UNIX emulator and applications provided with the CMU release. The first test action with each new kernel is to boot the existing system and ensure that the existing UNIX environment continues to operate correctly. After the Lites transition, a small set of applications was selected for regression testing purposes. They include the DTOS demo, and other applications that the Government provided to run on top of Lites. This was better than simply verifying that the system booted, however many problems were still missed. To provide adequate regression testing, a careful set of applications that exercise a wide range of system functions must be available.

3.2.6.2 Lack of Understanding of Implementation Details

The success of building a secure system by adding security to an existing system is affected by the extent of detailed knowledge about the system being modified. A review of the performance track on the prototype effort shows that this risk is real. We have had consistent problems in meeting cost and schedule estimates and lack of understanding is the most frequently referenced reason. There must be enough time allocated to study the existing system in detail. Taking the time to understand something before changing it can save time in the long run.

3.2.6.3 Needed State Missing in Low-Level Routines

The design of Mach, version MK83, is such that many low-level operations (e.g., allocating memory objects) are done in an environment where the addresses of important higher-level structure (e.g., the address map's owning task) are not available. This problem occurs to some extent in the IPC software and to a much greater extent in the VM software. As a result, there was a need to add parameters to existing entries and to add duplicate routines providing similar functionality but with modified parameters to provide more information. This problem was solved by adding back pointers to structures and changing parameters. When retrofitting a system, one should expect to make low-level structure changes.

3.2.6.4 Deficiency in Mach Code Structure

Whenever enhancements are made to an existing system, the success of the enhancements is dependent upon the ability to understand the structure of the existing code. On DTOS, the structure of the baseline Mach code is not as clean and logical as we had hoped. For example, there are many in-line "optimizations" that have been made to overcome a number of performance problems that were inherent in the initial implementation of the Mach system.

Another example is the poorly defined locking protocols. In some cases, locks that appear to have been missing result in invalid pointers showing up. Problems at this level of the system implementation are difficult to identify and resolve and may have a significant impact on integration and system testing efforts.

Lack of structural integrity of the system being modified makes it more difficult to correctly insert the required permissions checks. Inclusion of special case processing adds to the complexity and makes it much more difficult to ensure that the exact set of required permission checks are being made.

3.2.6.5 Code Changes from Outside Sources

Any time code developers make changes to code that they initially received from another source, adding code updates and patches received from that source can be difficult. On DTOS, the changes that SCC has made to the Mach 3.0 kernel code can make adding updates and patches from CMU more difficult. There is currently no automatic way to identify the differences between the current code and revised code from CMU without generating a large amount of extraneous information. This difficulty could have been overcome by minimizing the amount of changes to the original source code, or keeping the code changes in separate files from the original source code. However, this was not achievable because the permission checks required by DTOS necessitate the placement of security checks in CMU distributed source files. The lesson here is that the further one deviates from the baseline code, the more difficult it is to incorporate outside code changes.

This was initially expected to be a concern for the DTOS prototype, however, since CMU stopped supporting Mach shortly after the program began, there were no updates to incorporate.

3.2.6.6 Bugs in Code from Outside Sources Any time code developers make changes to code that they initially received from another source, bugs in the latter may cause problems with their system. The Mach 3.0 microkernel code, as delivered from CMU, contains bugs. This presents a problem during implementation and execution of the DTOS prototype. After the prototype is completed, the instability of the Mach microkernel might complicate attempts to determine if the system is functioning correctly. The project dealt with this problem by doing the following:

- Compile and keep a list of known bugs and work-arounds about the software that you are working with.
- Check with the Mach community for a list of known bugs and work-arounds.
- Cross-compile DTOS to remove the unreliability of Mach as a factor that affects software development efficiency.

The above steps provided adequate relief, especially the cross-compilation, and significantly accelerated the development effort.

3.2.6.7 Prototype Maintenance and Baseline Prototype Bugs When making additions and changes to an existing system instead of developing a system from scratch, problems may result in trying to identify which problems came from the changed and added software versus what problems came with the baseline system. On DTOS, problems could have occurred during the "Support for Fielded Prototype" task if sites that received our prototype had identified problems with the prototype that were problems in the baseline Mach system instead of problems that we introduced into the system. We needed to be clear on which problems came with the baseline Mach system and which problems were introduced by adding security to the system, so that we did not spend time trying to fix problems that were not our own. DTOS kept a list of prototype bugs and tried to determine the underlying problem for each of them. Additionally, it may have been helpful to distribute the list of known Mach bugs to the user community.

3.2.6.8 Side Effects of Changes Any change made to a system where the system makes undocumented assumptions could cause the system to break. For example, system implementations that rely on the timing of events could break when new events which cause context switches (such as security faults) alter the timing. It is necessary to be very cautious when making code changes. In several places, the kernel code had to be restructured to avoid these problems.

3.2.6.9 Functional Testing When retrofitting security to an existing system, it is important to perform more testing than when developing a secure system from scratch. While the developers of a new system have complete knowledge of the system, developers retrofitting security must reverse engineer the existing system.

In the process of reverse engineering a system, a developer constructs a set of properties that are assumed to be satisfied by the system. Depending on how these properties interact with the security extensions, security flaws can result from violations of the assumptions.

One example of this from DTOS is the assumption that port structures are not reused. An instance was discovered in which the kernel optimized the destruction of one port right and allocation of a second port right into reuse of the first port right. This was implemented by changing the pointer associated with the existing port right to the new port structure. However, the only place the pointer into the DTOS access vector cache (AVC) was set was during the creation of a new port right. So, the existing port right retained a pointer to the AVC entry for the old port but was associated with the new port which could have a different label.

Detecting such problems requires more “testing” of the assumptions. This testing could either be actual testing, the definition of assertions, or defensive programming in which more validity checks are performed within the code.

When retrofitting security to an existing system, the possibility that assumptions about the system are incorrect must be accounted for in the system development. Otherwise, surprises are likely to be encountered relatively late in the development effort.

3.2.6.10 Desired Robustness of the Prototype Some difficulties on the program arose from the different expectations about the required maturity and robustness of the prototype. The statement of work called for creation of a “rapid prototype”. This term seems to imply that the purpose of the prototype was to provide evidence that the concepts behind the planned system were sound, not to provide a robust implementation of those concepts. However, at the same time, the prototype was expected to be distributed to interested sites who would be using it as a platform on which to perform their own research. Clearly such users require reasonable stability from the prototype. Furthermore, it is unlikely that the government actually wanted a prototype in the sense described above, since they had already created a prototype of their own to demonstrate feasibility of the concepts.

This conflict led to a situation in which the processes followed for development of the prototype were inconsistent in their level of maturity.

Some processes used were certainly more appropriate for a prototype system:

- During the initial development, the main emphasis was on making changes to the kernel, with less emphasis on ensuring that all changes were made correctly. For instance, a major effort was made to insert control points into the kernel, though there was less emphasis on making sure that the correct access right was being checked at each control point. Also, as discussed in Section 3.2.2.3, little effort was put into ensuring that the kernel would always remain in a legal state if an access right was denied.
- After the initial release, little documentation of additional features or code changes was done before implementation, and often, very little was done after implementation.
- Most new features were incompletely tested. Generally, testing of a new kernel request consisted of using the request with a simple set of parameters. Full exercise of the code, testing of boundary conditions, etc. was not done.
- Even the minimal tests that were developed for new features were generally not repeated on subsequent releases.

On the other hand, some processes exerted excessive control over the prototype and resulted in excessive time spent in unproductive efforts:

- The process followed when releasing updates of the prototype was based upon a company wide release process for actual products.
- The process for tracking bug reports and enhancement requests required detailed logging.
- Rigorous version control and configuration management was used both on software and on documentation.

In retrospect, the development processes should have found a consistent path somewhere between these two extremes.

3.3 Security Server Design and Implementation

The security server is the component which makes security decisions within a DTOS system. It executes as a normal Mach task in user space. The implications of executing the security server as a distinct task will be addressed in Section 3.4. This section focuses on the security decision making within the example security server implemented as part of the DTOS prototype.

A security server has two basic functions:

- Provide a mapping between SIDs and security attributes.
- Respond to requests for security decisions by determining if a particular requested access right can be granted between two SIDs.

The DTOS prototype security server implements an MLS and Type Enforcement security policy. While it is not intended to be anything more than an example of a security server, it was designed with some modularity to make it easier to adapt to other policies if desired.

3.3.1 SID Interpretation

The security server maintains the map between SIDs and security attributes. It presents interfaces to convert from either representation to the other. Since the kernel is uninterested in specific security attributes, this interface is of use only to user space clients. Moreover, the data format in these interfaces is policy specific because the meaning of security attributes is policy specific.

Within the prototype security server, each SID maps to the following attributes:

- A classifier. This is the one field in the SID which is not policy dependent.
- A type enforcement attribute (domain or type).
- A hierarchical security level (unclassified, confidential, secret, top secret).
- A collection of categories (such as NATO) to augment the hierarchical level for the MLS policy.
- An authentication context (which typically indicates a user name).

Initially the SID was decomposed into five distinct fields and the mapping from SID to attributes was very direct. Later this mapping was changed to be opaque so that the SID could not be interpreted outside of the security server. One of the reasons for doing this was to eliminate the tendency of developers working on application outside of the security server to rely upon the particular format rather than always deferring to the security server's interpretation of a SID.

3.3.2 Security Decision Making

The heart of the security server is the decision making function. The prototype security server makes security decisions through a combination of hard-coded logic which defines broadly how the kinds of attributes interact and security databases which allow properties and interactions among specific attributes to be defined and altered without recompilation of the security server.

3.3.3 Security Databases

Ultimately, the security policy is defined in the security databases which are interpreted by the decision making code. Therefore the ability of a security administrator to accurately define and interpret the databases is essential. The tools available to an administrator in the existing prototype are terribly inadequate for this purpose, and even as a prototype, a much richer tool set would have been very helpful.

The internal complexity of the system requires a large security database and the addition of many security database entries for the addition of each new application. The presence of security database management tools at the same time as database development would have sped up the process tremendously. Database tools should have been developed concurrently with the development of a security server so they will be available even as the initial databases are constructed.

After the initial database development, the continued deficiency in security database tools meant that users of the prototype were required to build and maintain the database files by hand, which was both error prone and time consuming. The initial problem which this leads to are difficulties in setting up the databases to allow an application to even become operational. A longer term problem is that it is difficult to determine when the databases are becoming excessively permissive.

Finally, there is a concern that the lack of database tools will lead users of the prototype to conclude that the DTOS approach to providing a high degree of control over kernel operations is unmanageable.

3.3.4 User Level Policy Enforcement

From the point of view of the DTOS kernel, a security policy can be expressed as sets of access rights allowed between a source SID and a target SID. This allows for a potentially simple interface between the kernel and the security server, and a single format for the entire internal representation of the security policy within the security server. However, the security policy decisions required by higher level clients may not always be expressed this simply.

For instance, high level policy statements may define how an operation is to be performed instead of simply deciding if an operation is allowed to be performed. This may be the case when labeling output data or choosing cryptographic algorithms.

The DTOS security server provides one service of this kind, to the security enhanced Unix server developed by the government sponsor of the DTOS program. When a Unix process executes a file (by invoking the `exec` call), the Unix server needs to identify an appropriate SID for the task executing the file. Since the SID is policy dependent, a new interface was added to the security server to return the required SID for the task based upon the SID of the task making the `exec` request and the file to be executed.

If an application is developed which requires security policy decisions somewhat independent of the overall system security policy, it may even be desirable to provide a distinct security server for that application.

3.4 Kernel and Security Server Interface

As described so far, an interaction between kernel and security server occurs whenever the kernel reaches a control point, at which time it queries the security server for a security decision. While this is a very simple model, it leads to some problems because the security server is a distinct task:

- During bootstrap, the kernel is operational before the security server, so there is an initial state during which the kernel is unable to get security decisions from the security server.
- Deadlocks can occur because the kernel requires security decisions in order to send a message to the security server. Another source of deadlocks are communications with external pagers.
- The performance of the system is unacceptable if communication with the security server is required at every control point within the kernel.

The first two problems can be solved by preloading certain security decisions into the kernel during bootstrap processing. The third problem is much more significant, and is addressed in the prototype by introducing mechanisms for caching security decisions in the kernel when received from the security server.

So in the current system, when the kernel encounters a control point and requires a security decision, it first looks within its internal cache. If not found in the cache, it takes a *security fault* and requests the decision from the security server.

The additions to the kernel and security server to support the caching mechanism became quite complicated over the course of prototype development. The remainder of this section discusses why these complications became necessary and how the complications might have been avoided.

- Section 3.4.1 discusses the kernel cache and some of the optimizations provided by the cache.
- Section 3.4.2 discusses how the caching makes it much more difficult to support policy flexibility, and how additional mechanisms were added to restore some of the flexibility.
- Section 3.4.3 discusses how caching impacts the ability of the security server to audit security decisions, and how additional mechanisms were added to provide this function.
- Section 3.4.4 discusses whether in retrospect, the decision to separate the security server from the kernel was the correct decision. Some possible alternatives are presented.

3.4.1 Cache Optimizations

The simplest form of a cache would be to record results of particular security decisions, i.e., a collection of boolean values indexed by (SID, SID, access right) triples. This section describes three ways in which the cache and the interface for security server decisions were optimized to improve efficiency.

No testing was performed to attempt to quantify the value of any of these optimizations individually.

3.4.1.1 Access Vectors

The simplest and probably most powerful optimization is the concept of an *access vector*, which contains all security decisions between two SIDs. Each bit of the access vector corresponds to a particular access right and the value of the bit indicates whether the access right is granted. When a request is made to the security server, it responds by providing an entire access vector rather than a particular security decision. The kernel cache actually consists therefore of a collection of access vectors indexed by (SID, SID) pairs.

Since it is expected that there are usually multiple interactions between objects, the use of access vectors is expected to greatly increase the percentage of cache hits while also decreasing the size of the cache.

3.4.1.2 Elimination of Redundancies

While the SID is uninterpreted (other than the classifier) by the kernel, the SID does consist of two distinct fields, the mandatory identifier (MID) and authentication identifier (AID). It is expected that for some policies of interest (and in particular the policy supported by the prototype security server), the MID will encapsulate labeling attributes and the AID will encapsulate user attributes. Moreover, there is an expectation that for kernel level policies, user attributes are of less importance than labeling attributes and that many security decisions will be made independent of user attributes.

The conclusion is that many access vectors are actually independent of the AID values, and therefore there is potential for significant redundancy within the kernel cache. This is exploited by extending the security server interface to provide the kernel with an indication of whether a particular access vector is independent of the AID.

By expanding the logic for cache lookups to take advantage of this property, there is again potential for increasing the percentage of cache hits while decreasing the size of the cache.

3.4.1.3 Second Level Cache

Looking up a security decision within the kernel cache can potentially become rather time-consuming itself as the size of the cache increases. Since many of the kernel control points are encountered when accessing port rights held by a task, the structure which holds port rights was augmented with a pointer to a cache location that is expected to contain the access vector between the SID of the task holding the port right and the port itself. This location must still be verified, but that can be done with comparisons which are fast relative to the time potentially required to find an access vector with no hints.

3.4.2 Caching and Policy Flexibility

While caching of security decisions within the kernel is necessary for the prototype to meet its performance goals, caching is a significant impediment to meeting the goal of policy flexibility. This is due to the expectation that security decisions cached at some time will still be valid when the cache is referenced at a later time. Without further changes to the cache and the security server interface, this eliminates the prototype's ability to support security policies in which security decisions change over time.

For even the simplest security policies, this is usually unacceptable. For instance, even if an organization's high level security policy never changes, particular users come and go or their roles within the organization may change.

This section describes several features which were added to the kernel specifically to increase support for security policies which are allowed to change in some way.

3.4.2.1 Cache Flushing

The simplest way to allow the kernel to respond to changes in the security policy was to add a kernel interface to flush access vectors from the cache. The new request allows the security server to specify that all of the cache or a specified portion of the cache should be flushed. It also allows the security server to explicitly add to the cache.

Unfortunately, the intent of a cache flush can fail due to synchronization errors if messages sent by the security server are received out of order. A policy identifier was therefore added to all security server decisions in order to force synchronization.

3.4.2.2 Cache Control Vector

The ability to flush the kernel cache probably addresses the most common changes which may occur in a security policy. However, there is a delay between the time that the policy changes within the security server and the time that the kernel actually flushes the cache, and in some policies this time delay may be unacceptable.

What such policies require is for the original model of interaction between the kernel and security server to be restored: the kernel should ask the security server for a decision at the time that some client requires a particular access right. Because the performance implications of this are severe, it is important to allow the security server to specify exactly when this behavior is required.

The *cache control vector* was added to meet this requirement. The security server provides this vector to the kernel with every access vector. Each bit of a cache control vector indicates whether the corresponding bit of the access vector can be cached. For simplicity, the entire access vector is cached, but when the cache is consulted, the cache control vector is also consulted to verify that the cached security decision is usable. If not, the security server must be called to receive a current decision.

3.4.2.3 Timeout Values

Timeout values were added to the kernel cache as another means of ensuring that the cache is current. When providing an access vector to the kernel, the security server can specify an absolute time after which the access vector will be invalidated. If an access decision from that access vector is later required, the security server will be called for a current decision.

This feature was originally added when consideration was being given to a distributed systems design with security servers executing on only some nodes. In such a system, there is potential for a significant delay between the time that a security server makes a request for a kernel to flush its cache and the time that the kernel actually receives the request, for instance if the communications path between the nodes is broken. Timeout values provided a way to ensure that a kernel was required to periodically communicate with a security server.

Another potential benefit of timeout values are policies which are dependent upon the time of day. While it would be possible to implement such policies with a cache flush (assuming the policy can handle short delays), timeout values are somewhat simpler.

3.4.2.4 Adding Wired Cache Entries

The access vector cache is contained in wired kernel memory, and is therefore limited in size. An algorithm for removing lesser-used entries when the cache nears its capacity is used to make room for new entries. Since the cache is used to address the deadlock issues identified at the beginning of Section 3.4, it is necessary to ensure that certain entries are never removed. These entries are identified as wired and are generally provided only during bootstrap.

However, a change in security policy as simple as adding a new application may result in a need for additional wired entries to avoid deadlock situations. To address this, the interface which the kernel provides to flush vectors from the cache or specifically add vectors also allows the new vectors to be marked as wired.

3.4.2.5 Changing Security Servers

The ultimate change in security policy is for the security server itself to be replaced. Since the kernel only identifies the security server through the security server port (the port to which it sends requests for security decisions), security servers can be changed without any kernel action or knowledge. Alternatively, the kernel provides an interface through which the security server port itself can be changed.

If the change in security server is truly a change in policy, the new security server will likely immediately request the entire cache to be flushed. To avoid deadlock issues, the security server is expected to simultaneously add new entries to the cache and mark them as wired if necessary.

3.4.3 Audit of Security Decisions

In addition to specifying allowable access rights, security policies often specify that certain kinds of permission checks should be audited. In the basic model of a kernel asking a security server for every security decision, it is natural for the security server to be responsible for audit since it contains the policy about which decisions are auditable.

However, caching of security decisions within the kernel makes it impossible for the security server to perform the auditing. To remedy this, a *notification vector* is provided to the kernel with all security decisions. The bits of the notification vector indicate whether an audit record should be produced when the corresponding bit in the access vector is consulted for a security decision.

3.4.4 Reflections on the Interface

Sections 3.4.1, 3.4.2 and 3.4.3 describe complexities which were introduced to the system while trying to meet three apparently competing goals:

- Performance similar to unmodified Mach
- Policy flexibility
- Physical separation between policy enforcement and decision making (i.e., separate address spaces)

The first two of these goals are fundamental objectives of the prototype effort (see the introduction to Section 3). The third is a design choice which was made to support the policy flexibility objective as well as other less critical objectives. The purpose of this section is to reconsider, with the benefit of hindsight, the impact of relaxing this third goal.

This section does not reconsider the *logical* distinction between security policy decision making and security policy enforcement. There is overwhelming evidence that logical separation provides distinct benefits in the areas of assurance and policy flexibility. The purpose of this section is to consider the *physical* distinction between the kernel and security server, since physical separation is the source of the complexities introduced on DTOS.

The physical separation between policy enforcement and decision making was considered a fundamental part of the original design for several reasons:

- Microkernel systems encourage separation of logically distinct functions into distinct tasks.
- The standard practice at Secure Computing was to perform security decision making in a small, physically distinct module. In fact, for the LOCK system from which DTOS was partially derived, the equivalent of the security server actually executed on a separate processor.
- The security policy could be changed with no changes to the kernel, even while the system was operational, thus providing for maximal policy flexibility.

3.4.4.1 Design Alternatives

This section describes three general design alternatives which will be assessed in Section 3.4.4.2.

To define the alternatives, we define the following three functionally distinct components of the system:

Kernel Policy Enforcer The kernel enforces security decisions as described in Section 3.2.2 and 3.2.3. At every control point, it asks for a security decision, stated in terms of two SIDs and a required access right. This is the only information which the kernel requires to enforce the policy.

Security Decision Maker This component responds to requests for policy decisions from the kernel policy enforcer, providing yes or no decisions. It makes these decisions based upon a combination of hardcoded internal logic, internal transient security data, and persistent security data maintained in the security policy database. Included in the transient security data is the mapping between SIDs and security contexts. All other internal logic and transient security data is generally policy specific.

The security decision maker also must respond to requests for security policy decisions and possibly other policy information from user space clients.

Finally, in some history based policies, security decisions can result in changes to the persistent security data, so the security decision maker may need to request that changes be made to the database.

Security Policy Database This component maintains all persistent security policy data. Note that the distinction between the security decision maker and the security policy database is policy dependent, and therefore it is difficult to draw as sharp of a distinction as can be made between kernel policy enforcement and security decision making. It is also a less important distinction for these discussions.

This component interacts with the security decision maker to read or write the databases. Administrative clients may also read and write the databases, such as when adding a new user or application.

Figure 1 shows these components and the primary interactions among them and other client tasks. It also identifies five possible physical boundaries between components, and in particular, between kernel space and user space. We define three alternative designs based upon these physical boundaries:

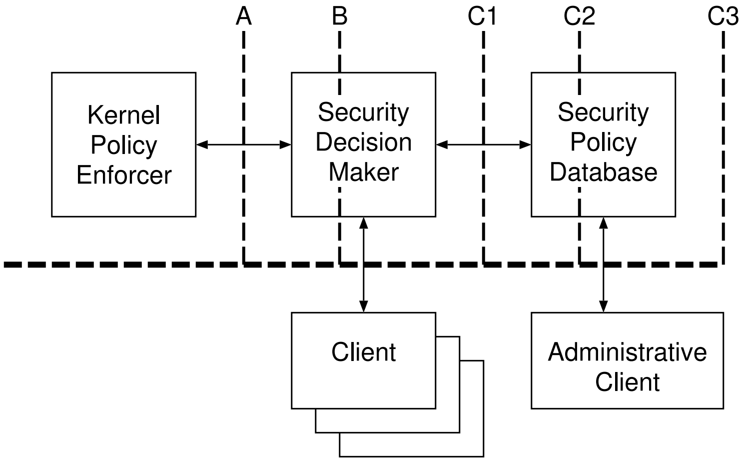

Figure 1: Kernel and Security Server Interfaces

Alternative A The security decision maker component and security policy database exist entirely in user space tasks. Every security decision needed by the kernel is found by calling a user space task.[5]

Alternative B Some of the security decision making functions are included as part of the kernel. These functions are designed to be policy independent, so that the same kernel executable can be used for any supported policy. All policy dependent decision making functions, and the security policy database, exist in user space tasks.

While there is obviously a range of specific implementations of alternative B, when discussing alternative B we consider the existing DTOS prototype implementation. The security decision making functions within the kernel are those which implement the access vector cache and the decision logic necessary to determine if a cache entry is usable and if so, whether it grants a particular permission.

The only aspect of the existing implementation which will occasionally be considered separately is the fact that this implementation does not provide the sharp distinction described above between kernel policy enforcement and security decision making. Primarily due to the second level cache, the internal interface across which most permission checks are requested actually provides two SIDs, the requested permission, and a hint as to where the necessary access vector may be found in the cache.

Alternative C Some policy dependent decision making functions execute within the kernel's address space. The kernel policy enforcer must be relinked with these policy dependent functions for each distinct security policy. Note that since the kernel only makes one request of the security decision maker, relinking is not complicated.

In this alternative the actual kernel boundary can vary with the policy. Figure 1 shows three possibilities marked C1, C2 and C3. The interface across this boundary can be tuned to the specific needs of a particular policy.

[5]This is a slight simplification, since to avoid deadlock situations, certain security decisions must be hardcoded into the kernel. But this is a relatively insignificant complication, and similar complications can arise in all three alternatives.

3.4.4.2 Tradeoffs

This section assesses each of the three alternatives against various criteria, identifying some of the tradeoffs made for each alternative.

3.4.4.2.1 Performance

One of the objectives of the DTOS prototype is to provide performance similar to that of the base Mach kernel.

Alternative A requires an outcall from the kernel to a user space task at every control point. Since there are often several control points for each single kernel request, this creates quite a bit of IPC and context switching overhead for each kernel request. Alternative A was rejected for the DTOS design based upon an expectation that this overhead would prove to have a significant impact on the overall performance of the system. The results of the performance testing in Section 4 indicate that under heavy IPC loads, the time required to perform some client/server interactions can increase by more than 100% if a security server outcall is required at each kernel control point.

Alternative B was developed to address this concern without requiring policy specific security server code to execute within kernel space. However, if a policy forbids caching of access decisions, then alternative B will provide performance comparable to alternative A, perhaps even worse because of the extra code complexity.

Alternative C provides the ability to tune the design to meet the specific needs of any particular security policy. In that respect, alternative C should always perform at least as well as alternative B.

There is however one potential advantage of alternative B over alternative C, which is the use of the second level cache to further increase the speed of cache lookups. Alternative C does not allow a second level cache because of the simple interface between the kernel and the security decision making component.[6]

3.4.4.2.2 Policy Flexibility

Another essential goal of the prototype is policy flexibility.

Alternative A seems to provides the maximum flexibility available, since anything outside of the kernel can be changed without the kernel's knowledge and without need for the kernel's knowledge. The only possible weakness of alternative A is the possibility that some dynamic security policies may require tight synchronization between the kernel and the security server, which may be more difficult to achieve with alternative A.

From one viewpoint, alternative C is as flexible as alternative A because it can always default to a case when the policy specific module simply directs a request to a user space security server. However, once significant policy specific code is executing within the kernel, there are certain limitations to alternative C, in particular, the elements of the security server executing within the kernel cannot be changed while the system is operational. Among practical security policies, we know of no example which demonstrates a need to change the basic policy logic while a system is operational.[7] Nonetheless, for a research platform such as DTOS this can still be considered to be a limitation on flexibility.

We've already seen in Section 3.4.2 that alternative B has the potential to negatively impact policy flexibility because of caching. The complicated caching protocol was provided in an attempt to provide flexibility comparable to that of alternative A. Because of this, alternative B appears to be capable of meeting the requirements of the practical security policies which are

[6] Though there is obviously another alternative C' which implements a second level cache and the more complex internal interface of alternative B.

[7] In fact, if a change in logic was known in advance, it could be incorporated directly into the security server so alternative C would still be available.

supportable within the kernel, though like alternative C other policies may be of interest in a research platform.

3.4.4.2.3 Code Complexity Alternative A clearly provides the simplest code because it provides a physical interface concurrent with the functional interface. Alternative C is similar.

The complexity of alternative B has already been discussed to some lengths in Sections 3.4.1 and 3.4.2, and this complexity is the reason for the current discussion.

3.4.4.2.4 Assurance The major factor affecting assurance is code complexity, hence the comments of the previous section are directly applicable. In particular, alternative B requires more assurance effort than the other alternatives.

It is often argued that assurance of a system is increased by dividing the system into physically distinct modules, and applying strict least privilege controls to each module. This argument would suggest that alternative B would be superior to alternative C, since alternative B separates policy dependent code into a distinct physical module. However, in this example, we do not believe that this is a sound argument.

The kernel and the security server are the most security critical components of the system, and must both be highly assured. Though executing both components within the same address space as in Alternative C could potentially cause an otherwise unexploitable flaw to become exploitable, this danger is not as significant as the potential problems caused by increased complexity of alternative B.

3.4.4.2.5 Ease of Debugging One of the advantages of a microkernel based system is that debugging can be performed on each module independently without worry about interference from other modules executing within the same address space.

Thus when considering the ease of debugging policy specific code in the security decision making and security policy database components, alternative A is the preferable alternative. Alternative B might be considered preferable over alternative C because it allows all policy specific code to be developed within a distinct address space. However, there are two mitigators:

- The additional complexity required by alternative B makes debugging more difficult.
- The system should have IDL tools available which make it possible to debug in a distinct address space and then execute within the kernel. Thus, debugging for alternative C could be done identically as alternative A, though for an operational system some of the security server code would be directly linked to the kernel.

3.4.4.2.6 Device Access The security policy database provides for storage of persistent policy information. Other than bootstrap info, the Mach kernel requires (and provides) no direct use of persistent data. Thus alternative C3 has the potentially serious drawback of requiring the kernel to be able to handle persistent storage. Unfortunately, for security policies which do not allow for caching between the security policy database and security decision maker, alternative C3 may be the only alternative which provides for sufficient performance.

3.4.4.2.7 User Space Client Access The security decision making component also has clients which operate in user space. Therefore we need to consider how the different alternatives affect the interface between these clients and the security server.

In alternative A, the physical interface between the client and the security server is the same as the interface between the kernel and security server. Each permission check requires a pair of IPC messages.

In alternative B, user space clients are expected to use the same interface used by the kernel. However if the client does not choose to cache access vectors, it can ignore most of the information passed on that interface. The only information it is required to pay attention to is the notification vector which indicates when the client must audit a permission check (see Section 3.4.3).

For clients which do cache, it should be possible to provide a standard cache module which can be linked to all clients, though no effort has been made to provide this for DTOS.

Alternative C provides, as usual, the most flexibility in the implementation. A simple implementation would be to add a kernel request which forwards a security decision request to the security decision maker through the same internal interface used by the kernel policy enforcer.

If performance requirements dictate that some kind of caching be performed in the user space clients, then policy specific code can be linked to all clients to provide this ability just as it is provided in the kernel. What is not clear is how this policy specific code would access the actual security decisions. In alternatives C1 or C2, the policy specific code may be able to call the user space elements of the security server. In alternative C3, a new policy specific kernel request would need to be added, which adds to the complexity and would require recompilation of the kernel for each policy, not just relinking (in order to incorporate a new request in the interface definition).

Therefore the impact of user space clients on the overall complexity is highly dependent upon the performance needs of those clients. If they require caching to maintain adequate performance, then alternative C not only requires added complexity to the user space clients (similar to alternative B), but it also requires additional complexity in the kernel.

No effort has been made on DTOS to determine the performance impact of policy enforcement in user space clients. Section 3.4.4.4 presents a suggestion to perform some user space policy enforcement within the kernel, which could be used to decrease the need for user space permission checks and therefore decrease the performance impact of performing those checks without caching.

3.4.4.2.8 Administrative Client Access The security policy database also has administrative clients. However, the needs of these clients are not performance critical, so a simple physical interface coinciding with the functional interface should be sufficient. Note that this interface is policy specific in all three alternatives.

3.4.4.3 Summary By all criteria other than performance, alternative A is clearly preferable. Unfortunately, the expectation on DTOS was that performance of alternative A would not be acceptable, and test evidence supports that expectation. However, DTOS did not further investigate the precise cause of the performance degradation, which is relevant when considering the remaining alternatives.

Section 3.4.4.2 identifies the following main points which should be considered when comparing alternatives B and C:

- The performance benefit gained by caching in alternative B assumes that the security policy allows caching through the kernel/security server interface defined for the DTOS prototype.

- If the security policy does not allow for caching between the security policy database and security decision maker, then the only acceptable performing alternative may be C3 which unfortunately requires direct device access while in the kernel.
- For policies which allow caching between the policy enforcer and security decision maker, alternative B may provide a faster cache lookup than alternative C because of the second level cache.
- In general, alternative B is more complex than alternative C, and is therefore more difficult to assure.
- The complexity benefits of alternative C can diminish if caching is required in user space policy enforcing clients.
- Alternative C provides the greatest flexibility across all of the criteria.

Therefore, the "correct" choice among the three alternatives depends upon the specific requirements of a system and specific data about the cause of any identified performance degradation.

3.4.4.4 Kernel Permission Checking Service

Editorial Note:
This section is somewhat out of place. However, it is referenced in the discussion above, so it is placed here as an "appendix" to the discussion.

The kernel currently performs permission checking only for the services which it provides. However, it would be possible for the kernel to perform certain kinds of permission checks for services provided by other servers.

Every IPC message in Mach contains an "operation id" field in the message header. This field can be used in client/server operations to indicate which particular request is being made. Since the kernel has access to the SIDs of the client and the port to which a request is being made, the kernel could provide permission checking between the client and the port based upon the operation id.

One way in which this could be done would be for the server, when it creates a request port, to request that the kernel perform additional permission checking on messages sent to that port. As part of this request, the server would provide a mapping between operation ids and bits in the access vector between clients and the server request port. The kernel already uses the access vector for permission checking itself, so this requires little additional effort by the kernel.

There are two potential advantages of providing this mechanism in the kernel. First, the server is simplified. It need not perform that kind of permission checking (and it already would need to have a mapping from operation ids to permissions). This may be enough to relieve the server of any need to cache permissions received from the security server. Second, the permission checking is slightly easier to assure (though the server must still be assured to provide the correct mapping).

This suggestion only is applicable if the server performs permission checking based upon the SID of the port through which a request is received, a condition which is unacceptable in some applications.

3.5 Prototype Distribution and Support

As part of the Synergy program, the DTOS prototype was used by government researchers and other researchers participating in Synergy. Secure Computing was responsible for distributing the DTOS prototype and the operating system components developed by the government.

The DTOS prototype was generally made available to anyone who asked, upon government approval. While we have no count of how many sites inquired about the prototype, twelve sites (not counting the Synergy program offices and Secure Computing) expressed sufficient interest that they were given access to the prototype. This access was granted by setting up a user account for each site on an internet access computer from which the prototype code and documentation could be downloaded.

At least three of these sites had significant active development efforts, including an RBAC project at NIST[8], a secure X Window System at Portland State University [14, 23], and a secure database management system at Penn State University [57, 56]. DTOS was also used as a platform for class projects at Penn State University in a graduate level computer security class.

After the initial release, DTOS was released approximately quarterly with a full set of updated documents. When the need arose, an interim release was distributed with no document updates.

To support users of the prototype, Secure Computing maintained a mailing list, `dtos-discuss`. This mailing list was used by DTOS users to report difficulties or ask questions and was used to announce new releases or patches. As of April 30, 1997, there were 26 subscribers to this list outside of Secure Computing and the government's Synergy team. Approximately 250 messages had been sent to the mailing list in its first 1 1/2 years.

DTOS support is currently scheduled to continue through September of 1997 under the DTOS Prototype Support contract, MDA904-97-C-0362. Users of DTOS are being encouraged to transition to the Fluke kernel and operating system.

3.6 Demonstration Application

To demonstrate the use of the DTOS control philosophy and the prototype security server at the application level, a client/server demonstration application was created. The application provides a simple database for the maintenance of patient records in a hospital.

The database server consists of two tasks, a trusted front end which performs all permission checking and a more complex task which actually manages the database. The database itself contains several fields for each user, such as personal identify information, billing records and medical records. Clients of the database can request to create a new patient record or read, modify or append individual fields of an existing record.

The security policy for this application grants different permissions to users based upon their role. The supported roles are administration, accounting, insurance, nurse, and doctor. The fields in the patient records are grouped into four collections based upon the security policy rules. The four collections are administrative, billing, vital signs and diagnosis.

The security policy defines the accesses allowed between each role and each collection of fields. For instance, vital sign and diagnosis fields can only be accessed by doctor and nurse roles, and only a doctor can modify or append diagnosis fields. Similarly, billings fields can only be

[8] See `http://hissa.ncsl.nist.gov/rbac/` for a description of RBAC projects at NIST, including this one.

accessed by administrative, accounting and insurance roles, but only accounting can modify or append those fields.

This policy is implemented by defining a distinct domain for each role, and distinct permissions for each access (read, modify, append) to each collection of fields (a total of 12 permissions). There is also a separate permission for adding or deleting an entire patient record. The identification of which permissions are granted to each domain is stored within the security databases accessed by security server.

The trusted front end to the database enforces the decisions of the security server. The demonstration shows that no other accesses are possible through the trusted front end.

Moreover, the trusted front end and the untrusted database server execute in different domains. The security databases are defined to allow only the trusted front end to communicate with the untrusted server. This part of the policy is enforced by the DTOS kernel. The demonstration therefore also shows that it is impossible for a database client to access the untrusted server directly, so the intended security policy is completely enforced.

Section 4
Performance Testing

One of the goals of the DTOS prototype was to add security features in a manner which minimized the performance impact of those features. The architecture of the security features was profoundly impacted by this goal. In addition, performance testing was performed on initial releases of the system in an attempt to quantify the actual performance impact of the DTOS kernel modifications.

This section provides an overview of the results of the performance testing and discusses some of the difficult problems that were encountered during the testing.

4.1 Testing Methodology and Results

System performance was measured in two ways: with a Mach performance test suite developed at the Worcester Polytechnic Institute (WPI) Computer Science Department [31, 32], and a simple kernel compilation test. The latter measures system time to compile the IPC portion of the Mach kernel. All tests were executed on a PC-clone with a 486DX2-66MHZ processor, 8 MB of memory and a 1 GB SCSI disk.

The tests were run on the baseline Mach kernel and each of the incremental versions of the system during the development. Table 1 provides a summary of the results as run on the baseline Mach kernel and the first version of the completed prototype system. The test runs on the prototype system included a best case situation, (100% cache hits), and a worst case situation, (0% cache hits). Table 1 provides a summary of the test results.

In addition to gathering performance test data, the system counted the number of permission checks and kernel-Security Server interactions made during a test. Each row, under Data Description, labeled *Permission Checks* in Table 1 indicates the number of permission checks made during the test. Each row labeled *Security Server Requests* indicates the number of times the kernel sent a permission check request to the Security Server. It should be noted that the difference between permission counts and Security Server interactions in the worst case reflects the fact that permission checks on Security Server operations are not allowed to result in a security fault. The Security Server is treated the same as all other tasks and thus all of its operations are subject to policy-defined permission checks. However, to avoid a deadlock, the cache is provided with wired access vectors that describe the allowed Security Server operations.

The three tests referenced in Table 1 are:

WPI Sdbase: This test uses TCP/IP sockets to communicate between a single server and multiple clients. The test was designed to evaluate the performance of IPC subsystem. The test was run using both 5 clients and 25 clients.

WPI Jigsaw: This test solves a mathematical model of a jigsaw puzzle. The test was designed to evaluate the performance of the memory management (paging) subsystem. The test was run with puzzle sizes ranging from 8x8 to 64x64.

IPC Compilation: This test measures time to compile the IPC portion of the baseline Mach kernel.

Table 1: Performance Results

Test	Data Description	Baseline	Modified Kernel Best Case	Modified Kernel Worst Case
WPI Sdbase				
5 Clients	Avg. Client Total Time(ms)	39344	40084	202278
	Avg. Client Communication Time(ms)	16308	16670	23178
	Avg. Server Time(ms)	27564	28628	187500
	Permission Checks	NA	110799	415086
	Security Server Requests	NA	0	108692
WPI Sdbase				
25 Clients	Avg. Client Total Time(ms)	205168	235434	1231272
	Avg. Client Communication Time(ms)	20904	23469	81598
	Avg. Server Time(ms)	180422	209395	1116058
	Permission Checks	NA	692010	2647666
	Security Server Requests	NA	0	682160
WPI Jigsaw		Average Values Over 10 Runs		
8 x 8	Time(ms)	19	21	24
12 x 12	Time(ms)	83	74	78
24 x 24	Time(ms)	185	181	186
32 x 32	Time(ms)	1941	1996	1998
40 x 40	Time(ms)	4320	4382	4381
48 x 48	Time(ms)	8130	8190	8233
55 x 55	Time(ms)	13061	13130	13233
64 x 64	Time(ms)	22100	22459	22433
	Permission Checks	NA	32630	66869
	Security Server Requests	NA	0	16337
IPC Compile		Average Values Over 10 Runs		
	Real Time(sec)	987	1031	1787
	User+Sys(Sec)	749	767	842
	Percent Utilization	76	74	47
	Permission Checks	NA	436046	1457665
	Security Server Requests	NA	0	469294

Due to the variation in test results from run to run, it is probably dangerous, at best, to try to draw narrow numeric conclusions from these test results. There are time changes that can only be explained as resulting from a change in the page alignment of kernel code. It is also believed that test results are influenced by the state of fragmentation on the disk.

A reasonable assessment of the test results is that the best-case performance of the prototype system tends to be slightly slower than the baseline. Some best-case tests are faster while others are slower. The worst case tests show significant differences, but the range of difference depends on the nature of the tests.

In one respect, the test results were largely as anticipated. In all tests for which DTOS showed a significant performance penalty, the severity of the impact in the “Worst Case” is much more serious than in the “Best Case”. This implies that as expected, the ability to cache security decisions in the kernel can provide a significant benefit in comparison to context switching to the security server whenever a decision is required.

The difference between the “Baseline” and the “Best Case” shows a difference of less than 5% for all tests except the Sdbase test with 25 clients. This is also very encouraging.

However, it is very difficult to draw meaningful conclusions about the performance of DTOS under realistic loads. The two WPI tests are extreme cases designed to test particular aspects of the system rather than realistic loads. The IPC compile test is also a very specialized test, though one which may be more in line with typical usage.

Due to these difficulties and others mentioned below, performance testing was not pursued on further releases of the prototype.

4.2 Problems and Lessons Learned

4.2.1 Testing Individual Operations

In addition to the testing described in Section 4.1, “micro” testing was performed in an attempt to measure the impact of DTOS changes to particular requests, in particular those requests with a `_secure` version. The tests measured the time required to perform each operation several thousand times in immediate succession.

Unfortunately, these tests did not even show consistent results between runs of the tests. Each test was generally run 10 times, and in some cases the slowest execution time was more than twice as long as the fastest. This made it impossible to gain any insight into actual impact of the DTOS changes.

The cause of these inconsistencies was unclear, though a paper by Brian Bershad, et al [7] suggests several reasons why it may be difficult to achieve consistent and meaningful microtest results. However, Bershad and David Black have both reported success in obtaining consistent results in microtesting of Mach, and OSF has even released a suite of microtests for its Mach kernel [24].

4.2.2 Baseline Reestablished for Every Release

The DTOS prototype was developed as a series of releases marking incremental change to the CMU Mach microkernel. Performance testing was performed on each release with the goal of measuring the effect of each relatively small set of kernel modifications. Unfortunately, the difficulties of obtaining consistent data were underestimated, with the result that the relative performance impact of each set of changes is unknown.

As these difficulties became clearer, test procedures were changed. For instance, initially kernel images from each release were compared directly. However, since some of the build processes and tools were updated between releases, this meant that testing would also identify performance impacts of these updates. In addition, steps were taken to ensure consistency of the environment between runs of the tests on each kernel release, such as running the tests in succession on a single machine.

4.2.3 Time Allotted to Performance Testing

Enough time must be allocated for performance testing during the development of any system if the system developers want to have adequate breadth of testing and not overlook performance testing on important parts of the system. Without adequate performance testing as a system is being built, a potential cost is that the system as an end product may be unusable due to its poor performance. This problem may have been corrected earlier during system development if enough performance testing had been done during system development. In addition, if the amount of performance testing is increased late during the development of a system, this testing must be repeated on the baseline and earlier versions of the system if one is to get adequate performance test numbers. Unanticipated re-testing of previous versions of a system may cause a slip in cost and schedule.

4.2.4 Inadequate System Resources

System resources may not be adequate for the number of iterations required in performance tests to overcome time granularity. This may result in inaccurate performance test numbers. Without proper performance test numbers, it is not possible to accurately assess the performance impact of security additions to the Mach system. Careful structuring of tests including the use of cleanup commands as part of each performance test allowed a larger number of iterations to be run.

4.2.5 Testing of Relevant Usage Patterns

If performance testing does not test relevant usage patterns of "real" applications, this may result in inaccurate conclusions being drawn about how the system will perform when run with real applications. Therefore, it is important to select tests that simulate real applications, like the WPI Mach performance test suite.

Section 5
Assurance Overview

Section 2.1.2 summarizes the overall approach to assurance for the DTOS program. This section introduces each of the assurance tasks performed on the program and discusses assurance aspects of the program which are not specific to any one assurance task. In particular, Section 5.2 discusses assurability of the prototype and Sections 5.3 and 5.4 discuss the use of formal methods and tools on the program.

5.1 DTOS Assurance Tasks

To describe each of the DTOS assurance tasks and the relationships between them, it is helpful to first describe the elements of a typical assurance analysis for a computer system. Figure 2 describes a possible layering of system descriptions within an assurance analysis:

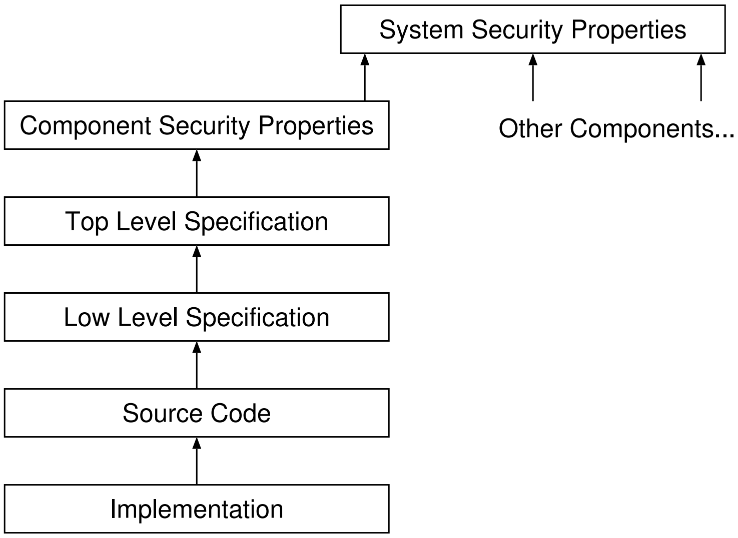

Figure 2: Layers in an Assurance Analysis

System Security Properties A collection of security properties satisfied by a system made up of several components.[9]

Component Security Properties A collection of security properties satisfied by a particular component.

Top Level Specification The highest level specification of a component.

Low Level Specification A lower level specification of a component.

[9]There is no reason that assurance must be limited to security properties, however the description here only considers security properties because little else was considered on DTOS assurance tasks.

Source Code The source code for a component.

Implementation The implementation of a component, i.e., the object code executing on some hardware. Because of inter-dependencies it may be impossible to completely separate the implementation of one component from all others.

The ultimate goal of an assurance analysis is to provide confidence that the system implementation satisfies all of the desired system security properties. A complete assurance analysis defines the system at each layer and provides justification that the system described at each layer satisfies the description at the next highest layer. The purpose of adding interim layers to the analysis is to lower the conceptual distance between adjoining layers, thereby increasing the reliability of the justification. The number of layers varies depending upon the complexity of the system and the amount of confidence desired.

The DTOS assurance tasks are not intended to provide a complete assurance analysis of the DTOS prototype. Such an effort would have been quite expensive, and moreover, techniques for performing some elements of a complete analysis are incomplete or immature. Addressing some of these shortcomings was a major emphasis of the DTOS assurance tasks.

The following describes each of the DTOS assurance tasks in terms of the layering defined in Figure 2 and the extent to which each task attempted to develop new techniques and theories. The tasks are described in depth in Sections 6 through 11:

Formal Security Policy Model (FSPM) The FSPM defines security properties of the microkernel component of a DTOS system. The properties described in the FSPM are in the form of control requirements which define all of the security controls implemented by the microkernel. The FSPM follows the strict separation of policy decision making and enforcement; it defines enforcement requirements on the microkernel independent of any particular security policy or security server.

Formal Top Level Specification (FTLS) The FTLS provides a top level specification of a portion of the DTOS kernel. While primarily an application of known concepts, the FTLS effort also pushes the specification to include some system elements which are often ignored. For instance, the DTOS microkernel is dependent upon two user space servers, the security server and the default pager. The FTLS includes an execution model that makes it possible to more accurately model these dependencies, as well as more accurately modeling the actual atomicity of operations within the microkernel.

Specification to Code Correspondence The specification to code correspondence task considers techniques for verifying that the microkernel specification within the FTLS correctly models the source code. Because of the large gap between the source code and the FTLS, an interim specification layer was added which corresponds to the low level specification of Figure 2. The verification is then performed between each pair of adjacent layers.

This task only considers a very small portion of the system because the focus is on developing techniques for performing the verification rather than generating verification evidence for the entire FTLS.

Composability Study The composability study is concerned with verification between the top most layers of Figure 2. It defines a formal mathematical framework for combining components into a system and relating properties of the system to properties of the constituent components. This approach allows for lower level analysis of each component to be performed independent of the other components.

The composability study is much more of a research task than the previous three. The only specific references to the DTOS prototype in the task are the use of simple specifications of the prototype in examples demonstrating the use of the framework.

Generalized Security Policy Specification (GSPS) The GSPS task surveyed a variety of security policies from the computer security literature, identified a list of policy characteristics and developed a classification scheme for policies with respect to these characteristics. These aspects of the task are independent of any particular assurance argument.

The other main element of the GSPS is a characterization of the range of security policies supportable by the DTOS architecture, particularly the relationship between policy enforcement and decision making. Referring to Figure 2, this aspect of the task considered the range of system security properties supportable based upon the security properties of the microkernel and security server components. The framework developed on the composability study was informally used to aid in this analysis.

Covert Channel Analysis Among the security policies which are intended to be supported by the Synergy architecture are information flow policies. The covert channel analysis task defines a very high-level approach for developing a complete assurance analysis of a system's ability to meet covert these policies. In terms of a general assurance argument, this high-level approach involves essentially all layers described above, with the security property defined in terms of allowed information flows between subjects.

However, the main efforts under the covert channel analysis task consider the use of noninterference to analyze a system at the design level. This is a verification between the top level specification and component security property layers of Figure 2. This aspect of the task is research oriented and, like the composability study, relates to DTOS only through some simple examples.

Within a complete assurance analysis of a single component, there are two verification layers which were not considered in any of the DTOS assurance tasks:

- There is no effort to provide any verification between the microkernel specification in the FTLS and the security requirements of the FSPM. This is ignored primarily because it is not considered to be a significant research issue, and can be performed by implementing known techniques.
- No verification is performed below the level of source code. In particular, there is no testing of the actual implementation. While testing of functional properties is a relatively well understood topic, testing of security properties is not at all well understood. The reason for not considering this subject under the DTOS program was simply a lack of resources.

5.2 Assurability of the DTOS Prototype and Impact on Assurance Tasks

One of the results [64] of the DTMach program was a determination that it would be possible to develop a highly-assured secure microkernel implementing the Mach interface. Unfortunately, a conclusion of the DTOS program is that it is not possible to accomplish this for an implementation, like the DTOS prototype, derived from the CMU Mach microkernel. This is due to complexity, lack of structure and lack of documentation for this code base. In retrospect this is not surprising, since the goal of the CMU project was primarily experimental.

Some of the problems with the software were encountered during prototype development and are discussed in Section 3.2.6. Several other problems of similar magnitude were encountered during the FTLS task. Whether all of these smaller problems would have been enough to lead to the conclusion is unclear, however, two overriding sources of complexity made the conclusion inevitable:

- Complexity of the IPC code. The biggest problem with the IPC code is the multitude of parallel paths through the code, each tuned for maximizing performance in a particular case based upon what types of data are being passed in a message. Since there is a need for some permission checking to be relatively deep within the IPC code, it is extremely difficult to assure that the permission checking is even being performed correctly. Assurance of all IPC functions would be essentially impossible.
- Complexity of the virtual memory system. Again, the problem is largely due to performance optimizations such as the technique of deferring virtual memory processing as long as possible. The lack of documentation makes it difficult to understand what is even being attempted in some cases.

It is important to recognize that this conclusion may not apply to all implementations of Mach, even though at the design level these are the two most complex portions of the Mach system. For instance, a more recent implementation of Mach, the MK++ implementation from OSF [54], was developed specifically to meet the needs of assurance concurrent with performance goals. While we have looked at this implementation only at the level of the documentation, it is very possible that MK++ could serve as a base for a highly-assured system, though the assurance effort required would still be considerable.

The realization that the DTOS prototype microkernel would not be able to transition to a product quality microkernel led to a change in focus on some of the assurance tasks. In the initial stages of the program, one of the secondary goals of the assurance tasks was to develop assurance evidence for the prototype which could be built upon during a follow-on effort to transition the prototype into a product. As it became clear that the underlying code base for the DTOS prototype would not support strong assurance, this goal faded in importance, and assurance evidence for the prototype began to be generated primarily as examples of techniques being developed or studied on the various assurance tasks. Nonetheless, some significant assurance efforts had already been undertaken prior to this change in objectives, most notably within the FSPM and FTLS.

5.3 The Use Of Formal Methods

As mentioned in Section 2.1.2, the use of formal mathematical methods was a fundamental element of the DTOS assurance tasks. The value of formal methods is the clarity, accuracy and completeness which they provide when describing and reasoning about a system.

Formal methods are generally applied in the areas of system specification, design, and verification. The remainder of this section discusses the DTOS use of formal methods in each of these areas, including the value of using formal methods and any difficulties encountered.

5.3.1 Specification

A specification records the design of a system at some level of abstraction. The use of a formal language provides for concise, unambiguous statements in the specification, and makes it possible to perform formal reasoning about the specification. It is also easier to ensure the completeness of a specification when it has been formalized, for instance when describing the side effects of an operation performed by the system or how the system reacts to “unexpected” inputs and failure situations.

The use of a specification beyond the initial design effort implies that the specification must be understandable to those outside of the specification team. Unfortunately, while a formal specification may be extremely precise and clear to someone who is an expert in a particular specification language, it may be of no value to anyone else. All too often the potential for lasting value is squandered when a formal specification consists of tens (or even hundreds) of pages containing nothing but formal statements in some specification language, with little or no accompanying text. Such specifications are of little or no use to anyone other than the authors, and it is likely that even an author is unable to make use of the specification after some time has passed.

Addressing this shortcoming was a high priority for the DTOS program. The key principle followed on DTOS is that a formal language specification is a supplement to an informal English language specification. An individual uninterested in the formalisms or unable to take the time necessary to learn a formal specification language should find as much as possible of the information that the individual needs simply by reading the English text. The formal specification is used to provide precise statements when the English text leaves ambiguities. The primary purpose of the English text is to explain the system, not the specification.

5.3.2 Design

Though formal languages are most obviously useful for recording design decisions which have been made, they can also be of use during the actual making of those decisions. Writing a specification for a proposed design, even an incomplete one, can have many of the same results as prototyping or simulation, such as identifying hidden assumptions and unexpected interrelationships. The simple act of writing a specification can force a systematic consideration of the design which is difficult to accomplish otherwise. If tools are available to execute the specification, then the specification can even substitute as a prototype.

This aspect of the use of formal methods was not explored to any significant extent under the DTOS program, except somewhat during the development of the requirements in the FSPM. The main reason for this was that most of the critical design decisions were made either during the design of Mach or during the design work performed during the DTMach program. Nonetheless, it is likely that some problems identified with the DTOS prototype would have been identified earlier had formal methods been applied during the design. The most notable of these are the unintended interactions among features added to the permission checking model.[10]

5.3.3 Verification

Formal methods can be used to verify properties of a system based upon the formal specification of the system. The value of formal verification is that it provides confidence in the completeness of the analysis. This is especially important for security properties, since a single failure can be catastrophic.

The biggest advantage of formal verification is also one of its biggest drawbacks. Completeness requires attention to details that are often tedious to address and contribute little to an understanding of a system or its ability to meet the desired properties.

[10] These interactions are discussed at length in entry 38 in the DTOS Notebook [77].

Formal verification was used on the DTOS program primarily within the Composability Study. Formal methods were used to verify the correctness of the theorems within the composability framework and were used to demonstrate how the framework applies to components within an extended example.

5.4 Formal Specification Languages and Tools

Though formal methods techniques can be implemented "by hand", the full power of formal methods is not achieved unless tools are used to support the effort. There is a range of tools available, supporting the use of formal methods in the areas of output formatting, syntax and type checking, theorem proving, proof checking, and prototyping. Tools increase the value of formal methods in many ways, such as:

- Tools can minimize the human errors that naturally occur even in a systematic use of formal methods.
- Tools can increase efficiency by performing some of the more mundane aspects of formal methods.
- Tools can increase the repeatability of a verification effort. If a proof checker is trusted, a person evaluating a system can easily verify that proofs are complete.
- Tools can increase maintainability. For instance, a single change to a specification can potentially ripple though other specifications and proofs. Tools can be used to identify precisely where the change is significant.

There are also some disadvantages to using tools:

- One advantage of formal methods is that it forces the analyst to think systematically about a system, which can identify subtle flaws. If automatic reasoning tools are used to do the "thinking" for the analyst, then this advantage is lost. There is an important (though sometimes gray) distinction to be made here between automated proof checking which verifies a proposed proof and automated reasoning (such as the "grind" command in PVS) which constructs proofs. In the latter case, proofs can be completed without gaining any understanding of why the proof succeeded or what properties about the system are necessary for the proof to succeed.
- Tools can have bugs which affect the validity of the analysis. A particular concern is theorem provers which can independently attempt many different approaches to a proof. A bug in such software which allows incorrect theorems to be proved may go unnoticed because users of a theorem prover generally only try to prove theorems which they expect to be true.
- Tools can place unnatural restrictions on the way in which a formal methods effort is carried out, especially in the way that a specification is written. Because tools often require a particular form of a specification or work best with a particular form of specification, there is a strong temptation to write a specification as if the primary audience were the tool, potentially lowering its value as system documentation.

This section discusses the formal specification languages and tools used on the DTOS assurance tasks, along with some observed advantages and shortcomings of each. There were two "families" of tools used, and each is discussed in a separate section.

5.4.1 The Z Specification Language and Tools

The Z specification language [90] was the primary specification language used for DTOS assurance tasks. It was also used exclusively on the DTMach program which preceded DTOS.

The most distinctive quality of the Z specification language has perhaps less to do with the language itself than with the way in which the language is often used. A typical Z specification consists of small elements of Z interspersed within a text document (see [38], for several examples). The Z language supports this use by providing for very concise statements of complex properties and a rich collection of operations for combining elements within a specification.

These aspects of Z made it an ideal choice to meet the goals of the DTOS specifications as stated in Section 5.3.1.

However, Z has a serious shortcoming: it was not designed with a complete formal definition, so there can actually be ambiguities within a formal statement. Z is actually too much like English in this respect.[11]

A related shortcoming of the language is that while complicated concepts may be described in a concise expression, it may be quite difficult to determine the precise meaning of the expression. This caused considerable difficulty in DTOS (see Section 8.4.1 for an example).

The tools that were readily available for use with Z at the onset of the DTOS program reflected the common usage of the language. Several tools existed for formatting specifications, but few for performing any kind of analysis on the specifications.

The DTOS program used the *f*UZZ [89] tools for formatting and typechecking Z specifications. For formatting, *f*UZZ does an excellent job of making it easy to incorporate Z specifications into a LaTeX document. The typechecking tool also worked very well for us, however, as a typechecker it is limited in its ability to identify flaws within a specification. Within these limitations, the only real shortcoming of *f*UZZ is that it requires all values to be defined prior to their use within a specification. When an English description might most naturally defer a specific definition of a term until after it has been used, a *f*UZZ specification cannot easily do this. This is an example of where the use of a particular tool puts potentially unnatural requirements on a specification.

The use of more sophisticated tools could have provided two valuable features:

- They could have discovered more significant flaws than the syntax/typing flaws identifiable by *f*UZZ. For example, it is easy to write a Z specification in which a partial function is applied to a variable which is not in the domain of the function. *f*UZZ would be able to identify if the variable were of an incorrect type, but not if it were outside of a specified domain for the function.
- They could have provided the ability to prove properties of the specification. This is valuable not only to assure that the specification satisfies the desired properties, but also as a way to discover more subtle errors in the specification.

Finding errors in a specification is somewhat analogous to finding errors within a program. A compiler can discover syntax and type errors in a program, roughly analogous to the errors detectable by *f*UZZ. Logical errors are most likely to be found when actually executing a program or when attempting to prove properties of a specification. Syntax and type checking of a specification are most likely to find errors within the specification, not in the actual system,

[11] There is a working group tasked with standardizing a formal definition of Z, see `http://www.comlab.ox.ac.uk/oucl/groups/zstandards/`.

while errors found during a proof are more likely to be logical errors that exist both within the specification and the actual system.

Two automated theorem provers for the Z specification language were considered for use during the program, though neither was adopted:

- Balzac[12]. This was investigated midway through the program, after a considerable amount of effort had been invested in the writing of the FSPM and FTLS. Balzac did not use the same format for input as required by *f*UZZ, so significant effort would have been required to convert existing specifications for use with Balzac. Moreover, Balzac did not appear to have a very sophisticated theorem prover.
- Z/Eves [47]. This was investigated near the end of the program. It was significantly better than Balzac in both compatibility with existing specifications and the sophistication of the theorem prover. However, it still would not work with the existing DTOS specifications because it did not allow certain formatting that is allowed by *f*UZZ. Because it was investigated so late in the program, there was no real need for the theorem proving capabilities, certainly not enough to justify the effort required to convert the specifications. In particular, the composability study task, which required the use of a theorem prover, had already made considerable progress using PVS.

5.4.2 Prototype Verification System (PVS)

The DTOS composability study required the use of a theorem prover. Since no sufficient theorem prover had been found for Z (Z/Eves was not even available yet), and PVS [55, 21] was already familiar to Secure Computing through its use on other programs, it was chosen for use in the composability study.

PVS was designed from the beginning to support efficient mechanized proofs as the primary goal. The specification language itself was chosen to be "expressive, yet mechanically tractable" [59]. The language does not lend itself as easily as Z to integration of text and formal specification, though tools are available which make it relatively easy to include PVS theories[13] within a LaTeX document.

But the available functions of PVS go well beyond what is available by using Z with *f*UZZ. PVS provides an integrated set of tools for writing and reasoning about a specification, including sophisticated type checking and theorem proving capabilities. It encourages an interactive approach to specification writing, by making it easy to immediately determine the ramifications of small changes in a specification on any previous analysis, all while working within the same application.

A particularly useful feature of PVS is the ability to define custom "proof strategies". A proof strategy is a way of combining primitive proof rules into a single command. If several proofs in a verification effort follow the same pattern, a proof strategy can be used so that less human interaction is necessary to complete each of the proofs.

One significant concern with formal methods tools for a complex system like an operating system is the ability of the tools to efficiently work with a very large specification. Our experiences with PVS on DTOS, and Secure Computing's experience with PVS on other programs, indicate that it scales well to large specifications (at least as long as computing resources are not overly stressed, e.g., no page thrashing).

[12] Balzac was developed by Imperial Software Technology in the United Kingdom under government sponsorship. They also offer a commercial tool, Zola, which was not investigated on the program.

[13] A theory in PVS is the basic unit of a specification.

Experiences using PVS have also pointed out some negative aspects of the tools. In general, the impacts of these concerns are much less serious than the positive aspects of the system.

- PVS is relatively young and is still buggy. Some of the bugs have been serious, such as logic errors which make it possible to prove a false statement or failures to generate type check conditions. A mitigator to this is that the PVS development team has been very responsive to bug reports.
- PVS suffers from the property that the way in which a specification is written, among logically equivalent choices, can have a significant impact on the efficiency of type checking. This limits the practical flexibility of the specification language, and unfortunately there is no documentation to identify these cases.
- In general, the system is not well documented. Some features and subtleties are discovered by trial and error.
- No complete formal semantics have been published for the PVS language. Mitigating this is the fact that the theorem prover can be used to investigate the apparent semantics when a question arises.
- New versions of PVS have not always been compatible with old versions. For instance, a sequence of proof rules which are sufficient to prove a theorem in one version may not be sufficient in the next. Therefore upgrading to a new version may require significant effort if existing proofs are to be maintained.

Section 6

Formal Security Policy Model

The security policy for a computer system describes the collection of security objectives which the system is expected to meet and the list of requirements to which the system and users of the system must adhere in order for the objectives to be satisfied. The DTOS Formal Security Policy Model (FSPM) [72] describes security requirements on the DTOS kernel which are intended to be general so that the requirements can support a wide range of security objectives on a complete system using the DTOS kernel as a base.

The DTOS FSPM consists of three major parts:

- A formal model of the state maintained by the DTOS kernel, describing the basic entities within the kernel and their relationships. This model is contained in Sections 4 and 5 of the FSPM.

  The state model defines the elements of the state of the kernel, their attributes, and the basic relationships between these elements. It is further divided into a model of the attributes of Mach kernel entities (such as threads, ports and messages) and the DTOS additions to Mach (such as security identifiers (SIDs), permissions and access vectors). A particularly important part of the model of the DTOS additions to Mach is the identification of the *controlled kernel entities*, i.e., the objects managed by the kernel which are controlled by the security policy. Each controlled kernel entity has a SID as one of its attributes.

  The purpose of the state model is to describe enough of the kernel to allow for formal statements of the security requirements. However, the DTOS FSPM contains more detail in the state model due to the use of a single state model shared by the FSPM and FTLS. This is discussed further in Section 7.3.2.

  Note that the model defines only the state of the kernel, and not the transitions which occur in the kernel. The possible transitions are modeled in the DTOS FTLS [74] (also see Section 7).

- The definition of the services controlled by the DTOS kernel. This is contained in Section 6 of the FSPM.

  There are two distinct types of services defined in the FSPM:

  - *Transformation Services* are those services which can be formally defined in terms of transformations of the kernel state model. Examples include creation and destruction of kernel entities and modification of attributes of existing entities. Transformation services are not directly associated with any particular kernel request.
  - Not all services that the kernel provides can be easily defined in terms of transformations of the kernel state. *Invocation Services* are defined to represent kernel requests providing such services. Generally, invocation services represent kernel requests which observe information about the kernel but do not necessarily alter any of the high level kernel data structures visible in the model.

  The service definitions also define which controlled kernel entities are associated with each service. All services are associated with a client task and generally one or more other entities. In some cases, determination of the other entity or entities is straightforward, for instance a service to suspend a task's processing is associated with that particular task.

In some cases, more than two entities are associated with a service. For instance, task creation is associated with the client task, the parent task, and the new task.

- The kernel security requirements, in the form of *service control requirements*. This is contained in Section 7 of the FSPM.

  For each service, one or more requirements define the permissions required between the entities associated with the service. In the typical case in which there are two entities associated with the service, the client and the target entity being acted upon, the requirement simply defines which permission the client is required to have for the target. The permissions allowed are dependent upon the SIDs of the entities, not the entities themselves.

  For services associated with more than two entities, the requirements may be more complex, but requirements are always stated in terms of pairs of entities. Thus a service, like task creation, associated with three entities may have three requirements defined, one for each pair of entities.

The *kernel request control requirements* are another important piece of the DTOS security policy which will be discussed in this section, though these requirements are documented within the DTOS SDD [76] rather than the FSPM. The kernel request control requirements define which permissions must be checked for each DTOS kernel request. These are similar to the service control requirements, but are at a somewhat lower level of abstraction. The relationship between the two sets of requirements is that the kernel request control requirements applying to a particular kernel request must include the service control requirements for any service provided by the request.

6.1 Innovations

Innovations in the DTOS FSPM include:

- Development of a security policy which defines control requirements independent of security contexts and semantics.

- Use of transformation service definitions as the preferred way to define the controlled services.

- Separate controls over each of the services provided by the kernel, instead of grouping services into categories such as "read services" and "write services".

- Use of simple tables to define the requirements, and automated tools which extract the requirements into formats appropriate for different documents and even the kernel code.

- Concurrent generation of mathematically formalized and informal versions of the FSPM from a single source.

Each of these innovations is described more in a separate section below.

6.1.1 Security Context Free Security Policy

One of the key aspects of the DTOS prototype is the separation of mechanisms for security policy enforcement and decision making. The DTOS FSPM mirrors this distinction by stating a security policy which places requirements only on the enforcement mechanisms in the kernel. This separation has much the same benefit for the analysis of a security policy as it does for the actual implementation, in terms of modularization and re-use.

For instance, a typical security policy contains statements like

> **Requirement 1** A subject shall be allowed to read a file only if the level of the subject dominates the level of the file.

All analysis of the system's ability to meet the security policy is performed against this requirement.

In the DTOS model, Requirement 1 is split into separate requirements on the enforcer and the decider:

> **Enforcer 1** A subject shall be allowed to read a file only if the subject has permission to read the file.
>
> **Decider 1** A subject shall be granted permission to read a file only if the level of the subject dominates the level of the file.

Suppose in a different environment, only the identity of a user is relevant to the security policy:

> **Requirement 2** A subject shall be allowed to read a file only if the subject is acting on behalf of a user in the group to which the file belongs.

In a typical system, the entire system would need to be reanalyzed against this requirement.

However, for DTOS to support this second environment, only the decider requirement must change:

> **Decider 2** A subject shall be granted permission to read a file only if the subject is acting on behalf of a user in the group to which the file belongs.

And therefore only the decider functions must be reanalyzed.

6.1.2 Use of Transformation Services

The FSPM attempts, as much as possible, to define requirements on transformation services rather than upon actual kernel requests. There are two significant benefits which can be obtained by following this approach:

- Analysis of a system's ability to support some set of security objectives depends upon properties of the system as stated in terms of the model. By stating the controls provided by the system in terms of the same model, the analysis is simplified. For instance, it is more useful to know that "a process can only modify a file when it has write permission" than to know which permission is required for each request and have to determine which requests modify a file.

- Security requirements stated in terms of transformation services, and all of the security analysis based upon these requirements, are more robust in the face of changes to the system. Security requirements which control transformation services need not change if a new request is added (or an old one is altered) without making fundamental changes to the microkernel itself.

  As a simple example, suppose a request increments some counter. If a new request is added to arbitrarily change the value of the counter, and the transformation service defined the service as any change to the value, then the security requirements need not change as a result of the new request.

6.1.3 Maintaining Distinct Permissions for Each Service

The DTOS FSPM generally requires a different permission for each service. This is different from most security policies, which often consider each service (which generally is a particular request) to be either a read or a write service on some object. The effect is that granting one permission (read or write) allows many services to be performed. On DTOS, multiple services are controlled by a single permission only in cases where two services are very closely related.

The advantage of the DTOS approach is the possibility of supporting much finer grained least privilege controls. There are at least two ways to view the advantages provided by tightly controlled privileges:

- It simplifies the assurance effort needed to justify that a privileged application is in fact trustworthy. There is no need to assure that an application does not perform some operation if it is prohibited to do so by the operating system.

- It reduces the risks introduced by granting privileges to untrusted applications.

  For instance, a standard application may require some particular privilege that would usually only be granted to trusted applications. It may still be possible to use such an application if the privileges granted are sufficiently narrow, and if other techniques, such as audit, are used to identify if the privilege is used inappropriately.

In essence, least privilege allows “trust” to be defined in much more of a continuum than a system in which a trusted application can perform virtually any operation.

The canonical example of the problems that are created by a system which does not provide fine grained least privilege controls is the Unix sendmail program. This program performs a few privileged operations, and therefore runs as root. However, the complexities of the program have allowed attackers to use it to perform many other operations it has no need to perform, and which could be prohibited by providing more least privilege controls.

The major disadvantage of providing such fine grained controls is the difficulty of administering the security databases which define the permissions granted to each subject. This is discussed further in Section 3.3.3.

6.1.4 Requirements Tools

All statements of requirements, both service control and request control requirements, are generated from a collection of simple tables. Automated tools are used to extract the requirements from these tables into the various formats used in the FSPM and SDD. A preliminary version of the code (the `avu_msgid_table`) defining the initial permission checks for kernel requests was also generated from these tables.[14]

There are several benefits from using these tables:

- The requirements can be stated in different formats without concern about inconsistency between the different versions. This is very important because one format is needed for code, another for the SDD, another for non-formal statement of requirements and another for formal statement of requirements.
- The format of the tables can be chosen to meet the specific needs of the program rather than the needs of a particular formal specification language or programming language.
- The tables can be used as a "contract" between the assurance and development teams as the common source of information for both teams.

6.1.5 English Only Version of FSPM

Development of a mathematically formal security policy has some important benefits, including:

- Formalization can eliminate much of the ambiguity inherent in English language statements.
- Tools are available for parsing, type checking and analyzing formal language statements. In fact, some errors in the security requirements defined for the DTOS kernel were only discovered by using a parser after the statements were formalized.

However, formal specifications can be intimidating and their use can therefore eliminate some of the potential audience for a document. A possible solution to this problem is to provide a version of the security policy which is free from the formal specifications, but this leads to a further difficulty in maintaining consistency between two versions of a document.

These difficulties have been solved on DTOS by generating both formalized and English-only versions of the FSPM from the same source file. In general, this is done by using typesetting (LaTeX) macros to distinguish text that should appear only in the formal version.

This process has also had the desirable side-effect of forcing more emphasis on the use of English in the formal version of the FSPM as well, rather than relying on the formalizations to present basic concepts.

6.2 Creating the FSPM

This section discusses issues raised during the initial creation of the FSPM. It is divided into three sections paralleling the three sections of the FSPM, along with an additional section to discuss general process changes that are suggested for any similar future efforts.

6.2.1 Kernel State Model

For the most part, creation of the state model was straightforward because the system was already created. The biggest question that needed to be answered during this process was the identification of the controlled kernel entities.

[14] The tools are also used to generate a few lists of entities for the kernel state model.

To make this determination, each class of objects in the system must be analyzed to determine the services that can be requested through the system interface. If control is desired over any of the identified services, then objects of that class must be labeled. Labeling can either be implicit or explicit. For example, one class of object in Mach is an address space. Services on an address space include allocating and deallocating regions and reading and writing data. Since control is desired over these services, address spaces must be labeled. However, each address space is associated with exactly one task. Thus, each address space can be implicitly labeled using the label of the owning task.

Not all system objects are necessarily visible as objects at the user interface. For example, consider messages in Mach. Although the kernel treats messages as kernel entities, the interface does not provide any services for operating on messages. From a user perspective, messages are simply portions of a region in an address space. Messages are constructed by writing data of a particular format into a region. They are read simply by reading data from a region. When constructing a system's security policy, decisions must be made whether to label objects that are not visible at the user interface. Such labeling often is unnecessary because there are no user visible object services to control. However, in some cases, such labeling could be used internally to provide finer control on user visible services. For example, if the sender of a message were permitted to specify a security label for the message, then the receiver's access to the message could be controlled based on both the label of the port to which the message was sent as well as the label of the message.

More generally, labels should only be attached to entities when they provide value. For example, consider port sets in Mach. They are system objects since the system maintains each task's logical grouping of ports into port sets. The interface provides services on port sets allowing additions and deletions to be made, messages to be received, and the contents of the set to be listed. However, port sets are not labeled entities in DTOS since:

- We felt that control on the basis of IPC namespaces was sufficient. In other words, we saw no reason to allow a task to add elements to, remove elements from, and view the contents of one port set in an IPC namespace while preventing it from doing so to other port sets in the same IPC namespace. In particular, the common case was expected to be that only the owner of an IPC namespace would be able to operate upon it.
- Control over the receipt of a message is controlled on a per-port basis. If a task is not permitted to receive messages through a port, it should not be able to circumvent this policy control by inserting the port in a port-set for which it does have receive permission. Conversely, if a task has permission to receive a message through a port, it should retain that permission when the message is added to a port set.
- Labeling both ports and port sets would require consideration be given as to how to maintain the consistency between the labels. For example, should it be required that all elements of a port set be labeled the same as the port set itself? If so, additional security processing would be required in the service that adds elements to a port set.

Another issue to consider when deciding what object classes to label is the granularity of access control provided. For example, consider memory regions which we chose to label in DTOS. If they were unlabeled, then little control can be provided over access to memory. A task needs read and write access to the region containing its stack and execute access to the region containing the code it executes. If these regions are labeled the same, then a task would need read, write, and execute access to all of its regions. This would preclude the possibility of securely mapping files into address spaces since once a file was mapped into a region, the binding between the file's label and file's data would be lost.

Unfortunately, few decisions here are clear cut. While the examples given above illustrate some issues to consider, the choice as to which object classes to label is dependent on system dependent factors such as the granularity of control desired and the interdependencies between different classes of system objects.

6.2.2 Service Definitions

6.2.2.1 Granularity of Transformation Services The definition of transformation services is at the heart of the creation of the FSPM. One of the most difficult decisions that must be made is the granularity of the the transformation services. On one hand, defining many transformation services that are very specific allows the possibility of extremely tight control over kernel tasks. On the other hand, it also leads to more complexity in the security model and particularly in the administration of the system. Furthermore, it is not clear whether it is always practical to distinguish various services, as a task that needs to perform one operation often actually needs to perform several related operations as well.

The general approach to identifying the services provided by the system is to first identify the user visible data structures and then identify the operations that can be performed on each data structure. For example, one data structure in DTOS is the set of existing ports. Obvious operations that can be performed on sets are to add and remove elements. This suggests system services representing the creation and deletion of ports.

Unfortunately, the identification of services is usually not completely straightforward. For example, consider an integer data structure. One approach would be to define a single service denoting the changing of the value. By associating a security permission with this service, the policy could be used to control which users can alter the value. However, it would not be possible to distinguish between incrementing and decrementing the value. If associating different levels of privilege with each is desirable, then separate services and security permissions should be defined for incrementing and decrementing. For example, this distinction is desirable when the integer represents a bound on resources that can be allocated to a task. Typically, decreasing its own bound is something a task can do itself while increasing a bound is something that only a privileged subject can do. Privilege is required to prevent a task from denying resources to other tasks by inappropriately increasing the bound associated with itself.

Another complication is that sometimes data structures are accessed in groups rather than individually. For example, the creation of a new task also creates several data structures such as a new task port, a new virtual memory space and a new IPC name space. Rather than defining a separate service for adding each of these data structures, a single service is defined to represent creating a new task.

A more subtle complication is side-effects of operations. For example, consider the deallocation of a port in Mach. As a side-effect of deallocating the port, Mach deallocates any messages enqueued at the port. Furthermore, the system deallocates any ports whose receive rights were in messages which are deallocated. Any existing port rights for those ports are translated into dead names. If dead name notifications have been requested, then notification messages are sent to those tasks that requested notification. Unless great care is taken in the statement of the security policy, such side-effects can result in the policy being impractical. For example, suppose that the security policy denotes the sending of a message by a service that adds a message to the set of messages queued at a port. Then, a task that deallocates a port right could accomplish a message send service to a port by causing a notification message to be sent to that port. If the security policy required that a client can only accomplish a message send when it has send permission to the port, then the client would need to have send permission to any notification port that could ever be associated with a port whose receive right could be in transit through the deallocated port. Since there is not generally any relation between a port and its associated notification port, this is impractical. In DTOS, such situations are addressed be either:

- Defining the service to exclude the side effects.

  For example, services associated with changing a task's data structures exclude the case in which the task has just been created. A single policy statement is made as to when a task can be created. The side effects of updating each of the individual task data structures then do not need to be treated as individual services.

- Taking care to identify the "true" client for a service.

  For example, by viewing the sender of a notification message as being either the kernel or the client that requested the notification, then the policy on message sending would be easier to satisfy. In particular, the client for a port deallocation request would not be required to have *send* permission to a notification port since the service would be viewed as being provided to some other subject.

There are two serious drawbacks to the latter solution. First, the solution often is unacceptable from a security standpoint. For example, consider an MLS policy. Suppose that a low-level subject passes the receive right for a low-level port to a high-level subject. If the high-level subject destroys the port to which the message is sent, then the result is that a notification message is sent to the notification port specified by the low-level subject. Since the intent of the MLS policy is to prevent such signaling from high to low, viewing the message as being sent by the low-level subject is unacceptable.

The second problem deals with the mapping of the FSPM to the FTLS (and actual system for that matter). The FSPM assumes a binding between a service and the client for the service. The interpretation of this binding in the FTLS is not completely specified in the current FTLS. In the simplest (and most common) case, the interpretation is simply that the client for a service is the currently active thread. In cases where we choose to view the "client" as being some other thread, the fact that this interpretation is not explicitly specified makes it unlikely that it is done properly in the system.

A final issue to consider in stating the policy is whether to address all operations on a data structure or only those operations that can actually occur in the system. For example, DTOS does not provide any mechanism for the system's host name port to be changed. Consequently, there currently is no service defined that represents the value of the host name port changing. Since no service is defined for reference in the policy, the policy itself does not prohibit the host name port from changing. An implementation of DTOS which allows the host name port to be changed by any task would not violate the security policy. But, such an implementation would obviously be "insecure" since the security requirements on host operations could be circumvented. For example, a task that did not have permission to perform a host operation could change the host name port to a port for which it had all accesses. Although the framework in the DTOS FSPM allows for such a service to be specified and marked as being prohibited for all tasks, a nontrivial amount of work would be required to identify and specify all of the services which are precluded by the implementation of the system rather than the security policy of the system.

In summary, the identification of services is based upon the data structures in the system but heavily influenced by the semantics of the system operation.

6.2.2.2 Impact of Existing Kernel Design In the development of a new system, the determination of which services are to be controlled is likely to impact the design of the system. However, on DTOS the system was already designed, and one of the goals of the DTOS enhancements was to minimize the impact on the existing design.

The design of the system can have a significant impact on the development of the FSPM. Ideally, the security processing can be localized in relatively small areas of code. For example, in a message-based system the easiest approach is to integrate security processing with the sending and receiving of messages. In DTOS, much of the Mach code for request processing is retained unmodified. Most security checks are performed when the kernel receives a message. If they pass, then the Mach code for processing the request is called. Otherwise, the request is rejected.

Using this approach encourages the security policy to be defined in a way that favors using information readily available when messages are received. For example, consider the operation for changing the queue limit for a port (**mach_port_set_qlimit**). This operation is implemented as a message sent to a task port and containing parameters indicating the port and new queue limit. Since interpreting the parameters within the message is nontrivial at the point when the message is received, the only information readily available is the task port through which the message was received. Consequently, the DTOS policy controls this operation based on the task owning the IPC namespace in which the port resides rather than based on the port itself. This is non-intuitive since really a single port is being operated upon rather than an entire IPC namespace. From a security standpoint, this approach does not support least privilege very well. For example, if a task needs to be able to change the queue limit for some port in the IPC namespace of a second task, then it must be allowed to change the queue limit of all ports in the IPC namespace of the second task. In some cases this limitation does not seem severe. For example, with setting the queue limit for a port we expect the common case to be that only the task owning an IPC namespace needs to change queue limits for ports in that IPC namespace. Trusting a task to set the queue limit for all ports in its IPC namespace does not seem to be that bad.

In addition to conflicting with least privilege, the tight coupling of the FSPM and design can make the FSPM less robust in face of design changes. For example, different releases of the OSF KID have specified different ports as the port to which some requests are sent. Generally this has happened when there are multiple entities associated with the operation. One entity has to be chosen as the one to which the request is sent. If the decision as to which entity to choose is subsequently changed, then the KID has to be changed. If the FSPM keys off this entity, then it needs to change, too.

This issue is probably easier to deal with when the system is developed from scratch rather than a secure retrofit of an existing system. For example, if we were not concerned with compatibility with Mach, we could redefine **mach_port_set_qlimit** as an "out-of-band" message sent to the port whose queue limit is to be changed. Then, the port being operated upon would be readily available for use in the security check. As another example, if the ways in which port rights could be transferred between tasks were more constrained, controlling the transfer of port rights on a task-to-task basis would be more practical.

6.2.3 Control Requirements

Once the services were defined, the requirements are generally quite straightforward. There are really only two questions which must be answered:

- For each service, in terms of which entities is the service controlled?
- For each service and pair of entities, what permission should be required?

Ideally, the entities considered in each security requirement would include all entities relevant to a given service. However, the DTOS implementation of security checks is optimized for the case in which there is a single object involved. As discussed earlier, this leads to resistance to performing checks on "secondary" objects unless a clear security advantage is perceived. This is especially bad since the "primary" object is always viewed as being the object to which the request is sent; this object is not always the same as the object one would naturally view as being operated upon by the operation. In the end, the decision is a judgment call by the developers of the system. Although different choices in DTOS would result in a slightly different range of supported policies, the differences appear relatively minor for practical purposes.

The other question concerns the relationship between permissions and services. To provide the most general system, for each service and pair of entities there should be a unique permission, so this is generally the default decision. Otherwise, policies are precluded from allowing access to one service while denying access to another service. For example, rather than having individual permissions for each of the task services, we could use a single permission to control all of them. Then, any task that needed permission to (for example) destroy a second task would have to be given permission to arbitrarily modify the second task. Then, control over tasks would be little better than the base Mach system where the holding of a port right for a task allows arbitrary operations to be performed on the task.

One instance in which the same permission can (and probably should) be used for multiple services is when the services are very tightly coupled. For example, consider external memory managers in Mach. There are a number of kernel services they need to request to properly back memory. If denied permission to any of the requests, then a manager probably cannot correctly back memory. This suggests that all of the kernel requests associated with the external memory managers should be controlled by a single permission. Tasks trusted to back memory properly would be given the permissions with all other tasks denied.

A related instance is when the security relevance of different services is judged to be the same. For example, consider the services associated with changing the queue limit and sequence number of a port. We felt that any task trusted to perform one operation could safely be trusted to perform the other operation, too. So, we defined a single permission to control both operations instead of separate permissions for each.

In summary, typically a separate permission should be associated with each service. This provides the finest granularity of control. However, if it becomes apparent that providing some accesses in a group while denying others either breaks the system or serves no purpose security-wise, then a single permission can be used to control each of the services in the group.

6.2.4 If We Had To Do It Over Again

If given the opportunity to start over again, the most significant change that would be made to the approach used for developing the FSPM would be to use a "vertical" approach rather than a "horizontal" approach. By this, we mean that rather than trying to first address all services at a high level, we would choose some set of services to consider and completely address those services before moving on to the other services. Completely addressing a service would require:

- Identifying how the service would be mapped to the FTLS.

  This still has not really been done for DTOS. Significant issues to be addressed include mapping the atomic view of services to the non-atomic request specifications in the FTLS and mapping the FSPM notion of "client" label to the FTLS.

- Identifying how the service would be mapped to the implementation.

This has been done on an *ad hoc* basis as assurance personnel have become more aware of the implementation and developers have become more aware of the FSPM. At the beginning of DTOS, there was a fundamental miscommunication between the developers and assurance. Since the FSPM is largely independent of the actual operations in the system, the FSPM focused on identifying the various services provided by the system. To provide some confidence that the identified set of services was complete, the assurance team stepped through each request in the system and mapped any *unique* services provided by the request to the services identified in the FSPM. In cases where there were multiple requests that provided the same service, the service was not always identified as relevant to every one of those requests. The main concern was whether it was identified as relevant to at least one. When the developers saw this mapping, they assumed it was the *complete* mapping of services from the implementation to the FSPM. In cases where some services relevant to a request were not identified in the mapping, the developers failed to implement some security checks that were required by the implementation.

In addition to this communication issue, the concerns with mapping the FSPM to the FTLS apply to the implementation, too. For example, figuring out which label to use as the label of the current "client" in the implementation is not always straightforward.

- Considering how a proof could be given of the consistency of the FTLS with the FSPM.

  No such proofs have been done on DTOS.

Once all of these issues have been resolved for the selected set of services, work can begin on other services. By doing a "complete" trial on a small set of services at the beginning, improvements in the approach for the FSPM can be made before stepping through each of the requests. In contrast, the DTOS approach was to:

- Develop the approach for the FSPM.
- Apply that approach to the entire system to obtain a complete first draft.
- As problems were identified by subsequent analysis, revise the approach and hope the necessary changes in the FSPM were not to severe.

6.3 Obstacles

This section discusses some of the difficulties that were encountered with the approach that we have taken, and which have not yet been completely addressed. Many of the difficulties arise from the fact that we are working with an existing system that cannot be arbitrarily changed, and require workarounds rather than complete fixes.

The obstacles are divided into sections paralleling the major parts of the FSPM, except for the state model which is discussed more generally as part of the FTLS in Section 7.

6.3.1 Service Definitions

This section discusses some of the difficulties encountered in creating or using the service definitions in the DTOS FSPM.

6.3.1.1 Some Services Cannot Be Defined As Transformation Services

The transformation services are defined entirely in terms of changes to the modeled system state during a state transition. Unfortunately, not all services can be defined in terms of state transitions. The most prevalent services of this kind are services which simply observe portions of the system state but have no effect on the state.

To be more precise, observations of data do actually have an effect on the state, as they are really copies of data from one location to another. The action of modifying the target location is definable as a transformation service. The problem is that to control observation of data, there needs to be control not only over where data is copied to, but where the data was copied from, and there is no change to the state of the original location to indicate that the data was copied.

The only cases in which any control can be provided over this kind of operation is when the data itself is typed and can be interpreted in terms of the model. For instance, consider a service of copying a port right from one task to another. There is no way to determine, from the model, which task was the source of the port right. But, the identity of the port right itself can be determined and controlled.

An analogous service for arbitrary data could not reasonably be provided because it makes no sense to assign security contexts to arbitrary data.

Due to the difficulties in addressing all requests as transformation service providers, we introduced the notion of invocation services into the FSPM. As discussed in Section 6.1.2, transformation services are preferred.

6.3.1.2 Some Services Cannot Be Defined As Either Kind of Service

Some of the "services" provided by the microkernel cannot be defined either as transformation services or invocation services, and therefore cannot be controlled within the DTOS framework.

The problem is that our intuitive sense of service may not really map to anything which can show up as a transformation or invocation service. Such services are those which cannot be modeled as transformation services (as discussed in Section 6.3.1.1), and which do not correspond to a single kernel request (and therefore cannot be considered as an invocation service).

One example of this in Mach is the transfer of port rights. Everyone agrees that a port right is transferred from task 1 to task 2 if the right used to be in task 1's space and now is in task 2's space. Stating a policy on when such transfers can take place would be desirable.

Unfortunately, this service cannot be modeled as a transformation service because in general there is simply no way to determine where a port right came from. While in some requests it is quite straightforward to determine the source of a port right, in general there are so many ways of passing and acquiring port rights that there is no general method for assigning a particular source for every port right.

And this "service" can also not be defined as an invocation service because there are many ways to transfer rights, some of which involve multiple requests. For example, task 1 might transfer a right to task 2 by setting task 3's bootstrap port. When task 2 gets task 3's bootstrap port, the right will have been transferred from task 1 to task 2. Since the transfer is split across two different requests, providing the desired control is difficult. For example, allowing task 1 to set the bootstrap port and allowing task 2 to get the bootstrap port might be appropriate independently even though allowing task 2 to get the bootstrap port after task 1 has set it might be inappropriate. Enforcing this policy would require more significant modifications to the kernel to record which task last set the bootstrap port and would require more complicated permission checking to make use of this additional information.

6.3.1.3 Uncontrolled Observation Services

The range of services defined and controlled in the FSPM are not sufficient to control observation of critical system data. This is not a problem in the DTOS kernel itself, just a problem with the practices followed in developing the FSPM.

This situation is best understood by describing the way in which the current set of services was developed. The first step was to define transformation services to describe the basic changes which can be made to the kernel data structures. The second step was to look at all of the kernel requests which did not provide any of these services, and to define an invocation service to correspond to each such request.

The problem with this approach is that a request which performs one service may perform other services that should also be controlled, and these services, in particular observation of data, are not controlled in the FSPM. A simple example is the following:

The request `mach_port_set_seqno` is used to change the sequence number on a port. Within the security requirements in the FSPM, it is controlled only to the extent that it provides the transformation service *SetsSeqNo*. However, suppose that the implementer of the system chose to follow the letter of the FSPM and wrote this request so that it only checked for the permission if the new value provided as a parameter to the request was different than the current value of the sequence number.

Now suppose that a task does not have permission to perform the request `mach_port_get_receive_status` on a port (this request returns the sequence number associated with a port), and the task did not have permission to perform the transformation service *SetsSeqNo*. The task could still determine the sequence number through trial and error, by making the `mach_port_set_seqno` request with different parameters for the new sequence number. If the parameter is not the current sequence number, then the request will return `KERN_INSUFFICIENT_PERMISSION`. If the parameter is the current sequence number, then the request will return `KERN_SUCCESS`.

This could be used easily to create covert channels. It is not a practical concern in the current system, since the system is not implemented in this way, but still the FSPM states no requirements controlling a service which must be controlled by the system.

One possible solution would be to provide a new set of "observation services" defining all of the various observations that can be performed. To avoid unnecessary changes to the prototype, these new services would be controlled by existing permissions.

6.3.2 Control Requirements

This section discusses some of the difficulties encountered in creating or using the service control requirements in the FSPM and the request control requirements. Because these two sets of requirements were generated concurrently and are closely linked, it is difficult to separate the issues encountered in developing each set of requirements.

6.3.2.1 Kernel Implementation Limits Possible Service Control Requirements

If the FSPM were being developed as part of a top-down development of a new system, then an appropriate way to define service control requirements may be to look at each service and define permissions between each pair of entities associated with that service. These requirements could then be passed on to the development team who would need to determine the best way to satisfy the requirements.

While a purely top-down development model is generally extreme, the model followed on DTOS was at the opposite extreme since the basic kernel implementation already existed without any of the DTOS security controls. This placed a great number of limitations on the possible request control requirements which could be met, and hence on the service control requirements as well.

Perhaps the most serious limitation was the desire to implement a simple object-oriented model of access control for as many requests as possible. In this model, each request is a "method" on some particular controlled object, and the permission check is a simple check of "can the client task perform this method on the target object". The value of the model is simplicity and the expectation that this kind of control can be performed with no noticeable performance impact.

The difficulty with this model is that Mach is not strongly object oriented. For instance, there are over 45 kernel requests which are received through a task port, though logically these requests include operations on tasks, IPC name spaces, address spaces and ports. Since IPC name space and address spaces remain associated with a single task, this presents no significant problem. However, a few of the requests received through task ports are logically requests on ports (e.g., `mach_port_get_receive_status`).

The DTOS program has not addressed this particular issue, and therefore some potentially desirable security policies cannot be supported on DTOS.

To expand upon a particular example, the kernel request `mach_port_get_receive_status` returns some information about a port, such as the number of messages currently queued at the port. For a client to perform this operation, the kernel requires the client to have `observe_pns_info` permission to the task holding the receive right for the port. There is no direct permission checking between the client and the port. Such permission checking could be added, but not within the simple permission checking model.

One interesting side effect of this lack of permission checking may occur during a policy change. Suppose that the policy changes in such a way that a task is no longer allowed to receive from a port for which it holds the receive right. Since a task almost certainly needs to the `observe_pns_info` permission to itself, there is no way for the policy change to block the task from performing `mach_port_get_receive_status` on the port after the policy change, even though it can no longer receive messages from the port.

6.3.2.2 Some Simple Service Control Requirements Are Difficult To Meet

Ideally, any service which can be defined as a transformation service should be easily controllable within the kernel. However, the kernel implementation may make this impossible. The general problem is that simple services are often performed as a side effect of other services. The most common example is deallocation of data structures.

As an example, the destruction of a single port may cause the receive right for other ports to become "orphans", in which case the other ports are destroyed as well. Suppose a client has permission to destroy the initial port but not some orphaned port. What should be done? Some choices are:

- Prohibit the entire operation which led to destruction of the initial port. This could be quite difficult to implement, since it requires the code to be "reversible", or else it requires searching for all side effects before beginning any of the processing of the operation (and would also require locking many data structures).
- Prohibit destruction of the orphaned ports. This would satisfy the security requirements, but deliberately creating orphans is bad programming and could confuse potential clients of the orphaned port. It also would violate the semantics of the existing system.
- Distinguish in the policy between side effects and direct requests, and do not control a service when it occurs as a side effect. In general, this can be difficult if not impossible.

The third choice is what is generally done in the security policy, but this diminishes the benefits of using transformation services.

Note that this problem could also be considered to have arisen from the fact that the implementation existed prior to the policy development. In some cases, that may be the case, but in others, including the given example, it is hard to see how another implementation could avoid the problem without significant change to the messaging system.

6.3.2.3 Use of Two SIDs in All Permission Checking

In order to simplify the decision making functions and the infrastructure for caching decisions within the kernel, it was decided early in the program that all permission checking in the kernel must be based upon determining whether a particular permission is granted between two SIDs. In particular, enforcement is never based upon decisions made simultaneously among three or more SIDs. This decision, while removing many potential complications from the implementation, may limit the ability of the DTOS kernel to support certain security policies.

From the point of view of defining security requirements, this decision means that all requirements must define permissions between two controlled entities, even if more than two entities are associated with the service. For services associated with three entities, there are often three service control requirements, one for each pair of entities. There are currently no services associated with more than three entities.

Providing control in terms of each pair of entities individually dramatically limits the theoretical range of policies which can be supported.[15] It is unclear how limiting this restriction is in terms of practical security policies. One example of a policy which cannot be supported is found in the following example.

Consider an unmodified server S which uses port rights to identify clients which pass messages through a common port. When a client C initiates contact with S, an authentication protocol is executed. Once S is convinced of the identify of C, it creates a port P for C and passes C a send right for P. C can continue to use this right to authenticate itself to S. It would be desirable for the DTOS controls to be sufficient to ensure that C could not pass the send right for P to some other task, but this cannot be done with the current prototype.

To see why, let C' be another legitimate client of S, which should use a port P' to identify itself to S. Since S is unmodified, the kernel assigns the same SID to all ports created by S. In particular, P and P' are assigned the same SID. Therefore C' must have permission to hold P. C must have permission to transfer P since it uses P to identify itself to S. Finally, if C and C' also communicate for other reasons, then C may have permission to transfer rights to C'. Within the current DTOS prototype, these permissions allow C to transfer the send right for P to C'.

However, if the permission to transfer rights were controlled based upon a triple consisting of the sender, recipient and port, then this scenario would be eliminated, since C would only need permission to transfer its right for P to S, and not to C'.

6.3.2.4 Some Security Requirements May Not Involve a Subject SID

The initial DTOS permission checking model was based upon a simple paradigm: all security relevant operations can

[15] Mathematically this can be explained simply. Consider a 3-dimensional space, where each axis represents the SID of one of the three entities associated with the service. The policy for that service can be represented by the set of points in this three dimensional space for which the service is allowed. Stating the service control requirement in terms of the three entities simultaneously allows for arbitrary policy sets. Stating the requirement in terms of three pairs of entities requires that the policy set be the intersection of three "cylinders".

be represented by a task acting on some object. One of the shortcomings of this model, the possibility of operations involving three entities (or three SIDs), was discussed in Section 6.3.2.3).

Another shortcoming of this model is the assumption that one of the SIDs involved in each permission check is a subject SID. There was actually no need on DTOS to remove this assumption, however, it is easy to imagine services for which it is not sufficient. For instance, suppose an operation were added to change the SID of a port. Analogous to changing the SID of a task, the three labels involved in the permission checking for this operation would be the client task's SID and the old and new port SIDs. There is no subject SID in the permission check comparing the old port SID and the new port SID.

Removing this assumption would be more of a nuisance than a significant change, though it should be removed on any similar effort.

6.3.2.5 Security Policy Needs to Consider Entity Relationships

It can be very difficult to express all of the relationships between controlled entities in terms of security contexts. A simple example is an ownership relation which may exist between one entity and another. Another example is an application made up of multiple entities. There may be several such applications on a system at a time, each with equivalent sets of security contexts, but still the individual entities in one instantiation of the application should not be interacting with the entities in a different instantiation.

This has proven to be a difficult issue to address. One possible solution is to assign every entity a unique SID, so that the relationships can be "recorded" in the SID. In this solution, a SID becomes a combination of a unique identifier (UID) and a security context. This solution has two significant drawbacks:

- SIDs may be assigned by "default", for instance, based upon the SID of the client creating an entity. If all SIDs are required to be unique, it becomes more difficult to assign default SID values.
- The security server has to somehow be aware of and record all of the relationships between entities that are not captured strictly in the entities' security contexts. It is not at all clear how this can be accomplished.

A small step towards a solution of this problem has been implemented in the DTOS kernel.[16] The kernel itself already captures one relationship between entities, the relationship of "ownership" of an entity by some task. In terms of controlled entities, this ownership relation applies only between a task and the threads within the task.

This relationship between a task and its threads is taken advantage of in a detail of the security requirements that has not been previously discussed. We have generally described the requirements as defining a required permission between the SIDs of two entities. In reality, the requirements are slightly more complicated. In particular, in some cases where the permission is checked between a client task and either a task or thread entity, the SIDs against which the permission is calculated are chosen as follows:

- The first SID is always the client task's SID.
- The second SID is the target task or thread's SID, except:

[16] This is perhaps as big of a step as could be taken without considering the security server as a main part of the solution.

- If the target is a task, and it is the same as the client task, then the second SID is set to a special value.
- If the target is a thread, and it is a thread in the client task, then the second SID is set to a (different) special value.

These "special SID values" indicate to the security server that the operation is being performed by a task to an entity belonging to itself.

The same approach taken for the kernel could potentially be used to represent relationships between entities managed in other servers. The real difficulties arise when dealing with relationships between entities managed by different servers, which is probably the most common case since often a task will be one of the entities.

Related issues are discussed in entry 9 of the DTOS Notebook [77].

6.3.2.6 Control Requirements Involving "Special" SIDs The simple model for determining the control requirements is to identify which entities are relevant to a service, and requiring a permission check between the SIDs of each pair of relevant entities. In order to provide more specific and flexible controls, "special" SIDs are occasionally used instead of the actual SID of an entity. There are three particular cases where this occurs:

- When a client is accessing itself or a thread belonging to itself, as discussed in Section 6.3.2.5.
- When the kernel sends a message in response to a client request, in some cases a special SID derived from the client's SID is used in message processing instead of the kernel's SID.
- When a client sends a message and specifies a particular sending SID (see Section 3.2.5), some kernel permission checking is performed against the specified SID rather than the actual SID of the sending task.[17]

Individually, each of these cases is simple to specify. However, the interactions between the three cases can become tricky, and have never been completely addressed on the program. For instance, suppose a client performs an operation on itself but after specifying a different sending SID. Should the special SID in the first bullet be used in the permission checking or not?

[17] Actually, this has not been implemented, though it is part of the design.

Section 7
Formal Top Level Specification

This section discusses the DTOS formal specification task, the output of which is captured in the DTOS Formal Top Level Specification (FTLS) [74]. After providing an overview of the task, the following topics are covered:

- Section 7.1 discusses some of the decisions which must be made for any specification effort and the way in which these decisions were made on DTOS.
- Section 7.2 discusses some of the aspects of the DTOS specification which appear to be innovative in some way, going beyond the simple writing of a specification.
- Section 7.3 discusses some of the obstacles encountered while writing the specification, and how they were dealt with on DTOS.

There are three typical uses for a system specification:

- Developers of the system may use the specification as a reference during implementation.
- Users of the system may use the specification as a reference to describe how the system should respond to inputs. In this context, "users" can include end users as well as system programmers and administrators. For a microkernel, programmers are the only significant users.
- Analysts who are evaluating the system for its applicability to a particular purpose may rely upon the specification in an analysis to determine if the system can meet their requirements.

As the DTOS program evolved, the primary purpose of the specification task changed considerably. Initially, its main purpose was to create a complete specification of a substantial portion of the prototype for use in a planned covert channel analysis. As the overall focus of assurance on the program shifted (see Section 5.2), and the covert channel analysis of the DTOS prototype in particular was scaled back (see Section 11), the FTLS was needed less as a basis for any specific analysis task. Like most of the other assurance tasks, the focus of the formal specification task became more research oriented. In general, the research pursued under this task included ways to make the specification more palatable to developers and users, and the specification of aspects of the system which are often left unspecified or specified only at a very high level.

The DTOS FTLS presents a formal model of the kernel as a state machine, written in the Z specification language. A typical state machine model consists of three basic parts:

- A definition of possible states of the system.
- A definition of the initial state of the system.
- A definition of all possible transitions from one state to another, including inputs and outputs associated with a transition.

The DTOS FTLS is organized into four main parts. Note that there is no definition of the initial state, primarily because that would seem to require a tedious effort with no identifiable research benefit.

- State model. As mentioned in Section 6, this model was shared with the FSPM.
- Execution model. The execution model includes extensions to the basic state model to encompass state elements representing the current execution state of the system, as well as a description of the basic transitions among these elements. This section describes an often ignored connection between the state model and the transitions. Sec section 7.2.2 for further discussion.
- Request specifications. Specifications of approximately one-fourth of the kernel requests were completed. Section 7.1.2 discusses the choice of requests to specify.
- Refinement. The state model was refined to a lower level of detail in some areas where the specification was at a considerably higher level than the code. This was done primarily to assist in the specification to code correspondence, as described in Section 8.4.2.

7.1 Basic Specification Issues

The DTOS program began with an understanding that the formal specification would model the DTOS prototype kernel as a state machine in the Z specification language. Within the framework of a state machine, there are still several basic questions that needed to be answered in order to write the specification. This section discusses several of these questions.

Because the decisions on these issues are fundamental to the specification task, they were made early in the program during the period in which the main purpose of the specification was to aid in the covert channel analysis. In some cases discussed below, the decisions changed as the goals changed.

7.1.1 When to Write the FTLS

The timing for writing a specification is often a difficult decision. If the specification is to be used during the development process, then timing is driven by that process. However, on DTOS this was not an intended use for the specification (see Section 5.3.2).

If the specification is to be used for a covert channel analysis, it may be desirable to start the specification and analysis as early as possible in order to identify covert channels before they have become too heavily embedded into the system. On the other hand, if the specification is written so early that it needs to change, then the analysis must be updated with the specification, which can be quite costly.

On DTOS this was not as difficult of an issue as it could have been, because the Mach kernel was already complete and it was expected that the most significant source of covert channels would be in the Mach kernel as opposed to the DTOS additions to the kernel. Therefore the specification was begun concurrent with the development of the DTOS additions.

7.1.2 What Portions of the System to Model

From the beginning of the program, there was a recognition that a complete specification would likely require more resources than were available on the program. It was therefore important to determine which portions of the system to emphasize.

As mentioned above, for the covert channel analysis the existing kernel was expected to be of more interest than the DTOS additions. The initial set of request specifications were written in three of the core areas of the kernel: virtual memory, port name space and thread management. Within each area, the simpler requests were typically specified first in order to gain experience before beginning the more difficult requests, including Mach IPC.

After the goal of the specification task changed, only a few additional request specifications were written, and these were chosen for reasons specific to a particular research goal. If the original goal of the task had been research, the chosen set of requests would have been quite different. In particular, requests would probably have been specified from among more kernel "subsystems" and throughout a broader range of complexity.

7.1.3 Single or Multiple Components

The Mach kernel is sometimes described as consisting of multiple distinct subsystems, for instance the virtual memory system, port name space management and thread management. The kernel specification could potentially have been written to take advantage of this modularity by modeling the kernel as a collection of cooperating components instead of a monolithic kernel. Unfortunately, the actual implementation does not provide a clean separation between any subsystems, and without a strong logical distinction in the implementation it may not even have been possible to present such a distinction in the model.

Therefore it was an easy decision to determine that the kernel should be modeled as a single component.

7.1.4 Amount of Detail in State Model

An important decision to make when writing any specification is to determine the level of detail to provide in the state model. In some cases portions of the state may be included in the model but at a higher level of abstraction. In other cases, portions of the state may be left out of the model entirely under the assumption that this portion of the state is unimportant to any user of the specification.

By definition a "top level" specification is at such a high level that implementation details are all abstracted out. The specification of the DTOS kernel is pushed to provide lower level details for two (somewhat related) reasons:

- As a general purpose microkernel, the DTOS kernel makes visible at its interface much more implementation detail than a traditional operating system.
- A covert channel analysis by its very nature requires implementation details that may be uninteresting for most other uses.

Based upon Secure Computing's earlier experience with covert channel analysis for the LOCK system [29, 26], the philosophy adopted for the DTOS FTLS was to model the state to include enough detail so that the exact output of any request is completely determined by that state in combination with inputs to the request.

In only a few places was this approach abandoned. Examples include:

- When a new port right is allocated in a particular task's name space, the name of this right is chosen based upon details of the actual data structures used to store the name space. These details were abstracted out in the model. It is interesting to note that most of the necessary details were later added as part of the state model refinement.
- The request `thread_get_state` returns low level state information about a thread, such as its current register values. There is certainly no way to reasonably model information at such low levels.

There are several aspects of this decision that are interesting to reflect upon:

- It is unclear whether the covert channel analysis really requires so much detail in the state model. It may be possible to combine state variables to simplify the specification without impacting the covert channel analysis significantly.
- While the amount of detail chosen in the model for use in the covert channel analysis may seem extreme, there is no reason to believe that other kinds of analysis might not require more detail. For instance, analysis of real time properties or fault tolerance properties are likely to require very different aspects of the system to be modeled in detail.
- A specification is only useful if it is accurate. The results of specification to code task suggest that even the level of detail provided in the model may not be close enough to the implementation to have much confidence in its correctness. This is part of the reason for the state model refinement.
- A highly-detailed specification will be more difficult to maintain over the lifespan of a system.

These often conflicting observations imply that the ideal specification is one which can be presented at various levels of detail for each of the different uses of the specification. The state model refinement can be considered a small first step towards this. An ideal specification would provide automatic formal derivation of higher level specifications from a low level specification.

7.1.5 Atomicity in Transitions

The previous section discussed the amount of detail in the state model. A somewhat analogous issue for transitions is the general size of transitions to model as atomic.

The initial decision was to follow common practice and to model each kernel request, from input to output, as a single state transition. This is sufficient granularity for the needs of the covert channel analysis.

However, after the initial set of request specifications was completed and the goals of the task changed, this decision was reconsidered. There were two main (related) concerns which indicated that a finer grained atomicity was required:

- The specification to code correspondence was overly complicated because the original atomicity combined so much processing into a single state change. Some of the processing could have been more easily analyzed as smaller transitions. Other processing which is common to many requests could have been analyzed once if that portion of the processing were atomic but instead had to be analyzed as part of every request which included that common processing.[18]

[18] It is possible that there may be some mechanism so that “common processing” could be analyzed once even if included in multiple atomic operations, but there was no serious attempt to do this on the program.

- The accuracy of the specification at the original level of atomicity was questionable. While the Mach kernel includes locking operations to ensure atomicity in some operations, the locking protocols are poorly documented and any atomicity guarantees are difficult to determine. Even more fundamental, complete processing of a single request can potentially require the kernel to make multiple outcalls to either pager or security server, and it becomes very difficult to justify atomicity across such outcalls.

As a result, the execution model and one sample request were rewritten completely to use simpler atomic transitions. This is discussed further in Section 7.2.2. Note that like detail in the state model, atomicity in transitions can be moved to a higher level by combining transitions if desired.

7.1.6 Amount of Determinism in Transitions

The issue of determinism in a specification, i.e., whether transition specifications should specify exactly the outputs and final state of any transition, is another fundamental question. For some purposes, complete determinism is undesirable because it limits the range of implementations that the model satisfies.

However, completeness of a covert channel analysis is assured by a deterministic model. Depending upon the methodology chosen, nondeterminism can either lead to undetected covert channels or to detection of channels which do not actually exist. For this reason complete determinism in the specification was a goal.

7.1.7 What Level Interface to Model

When modeling state transitions, the model describes not only how the elements internal to the state change during a transition but also the inputs and outputs to the system. However, the definition of inputs and outputs can be considered at many different interface levels. For the specification of the DTOS kernel, three possible interface levels were considered:

- The client interface through the MIG routines.

  This is the interface which is defined in the Kernel Interface Document [85]. If clients using this interface are to be analyzed against a set of application requirements, then this is the logical interface level to consider. However, because clients are not limited to using this interface, it is not sufficient for analyzing the potential actions of all clients.

- The internal kernel interface which corresponds to the client side interface defined in the KID.

  This interface usually involves the same values as the client interface except for certain translations, such as between names and ports. This slightly complicates analysis of application requirements but it provides more confidence in the analysis of all possible client actions. However, this interface is fairly high-level within the kernel and there is a considerable amount of low-level processing available to clients which is still not considered.

- The internal kernel interface to the kernel trap routines in the KID.

  This is in some sense a subset of the previous, because all non-trap routines in the KID are actually accessed through the `mach_msg` trap routine. Modeling the interface at this level clarifies that dependence and provides a convenient way to model the relationship between the trap and non-trap routines.

No serious consideration was ever given to modeling a lower level interface, such as the passing of register values during a context switch. While this would provide more complete coverage of the kernel, the details would likely be overwhelming.

The approach taken in the DTOS FTLS was to combine all three approaches. The organization of the FTLS suggests the second approach as each KID routine is modeled in a separate section. However, the execution model provides a means of formally specifying the processing between this level and the trap interface level, so the third approach is also supported. Finally, informal mappings are provided between the client and kernel interfaces, to support the analyis of applications.

7.2 Innovations

This section describes a couple of areas in which particular effort was expended on the task beyond what was necessary to write a basic formal specification. We believe that these represent unique aspects or innovations of the DTOS FTLS.

7.2.1 Presentation Style

One of the nicest features of the Z specification language is the available support for integrating explanatory text into the specification. In fact, Z specifications often look more like a text document with the formalism interspersed while specifications written in other languages tend to look more like programs with occasional comments.

On the FTLS we took advantage of this feature and attempted to create a specification which could be largely understood through the text alone by an individual unfamiliar or only vaguely familiar with Z. In terms of TCSEC[51] terminology, the goal was to present the FTLS and DTLS within the same document so as to decrease the overall specification effort and increase confidence in the consistency of the specifications.

Unlike the FSPM (see Section 6.1.5), this goal was not implemented to the point of creating a version of the FTLS with all formalisms removed. However, for the most part this could have been done without adding new text. The reason that an English-only version of the FTLS was not created was simply because it was not felt to be worth the time required to insert and maintain all of the necessary text processing macros.

To further increase the comprehensibility of the FTLS, a large collection of formatting, naming and usage standards were adopted. These standards covered the formal Z as well as the organization and wording. Automated tools were developed to verify some of these standards, while others not so amenable to automatic verification were included in checklists which were completed during review of the request specifications.

It turned out that these standards did in fact accomplish their goal. When a new person unfamiliar with the FTLS was brought onto the program to help out with the final revisions, that person reported that several of the conventions were very helpful. In particular, different logical elements of the specification were easily identifiable by their appearance.

The following are some simple examples of the kinds of standards adopted. The complete set of standards are described in the DTOS FTLS Plan [75] (not a program deliverable).

- Capitalization and underscore usage in identifiers.
- Indentation and line breaking in the Z formalism.

- Standard wording of various portions of the text.

7.2.2 Execution Model and Transition Specifications

As described in Section 7.1.5, the original viewpoint was that the proper atomicity for transitions was one transition per request. This is not exactly true. There were a few special cases in which request specifications were broken into multiple transitions, and special mechanisms were put into the model for each of these cases.

As we gained experience specifying some of the more complicated requests, and especially after beginning the specification to code analysis, it became clear that a finer choice of atomic transitions was needed. Moreover, the model needed a general mechanism for dividing a single request into multiple transitions. Some driving forces behind these realizations included:

- Requests with "deferred" permission checks (i.e., permission checks embedded in the processing of the request rather than at the entry point to a request) could potentially block several times to send messages to the security server.

- Processing common to multiple requests was often included into each request transition specification using the schema calculus. This made each specification more complicated than it needed to be and in some cases it resulted in completely incorrect specifications (see Section 7.3.5). It also complicated the specification to code correspondence since the common code needed to be considered as part of the analysis of every request incorporating the code.

- It was difficult or impossible to specify requests that involved recursive function calls. The prime example of such requests are those which destroy a port representing a message queue. As a byproduct of destroying the queue all messages in the queue are destroyed, hence all port rights in those messages, and possibly the ports themselves.

To deal with these problems an entirely new execution model was introduced in the final draft of the FTLS. The main feature of this model is a general technique for breaking a single request into many smaller transitions, replacing the ad hoc techniques used to do this in the original execution model.

In essence, this model provides a very abstract view of each thread's stack, as a way of maintaining enough state information to define the next transition that each thread will undergo. This is a very different model than typically used in specifications, which is to consider transitions as being essentially independent; each transition is caused by some external input and all processing caused by that input is completed in a single transition.

The new model also allows the specification to much more accurately reflect the implementation. It allows reuse of transition specifications to parallel code reuse, and the "calling" of one transition from another.

To give some flavor for the atomicity in this new model a permission check is broken into the following transitions:

- Processing up to the point where the permission check is needed. The state at the conclusion of this transition essentially indicates that a function call is required to find a particular permission.

- The kernel checks for the appropriate access vector in the access vector cache. If found the transition completes indicating that the transition can "return" to the previous processing. If not found, a message is sent to the security server, and the transition completes indicating that the kernel must now wait for a response.
- When a response is received, the kernel looks up the appropriate permission, and the transition completes indicating that the transition can return to the previous processing.

The first break between transitions in this example exists to separate out the common permission checking code. The second break exists because the processing truly blocks waiting for a response from the security server.

7.3 Obstacles

This section discusses some of the obstacles which were encountered while writing the specification, and how they were dealt with.

7.3.1 Existing Documentation was Inadequate

Initial plans for writing of the DTOS specifications called for using the following existing documentation as the main source of information about the Mach kernel:

- The Kernel Interfaces Document [41] and Kernel Principles [42] written for OSF's version of Mach.
- Mach specifications written by Computational Logic, Incorporated (CLI) [11, 12, 9, 10]
- The DTMach FTLS [65], which was written based upon some of the previous documentation.

These sources turned out to be completely inadequate, as the documentation was incomplete, and more seriously, sometimes inaccurate.

In retrospect, neither of these inadequacies are surprising. The OSF documentation was generally written for use by application programmers, not by analysts interested in the internal workings of the kernel, so it should not have been expected to be complete. Discrepancies between the OSF documentation and the prototype are most likely due to using the CMU implementation instead of the OSF implementation. Finally, the CLI documentation which was available early in the program was incomplete, as was freely noted throughout that documentation.

A certain amount of incompleteness was expected. For instance, if multiple errors are applicable to a particular request, the FTLS is expected to specify exactly which error message is returned. Typical users are unconcerned about details such as this which may actually be quite important in a covert channel analysis. These complex behaviors might be particularly likely in a system such as Mach where the interactions between kernel objects are intentionally rich.

Therefore it was expected that it would be necessary to consult the kernel source code occasionally to help resolve such questions of detail. But the serious deficiencies in the available documentation led to the use of the source code as the primary source of knowledge for writing the FTLS.

The lack of adequate documentation had two detrimental effects. One was the need for more time to understand the system since source code requires more time to understand than good documentation. It also caused inaccuracies in the early specifications which were written before the source code became the main source of information. Fixing these inaccuracies caused more delays as the changes sometimes rippled throughout the specifications.

7.3.2 Distinct State Models in FTLS and FSPM

The FSPM is used to define concepts that are at a much higher level of abstraction than the FTLS. Therefore it is reasonable to assume that the state model within the FSPM would be at a higher level of abstraction than the state model within the FTLS.

On DTOS, both state models were initially derived from the state model written for the DTMach program [65]. Because of the assumption about the required level of detail, the state models diverged for the first few drafts of each document. However, as corrections were made to the models, it became difficult to maintain them individually, and eventually the decision was made to recombine them into a single model which contained the necessary information for both documents.

This was unfortunate for the FSPM, because the state model there does not require nearly the detail required by the FTLS. But without a better way to maintain the two models in unison, it seemed that it was best to combine them. If a similar effort were to be started again, one solution would be to treat the FTLS state model as a refinement of the FSPM model.

7.3.3 Uniform Typing for Transitions

Since the FTLS is written using a typed specification language, when performing analysis of the specification it is desirable for all transitions to have the same type. This was difficult to accomplish for the DTOS FTLS and never was completely done. It's unclear whether this problem is unique to Z or if it is shared by other typed specification languages.

The essential cause of the problem is that a transition consists of four elements: an initial state, a set of inputs, a final state, and a set of outputs. For all transitions to have the same type requires that the types of each of these four elements is the same for all transitions. In the first draft of the FTLS, the initial and final states for each transition did not necessarily even have the same types, since local variables were occasionally carried from one transition to another through added state variables.

This was changed with the new execution model so that local variables were all combined into a single type.[19] However, inputs and outputs are still not of the same type, and a similar technique should eventually be used to provide a single type for inputs and another single type for outputs.

7.3.4 Lack of Analysis Tools

As mentioned earlier, there are excellent tools available for typesetting Z specifications. But tools for analyzing Z specifications are much less well developed.

The problem with this lack of tools is not just the inability to reason about the specification, but the inability to gain much confidence in the correctness and consistency of the specification.

[19] Actually, this was not done for all of the request specifications since they were not updated with the new execution model. But the execution model not only makes this possible but essential.

Much like a program, until a specification is used it is difficult to have much confidence in it. Informal reasoning about specifications is notoriously error prone because it is so easy to overlook details.

Very late in the program the Z/Eves theorem proving tools became available. Unfortunately, the first release of the tools turned out to be incompatible with the FTLS and by the time the next release was available the specification task was complete.

The only Z "analysis" tool used on the program was Fuzz, which does the following:

- Verifies that the syntax of a specification is correct.
- Verifies that the declared type of the operands in all operations is correct according to the declaration of the operation.

It is interesting to compare this to the `typecheck` operation performed by PVS, a popular specification tool developed at SRI [55]. In both languages, the domain and range of a function can be declared to be a restricted subset of a particular type.[20] The `typecheck` operation performs the same syntax and type checks as Fuzz, but in addition generates all *type-correctness conditions* (TCCs) which must be proven in order to demonstrate that any value to which a function is applied is within the declared domain of the function and that any defined values of the function are within the declared range.

The value of this extra type checking is quite significant. The kinds of errors detected by Fuzz tend to be simple errors of omission or minor type errors, such as using a single variable of some type where a set is required. Only rarely were actual logical errors within the specification uncovered by Fuzz. But PVS type checking is much more likely to uncover logical errors.

For example, consider a function whose domain and range are declared to be the positive integers, and suppose the function is then defined to decrement the input value by one. PVS would detect the error in this specification, or more accurately, it would generate an unprovable TCC requiring that 0 be a positive integer. However, using Fuzz with a Z specification of this function no error would be detected. The reason for this is that positive integers are not a basic type in Z, they are simply a subset of all integers, and the decrement function is a legitimate function on the set of all integers.

Since many functions within the DTOS FTLS are defined to have a domain or range which is a subset of the basic type, this is a serious weakness in the tools. Without a doubt the DTOS FTLS still contains errors which could be detected by PVS but went undetected by Fuzz.

7.3.5 Complications of Schema Calculus

The Z schema calculus is one of the most powerful features of the language. It allows individual transitions to be combined in what can be very complicated expressions. Unfortunately, our experience with using some of the operators of the schema calculus is that they are not always intuitive, and that the very power of the operators can make them very dangerous in that the resulting transition may be very different than expected.

Discovery of these problems during the specification to code analysis (see Section 8.4.1 for more discussion) led to new guidelines disallowing the use of the more complicated operators of the schema calculus except in controlled situations. This particularly affected the revision of the execution model and encouraged the use of small transitions in the model.

[20] Though the way in which this is done in each language is logically quite different.

7.3.6 Determinism and Z

The Z specification language lacks some familiar tools to aid in the specification of deterministic systems. In particular, there is no way to specify in a transition that the final state of the transition is the same as the initial state of the transition with particular values changed. Instead, every element of the final state much be explicitly defined, even if most of them do not change.

This can become quite awkward in specifications, and led to the habit of only specifying those state elements which do change. The formal interpretation of such specifications is however nondeterministic in any state element that is not specified, and therefore analysis of the specification could have been jeopardized.

Later in the program a general technique for getting around this problem was developed. It was not actually incorporated into any portion of the FTLS since a higher priority was placed on the revisions to the execution model.

Section 8
Specification to Code Correspondence

A specification to code correspondence analysis involves a rigorous examination of an implemented system to ensure that it is a faithful implementation of some specification of the system. Without such an analysis there is little real connection between the analysis performed upon the model and the system that is actually implemented. Ideally, the analysis should not only verify that the system correctly implements the specification but should also provide evidence that convinces others of this correctness. Theoretically, the correctness could be checked by locking a single person in a room for an extended period of time and asking that person to "analyze" the system. Such an exercise might discover several flaws in the system and result in a number of fixes that make the system better. However, to an outside observer, there is no evidence that an adequate analysis of the system was done, or that the fixes did not create some other problems. Documentation of the process followed and the results found is critical to convincing others that the system is sound. Thus, the analysis should not only be complete and thorough but also believable, verifiable, repeatable and maintainable, all at an acceptable cost. Achieving all of these goals simultaneously is probably beyond the current state of the art. The Specification to Code Correspondence task on DTOS has given priority to the documentation of the correspondence. As a result only a very small part of the complete analysis has been performed.

The DTOS Specification to Code Correspondence report has two primary sections: the *data correspondence* and the *functional correspondence*. The former correlates components of the formal state model in the FTLS with the constants, data types, and structures used in the code. The correlation is documented by a collection of tables that relate these concepts.

The second section includes a partial analysis of two DTOS requests: **thread_get_assignment** and **mach_thread_self**. This analysis is performed in three steps:

1. A module level description (MLD) is written. An MLD is essentially pseudo-code describing the possible paths that a particular request might take through the system modules.[21] It describes the changes made to the state, the conditions controlling those changes, the information returned by the request, and the communication between modules. Sections of code that modify parts of the system state that are not modeled can generally be ignored. However, a condition based upon unmodeled data structures that controls changes to modeled data structures probably indicates a discrepancy and should be noted. In writing an MLD we extract portions of the code that are relevant to the transition and present them together in one place. This requires that function calls be "expanded" and most local variables be replaced by the state components for which they stand.

2. The FTLS is analyzed to determine exactly what the FTLS claims as the properties of the transition. This must be done since the relevant properties are scattered throughout the FTLS in various Z schemas that define different aspects of the transition. The result of this step is one schema that describes the transition in its entirety. We will refer to this process as *FTLS expansion*.

[21] The term system module has not been precisely defined in our work. We typically consider it to be a component of the system that is isolated in some way from other components and communicates with them via a well-defined interface. Under this interpretation, for the DTOS implementation analysis there is only *one* system module to consider — the microkernel — since no other DTOS modules have been modeled.

3. The MLD and expanded FTLS are compared with the goal of showing that every property asserted by the FTLS is in fact satisfied by the processing described in the MLD. To perform this proof we must consider each property that an expanded schema claims for the transition. These properties are first translated into MLD terms by applying the data correspondence. The precondition properties of the transition (i.e., those that constrain the state in which the transition occurs) are used to determine which MLD is applicable and which path is taken through that MLD. The properties that make claims about the new state that results from the transition or about the output of the transition must be demonstrated from the identified MLD. This means that whatever is claimed by the property (as translated into MLD terms) must be supported by the MLD.

The following sections discuss conclusions and lessons drawn from the DTOS Specification to Code Correspondence work. Topics considered include the amount of analysis to perform, the methodology, and things to consider when implementing a system and writing a specification.

8.1 Amount of Analysis to Perform

Our work on this report has confirmed our initial view that a thorough, complete, documented implementation analysis is possible only with significant tool support. In the absence of tools we recommend that a data correspondence still be done. The data correspondence is relatively inexpensive (about 150 hours on DTOS). This correspondence helps assurance engineers and developers speak the same language. It allows for a more precise understanding of how the model relates to the code and exactly what it claims. Furthermore, a data correspondence is an essential preparation for FTLS-based testing.

The data correspondence also can point out problem areas that need to be addressed. One such problem area is the need to clarify the conditions that determine when an entity in the formal model comes into existence (or ceases to exist) in the implementation. For example, when creating a task in DTOS a task structure is created and then a long sequence of operations is performed to initialize the fields of this structure and to relate the structure of other entities (e.g., port structures). Context switching in the kernel means that at any point in this processing, the initialization of the task may be interrupted to allow some other operations to occur. There should be some well-defined point in this processing at which we consider the task to exist. This is the point at which the FTLS and FSPM begin to make claims about the object and the events in which it participates. Similar comments apply to the destruction of a thread.

After we wrote the data correspondence, we wished that it had been written much earlier. Ideally, it should be done as early as possible. If it can be done before the FTLS request specifications are written, this will not only reduce later changes as a result of state inconsistencies but will also help the assurance team to understand the system and the model more precisely while writing the specifications. This will lead to higher quality (and less costly) request specifications in the FTLS. We have found the data correspondence to be very useful in the specification work that has been done since it was written (and also when making improvements to the FSPM). Of course it is not possible to do a data correspondence until the code (or at least a very detailed design including extensive data structure information) is available. If both FTLS and Specification-to-Code work are to be done for an *existing* system, then the data correspondence part of the Specification to Code should be done in parallel with the development of the state model in the FTLS. Of course, there is a potential risk in doing the data correspondence too early — details of the design may change as the code is further developed. This would necessitate changes to the data correspondence. On the other hand, the data correspondence might be very useful in assessing the impact of detailed design changes on the formal model.

In addition to a data correspondence, we recommend that at least one or two well-selected requests be partially analyzed. This analysis should focus on the common processing parts of the model with the goal of catching the inconsistencies that will have the most pervasive effect on the system specification and on the analysis based upon that specification. It may also be advisable to analyze any requests that are particularly security-critical. Using the methodology employed in DTOS, it would be prohibitively expensive to analyze all requests without having some very sophisticated tools. However, the small amount of request analysis suggested here can, at a non-prohibitive cost, identify many of the errors and misconceptions that have the most pervasive effect. If complemented with well-designed FTLS-based testing (or with a less costly implementation analysis methodology) applied to the remainder of the system, this analysis can greatly increase the confidence in the consistency of the specification and code.

8.2 Methodology

8.2.1 MLDs

The chief goals of the DTOS Specification to Code Correspondence methodology were that it be believable, verifiable, repeatable and maintainable. A basic requirement therefore was that the correspondence would be *documented*, not just analyzed. MLDs were a cornerstone of documenting the correspondence since they summarize the relevant aspects of the code being analyzed. However, it has become clear that the hardest step in our methodology is the writing of the MLDs. We have concluded that for certain sections of the code, an MLD should not be written at all. This is particularly true of those sections that involve heavy use of recursion or iteration including those that traverse and modify recursive data structures (e.g., trees). If these sections of code are to be analyzed, then it would be better to use more traditional algorithm analysis techniques. Although this avoids the difficulty of writing an MLD for those sections of code, performing and documenting the algorithm analysis might still be very costly. For this reason other alternatives ought to be considered, and we suggest several.

The first is to refine the specifications to a level that allows direct correlation of the specification and code. It is implicit in this approach that the specification and the implementation modularize[22] the system in much the same way. When a different modularization is used for each, it will probably be necessary to *demodularize* both the code (by writing an MLD) and the FTLS (by expanding it) to obtain descriptions that can be correlated. This approach is really just the use of stepwise refinement of the model down to the code level and is therefore part of a known (but perhaps too rarely used) software development methodology. In the process of refining the original model down to a low enough level that the correlation is trivial, the refinement approach essentially does document the correspondence. However, it is not clear that this approach is any easier or less costly than writing MLDs. It probably involves writing multiple specifications of the system at different levels and showing that the lower levels really are refinements of the higher levels. Alternatively, a single low-level specification could be written, but this would probably be difficult and it would certainly complicate the other analysis performed on the specification. Further consideration should be given to determining the optimal compromise between ease in implementation analysis and ease in other assurance tasks. An important question to ask in making this decision is what sorts of tools can be applied in the various assurance tasks. Code analysis tools might have a significant impact in tasks such as

[22] We are referring here to the decompositions of the program into functions and the specification into schemas (or some analogous logical unit), not to the way in which the system is broken into conceptual modules such as VM and IPC.

writing MLDs, and sophisticated theorem provers might be a great aid in performing functional correctness analysis, covert channel analysis and the refinement arguments mentioned above.

A second alternative is to compare the specification to a detailed design (assuming one is written) rather than the code. If a suitable detailed design document exists, then it will presumably be much easier, and less costly, to do this correspondence. However, the document must contain sufficient information to support all the claims made in the FTLS. This alternative really only documents part of the desired analysis — we would also like to know that the code is faithful to the detailed design. It is also not clear whether it will be any easier to write a detailed design document that is sufficient for the specification to code correspondence than it is to write the MLDs. If a detailed design document is to be written anyway, then it makes sense to try to substitute this for the MLDs.

Next we consider the use of slicing tools. These are tools that can extract from a program a *slice* of code containing all and only the statements that have some affect on the action taken in a particular instruction of the program. It may be that a person could use a slicing tool to identify the code slice that is relevant to a particular property asserted by the FTLS and then analyze that slice to ensure that the property is supported by the code. It is not clear how well the resulting analysis process and output would achieve the goals assumed in our analysis (e.g., repeatability, documentability, etc.). Perhaps the slices could be included in a report to help document the correspondence. Even if it does not achieve these goals, this alternative might dramatically increase the confidence in the system, especially if all the requests can be analyzed. Of course, code slicing tools are not extremely mature at this time, and they have difficulty with some of the traits of programming languages such as C that are popular with system programmers (e.g., pointer arithmetic). This may constrain their use for specification to code correspondence.

The final alternative considered here is essentially the one rejected in the introduction to Section 8: lock a knowledgeable person(s) in a room with the specification and the code and ask them to compare the two. They are not required to document their analysis (except to keep track of the parts of the system they have examined), only to report inconsistencies they find. This approach clearly does not achieve the goals we have outlined. Nevertheless, a competent analyzer is likely to find a good portion of the inconsistencies and this should increase the confidence in the system. The confidence in the analyzer is crucial here, since there is necessarily much more trust involved than there is with a documented correspondence. The cost will also be a fraction of that for a full-blown, documented analysis.

8.2.2 Tools

Tools are essential for a complete specification to code analysis, especially if the analysis is to be documented. The precise set of tools that would be useful depends upon the methodology employed. Some the possibilities include

- tools for manipulating FTLS information (e.g., performing FTLS expansion),
- tools to help in writing MLDs,
- code slicing tools,
- browsers for FTLS, code and MLDs, and
- theorem provers.

8.2.3 Proofs

The proofs that must be performed are, mathematically speaking, usually more tedious than challenging. However, they do require the uniting of information from several sources: MLDs, the data correspondence, and FTLS preconditions. Furthermore, the number of properties that must be proven in a full analysis is immense. This poses an obvious scalability problem that can be dealt with only through automation. Tool functionality that might be useful includes browsing, hypertext, and automatic collection of the related information. Although theorem provers are generally very useful tools, it is difficult to apply them to this particular kind of analysis. Existing theorem provers are not well-suited to tying together information from such diverse sources as MLDs, correspondence tables and formal specifications. Before applying a theorem prover it would be necessary to develop tools that translate these representations into the modeling language used by the prover. This is probably feasible for correspondence tables, but it might be quite difficult for the pseudo-code in MLDs. The use of slicing tools or detailed design documents in place of MLDs would probably make this even more difficult. On the other hand, theorem provers are fairly well-suited to the arguments that would be necessary when refining a high-level specification down to a low level, and mathematical frameworks exist in which this analysis could be couched. This is a strong argument for the refinement alternative in the case where a documented correspondence is required.

8.2.4 Relation to Testing

Static implementation analysis is only one way to gain confidence that the specification agrees with the implementation. Another method is testing, based upon either the FTLS or DTLS. However, testing all paths is in most cases prohibitively expensive, so a smaller test suite must be selected. This decreases the confidence that the specification and implementation are consistent, especially if it is difficult to test for the most important properties of the system. For any given system an overall plan should be developed that indicates what implementation analysis will be performed and what testing will be done. The interaction between these tasks should be carefully considered. It is also likely that tool support would be of great value here. Ideally, a single tool could process the code, deriving interesting test cases and serving as an aid to people analyzing portions of the code by hand.

8.3 Implementation

It is conventional wisdom among those who study programming languages that the characteristics of the language and the way it is used have a profound impact on the ease with which a program can be written, understood, analyzed and maintained. This wisdom certainly applies to the analysis performed to establish a specification to code correspondence. Although a comparative study of code analysis using different programming languages and styles is outside the scope of this contract, it is clear that the choices made with regard to these questions can greatly facilitate or hinder the specification to code correspondence analysis.

In particular, we expect the analysis of a large system to be much easier if the system has a strongly modular design and follows the principles of data abstraction and information hiding. This allows the analysis to be more localized, reducing the need for things such as slicing tools. Furthermore, the system specification could probably follow the same modularization reducing the need for FTLS expansion and MLDs.

As an example, although Mach is conceptually divided into sections such as those dealing with tasks, threads, IPC name spaces, and memory objects, in the implementation these sections are interwoven. Nearly all IPC-based kernel requests touch both IPC name spaces and memory objects (for the message processing if nowhere else), and they do so by calling a variety of functions. The lack of a well-defined, advertised interface between these sections makes it harder to perform specification to code analysis and in particular to write MLDs. An MLD describes the possible paths that a request might take through the kernel. If a kernel is decomposed into components that communicate through well-defined interfaces, then the paths through each component can be analyzed independently with the analyses combined to perform an analysis of the entire system. This allows parts of the analysis to be reused in analyzing other requests and reduces the analysis task to more manageable pieces. With a monolithic kernel, it is harder to reuse pieces of the analysis in this way.

It is much harder to achieve this goal in a programming language such as C that not only fails to support data abstraction but also allows pointer arithmetic and casting. Unfortunately, pointer arithmetic and casting are two of the traits of C for which system programmers have a particular fondness. In the presence of these language features, it is probably the case that modules can be dependably separated only by ensuring separation of address spaces and requiring modules to communicate via a mechanism such as IPC. Even then it will be necessary to analyze each module to guarantee that all the ways in which it can receive requests from other modules are well understood and carefully analyzed. Much of this would come for free (and with less computational overhead) in a programming language that supports strong typing, data abstraction and information hiding.

8.4 Specification

8.4.1 Z

The Z specification language contains a set of powerful operators that comprise the *schema calculus*. Our work has indicated that unless these operators are used in highly constrained ways it can be very difficult to determine what is being asserted by the resulting specification. These operators were the cause of virtually all of the complexity encountered in FTLS expansion. The operators to which we are referring are piping and sequential composition. Z piping and sequential composition can be used to combine two Z schemas, A and B, to obtain a new schema, C. For sequential composition, the new schema represents the "execution" of A and B in sequence. For piping, the new schema represents the parallel "execution" of A and B. (For piping, A and B are assumed to be fairly independent, but this is not enforced in any way by Z.) The relationship of the logical predications in A and B to those in C is often quite complex. Predications made in A or B may appear significantly modified in C, or they may not appear in C at all. Furthermore, C may contain entirely new predications that are in neither A nor B but arise from interactions of the typing system and the constraints stated in A and B. To make matters worse, these operators are not very stable. That is, small changes in A or B such as a minor change in the signature (i.e., the declared components) of A or B can have significant effects on C.

Before analysis of an FTLS transition can be performed, one must determine exactly what properties are claimed for the transition. When piping and sequential composition are used, this can require a significant effort. Of course, this effort must be weighed against any potential benefits of using piping and sequential composition when writing the FTLS. At this point there is little evidence that these operators made it much easier to write the FTLS specification. In fact, they probably made it harder as evidenced by several errors in the use of these operators in the FTLS. These errors may well have been introduced because of the instability of the operators. Although this problem was uncovered during the specification to code correspondence analysis, it applies equally well to any task (e.g., covert channel analysis and JLPs) in which the FTLS is analyzed.

8.4.2 Refinement of the State Model

It was discovered very early in the data correspondence work that some portions of the state were modeled in the FTLS at a level that was much more abstract than the actual implementation. By itself, this is not a significant problem. In fact, it is good to abstract out any low-level details that have no bearing on the analysis required for the system since maintaining those details could make proofs of properties from the specification more difficult. However, from the perspective of establishing a data correspondence this is more troublesome. It makes it difficult to precisely state the correspondence, and this reduces the likelihood of identifying any existing inconsistencies between the state model and the implementation. The imprecision also increases the chance that different people will interpret the correspondence in different ways.

In DTOS we addressed this issue by refining the relevant portions of the state model. This did not imply any change to, nor replacement of, any portion of the model. Rather it provided a partial low-level model of the state, and it related this low-level model to the portions of the high-level model that were refined. In some sense, the refinements could be viewed as a definition or implementation of some of the more abstract high-level concepts in terms of lower-level concepts that could be precisely correlated to state information in the implementation via the normal data correspondence tables. Because the correspondence between the high-level and low-level concepts is precisely expressed in the specification language the correspondence between the low-level concepts and the implementation is precisely expressed in the correspondence tables, the correspondence between the high-level concepts and the implementation is also precisely expressed. Since the high-level model is not changed, we may still base proofs upon the high-level abstractions. Only now, we have a precise definition of what the proven theorem claims about the implementation.

Section 9

Composability Study

The Composability Study report describes a variation of Lamport's TLA specification language[2] and provides a framework for composition of specifications based on the work of Abadi and Lamport[2] and Shankar[88]. Composition is a technique for constructing more complex specifications by building upon simpler specifications. Viewed from the other direction, the composition framework allows the specification and verification of a complex system to be decomposed into the specification and verification of simpler components. Benefits of this approach to assurance are similar to those realized when using a modular approach to software development. In particular, complex reasoning about an overall system can be reduced to simpler reasoning about a collection of components and reusable system components can be defined.

The Composability Study report has two principle parts:

- **Composition Framework** — Provides a general framework for
  - specifying components,
  - reasoning about their properties,
  - composing several components into a single system specification and
  - reasoning about the properties of the system based upon the properties of the original components.
- **Example** — The specification and partial analysis of a system comprised of seven components. This part of the report demonstrates the use of the composition framework.

9.1 Innovations

The Composability Study report describes a PVS framework for specifying and analyzing a system as a composition of components. The advantages of this analysis methodology are that it

- reduces reasoning about a system to reasoning about its components,
- allows reuse of assurance evidence, and
- allows "plug-and-play" assurance analysis.

The report demonstrates the first of these advantages, but we have not yet had an opportunity to demonstrate the second and third. The framework defines a structure for component specifications that includes

- a strong distinction between operations of the component and those of its environment,
- agents (to support security reasoning), and
- fairness conditions.

It also defines an n-ary composition operator that returns a component defined as an interleaving of individual component transitions. It has been shown that under appropriate circumstances (i.e., no component violates the environment assumptions of any other component — this property is called *tolerance*) this composition operation is equivalent to intersection of behavior sets for the components. An example of non-trivial size has been considered to help explore scalability and practicality questions. The approach is scalable as long as the tolerance proof obligations can be dealt with effectively. The worked example demonstrates that for systems with a structure similar to that of the example, this can be done.

In terms of writing specifications, the framework seems quite usable. The operations supported by each component can be specified in a "standard" state-machine manner, and the framework can then be used to combine the individual operations into a component specification.

There are two other contributions made by this study:

- A definition of what it means for a composition operation to be intuitively "right". This is important since it recognizes that there are a range of possible definitions for composition, all of which allow the Composition Theorem to be proved, but many of which do not correspond to intuition about what it means to compose systems. While the Composition Theorem places an upper bound on the set of behaviors that the composite can perform, "composition is right" places a lower bound on that set of behaviors.
- The identification and analysis of the "*priv* problem" along with a general solution using *hidd*. The *priv* problem is the need for component-level (or finer) granularity in specifying what state information of each component is protected from other components. In the earliest version of the framework, each component included a specification of the data that it considered private and therefore not modifiable by other components. It was realized when working the example that this is inadequate for dealing with more than two components. The collection of shared data elements can differ for each pair of components drawn from the set of components being composed. At a minimum we needed the ability to specify data privacy on a component-pair basis. The solution adopted in the framework goes further by allowing privacy to be specified on a per-agent basis (a component may contain multiple agents) in the *hidd* field of each component specification. This has resulted in a general framework that can be used in analyzing and comparing other approaches.

Finally, although our framework was inspired by the work of Abadi and Lamport[2] and Shankar[88] we have not merely incorporated their results as axioms of our composition framework. The framework has been built from basic definitions of concepts such as *component* and *behavior* using a mechanical proof checker (PVS). This increases the confidence in the soundness of the framework.

9.2 Future Work

9.2.1 Analysis of System Properties

An obvious disadvantage of using the modular specification approach rather than specifying the example as a monolithic entity is that it is necessary to specify how the individual components interact. In the example, the shared components of the state are used to model the communication protocol between the kernel and the other components. This increases the size of the specifications. The indirectness of component interactions also complicates the analysis of the system. Multiple system transitions (i.e., kernel and security server transitions) occur for each interaction of the non-kernel components of the example. Furthermore, we cannot even talk about the IPC port used by two components to communicate without pulling in the kernel specification to resolve the names used by the components to the ports to which they refer. The information needed in performing proofs is distributed among several components in ways that are not always convenient.

There is of course a trade-off between the accuracy of the model and the amount of effort required to analyze the model. By explicitly modeling the communication between the components, the correspondence between the model and the actual system is more obvious. It is also important to note that the modular specification approach has advantages from a maintenance standpoint. For example, suppose the security server were later replaced by a different security server that satisfied the same properties used in the correctness proof of the overall system. Then, the analysis of the system could be updated by simply reproving the security server properties. It would not be necessary to reprove properties of the kernel.

We believe this problem can be addressed by specifying the "application-level" components at two levels of abstraction and using refinement analysis to show that the lower level is an implementation of the higher level and therefore satisfies all properties satisfied by the higher level. The high-level specifications would incorporate any properties that arise from the execution of the components on a secured kernel which are necessary for proving the desired properties of the system. The kernel and security server would not be specified as high-level components. Thus, at the high level, the applications police themselves and each other. At the low level the kernel and security server do the policing. We hope to explore this approach in future work.

We also note here that it can be difficult in some cases to separate system properties into component properties. Perhaps something could be done with the state information maintained to make this easier.

9.2.2 Proof Obligations

There is a potential problem in the framework with the number of proof obligations for the Composition Theorem when composing a large number of components. For n components there are $O(n^2)$ tolerance proof obligations. However, it appears likely that in practice this will not be a problem. First, for particular architectures, we may be able to reduce the complexity due to the structure of interactions between components. In the example, the non-kernel components interact directly with only the kernel. In this case the number of non-trivial obligations is reduced to $O(n)$. For architectures in which the components are more tightly coupled (e.g., all components share and may modify a given region of memory), this reduction in obligations is not so easily obtained. However, it does not seem unreasonable to require this amount of reasoning about such a tightly coupled system. We are essentially trying to prove that each component manipulates the shared memory according to the conventions agreed upon by all the components. If this is not true, the system probably does not work anyway, and we would like our analysis to uncover this flaw. Even in this case, if all components make the same assumptions regarding the manipulation of the shared memory and they all select from the same operations in manipulating that memory, the analysis could largely be reduced to a comparison of the common assumptions and operations.

We should note here that there may be a tradeoff between the tightness of coupling in component specifications and the level of abstraction as discussed in Section 9.2.1. Omitting the kernel mediation most likely increases the coupling of the other specifications. However, depending upon the assumptions made by the high-level components, this might not pose problems.

9.2.3 Translators

Finally, we note that the use of translators is rather clumsy. When specifying a component we typically use an underlying state type that includes only the information visible to that component. To compose a set of components we must first translate the components into a common state space that includes all the information visible to any component. Translator functions are used to achieve this.

We have found that in almost all cases (in fact, all cases used in the the Composability Study report) the translators are essentially trivial "inverse projection" functions. Nevertheless,

- the translators must be declared,
- we must prove that they satisfy the requirements on translators, and
- a significant number of proof steps deal with the translators.

Furthermore, when a translator is used as a way to specify a component, the properties of a component must also be translated. For the inverse projection translators, the translation is the obvious one. However, if a translator that is not an inverse projection function is used (perhaps by accident) the properties of the translated component might not be what is expected. On another program, we are experimenting with a way to virtually avoid the need for inverse projection translators.

Section 10 Generalized Security Policy Specification

DTOS was designed for a wide variety of uses, both military and civilian. Each of these uses has its own security requirements, and hence DTOS must be capable of supporting a wide range of security policies. The Generalized Security Policy Specification (GSPS) investigates how well that objective is met. It also helps identify ways in which the design might be modified to support additional policies, allowing a decision to be made as to whether these additions are important enough to warrant any increase in overhead to support them. As the document evolved its purpose was further generalized to study the policy flexibility of not only the DTOS microkernel but of all systems employing an architecture in which an object manager enforces a security policy, and a separate security server makes policy decisions. Thus, the GSPS report will hopefully be of interest not only to those familiar with DTOS, but to anyone interested in policy flexibility in an architecture with separation of policy decision and enforcement. The report identifies characteristics of object managers and their interfaces with security servers that limit policy flexibility. This information may be of value to secure operating systems implementors and to those developing policies for secure systems. The sections that may be of the most interest outside the context of DTOS are Section 4 *Security Policy Survey*, Section 6 *Security Policy Lattice*, and Section 11 *Conclusion*. This section of the Lessons Learned Report essentially contains those three sections of the GSPS, augmented with a general description of the document and some selected information from other sections. No particular knowledge of DTOS is required to read the GSPS.

The GSPS report contains several parts. The first contains a survey of security policies (see Section 10.1 in this report) from the computer security literature including dynamic policies that change over time. It then presents a framework that models the interaction between a generic object manager that enforces policy decisions and a generic security server that makes policy decisions. The DTOS microkernel is specified as an instance of the generic object manager, and each of three selected security policies is specified as an instance of the generic security server (see Section 10.3 in this report). For each policy, a composability analysis [81] is performed to demonstrate that the combination of the security server with the DTOS microkernel implements the policy. Any weaknesses in the ability of DTOS to support these policies are identified. Since the framework for specifying object managers and security servers is very general, a wide variety of other manager/security server combinations could be specified and analyzed within the framework.

The portions of the GSPS described in the preceding paragraph demonstrate how to define a security server for various policies and help establish a lower-bound on policy flexibility. To better study the limits of policy flexibility in DTOS, we have developed a lattice of security policies (see Section 10.2 in this report) where each node identifies a set of system characteristics required to support policies at that node. A policy is classified at the lattice node that indicates all the characteristics that are required to support it. An object manager is classified at the node indicating all the characteristics it can support. We have classified the DTOS microkernel and a variety of policies in this lattice. Any policy classified at a node that is not dominated by the node at which the DTOS microkernel is classified probably cannot be supported.[23] A policy

[23] It is difficult to say with certainty that a particular policy *cannot* be supported since there may be many possible

classified at a dominated node can be supported assuming that it only requires characteristics that have been used in constructing the lattice. This provides a slightly more general approach to the determination of the policy flexibility of the DTOS microkernel and other managers.

In the following sections we describe the results of the GSPS. In particular, we have included the policy survey (Section 10.1), the security policy lattice (Section 10.2) and the conclusions of the GSPS (Section 10.4). We also summarize the example policies that were analyzed in the GSPS (Section 10.3). Section 10.5 describes two obstacles encountered in performing the analysis in the GSPS.

10.1 Security Policy Survey

A computing system used in a variety of commercial and military environments must have a security policy general enough to handle the needs of each of those environments. We surveyed many of the policies discussed in the security literature, including those for both military and commercial systems, from which three were selected for use in testing the generality of our proposed security architecture and determining the range of security policies that can be supported. Policies surveyed include those based on lattices of levels, those defined in terms of some user identity (including roles and groups), those defined in terms of execution environments (including Type Enforcement and capability systems), and those that maintain well-formed transactions (integrity models).

A *security policy* is "the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information" [51]. For a computer system, the security policy must define what information is to be protected, the accesses that the various processes in the system are permitted to make to that information, and how these permissions may be modified. The system must also have a protection mechanism that enforces the policy. In the GSPS our primary focus was on *access control policies* which are stated in terms of the accesses that processes may make to the information on the system.[24]

The accesses permitted by a security policy can generally be described using an *access control matrix* with a row for each principal that requests accesses, and a column for each object to which access is controlled [36]. Each entry in the matrix is the set of the accesses that are permitted from a principal to an object. Policies differ in how a process is mapped to a principal, the objects that are protected, the accesses in the matrix, and how the matrix can be modified. Each row in the matrix is a list of (object, permitted access set) pairs and is called an execution *environment* [39]. Similarly, each column is a list of (principal, permitted access set) pairs and is called an *Access Control List* (ACL) [36]. The set of objects depends on the system and may include such things as memory, files, message buffers, and processes.

10.1.1 MLS

A MultiLevel Secure (MLS) policy is defined using a lattice of levels [22]. Level a is said to *dominate*, or be greater than, level b if a is higher than or the same as b in the lattice. In many cases, a level consists of two parts: an element of an ordered sequence of sensitivities (i.e. unclassified, confidential, secret, top secret) and a set of compartments identifying the subject matter of the information; however, a level need not have this structure. Each subject is assigned a level representing its level of trust and each object is assigned a level representing the sensitivity of the information that it contains. In the matrix model, each row represents a

ways to implement the policy on the system. One would need to argue that no implementation can exist.

[24] Section 10.1.9 briefly describes another family of policies, the information flow policies.

level from the lattice; the environment of a subject is the row that represents its level. Each access is classified as a *read* and/or a *write*.

The Bell-LaPadula version of MLS policies [4] allows a process to perform a *read* access only if its level dominates that of the object, and a *write* access only if its level is dominated by that of the object. Thus, *read* is included in those matrix entries for which the level of the row dominates the level of the column, and *write* is included in those matrix entries for which the level of the column dominates that of the row. More restrictive policies can be used for special purposes. For example to achieve non-bypassability of a filter, we could allow a *read* access only if the process and the object have the same level.

With an MLS policy, a process's environment can change if its level changes, if the level of some object changes, if an object is created or destroyed, or if the lattice of levels changes. Some policies prohibit changes to the level of processes and objects; however, level changes can possibly be simulated by creating a duplicate process or object at the new level. Other policies allow the level of an object to reflect the current sensitivity of its contents, or to be modified by the security administrator. For example, in a High-Water-Mark policy a high-level process is allowed to write to an object at a lower level, but the level of the object is then raised to that of the writer.

10.1.2 Integrity

One class of security policies is concerned with data quality, or integrity, rather than controlling its disclosure. Issues related to integrity include: who created or modified the data, what code was used to do so, what is the integrity of the inputs that were used, and how has the data been tested. For example, the integrity of a compiled program depends on the integrity of the source code (how experienced are the programmers, was it formally verified), the integrity of the compiler, and how thoroughly it has been tested. Data from sources known to be reliable have greater integrity than rumors. Integrity policies proposed by Biba [13] and by Clark and Wilson [19] have been especially well studied.

The Biba policy is based on the premise that the integrity of the output from an execution can be no better than the integrity of the inputs and of the code. A Biba integrity policy is equivalent to Bell-LaPadula with high members of the lattice representing poor integrity instead of high sensitivity. Thus, unreliable data with integrity levels high in the lattice cannot be used by a reliable computation with an integrity level low in the lattice. Likewise, an unreliable computation cannot produce reliable data. Related to this is a Low-Water-Mark policy [18] in which the level of an object of lowered when it is written to that of the writer.

Another form of integrity policy was defined by Clark and Wilson [19]. These policies guarantee that a protected object, known as a Constrained Data Item (CDI), can only be modified by a *well-formed transaction*, known as a Transformation Procedure (TP). "The concept of the well-formed transaction is that a user should not manipulate data arbitrarily, but only in constrained ways that preserve or ensure the integrity of the data." [19, p. 186] Each of these TPs corresponds to an access and the set of TPs that may be applied to a particular CDI defines its type. Another goal of some integrity policies is *separation of duty* in which different subparts of an operation are executed by different processes. Separation of duty is equivalent to the n-person policies discussed in [92] with each principal performing a different duty. In Clark-Wilson policies, separation of duty is implemented by allowing each process to invoke only some of the TPs. An example of this is discussed in [50] for the processing of an invoice in a purchasing department. The TPs for this example are:

1. Record the arrival of an invoice.

2. Verify that the goods have been received.

3. Authorize payment.

A 3-person policy would partition the processes into three groups, such as data entry clerks, purchasing officers, and supervisors. Only a data entry clerk could record the arrival, only a purchasing officer could verify receipt, and only a supervisor could authorize payment.

A special form of Clark-Wilson policy is the dynamic assignment of duties to principals [50]. These policies, called dynamic n-person policies, allow a principal to execute at most one of several TPs on a CDI. In the purchasing department example, any of the processes could record the arrival of an invoice. However, if a supervisor did so, it could not then also authorize payment; another supervisor would have to do this.

10.1.3 Type Enforcement

Another form of access control policy is Type Enforcement [15, 52]. Allowed accesses are specified with a Domain Definition Table (DDT), which is a coarse version of the access control matrix in which objects that have equal access privileges are clustered into groups called *types* and the principals that have equal access privileges are clustered into groups called *domains*. Because of the coarseness of specification, type enforcement is generally not used to control access to particular objects, but rather to constrain the flow of information between groups of objects (the types). A typical example of its use is for a guard between a group of sensitive objects and the rest of the system. The only domain in the DDT that permits reading from the sensitive object type and writing to objects of other types is the one in which the guard executes. Thus, information may only move from sensitive objects to the rest of the system by passing through a guard. Changes to the DDT potentially impact on many objects and therefore are usually reserved for a very few trusted domains.

Type Enforcement may also be used to implement role-based access control and least privilege which are discussed in the next section. Furthermore, although much of the work in these areas has been within the context of discretionary control, Type Enforcement can implement them in mandatory control.

10.1.4 IBAC

An Identity-Based Access Control (IBAC) policy [94] defines the allowed accesses to an object according to the identity of the individual making the access. Each row in the matrix represents the accesses allowed to processes operating on behalf of some particular individual, and the row is named by that individual's identity. A column in the matrix (i.e., an ACL) associates with each identity a set of allowed accesses for the corresponding object. A process is permitted an access to an object if the requested access is included in the set of allowed accesses associated with the identity of the process by the ACL of the object. All processes that share an identity have the same permissions (they all have the same environment). Changing the access controls can be controlled by placing each ACL in an object (such as a directory) with its own ACL. Another way to control changes is to include the ACL as part of the object and to make changing the ACL one of the controlled accesses.

Variations to the IBAC policies include groups, roles, negative accesses, and delegation [1]. In the simplest case, a group is a set of individuals and accesses are authorized for groups rather than individuals. A process is associated with some collection of the groups that contain its responsible individual, possibly including a group whose only element is that individual, and its set of allowed accesses is the union of the sets of accesses for each of the groups to which it is associated. The accesses allowed to a process can be restricted by not associating it with all of the groups containing the individual. A principal is therefore a set of groups that share a common member (under this definition, each group can be thought of as the principal whose only member is that group). An environment in the matrix for a principal is the union of the environments for each of the principal's member groups. Thus, the set of accesses for a process is the union of the accesses for each of the groups associated with that process. For example, assume that there are groups:

$$\begin{aligned} W &= \{b, c\} \\ X &= \{a, b, c\} \\ Y &= \{a\} \\ Z &= \{a, b\} \end{aligned}$$

Assume that an object's ACL indicates that $\{W\}$ has no permissions, $\{X\}$ has *execute* permission, $\{Y\}$ has *read* permission, and $\{Z\}$ has *write* permission. One process of individual *a* might be mapped to principal $\{X, Y\}$ and therefore has *execute* and *read* permission, while another process of *a* might be further restricted to only *execute* permission by mapping it to principal $\{X\}$. The advantage of using groups to define the IBAC policy is that the policy can be changed just by changing membership in the groups. Thus, *b* might be given *read* permission without changing any ACLs by adding it to group $\{Y\}$. Groups are objects of the system so that these changes can be controlled by the security policy. A more complex case is to allow a group of groups. The allowed accesses for an interior group (one contained in other groups) is the union of the accesses of each group in which it is contained.

When the principal to which a process is mapped contains only some of the groups that contain the individual associated with the process, then the environment of the process might contain only some of the permissions to which the responsible individual is entitled. This preserves the principle of "least privilege" by allowing the process to operate with only those accesses that it needs. Such a principal is referred to as a *role*. Assumption of a role by a process can be controlled by treating roles as system objects with accesses *assume*, *create_process_in*, and *delegate* (see below) [62].

Another variation of IBAC is to allow *negative accesses* to an object by some groups. The set of accesses permitted to a process is the union of the positive accesses for each of its associated groups, minus the union of the negative accesses for those groups. Usually, permissions granted (denied) a group override permissions denied (granted) a containing group (specificity takes precedence over generality).

Delegation is the ability for a process to allow another to act on its behalf. For example, a client on one node of a distributed system that needs to use a service on another node might delegate its permissions for the service to a network server that will make requests on its behalf. Sometimes, the delegated permissions may even be greater than the original ones, as in the case of a virtual memory manager. A client of the memory manager may only have *execute* permission for a file, but the memory manager will need to read the file in order for the client to execute it. This facility is provided using principals '**B for A**', where **B** and **A** are principals. The ability to become this principal is restricted to **B** and requires the permission of **A**. Also, delegated permissions should expire after some time limit, possibly defined when the delegation occurs.

Abadi et al. [1] have created an access control calculus that includes groups, roles, and delegation. It also allows for operations that require permission from multiple principals. The calculus controls commands of the form *A says s*, where **A** is a principal and *s* is a statement, possibly a request for an operation to be performed on an object. The calculus has the following elements:

$A \Rightarrow B$**:** principal **A** is stronger than principal **B** (if **B** is allowed to say s, so is **A**).

$A \wedge B$**:** principal **A** and principal **B** together make a statement (this can be generalized to n principals to represent an n-person policy).

$A \mid B$**:** principal **A** quoting principal **B** (**A** says **B** says s).

10.1.5 ORCON

In an IBAC policy, processes that are able to read information from an object can usually make that information available to other processes at their discretion. This discretionary aspect of an IBAC policy can be eliminated by using an Originator Controlled (ORCON) policy [3]. With an ORCON policy [37, 43], the allowed accesses for an object are used to determine the allowed accesses for any object derived from it (shared memory segments must be treated as objects in this policy). In the simplest version, each process has a Propagated Access Control List (PACL). Whenever a process reads an object, the intersection of the allowed accesses for the object and the process's PACL form a new PACL. The allowed accesses for any object written by a process must be a subset of its PACL. For example, assume that

- a process b reads from two objects m and n,
- only a and b have $read$ permission to m, and
- only b and c have $read$ permission to n.

The only process that can have $read$ permission to any object that b subsequently writes is b itself. Alternatively, a process's PACL or an object's ACL can be replaced by an ACL and a set of pointers to the inherited ACLs, allowing an originating process the ability to change the allowed accesses. The allowed accesses for a process are again computed by finding the intersection of the allowed accesses from the referenced ACLs.

10.1.6 Capabilities

Instead of representing the access matrix as a set of ACLs (columns of the matrix), a set of environments (rows of the matrix) can be used. Each element in an environment is called a *capability* and consists of an object identifier and a set of accesses (referred to as *rights*) [20, 39]. An access to an object is allowed if the process's environment contains a capability with that object and the desired access. Included in the accesses are controls on the use and transfer of the capability. For example, a capability may be 'use-once', which causes it to be removed from the environment when it is used. Also, a capability may be 'no-transfer', which prohibits it from being copied to another environment, or 'no-copy', which allows it to be moved to a new environment only if it is simultaneously removed from the old environment. Some of these controls may be set when an object is created, for example setting a particular capability for an object to be 'no-copy' so that only one subject may hold the capability at a time. Other controls can be set before one process passes a copy to another process.

Capability systems can be enhanced with *rights amplification*, which provides a means of implementing protected subsystems [20, 39]. For example, assume that a file system process provides support for a virtual memory. To get a program file, the file system process requires that both *open* and *read* permissions be held by the client. Various users can be given capabilities for a file with only *read* permission. The users can only make use of these capabilities indirectly by first passing them to the virtual memory manager, which amplifies the capability to add *open* permission (permission to amplify requires possession of a special capability). Rights amplification also provides for "courier processes" that can be given a capability that only has *courier* permission. This permission does not allow the courier process that holds it to make any accesses to the object. However, when the capability is delivered to a process that is allowed to amplify its rights, the recipient can gain the needed permissions.

10.1.7 Chinese Wall

A Chinese Wall policy constrains accesses based on the history of previous accesses [17]. It was motivated by the need to restrict insider information, such as using knowledge about one firm's activities in the analysis of a competitor's best course of action. It can also be used for cases in which an aggregate of information is more sensitive than individual pieces [46]. The objects of interest are partitioned into data sets and the data sets are partitioned into conflict of interest classes. For example, all objects containing information about a single firm would form a data set. All data sets associated with petroleum companies would form a conflict of interest class, and all data sets associated with computer firms would form another (a conglomerate spanning several industries would presumably belong to multiple conflict of interest classes). When a process accesses an object, it (and any other process with which it cooperates) loses access to any objects in the same conflict of interest class but different data set as the accessed object. Thus, once a process reads confidential information about Unisys, it cannot read confidential information about IBM, but it could read additional information about Unisys or information about Texaco. Also, a process may not write to an object after it has read from an object in a different data set. Thus, it cannot copy information about IBM into a Texaco file where it can be read by a process that already has accessed information about Unisys. The policy does permit "trusted" processes that may read information from one data set, sanitize it, and write it into another data set.

10.1.8 Combinations of Policies

The security requirements for many systems require a combination of policies. For example, a military system might require both an MLS and an IBAC policy. A simple combination can be made by taking the intersection of the various policies: an access is permitted by the combined policy if it is permitted by each of the constituent policies. Sometimes, however, the combination is more complex. For example, MLS and type enforcement can be combined such that a *write* is permitted if either it is permitted by both the MLS and type enforcement policies (as in the simple combination), or *trusted write* is permitted by the type enforcement policy (a *trusted write* is only allowed to those processes with the authority to declassify information or to extract nonsensitive from sensitive information) [15, 52]. This is an example of a policy in which permission in one subpolicy overrides lack of permission in another subpolicy.

Another form of combination is to use different policies for different objects. Control of a group of sensitive objects might use a Chinese Wall policy, control of a shared virtual memory might use capabilities, and control of a file system might use an IBAC policy. These policies might interact, such as storing the capabilities for the virtual memory in the file system. Thus, the initial linking between a process and the virtual memory would depend on the identity of the process, but then the accesses would be controlled by capabilities.

10.1.9 Information Flow

The policies given above have been stated in terms of allowed accesses. An alternative is to state them as noninterference requirements [35]. One group of subjects is *noninterfering* with another group if what the first group does has no effect on what the second group can see. For example, an MLS policy can be expressed such that subjects at one security level must be noninterfering with subjects at another level if the second level does not dominate the first level. Note that if having an effect includes more than being able to write to an object that the other group can read, this MLS policy is stronger than the Bell-LaPadula policy in Section 10.1.1 and prohibits certain *covert channels* [51]. The advantage to a noninterference formulation is that the policy can be stated formally and a model of the system can be proved to satisfy the policy. Several versions, differing in what "having an effect on what can be seen" formally means, have been given [30, 44, 45]. There seems to be general agreement, however, that the formalization must be *composable* in that if two systems are both noninterfering, then the larger system formed by composing them also is noninterfering [44].

10.2 Security Policy Lattice

In this section we describe a lattice of security policies. We begin by defining a list of policy characteristics. This set of characteristics defines a lattice of policies. For any interesting policy, we can place the policy at a node in the lattice by determining the set of policy characteristics that it has.

This lattice is useful in analyzing the capabilities of a manager to support security policies. We first determine the set of policy characteristics that are supported by the manager. This places the manager in the lattice at a node N. The manager can support each policy that is located at a node that is dominated by or equal to N. Furthermore, the placement of policies in the lattice will help to identify policy characteristics that are important to support in a manager for which policy flexibility is a design goal. Finally, if m managers are to be analyzed with respect to p policies, the complexity of doing this with the lattice is $m + p$ as opposed to mp for analyzing each manager-policy combination separately.

10.2.1 Security Policy Characteristics

In this section we informally describe the dimensions along which we will classify security policies. They are

Input — Many policies vary on the amount of information used in a single policy decision. For example, Type Enforcement considers only two security contexts, a domain and a type. In contrast, Clark-Wilson (in its pure version discussed below) considers an arbitrary number of security contexts contained in an access triple when making a single policy decision. At the Mach microkernel level, it would probably be useful to control port requests based upon a triple containing the client, the target task, and the target port. We will use C to denote the number of contexts used in making a policy decision and will say that $C = 2$ for policies such as Type Enforcement and $C > 2$ for policies like Clark-Wilson.

In addition to security contexts a policy decision might also take into account other data describing the operation for which a subject is requesting permission. For example, a subject might wish to change its scheduling priority. A policy might associate a range of allowed priority values with each subject and include in its decision-making process an examination of the priority requested by the subject. Since the priority is most likely a parameter of the kernel request submitted by the subject, we will say that a policy is *parametric* if it takes this type of information into account in making its decisions.

We note here that with regard to both the number of contexts and the inclusion of parameters we are considering security server permission requests, not manager requests. This policy characteristic deals with the "language" used to request permissions. The processing of a single manager request may involve multiple permission requests to the security server. The number and content of these requests will certainly be affected by the manager request, including the values of the parameters and the number of entities in the request. This, by itself, does not make the policy $C > 2$ nor parametric. If the manager never supplies more than two contexts in permission requests and never includes any parameter information, then the policy decisions made in the security server are based upon just two contexts.

Sensitivity — While policy decisions are obviously influenced by the input (i.e., contexts and parameters) to the policy, many policies are also sensitive to other information. Such sensitivity can cause a policy to change so that it might make different decisions in two cases that have identical input. This dimension indicates whether a policy is sensitive to other information and, if so, the kind of information.

We will call a policy that is insensitive (i.e., decisions are based solely upon the input information) a *static* policy. Static policies can change only as a result of a specific policy modification request by a *privileged* user. MLS policies are typically static. If a policy is not static, we call it *dynamic*. A Chinese Wall policy is dynamic. Initially, a user can read all files in the system. However, if a user u reads a file f, that user may no longer read any files that are in the conflict of interest class for f except for those that are in the same company data set as f. Thus, as a side effect of reading a file, u has lost read access to other files, and the policy has changed automatically. In terms of sensitivity, the Chinese Wall policy is sensitive to the history of file read accesses.

For a dynamic policy, it is important to know which events cause the policy to change — the events to which it is sensitive. We identify several types of sensitivity. A single policy may have multiple sensitivities.

History — Many policies are sensitive to accesses to files (see the Chinese Wall example above). We generalize this concept and say that any policy that is sensitive to the execution of manager requests is *history-sensitive*.

Environment — Other policies are sensitive to properties such as the time of day (e.g., no write access between 5 PM and 9 AM) or operational mode of the system (e.g., training, normal, emergency). We will call such policies *environment-sensitive*.

Discretionary — A policy such as IBAC changes as a result of an explicit request by a user (privileged or unprivileged) to change the portion of the policy that applies to objects owned by that user. We call such policies *discretionary*.

Relinquishment — In addition to making requests for permissions (e.g., read access to a file) processes also logically *relinquish* the permissions that they hold. For example, when a process closes a file, it is relinquishing its access to that file. True relinquishment implies that it is impossible for the process to reopen the file without making a new permission request. When a process terminates, it relinquishes all the permissions that were granted to it. Note that relinquishment and history-sensitivity are related in that permission is often required to perform a relinquishment action such as deallocating a region or terminating a task. They are different in that the permission(s) relinquished are different from the permission that enables the relinquishment. When a task deallocates a region to which it has read permission, it adds deallocation permission to the history and relinquishes read permission to the region.

If a policy changes in response to relinquishment, we will say it is *relinquishment-sensitive*. For example, if we depended upon the security policy to enforce locking of files (e.g., at most one process using each file at a time), the policy would change whenever a process opened a file. When a process closed a file, the policy should change again to allow access to the file for other processes.[25] Although our discussion has focused on file access, relinquishment can in principle apply to any permission.

The lines between these sensitivity types are somewhat vague and arbitrary. For example, if we defined history-sensitive as "sensitive to any system event" rather than "sensitive to the execution of a manager request", then history-sensitivity would include environment, discretionary and relinquishment sensitivity since changes in the environment, requests to change discretionary policy and relinquishment are all system events. By restricting history-sensitivity to consider only the execution of manager requests we are singling out those policies where the Security Server must observe what the manager is doing. We consider sensitivity to operational mode to be environment sensitivity. However, to implement a change in operational mode might require a manager request, and it probably is initiated by a user and thus has a discretionary flavor. We choose not to consider it discretionary due to its potentially global effect on security decisions, reserving discretionary sensitivity for modifications performed by a user to the parts of the policy dealing with objects owned by the user. We choose not to consider it history sensitive since the primary purpose of a change-of-mode request would be to modify the Security Server's internal data, not that of the manager. Intuitively, it is really a Security Server request rather than a manager request. Relinquishment sensitivity is really just a special case of history sensitivity.

Retraction — Many dynamic policies need the ability to retract permissions that have been previously granted. We call such policies *retractive*. Note that this is a fairly narrow definition of retraction. We only consider retraction of *granted* permissions. If the removed permission is merely grantable and has never actually been granted, this will not make the policy retractive. Consider the Chinese Wall example, and let f_1 be a file in the same conflict of interest class as f such that f and f_1 are in different company data sets. Initially, read access for u to f_1 is grantable but not granted. When u reads f, read access for u to f_1 becomes nongrantable. We know at this point that read access was never granted for u to f_1 since otherwise read access to f could not be granted. The policy does not need to retract a permission that it has never granted, so no retraction is necessary here. Retraction is similar to relinquishment. The difference is that retraction is an action of the policy to forcibly remove permissions from a process while relinquishment is a voluntary action of a process to give up those permissions.

Transitivity — In defining security policies we are often concerned with the ways that multiple subjects can combine to obtain information or affect the system state and output. For example, in an MLS system a subject may write to an object only if the level of the object dominates the level of the subject. Since the dominance relation on levels is transitive, so is the permission to write. That is, if a subject at level A can write to an object at level B, and a subject at level B can write to an object at level C, then a subject at level A can write to an object at level C. An analogous relationship exists for reading. Such policies are called *transitive*. More generally, we will say that in a transitive policy if a subject A

[25] Locking can in most cases be handled by the file server rather than the Security Server. However, if file locking is related in some way to security policy decisions, then the policy will probably play a role in the control of locking.

can modify a data item d_A (e.g., a file or a piece of system state) and if a subject B can detect the modifications made by A to d_A and can itself modify a data item d_B, then A can also modify d_B.

Transitivity is not a desirable property in all circumstances. For example, consider the problem of output labeling in an MLS environment. Suppose every page that is printed must be marked with its classification level. A page labeling program could be used to do this automatically. The normal way to print a document then would be to send it to the page labeler program to be marked and then pipe the result into the print spooler. A user must be able to send information to the labeler and the labeler must be able to send information to the print spooler. It is important, however, that users not be allowed to send information directly to the print spooler. If this happened, unlabeled pages could be printed. Thus, an *intransitive* policy is needed here. It should be noted that few if any policies are purely transitive without any exceptions. For example, an MLS system will typically have trusted subjects that are able to downgrade information to a lower classification level. This downgrader can be used by other subjects to write downgraded information to levels to which they themselves cannot write.[26]

Editorial Note:
Another characteristic to consider is whether entities must have unique SIDs.

10.2.2 Classification of Some Well-Known Policies

Now we will consider a number of well-known security policies and classify them with respect to the dimensions identified in Section 10.2.1. We will also discuss whether the policy can be implemented in the current DTOS prototype. To gain a better understanding of the issues involved, some variants of these security policies that have slightly different characteristics from the originals will also be considered. These variants are called Nonretractive IBAC, Piecemeal Clark-Wilson, Piecemeal Dynamic N-person, Locking ORCON and Static Chinese Wall. The discussion in this section is summarized in Table 2. Section 10.2.3 will summarize the conclusions regarding the ability of DTOS to support the policies in this section and will give a more general discussion of the issues involved.

10.2.2.1 Type Enforcement In Type Enforcement (TE) subjects with identical access privileges are grouped into a *domain* and objects that may be accessed in precisely the same ways by each domain are grouped into a *type*. Access controls then restrict the access of domains to types. The other policies discussed in the GSPS have explicit goals such as preventing downward flow of classified information, ensuring data integrity, and preventing fraud or insider trading. TE, by itself, has no similar goal. Thus, we consider TE to be more a framework through which security policies can be implemented than a policy in its own right. We discuss it here since the DTOS security mechanisms are based upon TE, and any policy that can be achieved through TE can be supported by DTOS.

TE makes decisions based upon the domain and type, so it is a $C = 2$, nonparametric policy. We consider it static, but if mechanisms are provided for changing the domain definition table, TE can be used as a basis for dynamic policies. TE by itself does not require retraction, and it can support intransitive policies.

[26] It is also worth noting that trusted pipelines, such as the page labeler, and downgraders seem to be the main, if not only, places where intransitivity is needed in systems that are otherwise MLS.

Policy	Input		Sensitivity							
	$C > 2$	Parametric	Dynamic/Static	Discretionary	History	Environment	Relinquishment	Retractive	Transitive	DTOS Supports
MLS/BLP			S						$\bullet$	Y
Biba			S						$\bullet$	Y
Type Enforcement			S							Y
IBAC										
Retractive			D	$\bullet$				$\bullet_{rwe}$		N_b
Nonretractive			D	$\bullet$						Y
Clark-Wilson										
Pure	$\bullet$		S							N_a
Piecemeal			D		$\bullet_{we}$					Y
Dynamic N-Person										
Pure	$\bullet$		D		$\bullet_{e}$					N_a
Piecemeal			D		$\bullet_{we}$					Y
ORCON										
Pure			D		$\bullet_{rwe}$			$\bullet_{re}$		N_b
Locking			D		$\bullet_{rwe}$		$\bullet$			N_c
Chinese Wall										
Pure			D		$\bullet_{rw}$					Y
Static	$\bullet$		S							N_a

For history-sensitive policies the entry includes an annotation indicating the actions that cause the policy to change. The history-sensitive policies studied here are sensitive to reading (r), writing (w) and executing (e). For policies that are retractive we indicate the permissions that need to be retracted using the same annotations as for history-sensitivity. The rightmost column in the table indicates the policies supported by DTOS. A "Y" indicates support while an "N" indicates lack of support. A subscript attached to "N" denotes the reason why DTOS does not support the policy. The meaning of the subscripts is

- N_a - Arbitrary number of SIDs in a request,
- N_b - Retraction,
- N_c - Relinquishment.

Table 2: Characteristics of Security Policies

10.2.2.2 IBAC Identity-Based Access Control (IBAC), also known as Discretionary Access Control (DAC), has as its guiding principle the idea that control of access to a file is up to the individual who owns that file. This owner determines the permissions of other users to access the file. The owner may change the permissions at any time. Thus, the policy is sensitive to discretionary controls. It is trivial for a user to set file permissions in a way that makes the policy intransitive. Since an IBAC policy makes most decisions based upon the user associated with a process and the access controls assigned to a file, IBAC is essentially $C = 2$ and nonparametric. There are, however, examples where a request in an IBAC system involves checks on more than two entities and where the specific checks required depend upon parameters of the system request. For example, consider unlinking a UNIX file from a directory with the sticky bit set. The permission logic is:

- the user must have write access to the directory, and
- either
  - the user must own the directory, or
  - the user must own the file being unlinked, or
  - the user must be the superuser.

So, the logic must deal with the contexts of three entities: the process, directory and file. Whether this makes the policy $C > 2$ depends upon how the policy is implemented. If the manager breaks the permission checking up into the individual requests indicated in the bulleted items above and sends each as a separate security server request, then the decisions made in the security server are still based upon a pair of contexts. If on the other hand the manager sends a permission request $unlink\_file(file, dir, process)$ to the security server, then the implementation of the policy is $C > 2$. We anticipate that the most common implementation will be the former one involving multiple permission requests,[27] so we will classify IBAC as $C = 2$. The permission checks required are sufficiently independent that no semantic information is lost by doing so.

As an example of parametric control, consider the UNIX `chmod` command. If the `setgid` bit is set in the new file mode, then the user must be a member of the group that owns the file. As with the example involving unlinking a file, this does not necessarily imply that the policy is parametric. Again, we assume the implementation involving multiple permission requests to be the typical one, and we classify IBAC as nonparametric.

Retraction is slightly complicated. A file owner may remove accesses of other users for a file at any time. If another user is reading a file at the time when read access is removed, the access could be disabled immediately (retractive) or it could remain as long as the file is open (nonretractive). In Table 2 we include both versions labeled as Retractive and Nonretractive, respectively. DTOS supports only the nonretractive version.

10.2.2.3 MLS and Biba MLS and Biba are both defined in terms of a lattice of levels. In MLS each level represents a classification (e.g., unclassified, secret and top secret) with high classifications at the top of the lattice. In Biba each level represents a given amount of integrity with low integrity at the top of the lattice. The same restrictions on reading and writing apply to both lattices. A process executing at level l may only read a file whose level is dominated by l and may only write to a file whose level dominates l.

[27] Actually, we expect that in many implementations directory and file ownership will be managed entirely within the file system and identity as the superuser will be managed entirely within the operating system. Thus, only one permission request will be sent.

Both of these policies are static. Since decisions are based only upon the level of the process (subject) and file (object), $C = 2$ and the policies are not parametric. Both policies are also nonretractive and transitive. DTOS is able to support both policies.

10.2.2.4 Clark-Wilson

10.2.2.4.1 Pure Clark-Wilson The Clark-Wilson policy is defined in terms of access triples of the form $(UserID, TP, (CDI_1, CDI_2, \ldots, CDI_n))$ rather than the more common access pair $(subject, object)$. Although this policy can be implemented through access pairs by requiring subjects to submit a properly constructed sequence of permission requests, it is more consistent with the definition of the policy to think of a subject requesting access to a *set* of CDIs all at once with one security decision made for the entire set. Access to the CDIs here is all or nothing. This type of system interaction is quite different from what typically occurs in a UNIX system where programs request access to a single file at a time. However, it is closely linked to the way in which file access has historically been controlled on many commercial mainframe systems using COBOL. To run a program on such a system, a user must prepare and submit a sequence of instructions in a Job Control Language (JCL). These JCL instructions state the program that is to be run along with the name and desired access mode of each file to be used by the program. Each file is assigned some code (e.g., a unique letter) by the JCL instructions. These codes are used inside the program to refer to all files. No other way to access files is provided to the program. Before the program begins execution, the system tries to obtain access to each of the files requested in the JCL instructions. If access to any file cannot be obtained, the entire execution terminates without performing a single instruction of the COBOL program. We call this interpretation of Clark-Wilson *Pure Clark-Wilson*.

In many respects Pure Clark-Wilson is a very simple policy. It is static in that no decisions on access to a set of CDIs depend upon earlier decisions. A static policy needs no history, and unless it also includes discretionary control it is nonretractive. However, Pure Clark-Wilson assumes a style of interaction between individuals and the system that is at least similar to the JCL example discussed above. It also means that the Security Server must handle requests containing an arbitrary number of SIDs. This is outside the capabilities of DTOS.[28] A Clark-Wilson policy is almost certainly intransitive since reading is unconstrained but each user/TP pair is granted write access to only certain files.

10.2.2.4.2 Piecemeal Clark-Wilson If we wish to apply the Clark-Wilson policy to a system that follows the UNIX paradigm for file access (i.e., one file at a time), the way that we think about Clark-Wilson must change dramatically. We will call this view of Clark-Wilson the *Piecemeal Clark-Wilson* policy.[29]

The first thing we observe is that Piecemeal Clark-Wilson is not a static policy since obtaining access to a particular CDI can affect future access decisions. For example, assume a transformation procedure TP_1 is certified to manipulate the following sets of CDIs:

$$\{\{CDI_1, CDI_3\}, \{CDI_2, CDI_3\}, \{CDI_2, CDI_4\}\}$$

Initially, TP_1 could obtain $Have\_write$ permission to any of $CDI_1, \ldots, CDI_4$. However, if TP_1 is granted $Have\_write$ permission to CDI_2, then the policy should no longer grant $Have\_write$ permission to CDI_1 since there is no set above that contains both CDI_1 and CDI_2.

[28] A complex request with arbitrarily many SIDs can be broken down into a sequence of requests based upon SID pairs (which is what happens with Piecemeal Clark-Wilson). However, it is important for Pure Clark-Wilson that this sequence of requests be recombined to form the access triple and that access be all or nothing.

[29] This is the version of Clark-Wilson that is modeled and analyzed in the GSPS.

Since the policy changes in response to file access we consider it to be history-sensitive with writing and executing as the events that cause policy changes.[30] Retraction is not necessary since actions can only affect future access decisions and never invalidate prior decisions. As with the pure version, a Piecemeal Clark-Wilson policy is almost certainly intransitive. Since DTOS can support nonretractive, history-sensitive policies, Piecemeal Clark-Wilson can be implemented on a DTOS system.

We note here that the pure and piecemeal versions of Clark-Wilson exercise equivalent control over the system. It is only the implementation of the policy that changes. The fact that this has a profound effect on the classification of the two policies suggests that distinctions such as static versus dynamic, while appearing to describe a policy in abstract terms, implicitly incorporate details of how the policy might be implemented. We discuss this further in Section 10.2.4.

10.2.2.5 Dynamic N-Person

A dynamic N-person policy is similar to a Clark-Wilson policy except that the roles played by the users are not fixed as they are in Clark-Wilson. To see why fixed roles can be undesirable, let us consider the following example. In a Clark-Wilson policy we might allow a purchasing clerk to initiate a purchase order and then require that the purchasing supervisor approve the order. In order to require that two people be involved in this transaction we must prevent the supervisors from initiating purchase orders and prevent the clerks from approving them. This enforces separation of duty. One drawback of this static assignment of roles is that it does not allow any flexibility in responding to special situations such as the absence of all purchasing clerks (e.g., due to illness or vacation). Dynamic N-person policies add this flexibility while still maintaining separation of duty. The purchasing supervisors are allowed to initiate a purchase order, but a single supervisor cannot both initiate and approve any given order. Instead, the order must be approved by another supervisor. Thus, a supervisor is allowed to fill more than one role but can fill at most one role with respect to any given order.

To define such a policy, we must have a concept of a *valid sequence* of TP executions. The policy defines the valid sequences and for each step in the sequence the CDIs that may be manipulated by the step. When a TP process is created, it must either establish a new sequence or continue an existing one. The granting of permission to create the TP process would be sensitive to the valid TP sequences. The granting of CDI accesses could then be decided on the basis of the user, TP and perhaps the sequence.

10.2.2.5.1 Pure Dynamic N-Person

As with Clark-Wilson we distinguish two versions of dynamic N-person policy. In the pure version, access for a TP process to the CDIs is all-or-nothing just like it is with Pure Clark-Wilson. This policy has $C > 2$. Unlike Pure Clark-Wilson it is dynamic. This comes not from the CDI access, but from the sensitivity to the position of the TP process in the valid sequence. This policy is nonretractive and will usually be intransitive. It cannot be supported by DTOS due to lack of input flexibility.

10.2.2.5.2 Piecemeal Dynamic N-Person

This version corresponds to Piecemeal Clark-Wilson — CDIs are accessed one at a time, and the access permissions are sensitive to the history of

[30] We consider Clark-Wilson to be primarily concerned with the writing of CDIs and not the reading of them. Input to TPs is unconstrained by the policy, and the TPs themselves are responsible for making sure that they are given correct input. Thus, the reading of a CDI does not cause the policy to change. This focus on writing rather than reading is consistent with the author's experience in a business information systems environment. With the exception of sensitive information (e.g., personnel files) programmers were allowed to execute programs that read virtually any files on the system. However, they were not allowed to write any production data files (the CDIs of the system). This allowed them to debug programs using production data while preventing them from destroying information or committing fraud.

CDI accesses granted to the TP process. It is thus sensitive to the write and execute accesses granted to files. $C = 2$, and the policy is nonretractive and generally intransitive. This policy can be supported by DTOS.

10.2.2.6 ORCON

10.2.2.6.1 Pure ORCON In the ORCON policy if a process p_1 reads a file f_1 the permissions associated with f_1 place a new upper bound on the permissions associated with any file f_2 to which p_1 subsequently writes. If a process p_2 is reading f_2, the change to the permissions for f_2 is propagated to the files to which p_2 is writing. This constitutes a change of policy so ORCON is dynamic. Since the change is in response to file access, ORCON is history-sensitive with sensitivity to reading, writing and executing. In the above example it is possible that p_2 might no longer have read or execute access to f_2 when its ACL is changed as a result of p_1 writing to it. In this case ORCON must be able to retract read and execute permissions. In DTOS, read, write and execute permissions are what we call *migrating* permissions. A migrating permission is one that is retained in the protection bits associated with memory. When one these permission is flushed from the cache, a “flush thread” is spun off to search the page tables flushing the associated protection bits. This thread will eventually remove all the migrated permissions. However, the cache flush request can return before the flush thread finishes its job, and there is currently no way for the security server to determine when the flush thread has completed the operation. Thus, DTOS provides rather poor support for any policy that needs to flush one of these migrating permissions.

Since read permission migrates in the DTOS microkernel, the prototype cannot support the retractions required in ORCON. The analysis of ORCON in the GSPS further suggests that even if we modified DTOS so that the security server could determine when the page table protection bits have been cleared, DTOS would still not provide good support for retractive policies. The security server would also need the ability to abort or restart active kernel requests that make use of the permissions being retracted.

10.2.2.6.2 Locking ORCON We now consider a modified version of ORCON which we call *Locking ORCON*. Retraction only needs to be done in ORCON if the process p_2 currently has read access to the file f_2 when process p_1 writes to f_2. If p_2 does not have read access when the writing occurs (i.e., read access was either never granted or was relinquished), then read access can simply be denied when p_2 requests access at a later time. If we can guarantee that no file may be open for reading and writing by different processes at the same time, then no retraction is necessary. Locking ORCON consists of standard ORCON together with this new restriction that a process must have a file locked (i.e., no other processes have it open for reading) in order to write to the file.

Editorial Note:
We suspect that even Locking ORCON cannot practically be supported by DTOS since the Security Server has no reliable way to find out that the file is no longer locked (even if p_2 terminates). Even worse, if the Security Server itself is enforcing the locking as part of the policy, then once a file is read it would be tied up in perpetuity. The SS needs to be able to respond to relinquishment. This also relates to the issue of how the Security Server recognizes the destruction of a process.

10.2.2.7 Chinese Wall

10.2.2.7.1 Pure Chinese Wall As described in Section 10.2.1 the Chinese Wall policy is history-sensitive. Its sensitivity to reading has already been discussed. It is also extremely sensitive to writing. If a user has write access to any file in an unsanitized data set D, then that user may only read files that are either in D or in the sanitized data set. Thus, granting process p write access to a file makes read access ungrantable for p to most of the files in the system.

This observation casts some question on the claim of Brewer and Nash that a Chinese Wall system may be operated with a number of users that is no more than the largest number of data sets in any conflict of interest class. Since for each company data set there would most likely be at least one user with write permission to that data set, and since that user cannot read any other company data set, the number of users must be at least the number of company data sets. This is most likely a much larger number of required users.

The breach of security that Brewer and Nash are trying to prevent is the following:

1. User-A has access to data on Oil Company-A and Bank-A.
2. User-B has access to data on Oil Company-B and Bank-A.
3. User-A writes information on Oil Company-A to a file in the data set for Bank-A.
4. User-B reads that file and now holds information on two oil companies, in violation of the policy (and United Kingdom laws).

The problem is that, although we can trust a financial analyst not to write any information on Oil Company-A to the data set for Bank-A, we cannot necessarily trust a computer program in this way. The program has no understanding of the data, and furthermore, it could have been subverted by User-B to write information without the knowledge of User-A.

The Chinese Wall policy is nonretractive since each access, when allowed, affects only the permissions that are grantable and ungranted. Note that write access is granted only if no objects have been read in any other unsanitized data set. Because of the extreme constraints on writing, the policy is nearly transitive with only one type of intransitive behavior. Assume A can write to the sanitized data set (and therefore to no other data set), and B can write to an unsanitized data set D. Since every user can read the sanitized data set, B can read files written by A. However, A cannot write to D. This policy can be supported by DTOS.

10.2.2.7.2 Static Chinese Wall In Section 10.2.2.4.2 we modified the static Pure Clark-Wilson policy ($C > 2$) to obtain a dynamic version which we called Piecemeal Clark-Wilson ($C = 2$). The restriction of C to 2 allows us to implement the piecemeal version on DTOS. The fact that this could be done suggests a relationship between history-sensitivity and implementation. To explore this relationship further we now attempt to apply the reverse process to Chinese Wall to obtain a static version with $C > 2$.

Any Chinese Wall system progresses toward a static state. If it reaches a point where every user either has write access to some data set or has read access to one data set from each conflict of interest class, then no more policy changes can occur unless a new user is added or a new conflict of interest class is created. Along the way to this static state the policy changes whenever any user is granted permission to write a file or is granted permission for the first time to read a given data set. Our static version collapses this progression by requiring the administrator to specify at the time when a new user is added to the system all of the data sets to which the user is given read and/or write access.

This policy is static in that the only changes are due to explicit requests from the administration. $C > 2$ since we assume the administrator submits the entire set of allowed accesses all at once.

The policy is nonretractive since it is static. It is intransitive for the same reasons as Pure Chinese Wall. Admittedly, this policy is less convenient for everyone involved, but it does allow the same proof of compliance with United Kingdom law as is allowed by the pure version. Since the pure version can be supported, this variant is only of theoretical interest.

10.2.3 Classification of the DTOS Kernel

In this section we consider DTOS with respect to its ability to support policies with the characteristics we have been discussing. We both collect in one place the earlier conclusions regarding policy support and draw some more general conclusions. In supporting security policies DTOS does have several limitations, and we discuss a number of them in this section. We should point out that these are limitations of the DTOS microkernel and the prototype Security Server, not of the general idea of separating policy decisions from policy enforcement. These limitations could typically be skirted by developing a new manager and Security Server that interacted to enforce a given policy.

The first limitation we discuss is that DTOS makes decisions based upon a *pair* of security identifiers (SIDs), one for the subject (active entity) and another for the object (passive entity). Thus, $C = 2$. While this works well for most policies, there are certain cases where the policy is most naturally thought of in terms of more than two SIDs ($C > 2$). For example, in the Clark-Wilson integrity policy, access is typically defined in terms of access triples of the form $(UserID, TP, (CDI_1, CDI_2, \ldots, CDI_n))$. Since we would most likely consider each CDI_i to be an object with its own unique SID, these access triples really include an arbitrary number of SIDs. Note that one could develop a file server and a Security Server that could send lists of SIDs back and forth, so this is not a general limitation of the architecture. It applies only to the DTOS microkernel and the prototype Security Server. Furthermore, DTOS does not support parametrized policies well. The only input information for a policy decision in addition to the SIDs is the requested permission. This could also be resolved by implementing more extensive detailed communication between the manager and Security Server.

A second limitation relates to the ability of DTOS to retract permissions. DTOS has improved in this regard over the course of its development and maintenance. However, it still does not support retraction entirely. In DTOS, the results of permission requests are stored in a cache to improve performance of the system. To allow for the removal of permissions the cache may be flushed. Furthermore, individual permissions stored in the cache may be marked as non-cachable (i.e., they may be used only once). In certain cases granted permissions may migrate out of the cache and into the microkernel. In early versions of DTOS the migrated permissions were not removed when a cache flush was performed. This placed a very serious limitation on retractive policies. When a page fault occurs in DTOS the cache and Security Server are consulted to calculate the permissions of the task to the page. The result of this computation is stored in the page table so that permissions to the page may be efficiently checked on future page accesses. In early releases, a subsequent cache flush had no immediate effect on the page table. The permissions $Have\_read$, $Have\_write$ and $Have\_execute$ had effectively migrated into the microkernel and were not retractable. This migration problem was solved by modifying the microkernel so that when permissions are flushed from the cache they are also flushed from the page table and are recalculated at the next access to the page. This has been implemented by a flush thread that resets the appropriate page table bits. Since the flush request may return to the Security Server *before* this thread finishes its job, there is a time delay in the retraction. With the current implementation of the flush thread, the Security Server has no way of knowing when the flush thread is done. Thus, retraction is still rather difficult to achieve in DTOS. The situation could be improved by

- having the flush thread send a notification to the Security Server when it is done cleaning the page tables, and
- running the flush thread at a very high priority to ensure that it does its job quickly.

However, even with this approach there is a time delay involved in flushing the cache — the flush request must be sent and processed — and this might make it difficult to implement a strict retraction policy.

Permissions may also migrate to other servers. When a request is made by sending a message to a server S, the access vector of the requesting task to the server's port is included in the message. S may use this access vector in any way it wishes. If it continues to use this vector over a period of time to make service access decisions regarding the client task, then a retraction of permissions will not be reflected in the actions of S. S might also be able to make requests directly to the Security Server, and the same issue of stale, migrated permissions applies in this case. To solve this migration problem servers must be notified when permissions are removed for any subject. The later releases of DTOS do provide for this notification.

DTOS also is limited in its sensitivity. For history-sensitive policies it is important to know what permissions have been used. However, if a Security Server grants a permission, it cannot tell whether the permission is actually used. By use here, we mean that the service controlled by the permission has occurred. For example, the fact that write access for a file was requested and granted does not mean the requesting process actually wrote to the file. This can partially be overcome by having the Security Server assume that any granted permission is in fact used. We call this the *use-of-permission assumption*. When using this assumption in a Security Server, it will typically be necessary to design the server so that it only grants permissions that are actually requested. Otherwise, the server might unnecessarily restrict future policy decisions making the system hard to use. For example, if a task originally requests read access but not write access, it is only given the former even if the latter is also allowed by the policy. If the task later wishes to write, it must make another request, this time for write access. This allows tasks to declare to the Security Server which of their grantable permissions they wish to exploit. We call this Security Server behavior *stinginess*. If applied to all permissions, stinginess could greatly increase the number of interactions between a task and the Security Server and thus slow the system down. To prevent this a DTOS Security Server is allowed to be stingy with some permissions and generous with others. Selective stinginess can in principle be implemented in DTOS with the use of the cache control vector to make denied, stingy permissions non-cachable. This will cause a new query to the Security Server when the denied permission is checked. However, for the migrating permissions the denied permission is checked from the page table rather than the cache and a new security server request is not generated. To fix this, it would be necessary to make the permission checking mechanisms in the page table sensitive to cachability of permissions. A typical policy need only be stingy with some subset of the permissions read, write and execute, but stinginess can in principle apply to any permission.

We must, however, be judicious in applying the use-of-permission assumption. If the sensitivity of the policy implies that the use of a particular permission reduces the set of grantable permissions, then the assumption is safe. If, on the other hand, the use of the permission can cause an ungrantable permission to become grantable, the use-of-permission assumption is dangerous and should be avoided. An implication of this is that DTOS does not support relinquishment well. There is no way for a task to notify the Security Server that it has relinquished a permission. A task would most likely relinquish access by deallocating a region of memory. There is a permission check at the start of this operation, so the Security Server is told when this operation is being attempted. However, the Security Server is never notified that this operation has actually succeeded. The Security Server could make the use-of-permission assumption and adjust the policy to allow access for the file to other processes. However, if the deallocation fails for some reason, the relinquishing task might still be able to access the file. This would violate the policy.

DTOS can support environment-sensitivity through mechanisms such as the **SSI_load_security_policy** interface and the ability to dynamically swap in a security server by setting the security server port (**host_set_special_port**). The current implementation of **SSI_load_security_policy** may not be adequate to support policies that are both history- and environment-sensitive if the history information must survive an environment change either to affect the decisions in the new environment or to be resumed later when the system returns to the original environment. However, this is only a limitation of the prototype Security Server. A different Security Server could reimplement this operation.

The DTOS prototype Security Server could in principle support discretionary policies through the **SSI_load_security_policy** interface, but this would be clumsy and difficult to use. It would be better to implement an additional interface to the Security Server for requesting changes to the discretionary policy.[31] Another option is to not put the discretionary policy in the Security Server at all but to locate it in another server (e.g., the file server).

Finally, DTOS does support intransitive policies since the policy in the Security Server may implement any general relation.

In summary, the current DTOS prototype microkernel has the following properties with respect to support of the policy characteristics:

- $C > 2$ — No support.
- **Parametric** — No support.
- **Discretionary** — Yes, but clumsily.
- **History Sensitivity** — Yes, but only in cases where the use-of-permission assumption is valid.
- **Environment Sensitivity** — Yes.
- **Relinquishment Sensitivity** — No.
- **Retractive** — Not entirely, because of the time delay in flushing the cache and the inability of the security server to determine when migrated permission have been eliminated. (Early releases failed to support retraction of read, write and execute permissions at all.)
- **Transitivity** — Supports both transitive and intransitive policies.

The fact that it is difficult to provide "yes or no" answers on whether DTOS supports these characteristics suggests that it might be worthwhile in further work to attempt a further decomposition of the characteristics. For example, perhaps history-sensitivity could be split into two classes depending on whether the use-of-permission assumption is valid.

10.2.4 History Sensitivity and Implementation Methods

As we have seen the sensitivity of a policy to the history of file accesses depends in part on the style of interaction between users, processes and the Security Server. Piecemeal Clark-Wilson is history-sensitive because the Security Server cannot grant access to a CDI for a TP without knowing the other CDIs to which the TP has already been granted access. This cannot be

[31] Discretionary policies typically, in their most natural interpretation, require a 1-1 relationship between objects and security identifiers. While the DTOS microkernel does not prohibit this, it provides virtually no assistance to processes wishing to label objects uniquely. We view this as an inconvenience in using DTOS rather than a total lack of support for discretionary policies.

determined based upon a single policy input with $C = 2$. However, Pure Clark-Wilson receives in a single input (with C arbitrarily large) enough information to make a policy decision without sensitivity to any prior file access.

Working in the reverse direction, we modified the Chinese Wall policy, which is typically viewed as history-sensitive, to obtain a static policy (Static Chinese Wall). In this version, an administrator must certify each individual for a particular non-conflicting group of company data sets when the individual's account is set up. While this static version is clearly less flexible than the original, it can still prevent insider information in much the same way as the Chinese Wall policy.

The difference in both cases is in the amount of information received in a single policy input. We call this *chunking*. It is worth looking at various policies and asking whether there is some level of chunking that would make the policy static. We start with the Pure Dynamic N-Person policy. For this policy the history-sensitivity is not in the CDIs which are accessed in a chunk as with Pure Clark-Wilson. The sensitivity is in the valid sequences of TP executions. The determination of whether an individual i is allowed to perform the next step in a TP sequence depends upon the history of the sequence. That is, which individuals have executed the preceding steps. To remove the sensitivity, the entire sequence must be placed in a single chunk. Thus, an entire sequence would have to be requested (or at least declared in some way) all at once. This would include a specification of who could perform each step. This is clearly harder to use than the standard policy.

The prospects for a static variant of ORCON are much worse. Individual process executions are much more intimately linked than they are in a system with the Pure Clark-Wilson policy. Every read and write operation throughout the life of the system can potentially change the policy. It is therefore likely that we could only use one chunk containing the entire life of the system. This is unworkable.

In summary, history-sensitivity is closely linked to the way in which a policy is implemented. For some policies such as Clark-Wilson the interactions between individual policy decisions are localized enough that a static implementation is feasible. For other policies this seems unlikely.

10.2.5 The Lattice

Figure 3 graphically shows a portion of the lattice of policies defined by the characteristics presented above. The bottom node of the lattice represents those policies that have none of the characteristics and can be supported by a manager that supports none of the characteristics. The nodes immediately dominating the bottom node represent those policies that have exactly one of the characteristics. Each node is labeled with the set of characteristics held by policies classified at that node. Managers can also be placed in the lattice with the following interpretation. A manager is placed at node N if it supports exactly the characteristics indicated by the label of N. We have placed DTOS in the lattice. See Section 10.2.3 for a discussion of its placement. The nodes in the lattice that indicate classes of policies that can be supported by DTOS are shaded.

It should be noted that we have taken *in*transitivity (rather then transitivity) to be a characteristic that might be supported by a manager or required by a policy. Transitivity provides an assumption about the ways in which permissions may be assigned in the system. If a system is designed with transitive policies in mind, then it is likely that this assumption will be used to gain efficiency or make it easier to define the set of allowed permissions. Transitive relations are more restricted than general relations, and this provides more information that can be used by designers. If a system is designed with intransitive policies in mind, no such

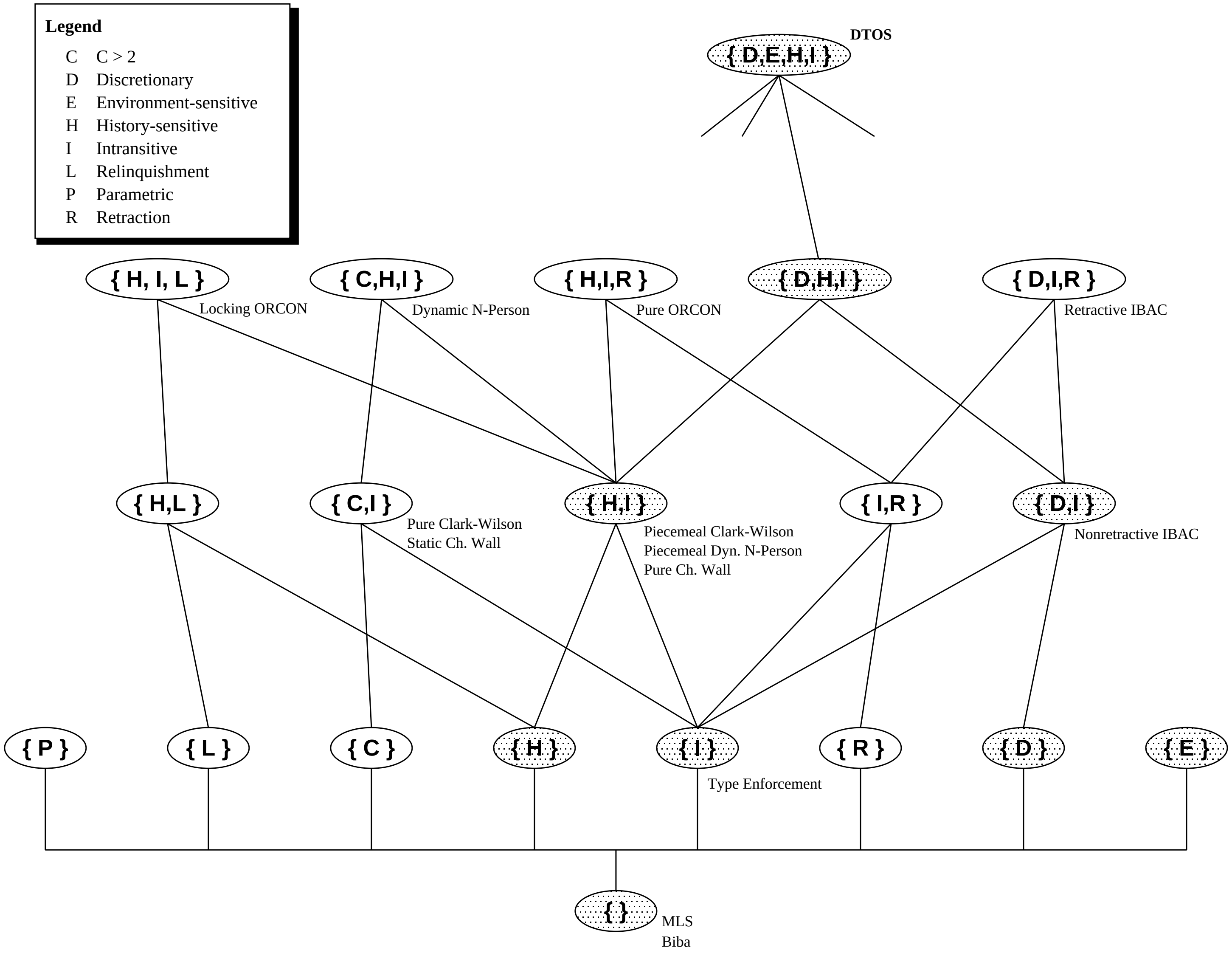

Figure 3: Partial Security Policy Lattice

assumption is made. It also seems unlikely that the design would incorporate an assumption that no potential policy for the system can be transitive. Thus, while any system that supports intransitive policies will most likely also support transitive ones, the converse is not true.

10.3 The Example Policies

Three example policies were considered in the GSPS: MLS with Type Enforcement, Clark-Wilson and ORCON. In this section we summarize these three policies and outline the conclusions of the GSPS regarding the level of support provided by DTOS for each.

10.3.1 MLS/TE

A MultiLevel Secure (MLS) policy is defined using a lattice of levels [22]. The lattice defines a partial ordering on levels called a dominance relation. Each subject is assigned a level representing its level of trust and each object is assigned a level representing the sensitivity of the information that it contains. Each access is classified as a *read* and/or a *write*. The Bell-LaPadula version of MLS [4] allows a subject to perform a *read* access only if its level dominates that of the object, and a *write* access only if its level is dominated by that of the object.

In Type Enforcement [15, 52], allowed accesses are specified with a Domain Definition Table (DDT), which is a coarse version of the access control matrix in which objects that have equal access privileges are clustered into groups called *types* and the environments, that may represent a cluster of individuals, are called *domains*. A typical example of its use is for a guard between a group of sensitive objects and the rest of the system. In this case, the only domain in the DDT that permits reading from the sensitive object type and writing to objects of other types is the one in which the guard executes and no other process can execute in this domain. Thus, information may only move from sensitive objects to the rest of the system by passing through a guard. This movement of information would be in violation of a strict MLS policy, but it is usually necessary in special cases to obtain a workable system.

The MLS/TE policy is a combination of MLS and Type Enforcement. In MLS/TE, all operations must obey Type Enforcement. Furthermore, operations by normal subjects must obey MLS. A specified set of *exceptional* subjects are allowed to violate MLS. Exceptional subjects require close analysis to ensure proper functioning (e.g., to ensure that no classified information is written to an object labeled “Unclassified”). MLS/TE can be thought of as relaxation of MLS, within the bounds of Type Enforcement, that allows for the existence of subjects such as the guard discussed above (a form of downgrader) without unnecessarily complicating the task of analyzing the system. The analysis can focus on the subjects that can violate MLS. Furthermore, the separation provided by Type Enforcement can make it easier to analyze these subjects since their access to system objects may be tightly constrained.

The GSPS contains the specification of a security server to implement MLS/TE for the DTOS kernel. The security server encodes the MLS and Type Enforcement access restrictions in its internal policy database. A non-exceptional subject must satisfy both MLS and Type Enforcement. An exceptional one need satisfy only Type Enforcement. The MLS/TE section of the GSPS includes a proof that, under the assumption of tranquility of security labels in the kernel, the combination of this security server with the kernel implements the MLS/TE policy.

10.3.2 Clark-Wilson

The Clark-Wilson security policy is an integrity policy. As such it is concerned with the correctness of data and the prevention of fraud rather than the prevention of disclosure. The data items that are to be protected are called *constrained data items* (CDIs). The correctness of a CDI is protected in two ways. The first is through *integrity verification procedures* (IVPs). An IVP is intended to verify the integrity of a CDI in one or more of the following senses:

- the internal consistency of a particular CDI,
- the consistency of CDIs with each other, and
- the consistency of CDIs with the external world.

The second way in which CDI correctness is protected is by allowing CDIs to be modified only by certain programs, called *transformation procedures* (TPs), that have been certified to take the set of CDIs from one valid state to another. Validity is defined in some application-specific way.

Prevention of fraud is furthered by providing mechanisms for the separation of duty. Only certain users are allowed to modify a given CDI and then only by using a particular TP. Thus, we can prevent the person who can run the check-writing program from also running the equipment purchasing program. In this way no single person can produce a purchase order, discard it, and then write a check to pay for an item which is never ordered. Fraud then requires at least two people conspiring together.

This policy (in its piecemeal variant) can be handled within the DTOS architecture. The specified security server maintains for each subject security context (which denotes to the security server a particular user executing a particular TP) the set of CDI contexts to which the subject context has been granted write permission. For each new request for write permission from a subject s, the security server determines whether the newly requested CDI write permission together with all those CDI write permissions previously granted to s forms a set consistent with the policy. If so, the new permission is granted. Otherwise, it is denied.

10.3.3 ORCON

Finally, the GSPS considers the implementation of an Originator Controlled (ORCON) [37, 43] policy on a DTOS system. An IBAC security policy usually allows a process that is able to read information from an object, as identified by the object's Access Control List (ACL), to make that information available to other processes at its discretion. This discretionary aspect of an IBAC policy can be eliminated by using an ORCON policy [3]. With an ORCON policy, only those processes allowed to read an object are able to read from any objects that may have been derived from that object.

The root policy is that the originator of a piece of information can specify who may see that information and that everyone who does see the information obeys the originator's wishes. Given the current state of the art, it is much more practical to use this policy with people than it is with computers. The originator of information trusts those to whom access has been granted not to divulge the information to others who are not on the list. However, on a computer system it is processes, not people, which are seeing the information and deciding how they can pass it on. Since these processes are frequently not entirely trusted, it is necessary to assume that when they write to a file they will divulge all information they have previously read. So, any time a process writes to a file, this will likely reduce the set of processes that may read that file. Such a system tends to converge on a state where each process can read only the files to which it can write, resulting in a rather segregated file system. Despite the impracticality of this policy for use on computers, it is still an interesting example to consider in the GSPS because it involves retraction of permissions.

ORCON can be defined in terms of two access control lists. Each object has an associated Access Control List (ACL) as described above. In addition, each process has a Propagated Access Control List (PACL) that lists those processes allowed to receive all of the information that the process possesses. Whenever a process reads an object, the intersection of the processes allowed to read that object and the reader's PACL form a new PACL (a process will always be on its own PACL). Whenever a process writes an object, read access must be removed from the object's ACL for any processes not on the writer's PACL.

In the GSPS, we take the view that the manipulation of ACLs and PACLs as described above is really an implementation rather than a specification of the high-level policy. It belongs in the security server. As stated above, the high-level policy is that the originator of information may define who can see the information. It can be rather difficult to formally state a policy in terms of information.[32] However, rather than retreating to ACLs and PACLs in our policy definition, we state the policy as follows: No system behavior is allowed that contains a finite sequence of alternating read and write actions by which it is possible for data to be transferred from an object j to a process p such that p is not allowed to read j.

DTOS cannot really support this policy. There are two root problems, both of which have to do with permission retraction:

- There is no way for the security server to tell when a request to flush read, write and execute permissions has completely taken effect. These permissions migrate into the page table protection bits in DTOS. Whenever a request to flush the permission cache is being processed, a "flush thread" is dispatched to clear these bits. However, the flush thread does not notify the security server when it is done.
- There is no way for the security server to abort the processing of all requests that are "using" a permission that is to be retracted. This is essentially a check-before-use problem.

Rather than simply concluding in the report that ORCON cannot be supported, we have show that it could be supported on a slightly modified DTOS system. We assumed a system in which a "flush complete" notification is sent and in which no write operations to an object j can occur from the time the permission checking begins for a read request to j until the read request has been completed. The report specifies an algorithm for the security server to follow in responding to permission requests. The security server first determines what the permission response should be and whether any permissions need to be flushed. It also changes its internal policy data to record the effect that the granting of permissions for this request has on future policy decisions. If any permissions need to the flushed, the security server sends the flush request to the kernel. Once the "flush complete" notification has been received the security server sends its reply to the permission request. It can then return to the top of the loop and process another request.[33]

[32] This is probably the reason why one often sees an *implementation* of a policy serving as a *definition* of the policy. This allows the security analyst to avoid the difficulties inherent in an information-based definition. However, any proof that a system complies with the policy is really just a proof that a more detailed implementation (e.g., a decider-enforcer implementation) is a correct implementation of the less detailed implementation. In the case of ORCON, an organization probably does not really care that ACLs and PACLs are manipulated properly. The real concern is that the desires of object originators are enforced.

[33] The security server is allowed to receive, but not process, new permission requests at any point in its processing loop.

10.4 Conclusions of the GSPS

In the GSPS, we used a formal specification language, Z [90], to model a generic framework for systems containing an object manager that enforces policy decisions made by a security server. The DTOS microkernel was specified as an instance of the generic manager and three example security servers were modeled and analyzed. A lattice of policy characteristics was developed to study the implications that the separation of enforcement from decision-making has on the policy flexibility of a system. The key factors limiting policy flexibility under this separation are

- the interface between the object manager and security server,
- the ability to retract permissions, and
- the degree of synchronization required for dynamic policies.

An example of an interface limitation is that a manager will only send selected pieces of information to the security server in a permission request. Any policy that requires additional information may be difficult or impossible to implement. For example, the DTOS microkernel sends a requested permission identifier and a pair of security identifiers in a permission request. Any policy that requires, for example, a requested priority level or the amount of memory to be allocated could not be implemented. Early versions of DTOS were not completely capable of retracting permissions because Mach caches the permissions in the page protection bits. Although this has been resolved in later releases, there are still issues regarding the order in which interactions between the manager and security server are processed and the degree of synchronization required. Great care is required when implementing and analyzing a security server for a retractive policy.

It has been found in this study that the DTOS kernel supports MLS/TE with little difficulty. It only supports the piecemeal version of Clark-Wilson since the kernel will request permissions only with respect to a single pair of SIDs. Of course, this relates to the more fundamental inability of Mach tasks to request access to multiple objects in a single request. DTOS, as is, does not support ORCON, but with some modification it could. We reiterate that the failure of the kernel to support any policy is not a reflection on the general DTOS architecture, but only on the particular DTOS kernel. A modified kernel or a new object manager, together with an appropriate security server, could support Pure Clark-Wilson or ORCON on the objects that it manages.

10.4.1 Open Issues

- The formalization of the various sensitivities is not entirely satisfying. For example, the formalization of history sensitivity is too dependent upon the implementation of a policy rather than the policy itself. (This criticism also applies to the static/dynamic distinction.) Furthermore, the history sensitivity characteristic is probably too broad; a manager might support some types of history sensitivity but not others. It is also rather difficult to formally distinguish the other types of sensitivity from history-sensitivity as defined. This suggests that the identified types of sensitivity are not an ideal set of characteristics for use in the analysis of policies.
- When defining the policy characteristics and placing policies into the policy lattice it was at times necessary to consider how a policy might be implemented rather than the abstract properties of the policy itself. This suggests that it might be more appropriate

to focus entirely on ways of implementing policies rather than on the policies themselves. Further work would be needed to determine if this leads to a better classification scheme.

- The GSPS suggests that the problems of obtaining policy flexibility may be exacerbated by the level of separation between manager and security server present in the DTOS architecture. It is an open question to what degree this is actually true. Answering this question would require performing an analysis similar to the one in the GSPS but with a more tightly coupled manager and security server. Of course there would also be disadvantages to this tighter coupling. For example, if the security server code were actually incorporated into the managers, then we could not change the security server code without at least relinking the managers. This, too, is counter to policy flexibility. The ideal solution most likely lies in solving the problems discussed in the GSPS.

- Another possible security policy characteristic to which we have given some thought is whether each entity must have a unique SID and context. However, there are some subtleties to this question, and we have not included it in our list.

  Consider Clark-Wilson for example. Assume that individual i is authorized to execute TP p on each of the sets of CDIs $\{a, b, c\}$ and $\{d, e\}$, but no other CDI sets. Assume i creates one task t_1 for executing p on a, b and c and then, while t_1 is still executing, asks to create another task t_2 for executing p on d and e. There are two options: consider t_1 and t_2 to be separate Clark-Wilson processes, or consider them to be two tasks operating within the same Clark-Wilson process.[34] If we choose the latter, that means that i may only have one process at a time executing a given TP. It also makes it imperative that the manager notify the security server when a process is terminated so that the TP can be executed again. In this case t_2's requests for access to d and e must be denied since i is not allowed to have a process executing p on $\{a, b, c, d, e\}$. To ensure this, we must give the same SID to t_1 and t_2.

  If we choose to consider the two tasks to be different Clark-Wilson processes, then the second request should be allowed. We also must be careful that t_1 are t_2 are separated in such a way that t_1 cannot access d and e and t_2 cannot access a, b, and c. To achieve this, we will need different SIDs on t_1 and t_2. In fact, every Clark-Wilson process must have a unique SID. This approach is more flexible than the other one since an individual may simultaneously run p on two different sets of CDIs. It also makes it less crucial that the security server be notified of process termination.

  To answer the question of whether unique SIDs are needed we must decide how abstract concepts such as Clark-Wilson processes correspond to low-level, labeled entities such as tasks, threads and ports. The above example suggests that, for Clark-Wilson, it would be good to have a unique SID for each task. We suspect that this will also be true for many other policies.

  The DTOS design, but *not* the kernel, offers as least some support for this 1-1 relationship. A security policy may be defined in such a way that the default version of task creation (i.e., **task_create**) in which the SID is inherited from the parent task is not allowed. Furthermore, **task_create_secure**, in which a SID for the new task is specified as a parameter, is disallowed unless the specified SID is different from that of any other task. The DTOS design contains a security server request **SSI_transition_domain**[35] that may be used to obtain a SID for use in creating a new task. This means that the task submitting the **task_create_secure** need not guess an unused SID but can simply use **SSI_transition_domain** to ask for one. This will typically require modified versions

[34] We assume the former in our specification of Clark-Wilson.
[35] This request was not added to support a 1-1 relationship, but it can be used for this purpose if desired.

of all applications that create tasks through explicit kernel calls. Since the operating system would probably be modified to request distinct SIDs automatically during process creation, applications that create new tasks only through operating system calls need not be modified.

10.5 Obstacles in the GSPS

There were two primary obstacles that were encountered in performing the analysis in the GSPS and writing the report. The first of these is that the scope of the report was limited to analysis of the kernel level. That is, we analyzed the interactions between a secured kernel and a security server. Since all of the policies analyzed are typically viewed at the file system and operating system level, it was necessary to interpret them at the kernel level. This was further complicated by the second obstacle, the complexity of Mach (and hence the DTOS kernel). We are referring here to the large number of requests in Mach, the corresponding large number of permissions controlling the DTOS kernel, and the occasionally subtle behavior of Mach requests. This complexity means that a large amount of work must be performed to verify that a policy is satisfied. In the GSPS we essentially ignored all but a handful of the defined permissions, focusing on those that deal with reading, writing and executing memory and creating tasks. Whether these are the correct permissions to consider in enforcing a file system policy at the kernel level probably depends upon the way in which the file system and operating system are implemented. In all likelihood, IPC would also need to be constrained by the policy. However, a file-oriented policy would typically be enforced not by the kernel, but by the file system, so it is not clear how appropriate it would be to perform a more extensive analysis at the kernel level.

Section 11 Covert Channel Analysis

The DTOS covert channel analysis task studied techniques for performing an assurance analysis of an information flow policy, and the applicability of those techniques to DTOS in particular.

"Covert channel analysis" is perhaps an unfortunate term to describe this task, since it can project an image of low-level implementation analysis of a system of use only in sensitive military installations, while the main focus of the task was high-level design analysis of general information flow policies, especially policies other than multilevel security. It is perhaps surprising that multilevel security is the easiest kind of information flow policy to analyze.

Information flow policies are also important outside of military environments. For example, unbypassability of some module as data passes between two subjects is a requirement about information flow. Unbypassability cannot be expressed as an example of multilevel security (see Section 11.2), but it is a very common requirement in both military and commercial applications. Unbypassability of a filter or proxy is one of the fundamental requirements of internet firewalls.

The covert channel analysis task was heavily affected by the evolving goals of DTOS assurance (see Section 5.2). Initially, the goal for the task was to write a plan for a complete covert channel analysis of the DTOS prototype and to perform a design level search for covert channels based upon the DTOS FTLS [74]. As the goal of providing assurance evidence for the DTOS prototype diminished in importance, more opportunities arose for extending the understanding of information flow policies.

The results of the task include the following:

- A high-level plan for a complete covert channel analysis for the DTOS prototype or any other single operating system component. This is discussed further in Section 11.1.
- A study of a general framework for analyzing a single system component against a general information flow policy, including policies other than multilevel security. This is discussed further in Section 11.2.
- A discussion of some of the basic issues involved in performing an information flow analysis of a multiserver operating system. This is discussed further in Section 11.3.
- Some simple examples of analysis performed on the DTOS prototype. These are presented in the Journal Level Proofs [69] and Covert Channel Analysis Report [67], and are not discussed any further here.

Suggestions for future work building upon these results are provided in Section 13.5.

11.1 Covert Channel Analysis Plan

The first effort undertaken as part of the DTOS covert channel analysis was to write a plan for a complete analysis of the DTOS prototype (or a similar system component) against a general information flow policy. The complete plan is presented in Section 4 of the Covert Channel

Analysis Plan [82]. An outline of the plan is shown in Figure 4, which is a particular instance of the general outline for an assurance analysis depicted in Figure 2. Accompanying each arrow is the title a task described in the plan. Each of these tasks provides justification that one layer of specification or implementation meets the information flow requirements.

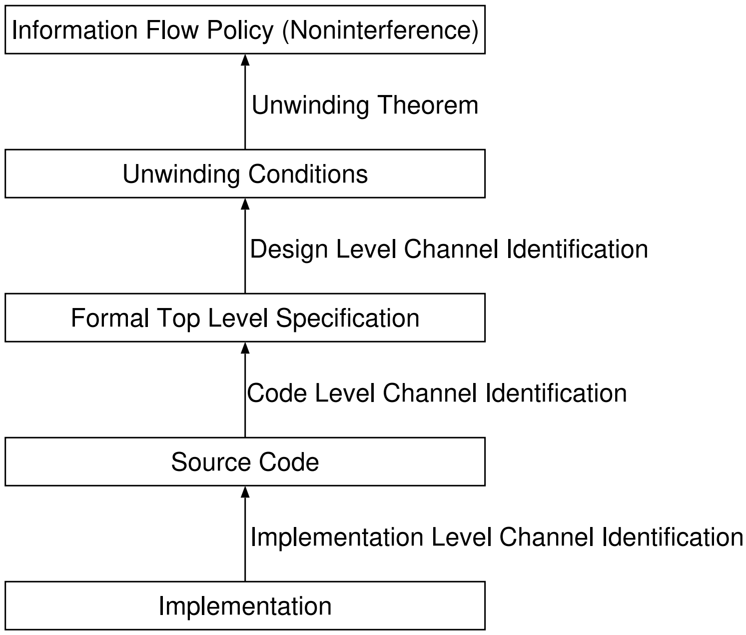

Figure 4: Layers in a Covert Channel Analysis

Because information flow policies can be very difficult to satisfy precisely, the plan also includes tasks to analyze those violations (i.e., covert channels) which are found. This analysis is directed at determining the severity of each channel (e.g., bandwidth, ease of exploitation) and possibilities for eliminating or mitigating the channels.

Because the focus of the remainder of the covert channel analysis task is on the upper levels of Figure 4, in particular the top three levels and the tasks connecting them, these levels are emphasized in the plan. The highest level policy statement is stated as a noninterference policy. Noninterference provides a formal definition of information flow between subjects through arbitrary sequences of operations.

An unwinding theorem decomposes a statement about sequences of operations into unwinding conditions, which make statements about individual operations. The identification of appropriate unwinding conditions is very important because it takes a generally intractable problem of analyzing a system through arbitrary sequences of operations and transforms it into an easier problem of analyzing individual operations.

The design level channel identification compares the definitions of system operations in the FTLS to the unwinding conditions in an attempt to verify that the conditions hold. If the unwinding conditions hold for every operation, then the design is free of information flows that violate the policy. If the conditions cannot be satisfied, then a potential covert channel will be identified.

11.2 High Level Analysis of Information Flow

Multilevel security policies have a transitivity property which simplifies analysis of information flow. This property states that if information flow is allowed from subject A to subject B, and from subject B to subject C, then information flow is allowed directly from subject A to subject C. For instance, information flow is allowed from Unclassified to Secret and from Secret to Top Secret, or directly from Unclassified to Top Secret.

However, this property does not hold in general. In fact, the purpose of unbypassability is to force information flow between subjects (such as subject A and subject C) to pass through some interim subject (such as subject B). Therefore intransitive policies also should be considered, especially for a system like DTOS which is intended to support a range of security policies.

In [58], John Rushby presents a high-level framework for general information flow policies. This framework was the first formal definition of information flow which is applicable to intransitive information flow policies and which includes proofs for the sufficiency of the unwinding conditions. It was therefore the starting point for our investigations.

This framework presents formal mathematical definitions and proofs for the top layers of Figure 4:

- A formulation of noninterference for general (transitive or intransitive) information flow policies.
- Three unwinding conditions which place requirements on individual operations provided by a system.
- A proof that if a system satisfies the three unwinding conditions, then it satisfies the formulation of noninterference, and is therefore secure according to the information flow policy.

Our original goal was to expand upon the theoretical framework by developing a methodology for design level channel identification (see Figure 4) and demonstrating the applicability of this algorithm to the DTOS prototype in particular. The ideal methodology would be representable as an algorithm satisfying four requirements:

Complete If the unwinding conditions can be proven to hold, then there are no covert channels in the design as captured by the top level specification. However, it is unrealistic to expect any complex design to be without channels. Therefore, when a proof fails, the algorithm must provide a way to identify all of the potential covert channels which could be causing the proof to fail. It is not acceptable if a proof fails due to multiple channels but only one is identified.

Maintainable As the system evolves, the analysis must evolve with it. A small change in the system should require only a correspondingly small addition to the analysis.

Policy Independent To avoid repeating analysis when a system is used with different security policies, the algorithm should minimize dependencies upon the specific policy being enforced by the system.

Amenable to Automation It should be possible to automate execution of the algorithm. Analysis of every system operation against the unwinding conditions can be a time-consuming, tedious effort when performed by hand, and is therefore prone to errors.

Only the completeness requirement is absolute. An algorithm which fails to identify all covert channels is unacceptable. Among the other requirements, tradeoffs are possible and probably necessary. For example, if an algorithm can be automated, then the desire to reuse the results of earlier analysis is not as strong.

The starting point for developing such an algorithm was [26], which presents an algorithm meeting these requirements (other than policy flexibility) for noninterference analysis of a multilevel security policy. For DTOS, this algorithm was adapted to work with the general noninterference framework (see Section 5.2 of [82]). The resulting algorithm satisfies the four requirements above with one potentially significant weakness; one of the steps in the algorithm presents a choice of two actions. Even when doing the analysis manually, it is unclear what choice should be made in some situations, so automation of that portion of the algorithm is not practical.

This weakness in the algorithm, and other difficulties encountered when attempting to apply the algorithm and the theoretical noninterference framework to the DTOS prototype, resulted in the devotion of most of the remaining effort to consideration of a number of issues with the framework and algorithm. Discussion of the most significant of these issues follows.

11.2.1 Issues Arising from Weak Step Consistency

The unwinding conditions require the definition of the portion of the system state which is *visible* to each subject. Intuitively, an element of the state is visible to a subject if the subject can determine the value of the element from outputs from a request (or sequence of requests) made by the subject. The problem with the proposed algorithm arises from the statement of one of the unwinding conditions, Weak Step Consistency, which can be paraphrased as follows:

> If some element of the state is not visible to a subject A and also not visible to another subject B, then no operation performed by subject A can make the element visible to subject B.

As an example of an operation which apparently violates this condition, suppose there is a file Foo which is not readable by either subject A or subject B, but subject A is allowed to change the permissions so the subject B can read Foo (perhaps subject A is the DTOS security server). The operation which changes these permissions makes Foo visible to B though it was not visible to A or B in the previous state.

It is possible to interpret the Weak Step Consistency as a prohibition on this operation, so a system with such an operation could not be analyzed at all using the framework. However, we would like to make a less drastic interpretation, which requires placing a less restrictive interpretation on the meaning of "visible". This is certainly allowed by the framework, since visibility is just a mathematical abstraction in the framework and can be interpreted in any way. In the example, this interpretation requires that Foo be defined to be visible to either subject A or subject B prior to the operation of changing permissions.[36]

The weakness in the algorithm mentioned above is caused by this possibility. In particular, for which of the two subjects should Foo be considered visible? Not only is the answer unclear, but the criteria for determining an answer are also unclear. Either answer could be the intuitively "correct" answer in some situation:

[36] Note that if subject A is the same as subject B, then the Weak Step Consistency requirement is always interpreted as identifying that something visible to subject A was incorrectly identified as not visible.

- If subject A is a highly trusted subject (like a security server) which only denies access to Foo to itself out of least privilege concerns, then it seems reasonable to suggest that Foo is really visible to subject A.
- If subject B merely has to request permission to read a file before opening the file, then it seems reasonable to suggest that Foo was always visible to subject B.

We conclude with discussions of three specific topics related to this difficulty with the Weak Step Consistency condition.

11.2.1.1 An Alternative Algorithm

As just described, the proposed algorithm for performing a design level analysis suffers from an inability to automate a decision which may arise from failure to satisfy Weak Step Consistency. If no such operation exists, then the algorithm will still be successful since this decision will never need to be made.

However, it is also possible to remove this problem altogether by revising the algorithm to more closely resemble the algorithm in [26]. Such a change solves the automation problem but also requires introduction of the security policy into the algorithm at an earlier stage, so that the algorithm is less policy independent.

This alternative algorithm is described in Section 6.3 of [82].

11.2.1.2 Equivalence of Unwinding Conditions and Noninterference

In [58], it is shown that if the unwinding conditions are satisfied by a system, then the system satisfies the noninterference statement of its information flow policy. It is also shown that for transitive policies, if noninterference holds, then the unwinding conditions must also be satisfied. It is left open whether the same equivalence holds for general (intransitive) policies.

During our investigations of the problems arising from Weak Step Consistency, we found that this equivalence does not hold. Specifically, it is possible to design a system which meets the definition of noninterference but which fails to satisfy the three unwinding conditions, no matter how "visible" is defined. This system includes an operation similar to the one described above which "violates" Weak Step Consistency. Details of the example can be found in Section 6.1.3 of [82].

The significance of this example to actual analysis of a system is that it demonstrates that an attempt to prove a system is secure may fail due to the fact that the unwinding conditions are too strong rather than due to an actual insecurity. Stated in terms of any proposed algorithm, it means that covert channels may be identified in the analysis though they do not actually exist in the design.

11.2.1.3 Seriousness of the Issues

The seriousness of the problems with Weak Step Consistency is unclear because it is unclear how widespread operations which "violate" Weak Step Consistency actually are. Because most operations appear to be some direct combination of read and write, these would seem to be unusual operations. On the other hand, in a complex system there may be many unusual operations, and the main purpose of performing a formal analysis of a system is to uncover precisely those operations which are unusual or have surprising side-effects.

We have not identified any specific examples of such operations, though neither we have made no significant attempt to do so. The simple example expressed above could certainly occur in a DTOS system. In other systems this may even be more likely if two permissions are required to perform an operation, and the permissions (or capabilities) are retrieved from different sources.

11.2.2 Issues With The Definition of Noninterference

Here we describe three shortcomings of the basic definition of noninterference from [58], and potential solutions.

11.2.2.1 Output Redirection The definition of noninterference is based upon a model of a system which assumes that all output from an operation is returned to the subject which made the request. This model precludes many kinds of operations which occur in real systems, including asynchronous events, broadcasting and even signaling.

In Section 6.1.1 of [82], the system model is extended to allow outputs to be presented to multiple subjects. Noninterference is stated and an unwinding theorem is proven. A fourth unwinding condition must be added, but it is straightforward. Unfortunately, one of the existing unwinding conditions must be changed to a form which more closely resembles the Weak Step Consistency condition. No effort was made to understand the ramifications of this condition on the proposed algorithm.

11.2.2.2 Simplistic Computation Model The definition of noninterference is based upon a model of computation in which only fixed sequences of operations are considered. In particular, operations do not depend upon the output from previous operations. This raises two concerns:

- The framework may fail to identify insecurities in a system which arise by executing real programs.
- The definition of noninterference relies upon "purging" operations from a sequence to construct a shorter sequence of operations. Because operations do not depend upon the output of previous operations, this may result in a completely different meaning for the shorter sequence.

These concerns are elaborated on briefly in Section 6.1.6 of [82]. It is possible that a definition of noninterference based upon more realistic computation model could be shown to be equivalent to the existing definition, but we know of no effort to do this and it was outside of the scope of the DTOS efforts.

11.2.2.3 Unsupported Policies The definition of noninterference provides a precise definition of security for information flow policies with a predefined set of allowed flows between pairs of subjects. Section 6.1.2 of [82] discusses four kinds of policies which are not completely supported by this definition. The two limitations which appear to be most significant are the following:

- The policy must be "predefined". More specifically, the definition of the policy is not allowed to change over the lifetime of a system (i.e., after booting). It may be possible to extend the definition to include the policy as a function of the current system state, but this has not been pursued for DTOS.
- The policy is defined between pairs of subjects. In particular, there is no provision for any statements about objects in the security policy. It may however be valuable to control not only the subjects but the particular objects through which information flows from one subject to another.

An alternative definition of noninterference which begins to address this issue is presented in [8]. The proposed algorithm does not directly support this alternative definition, though it may be possible to support it through a somewhat more complicated algorithm. It is not surprising that the algorithm would need to be more complex since objects have been added to the definition of noninterference.

11.3 Analysis of Multiserver Systems

All of the discussion so far has been focused at analyzing the DTOS prototype kernel, or more generally, any monolithic system. However, the DTOS prototype is intended to be used to support a range of operating system and programming environments through various configurations of user level servers. It is desirable to have a technique for performing covert channel analysis which is localized to each server as much as possible. Ideally, each server could be analyzed once and each use of the server in a particular environment would only require analysis of how the server interacts with other servers in that environment.

Previous efforts at such decomposition of the analysis have focused on the connection of two otherwise stand-alone systems which satisfy a multilevel security policy. Each system is modeled to be communicating with a collection of clients, each at a single security level. If it can be shown that no covert channels exist between clients of a single system in some suitable framework, then it can be shown that no covert channels exist when two such systems are connected.

The shortcoming of this work for our purposes is the way in which the systems are connected. Since each system only has single level clients, communications between from one system to the other must always be considered to be passing through one of these clients. For example, if one system requests that the other system perform some operation, it must label the request as if it came from some single-level client.

This kind of connection is too limiting within a multiserver operating system, which must deal with the following types of communications:

- A server may request a service from another server even though the first server cannot identify that the request is on behalf of any particular client.
- Even if the first server can identify a particular client, that client may not be allowed to make the request being made by the first server.

One potential solution to this is to expand the model of each server to specifically identify the other server as a particular client. Unfortunately, the information flow property which can be proven about the composition of two servers modeled in this way is much weaker than what is needed.

Section 7 of [82] elaborates on these problems and presents some preliminary thoughts for possible solutions.

Section 12 General System Security and Assurability Assessment

The goal of the General System Security and Assurability Assessment Study is to provide guidance for future secure system development by providing design assessment criteria and examples of designs which meet and do not meet those criteria.

The output of this study is a report [83] which presents criteria for assessing microkernel based systems in their ability to be configured to meet a range of security policies, and the feasibility of assuring that a system meets the security requirements of the policies. Five systems are then assessed against these criteria: Amoeba, Spring, KeyKOS, Trusted Mach and the DTOS prototype.

The report is not intended to serve as a rating of the applicability of the systems for any particular purpose. Each of the systems was designed with its own goals which are often quite different than the criteria used for assessment in the study. The purpose of assessing specific systems in the report is to provide real examples of systems which meet or do not meet particular criteria, not to provide any judgment about the overall value of each system.

This section presents a summary of the study, focusing on two topics; an overview of the assessment criteria identified in the report, and a discussion of those criteria which are not met by any of the systems under consideration.

12.1 Summary of Assessment Criteria

The assessment criteria are organized broadly into two categories, security criteria and assurability criteria. We begin by discussing the security criteria.

12.1.1 Security Criteria

The security assessment criteria are all ultimately derived from consideration of the range of security policies which a system is capable of supporting through changes limited to some small set of policy-dependent modules in the system. The highest level criteria describe some common characteristics of security policies, organized into four categories:

General Characteristics These describe the basic kinds of security policies, including mandatory access control policies, discretionary access control policies and information flow policies. Systems which cannot be configured to satisfy these kinds of policies are given less emphasis in the report because many of the other criteria are only relevant with respect to these policy characteristics.

Least Privilege and Granularity Once it has been established that certain policy characteristics are supportable, it is important to determine how much fine tuning is possible in the control mechanisms. The ability to provide an application with the least amount of privileges that are necessary for the application to function can be critical in the overall security of a system. These criteria describe different kinds of least privilege which may be desired.

Dynamic Policies It is typically unrealistic to expect that a security policy will not change while a system is operational. As a simple example, users of a system may come and go and these changes must be reflected in the security policy. More complex changes in policy are possible and often the changes must occur essentially immediately. These criteria identify some of the ways in which a policy may change.

Active Policies Security policies are generally thought of as making statements about what a system must not allow. But it is often just as important for overall security to ensure that some action does occur. These criteria briefly discuss availability of resources, accountability (audit) and intrusion detection.

The remaining security criteria consider the characteristics of common system features, both those specifically providing security and those which provide basic operating system functions. Some categories of criteria in this section include security management, ramifications of failures, capability systems, process integrity and interprocess communication.

While the higher level policy characteristics can be considered to be "universal" across systems, the lower level security criteria are targeted much more specifically at the particular systems considered in the report. If other systems are considered, the criteria would probably need to be expanded. This is especially true of systems which rely upon very different processing or security models, such as SPIN [6].

The security criteria do not consider some categories of criteria which are very important to secure systems, generally because of the lack of documentation (or implementation) of the relevant features in any of the systems studied. These include two-way authentication between the user and a workstation, trusted path, input and output labeling, and secure system boot.

12.1.2 Assurability Criteria

The assurability criteria consider the feasibility of providing assurance that a security policy is met by a system. Like the assurance tasks performed on the DTOS program, these criteria were developed considering formal mathematical methods as the primary form of verification evidence (see Section 2.1.2). It is important to recognize that while these criteria are developed specifically as assurability criteria, any property of a system which makes it easier to assure will typically also increase the likelihood that the system is implemented free of unexpected security flaws.

The assurability criteria are not as well developed as the security criteria, for two reasons:

- It is very difficult to quantify specific requirements for assurability. All that can usually be done is to identify what kinds of properties of a system make it easier or harder to generate assurance evidence. Therefore the assurability criteria tend to be much more relative than the security criteria.

- More knowledge of the details of a system's design and implementation are required to assess assurability than to assess the security criteria. The resources available for the study (both the amount of time available for the study and the amount of information available about each of the systems) were too limited to evaluate the systems to the necessary level of detail. Since most of the assurability criteria would not be used in the assessments, less emphasis was placed on generating the assurability criteria.

Like the security criteria, the assurability criteria are presented first at the highest level and then at lower levels of detail. The high-level criteria consider the complexity of the proofs which must be generated as assurance evidence. Lower level criteria identify how particular aspects of the system's design and implementation affect the complexity of the proofs. Like the security criteria, these lower level criteria are identified based upon the design of the systems being assessed, and may not be directly applicable to all systems.

12.2 Unsatisfied Criteria

One of the results of the study is that there are several criteria that none of the systems satisfy. Since the two basic criteria of policy flexibility and assurability were design goals for the DTOS prototype, it is perhaps surprising that DTOS does not satisfy all of the criteria. There are three reasons for this:

- The details of the criteria were not developed until long after the DTOS design was complete and the initial prototype released. Especially in the area of dynamic security policies, the need for some of the criteria only became apparent through experience gained with the prototype.
- Some of the criteria are dependent upon the underlying kernel, not just the security mechanisms, and the DTOS prototype emphasized the addition of security enhancements without changing the kernel.
- In some cases, such as audit mechanisms, the DTOS prototype was not designed to meet the criteria even though they were known. This was primarily due to limitations on the resources available.

This section concludes with a summary of the criteria which are not satisfied by any system considered in the study. The criteria are organized by the categories in [83], beginning with the high-level security policy characteristics.

General Characteristics The only criteria in this category which is not addressed substantially in any of the individual system assessments is the covert channel criteria for information flow security policies. In part, this is because covert channel analysis requires much more consideration than could be undertaken while assessing the systems. Moreover, information about any covert channel analysis which may have been performed for the systems is unavailable.

Another fundamental reason that covert channels have not been addressed is that covert channel analysis for security policies other than multilevel security is still poorly understood. Even for multilevel security, covert channel analysis in multiserver systems can be difficult. The DTOS Covert Channel Analysis Plan [82] discusses some of the open issues in this area (these issues are also summarized in Section 11).

Least Privilege and Granularity Implementations of mandatory access control often label subjects and objects with security attributes and make security decisions based entirely upon those attributes. This is valuable for simplifying configuration of the policy. However, some security policies require individual labeling of subjects or objects and even in policies which do not require it, individual labeling can be valuable for providing least privilege.

The difficulty with implementing mandatory access control and providing unique labels is that labels are usually assigned through default rules. None of the systems support (at least in a natural implementation) requirements for unique labeling of all entities or of all entities of some type (e.g., all files).

Dynamic Policies DTOS is the only system that provides any significant support for dynamic policies, and DTOS is still limited by the inability to retract permissions which are currently in use. This ability is necessary for any policy which changes in such a manner that previously granted permissions are no longer valid.

The problems with retraction are considerable, though they have not been systematically studied or documented. In the simplest case, suppose a request by a client to perform an operation on some kernel object is currently in progress. This operation could become invalid if either the permissions change in the security policy or if the security attributes of either the client or the object change. It is conceivable that the kernel could be enhanced to identify whenever a change of this type occurs.

There are however cases where it is much more difficult to even conceive of a solution. Consider Mach IPC for example. A security policy is likely to state IPC requirements in terms of the communicating processes. But because of Mach's indirect messaging through a kernel buffered message queue, at any time only one of the processes is involved in an IPC operation. The sending process may have even been destroyed at the time that the message is being received. It could be quite complex to identify that the permissions of the sender have changed, especially considering the multitude of permissions which may be required to send a message.

Potential future work to address this shortcoming is described in Section 13.3.

Active Policies All three services, availability, accountability, and intrusion detection, are insufficiently developed in all of the systems under consideration. Availability is receiving considerable attention, but generally outside of the security community and sometimes with mechanisms that are difficult to incorporate into a secure system. The proper use of security auditing and intrusion detection in a microkernel based operating system is not well understood, especially the proper allocation of responsibility between the microkernel and higher level servers.

Potential future work to address audit mechanisms is described in Section 13.2.

Security Management None of the systems under consideration provide any significant mechanism for managing security policies. This is a shortcoming which must be addressed before policy flexible secure systems (or any secure systems) will be widely accepted and deployed. Ultimately, the mechanisms for managing a security policy are as important as the basic control mechanisms, though they have received considerably less attention.

Potential future work to address this shortcoming is described in Section 13.1.

Process Execution None of the systems under consideration are able to separate the permissions to read and execute data in a process' address space. This decreases the confidence that a process is actually executing the "correct" code. For instance, providing an execute-only access mode (or execute/read) distinct from read or read/write modes can make it significantly harder for a stack overrun attack to succeed.

Assurability The assurability criteria are generally relative rather than absolute, so it is difficult to say that any criteria have not been met. Nonetheless, all of the systems present some significant concerns about assurance, either because of the security mechanisms themselves or simply the complexity of the overall system.

Section 13
Future Work

This section discusses a selection of the most significant issues which must be addressed before a system like DTOS can be widely used and before high assurance can be obtained.

13.1 Security Policy Management Tools

The DTOS prototype effort focused almost exclusively on incorporating security mechanisms into the kernel. With those mechanisms in place, there remains the equally important need for tools to configure and manage the mechanisms. The potentially complex configurations required because of the fine-grained control mechanisms make such tools even more of a necessity.

There are three distinct user groups for security management tools:

Security Server Developers Developers of security servers probably have more use for documentation than for software tools. They require descriptions of the security mechanisms provided by the kernel (not just the control mechanisms), and the requirements which the kernel places on the security server. They require documentation of the "minimal" security policy which must be satisfied by all implementations (for instance, ordinary clients must not be able to gain the privileges reserved for the security server and default pager). They can also be assisted by identification of any relationships among permissions. For instance, there are several different permissions required by a task to act as a user pager.

Basic Policy Configuration At any particular site where a DTOS system is used, there is a need for performing the basic configuration of the policy, for instance, definition of a hierarchy of roles or levels. These are the aspects of the policy which change only rarely but also must be configured with great care and an understanding of the security policy of the overall organization.

There may need to be a selection of security servers available from which to choose based upon the needs of a particular organization. But more importantly, tools must be available for translating the policy of a particular organization, which is generally stated at a fairly high level, into the low-level primitives supported by the security server. This gap between low-level mechanisms and high-level policy statements is probably the most significant part of any tool development.

System Administrators System administrators are less likely to have knowledge of the organization's security policy and less privilege to change the basic configuration. However, within that configuration they must be able to handle day-to-day duties with ease and confidence that the policy is not being violated. For instance, new applications and users will be added or their basic attributes updated.

The way in which these operations are performed will be determined as part of the basic policy configuration. The main need for tools for system administrators is a simple interface so that proper security administration is not such a burden that rules are ignored.

Clearly, a complete suite of tools to meet the needs of all of these users in all environments would take a tremendous effort. However, limiting the scope is both possible and desirable. For instance, the second category of tools could first be considered for use with the DTOS prototype security server implementing MLS and type enforcement. Also, by focusing on the latter two groups of users there is little need for dependencies on a particular underlying system so any successful results could be transferred among systems.

13.2 Audit

Another basic security tool which was not developed as part of the DTOS prototype is audit. Audit provides a final layer of defense if security mechanisms fail or are configured incorrectly. Audit can be relied upon as a replacement for access controls for some purposes, such as allowing operations to be performed even if they can potentially be used to exploit a covert channel. Finally, even in a perfectly secure system audit can be useful for identifying attempts to penetrate the system.

The only audit mechanism incorporated into the DTOS prototype is the simple mechanism for audit of permission checks which is described in Section 3.4.3. Ideally, flexible audit mechanisms would be introduced to the system with an architecture paralleling the security architecture. In particular, auditing could be turned on or off at the various locations in the code based upon the decisions of an audit server. These decisions could be changed as the audit trail is analyzed, for instance, if an attempt to penetrate the system is suspected.

Perhaps the most fundamental question that must be asked when incorporating audit mechanisms into the system is the proper division of responsibilities between the kernel and the external servers in the operating system. There are certain things that only the kernel can audit, for instance, operations of tasks which operate independent of any other servers, or potential attempts to exploit a covert channel through the kernel.

However, audit of kernel operations in general does not provide the kind of information which can easily be interpreted. For instance, the kernel can audit the fact that a particular task sent a message to a particular port. The file server may audit the same operation as a particular Unix process requesting to open a particular file, which is likely to be much more useful information.[37]

The Adaptive Security Policy Experience (ASPE) program (see Section 2.4) has implemented further audit mechanisms in the DTOS prototype. These mechanisms allow audit of messages sent to ports according to rules defined by the tasks holding receive rights to those ports. This mechanism has the potential of satisfying many of the audit requirements of the kernel in addition to providing a service for other servers to rely upon the kernel to provide audit.

Tools for audit analysis are needed before there is any real use for audit mechanisms. In conjunction with the development of such tools, the ASPE additions to DTOS should be evaluated in their ability to provide the necessary information.

13.3 Dynamic Security Policies

The General System Security and Assurability Assessment Report [83] identifies support for dynamic security policies as a significant weaknesses across all systems studied. We know of

[37] Less useful still is the information provided by the current permission audit mechanism: a task with a particular SID has requested to send a message to a port with a particular SID.

no system which provides more support than DTOS for dynamic security policies, even though DTOS support is incomplete.

Efforts to expand support for dynamic policies should have two elements, though they can each be pursued somewhat independently:

- Development of a security server to implement dynamic policies.

  Perhaps the best candidates for dynamic policies to implement are dynamic role based access control (RBAC) policies [25], because these policies are of interest to many people and because RBAC policies integrate very well with type enforcement, which has been implemented in the DTOS prototype security server.

  The process of developing such a security server will clarify the requirements which must be met by the kernel, in particular, the interfaces which must be provided to coordinate policy changes. Testing of the security server will require kernel modifications to provide stub routines to satisfy the interface requirements.

- Changes to the kernel to support dynamic security policies.

  The DTOS Generalized Security Policy Specification [84] and General System Security and Assurability Assessment [83] identify several issues with the ability of the DTOS prototype to support dynamic security policies. The most significant issues concern the kernel's ability to identify permissions which are "in-use" and to revoke those permissions when asked.

  For a permission to be "in-use" means that a permission check has been successfully passed, but the operation controlled by that permission check has not yet completed. It is usually sufficient for the kernel to ensure that the operation will complete before a policy change is acknowledged. However, it is unclear whether this is sufficient for all policies. And worse, the kernel cannot always ensure that an operation will complete in a timely manner, because this may require the cooperation of external tasks. The kernel must have the ability to abort such operations.

  These kinds of fundamental changes to the kernel should probably not be implemented as part of the DTOS prototype. It would be useful to identify the changes needed, but implementation may be more appropriate for a kernel which can provide better support for revocation operations.

There is an equally important need to be able to identify permissions which are in-use within higher level servers. An kernel supporting RPC based messaging may be able to abort certain operations by aborting the RPC. A complete solution does require implementation within the servers as well.

13.4 Integration of Formal Methods with Other Forms of Assurance Evidence

Among the various forms of assurance evidence listed in Section 2.1.2, DTOS focused almost exclusively on the use of formal methods. This focus 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 something does not happen, while functional testing is concerned with demonstrating that something does 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.

13.5 Covert Channel Analysis

The DTOS Covert Channel Analysis raised several issues which should be resolved before pursuing significant efforts attempting to actually analyze a system like DTOS. These issues fall into two categories, which are discussed in separate sections below.

- Analysis of a monolithic system or a single system component with respect to an intransitive information flow policy.
- Modular analysis of a multiserver system against any information flow policy.

13.5.1 Intransitive Information Flow

Section 11.2 describes the efforts undertaken to develop a methodology for performing analysis of general information flow policies with intransitive flow requirements. There were two basic sets of issues raised in that section.

The first set of issues are of interest if the system contains operations which “violate” the Weak Step Consistency condition. Therefore the first task to undertake as part of any future effort is to determine how widespread such operations are, in both high-level and low-level system interfaces. If it is determined that such operations are rare, then the proposed methodology should be sufficient, with such operations considered as special cases. If however they turn out to be more common, then a different methodology should be developed, perhaps based upon the alternative algorithm mentioned in Section 11.2.1.1.

The second set of issues considered the basic definition of noninterference itself. These issues should also be better understood before expending effort on an analysis derived from this definition of noninterference. In particular, the possible limitations on the range of information flow policies which can be placed into this framework should be considered. While it is easy to identify potentially unsupportable policies, it is unclear whether such policies are realistic or how difficult it would really be to expand the framework to incorporate them.

13.5.2 Multiserver Analysis

Section 11.3 discusses briefly the lack of a theoretical (or practical) basis for combining the analysis of individual servers into a claim about the ability of a multiserver system to satisfy an information flow policy. In particular, it is unclear how to state a property on each server so that when composed, the result is the desired information flow property. Any attempt to state such a property must take into account the various ways in which servers can influence the behavior of each other and how the clients of a particular server can influence the requests which that server makes to the other servers in the system.

13.6 System Composition

Section 9.2 describes potential for future work in the study of system composition and modular specifications.

Section 14
Notes

14.1 Acronyms

ACL Access Control List

AID Authentication IDentifier

API Application Program Interface

AVC Access Vector Cache

CAR Corrective Action Request

CCA Covert Channel Analysis

CDI Constrained Data Item

CLI Computational Logic Incorporated

CMU Carnegie Mellon University

DAC Discretionary Access Control

DDT Domain Definition Table

DLL DTOS Lesson Learned

DTMach Distributed Trusted Mach

DTOS Distributed Trusted Operating System

FSPM Formal Security Policy Model

FTLS Formal Top-Level Specification

GSPS Generalized Security Policy Specification

IBAC Identity Based Access Control

IDL Interface Definition Language

IPC InterProcess Communication

KID Kernel Interface Document

MAC Mandatory Access Control

MID Mandatory IDentifier

MIG Mach Interface Generator

MLS MultiLevel Secure

ORCON Originator Controlled

OSF Open Software Foundation

RPC Remote Procedure Call

SDD Software Design Document

SID Security IDentifier

SRS Software Requirements Specification

SSDD System Segment Design Document

SSDTEP Software System Development Test and Evaluation Plan

STR Software Test Report

TCB Trusted Computing Base

TCSEC Trusted Computer System Evaluation Criteria

TE Type Enforcement

TP Transformation Procedure

UDI Unconstrained Data Item

VM Virtual Memory

Appendix A
Bibliography

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] Martín Abadi and Leslie Lamport. Conjoining specifications. Technical Report 118, Digital Equipment Corporation, Systems Research Center, December 1993.

[3] Marshall D. Abrams. Renewed understanding of access control policies. In *Proceedings 16th National Computer Security Conference*, pages 87–96, Baltimore, MD, September 1993.

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

[5] Terry C. Vickers Benzel, E. John Sebes, and Homayoon Tajalli. Identification of subjects and objects in a trusted extensible client server architecture. In *Proceedings of the 1995 National Information Systems Security Conference*, pages 83–99, 1995.

[6] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, December 1995.

[7] Brian N. Bershad, Richard P. Draves, and Alessandro Forin. Using microbenchmarks to evaluate system performance. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 148–153, 1992.

[8] William R. Bevier, Richard M. Cohen, and William D. Young. Connection policies and controlled interference. In *Proceedings of the 8th Computer Security Foundations Workshop*, June 1995.

[9] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Atomic actions and locks. Technical report, Computational Logic, Incorporated, February 1993.

[10] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Entities and relations. Technical report, Computational Logic, Incorporated, February 1993.

[11] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel. Technical report, Computational Logic, Incorporated, August 1994.

[12] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Kernel requests. Technical report, Computational Logic, Incorporated, August 1994.

[13] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre Corp., Bedford, MA, 1977. Also available through Nat'l Technical Information Service, Springfield, Va., Report No. NTIS AD-A039324.

[14] Kirk Joseph Bittler. A policy-independent secure X server. Master's thesis, Portland State University, 1997.

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

[16] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[17] 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.

[18] Maureen Harris Cheheyl, Morrie Gasser, George A. Huff, and Jonathan K. Millen. Verifying security. *ACM Computing Surveys*, 13(3):279–339, September 1981.

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

[20] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the Fifth Symposium on Operating Systems Principles, Operating Systems Review 9,5*, pages 141–160, Austin, TX, November 1975.

[21] Judy Crow, Sam Owre, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available from the WEB page WWW://www.csl.sri.com/sri-csl-fm.html.

[22] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[23] Department of Computer Science, Portland State University. *DX: A Secure Window System User's Manual*, February 1997.

[24] Francois Barbou des Places, Phillippe Bernadat, G. N. Madhusudan, Yves Paindaveine, and Nick Stephen. *OSF Microkernel Performance Test Suite*. OSF Research Institute, April 1996.

[25] David F. Ferraiolo, Janet A. Cugini, and D. Richard Kuhn. Role-Based Access Control (RBAC): Features and motivations. In *Proceedings of the Eleventh Annual Computer Security Applications Conference*, December 1995.

[26] Todd Fine. Constructively using noninterference to analyze systems. In *IEEE Symposium on Security and Privacy*, pages 162–169, May 1990.

[27] Todd Fine. Defining noninterference in the temporal logic of actions. In *IEEE Symposium on Research in Security and Privacy*, pages 12–21, May 1996.

[28] Todd Fine. A framework for composition. In *Proceedings of the Eleventh Annual Conference on Computer Assurance*, pages 199–212, June 1996.

[29] Todd Fine, J. Thomas Haigh, Richard C. O'Brien, and Dana L. Toups. Noninterference and unwinding for LOCK. In *Proceedings of Computer Security Foundations Workshop II*, pages 22–28, June 1989.

[30] Todd Fine, J. Thomas Haigh, Richard C. O'Brien, and Dana L. Toups. Noninterference and unwinding for LOCK. In *Proceedings of Computer Security Foundations Workshop II*, pages 22–28, Franconia, NH, June 1989. IEEE.

[31] David Finkel, Robert E. Kinicki, Aju John, Bradford Nichols, and Somesh Rao. Developing benchmarks to measure the performance of the Mach operating system. In *Proceedings of the USENIX Mach Workshop*, pages 83–100, 1990.

[32] David Finkel, Robert E. Kinicki, Jonas A. Lehmann, and Joseph CaraDonna. Comparisons of distributed operating system performance using the WPI benchamark suite. Technical Report WPI-CS-TR-92-2, Worchester Polytechnic Institute, Worcester, MA 10609, 1992.

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

[34] Bill Frantz. KeyKOS - a secure, high-performance environment for S/370. In *Proceedings of SHARE 70*, pages 465–471. SHARE Inc., Chicago, February 1988.

[35] Joseph A. Goguen and José Meseguer. Security policy and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982.

[36] G. Scott Graham and Peter J. Denning. Protection – principles and practice. In *Proceedings AFIPS 1972 SJCC*, volume 40, pages 417–429. AFIPS Press, 1972.

[37] Richard Graubart. On the need for a third form of access control. In *Proceedings 12th National Computer Security Conference*, pages 147–156, Baltimore, MD, October 1989.

[38] Ian Hayes. *Specification Case Studies*. Prentice Hall International, 1993.

[39] Anita K. Jones and William A. Wulf. Towards the design of secure systems. *Software – Practice and Experience*, 5:321–336, 1975.

[40] Key Logic, Inc. Introduction to KeySAFE. Key Logic Document SEC009.

[41] Keith Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, November 1992.

[42] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, May 1993.

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

[44] Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988.

[45] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 1994.

[46] Catherine Meadows. Extending the Brewer-Nash model to a multilevel context. In *IEEE Symposium on Security and Privacy*, pages 95–102, Oakland, CA, May 1990.

[47] Irwin Meisels and Mark Saaltink. *The Z/EVES Reference Manual*. ORA Canada, April 1996.

[48] Spence Minear, Dick O'Brien, and Lynn Te Winkel. Supporting a Secure DBMS on the DTOS Microkernel. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1996.

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

[50] Michael J. Nash and Keith R. Poland. Some conundrums concerning separation of duty. In *IEEE Symposium on Security and Privacy*, pages 201–207, Oakland, CA, May 1990.

[51] NCSC. Trusted computer systems evaluation criteria. Standard, DOD 5200.28-STD, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, December 1985.

[52] Richard C. O'Brien and Clyde Rogers. Developing applications on LOCK. In *Proceedings 14th National Computer Security Conference*, pages 147–156, Washington, DC, October 1991.

[53] Duane Olawsky, Todd Fine, Edward Schneider, and Ray Spencer. Developing and using a "policy neutral" access control policy. In *New Security Paradigms '96*, September 1996.

[54] Open Software Foundation, Inc. *MK++ Kernel High Level Design*, January 1996.

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

[56] Vikram R. Pesati. *The Design and Implementation of a Multilevel Secure Log Manager*. Department of Computer Science and Engineering, Pennsylvania State University, December 1996. Master's Paper.

[57] Vikram R. Pesati, Thomas F. Keefe, and Shankar Pal. The design and implementation of a multilevel secure log manager. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 55–64, May 1997.

[58] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI International, December 1992.

[59] John Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113. Springer-Verlag, September 1995.

[60] O. Sami Saydjari, S. Jeffrey Turner, D. Elmo Peele, John F. Farrell, Peter A. Loscocco, William Kutz, and Gregory L. Bock. Synergy: A distributed, microkernel-based security architecture. Technical report, INFOSEC Research and Technology, R231, November 1993.

[61] Edward A. Schneider, William Kalsow, Lynn Te Winkel, and Michael Carney. Experimentation with Adaptive Security Policies. ASP CDRL A005, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1996.

[62] Edward A. Schneider, Stanley Perlo, and David Rosenthal. Discretionary access control mechanisms for distributed systems. Technical Report RADC-TR-90-275, Rome Air Development Center, June 1990.

[63] Secure Computing Corporation. Software Requirements Specification for Distributed Trusted Mach. DTMach CDRL A005, v1, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, October 1992.

[64] Secure Computing Corporation. Final Report for the Distributed Trusted Mach Program. Technical Report RL-TR-93-235, Rome Laboratory, Griffiss Air Force Base, NY, December 1993.

[65] Secure Computing Corporation. Formal Top Level Specification for Distributed Trusted Mach. DTMach CDRL A012, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, April 1993.

[66] Secure Computing Corporation. System/Segment Design Document for Distributed Trusted Mach. DTMach CDRL A006, v2, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, March 1993.

[67] Secure Computing Corporation. DTOS Covert Channel Analysis Report. DTOS CDRL A007, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, May 1995.

[68] Secure Computing Corporation. DTOS Demonstration Software Design Document. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, July 1995.

[69] Secure Computing Corporation. DTOS Journal Level Proofs and Conclusions. DTOS CDRL A016, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, May 1995.

[70] Secure Computing Corporation. DTOS Software System Development Test and Evaluation Plan. DTOS CDRL A012, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, April 1995.

[71] Secure Computing Corporation. DTOS Software Test Report. DTOS CDRL A013, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1995.

[72] Secure Computing Corporation. DTOS Formal Security Policy Model. DTOS CDRL A004, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1996.

[73] Secure Computing Corporation. DTOS Formal Security Policy Model (Non-Z Version). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1996.

[74] Secure Computing Corporation. DTOS Formal Top-Level Specification. DTOS CDRL A005, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1996.

[75] Secure Computing Corporation. DTOS FTLS Plan. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, November 1996.

[76] 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.

[77] Secure Computing Corporation. DTOS Notebook of Technical Issues. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1996.

[78] Secure Computing Corporation. DTOS Software Requirements Specification. DTOS CDRL A001, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1996.

[79] Secure Computing Corporation. DTOS Specification to Code Correspondence. DTOS CDRL A018, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1996.

[80] Secure Computing Corporation. DTOS Users Manual. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, October 1996.

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

[82] Secure Computing Corporation. DTOS Covert Channel Analysis Plan. DTOS CDRL A017, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, May 1997.

[83] 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.

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

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

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

[87] Secure Computing Corporation. DTOS Risks and Mitigators Report. DTOS CDRL A015, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1997.

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

[89] J. M. Spivey. *The fuzz Manual*. July 1992.

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

[91] Lynn Te Winkel, Ray Spencer, and Todd Fine. Adding Security to Commercial Microkernel-Based Systems. Technical Report RL-TR-96-75, Rome Laboratory, Griffiss Air Force Base, NY, January 1996.

[92] Phil Terry and Simon Wiseman. A 'new' security policy model. In *IEEE Symposium on Security and Privacy*, pages 215–228, Oakland, CA, May 1989.

[93] Trusted Information Systems, Inc. *Trusted Mach System Architecture*, October 1995.

[94] Thomas Y. C. Woo and Simon S. Lam. Authorization in distributed systems: A new approach. *Journal of Computer Security*, 1994.