Part Number 83-0902028A001RevA
Version Date 26 June 1997

DTOS General System Security and Assurability Assessment Report

CONTRACT NO. MDA904-93-C-4209
CDRL SEQUENCE NO. A011

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	______________________

Secure Computing Corporation

Technical Report

DTOS General System Security and Assurability Assessment Report

Secure Computing Corporation

Abstract

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.

Part Number	83-0902028A001RevA
Created	26 February 1996
Revised	26 June 1997
Done for	Maryland Procurement Office
Distribution	Secure Computing and U.S. Government
CM	`/home/cmt/rev/dtos/docs/mer/RCS/mer.vdd,v 1.10 26 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 report describes the results of a study into security and assurability of microkernel based operating systems performed on the Distributed Trusted Operating System (DTOS) program, contract MDA904-93-C-4209. The goal of the study is to assess the security architecture presented by several microkernel based operating systems and to provide guidance for future development of secure systems.

1.2 Document Overview

The report is structured as follows:

- Section 1, **Scope**, defines the scope and this overview of the document.
- Section 2, **Applicable Documents**, describes other documents that are relevant to this document.
- Section 3, **Assessment Criteria**, describes the goals of this study and a description of the general criteria used to assess the systems under consideration.
- Section 4, **Overview of System Assessments**, provides an overview of the content and organization of the system assessments which follow.
- Sections 5 through 9 contain the assessments of the Amoeba, Spring, KeyKOS, Trusted Mach and DTOS systems against the criteria of Section 3.
- Section 10, **Conclusions**, discusses the conclusions of the report.
- Section 11, **Notes**, contains an acronym list.
- Appendix A, **Bibliography**, provides complete bibliographical information for all cited documents.

Section 2 Applicable Documents

This section contains brief descriptions of the primary sources of information consulted during the assessment of each system.

2.1 Amoeba

An excellent general reference on Amoeba is Chapter 7 of *Distributed Operating Systems*, by Andrew Tanenbaum [33].

More detailed information can be found in the collection of manuals which are included in the Amoeba distribution and are available electronically (currently from `http://www.am.cs.vu.nl/amoeba.html`). These include:

- *The Amoeba Reference Manual—User Guide* [55]
- *The Amoeba Reference Manual—Programming Guide* [53]
- *The Amoeba Reference Manual—System Administration Guide* [54]

For this report, the manuals for Amoeba 5.2 were consulted. Version 5.3 has since been released, but the differences between the releases do not appear to have any effect on the conclusions of Section 5.

2.2 Spring

An overview of Spring can be found in An Overview of the Spring System [18]. This paper is included in *A Spring Collection* [32], a collection of published papers and previously unpublished technical reports on various elements of the Spring system.

The assessment of Spring in this report is based upon the final report [24] of the "Adding Security to Commercial Microkernel-Based Systems" project. This project, conducted by Secure Computing, investigated the possibility of using Spring as a base for a secure system using DTOS methodology.

2.3 KeyKOS

Unlike the other systems considered in this report, there is no single standard reference or set of references for KeyKOS. The information on which the KeyKOS assessment in this report is based has been gathered from a variety of sources. Of these, the documents which have directly contributed to the final assessment are listed here. At the time of the writing of this report, all of these documents are available at `http://www.cis.upenn.edu/~KeyKOS`.

- KeyKOS - A Secure, High-Performance Environment for S/370 [6]

  This paper contains a short but highly readable introduction to the KeyKOS kernel and the KeySAFE security architecture.

- The Confused Deputy [10]

  A very short but enlightening paper describing some of the motivation behind the basic security features of KeyKOS.

- The KeyKOS Architecture [8]

  This paper provides an overview of the KeyKOS kernel, with no particular emphasis on security.

- KeyKOS Principles of Operation [13]

  This paper provides additional details about the KeyKOS kernel implementation, again with no particular emphasis on security.

- A Patent [9]

  This patent document provides by far the most detailed description of the features of the KeyKOS kernel and the basic KeyKOS servers.

- Introduction to KeySAFE [12]

  This paper provides the only specific treatment of the KeySAFE security architecture, though at a high level.

2.4 Trusted Mach

Trusted Mach (TMach) is built upon the MK++ kernel from the Open Software Foundation (OSF). The MK++ external interfaces are described in

- *MK++ Kernel Executive Summary* [21]
- *MK++ Kernel Interfaces* [22]

The internal design of MK++ is described in

- *MK++ Kernel High Level Design* [23]

Overall views of the architecture and security policy for TMach can be found in:

- *Trusted Mach System Architecture* [52]
- *Trusted Mach Philosophy of Protection* [35]
- *Trusted Mach Security Features User's Guide* [49]
- Identification of Subjects and Objects in a Trusted Extensible Client Server Architecture [1]

The TMach servers are described in a series of manuals generally consisting of an executive summary and interface description for each server. In most cases these manuals describe the programming interfaces to a server, much of which actually resides in client side libraries. In a few cases, the "server" consists completely of library code executing in other TMach servers or clients. The following server manuals are specifically relevant to discussions in this report:

- *Trusted Mach Class Library Executive Summary* [36]

- *Trusted Mach Class Library Interface* [37]
- *Trusted Mach Device Server Executive Summary* [38]
- *Trusted Mach Device Service Interface* [39]
- *Trusted Mach File Server Executive Summary* [40]
- *Trusted Mach File Service Interface* [41]
- *Trusted Mach Host Control Server Executive Summary* [42]
- *Trusted Mach Host Control Service Interface* [43]
- *Trusted Mach Item Control Service Interface* [44]
- *Trusted Mach Item Manager Service Interface* [46]
- *Trusted Mach Item Manager Library Interface* [45]
- *Trusted Mach Root Name Server Executive Summary* [48]
- *Trusted Mach Name Service Interface* [47]
- *Trusted Mach Subject Server Executive Summary* [50]
- *Trusted Mach Subject Service Interface* [51]

Finally, the first version of the Trusted Mach assessment was provided to Trusted Information Systems, Inc. for review. Changes were made to the assessment based upon comments received from Jeff Graham [7].

2.5 DTOS

DTOS is built upon the Mach 3 kernel from Carnegie Mellon University. The standard references for Mach 3 are the following:

- *OSF Mach Kernel Principles* [16]
- *Mach 3 Kernel Interfaces* [15]

In addition, Chapter 8 of Andrew Tanenbaum's *Distributed Operating Systems* book [33] provides a nice introduction to Mach 3.

The designs of the DTOS microkernel and security server, and the interfaces to each, are described in the following:

- Providing Policy Control Over Object Operations in a Mach Based System [17]
- DTOS Kernel and Security Server Software Design Document [27]
- DTOS Kernel Interface Document [30]
- DTOS Formal Top Level Specification [26]

The low-level security control policy enforced by the DTOS Microkernel is described in the DTOS Formal Security Policy Model [25].

The ability of this control policy to be used to meet a range of higher level security policies is discussed in the DTOS Generalized Security Policy Specification [29].

An overview of the control policy and the range of supportable policies can be found in Developing and Using a "Policy Neutral" Access Control Policy [20].

Section 3

Assessment Criteria

The basic criteria by which operating systems are assessed in this report are their ability to support some collection of security policies, the strength of the mechanisms which provide this support, and the feasibility of providing assurance evidence to give confidence that the policies are supported. This section provides details of these criteria.

It begins in Section 3.1 by presenting a list of security policy characteristics with a description of each. The system assessments will address each system's ability to meet policies based upon this list of characteristics rather than considering specific security policies.

Sections 3.2 and 3.3 describe criteria for assessing the support provided by operating system mechanisms for satisfying the various policy characteristics. Section 3.2 discusses mechanisms specifically provided for security purposes while Section 3.3 discusses some typical low-level operating system mechanisms.

Section 3.4 describes criteria for assessing the ability to assure that a system satisfies some security policy.

The criteria considering specific mechanisms and some of the assurability criteria were created with a basic understanding of the systems investigated in this report. If different systems are considered, it is likely that some other criteria will need to be added. This is especially true of systems which rely upon very different processing or security models, such as SPIN [2].

The criteria focus on basic operating system and security mechanisms. Higher level elements of an operating system or applications are generally not considered except when they can provide examples to help explain the criteria.

Several specific security features are not considered within the criteria because of the lack of implementation or documentation of the features in the systems under consideration. These include the following:

User Authentication to the System More generally, the criteria ignore authentication of any external entities such as a user or other system connected by a network.

System Authentication to the User It is often equally important for a user to correctly identify a computer system during input or output of critical data. In most environment, this is provided by physical security rather than specific system features.

Application Authentication to the User A trusted path mechanism can be used to guarantee that user inputs are sent to a particular application and outputs are received from it.

Input/Output Labeling Even in the absence of a trusted path some form of labeling of input and output is often desirable.

System Boot and Initial State A system must always execute a boot sequence which results in a "secure" initial state. This is rarely described in system documentation and is not discussed further in this report.

3.1 Security Policy Flexibility

This section presents criteria to assess the ability of a system to be configured to meet a range of security policies. The criteria are presented as a list of characteristics of security policies. The ability of a system to satisfy a particular policy then depends upon its ability to satisfy all of the characteristics of the policy.

The DTOS Generalized Security Policy Specification [29] presents a more in-depth discussion of policy characteristics than is presented here. It describes many of the better known security policies and a taxonomy for classifying security policies according to a similar collection of policy characteristics. However, the list of characteristics presented here differs somewhat from [29] because [29] assumes a specific security architecture and focuses primarily upon security decision making.

3.1.1 General Characteristics

Mandatory Access Control (MAC) To support a MAC policy, all subjects are assigned security attributes, as are all objects which are controlled by the policy. Every security policy decision is a function of the attributes of the subjects and objects involved in a particular attempted access. The definition of these policy functions and the assignment of security attributes are defined into the system, or when configurable, are tightly controlled by a security administrator. An ordinary user may be allowed to make changes but only if they further restrict the policy.

At the user level, a MAC policy is typically implemented when a person's privileges within an organization are dependent upon some status which the person holds rather than just the person's identity. Examples of different kinds of status include clearance levels within a government organization, the chief financial officer for a corporation, an auditor for that corporation, or a customer.

Often, the term "mandatory access control" is used to refer to multilevel security (MLS) policies which perform access control based upon the clearance level of subjects and the classification level of objects. In this document, we use the term more generally, so that MLS is just one example of a MAC policy.

Discretionary Access Control (DAC) A DAC policy assigns some kind of security attributes to subjects and objects, like a MAC policy. The key characteristic of a DAC policy is that the definition of the policy functions and/or the assignment of security attributes can be changed at least in some ways by subjects executing on behalf of "ordinary" users of a system.

A typical implementation of a DAC policy assigns a user to every subject and an access control list (ACL) to each object. The ACL defines the security policy by indicating the privileges to the object granted to each subject based upon the subject's user. If ordinary users are allowed to change the ACL, then this is indeed an example of a DAC policy implementation. However, it is also possible to implement a MAC policy using ACLs by restricting who can change the ACLs, though configuration of a MAC policy using ACLs could be difficult and inefficient.

The most familiar example of DAC is the permission mode bits mechanism in a Unix file system.

DAC and MAC are not mutually exclusive and in fact can serve well as complementary mechanisms within a single system. The standard example of this combination is the security policy considered in the TCSEC [19], which uses MAC to segregate information between classification levels. Within each classification level, DAC is used to provide least privilege "need-to-know" controls which restrict the accesses of users who are otherwise cleared for the data. Cleared users having access to some data are allowed to determine when other cleared users have a need-to-know without placing the burden of determining this on a central security administration.

Information Flow-Transitivity Transitivity is a characteristic of security policies which control the flow of information between subjects based upon some attributes assigned to the subjects. Such a policy is said to be transitive if whenever the policy allows information flow from subject A to subject B and from subject B to subject C, information is actually allowed to flow from subject A directly to subject C, without mediation by subject B.

MLS is an example of a transitive policy (if level B dominates level A and level C dominates level B, then level C must dominate level A). However, intransitive policies are also quite common, even in MLS environments. For instance, data may be allowed to flow from a Secret network to an Unclassified network if it passes through an encryption device. This policy is intransitive because while data flow is allowed from Secret to the encryption device and from the encryption device to Unclassified, it is not allowed directly from Secret to Unclassified.

Therefore it is desirable for systems to be able to support both transitive and intransitive security policies.

Information Flow-Covert Channels Security policies which are stated in terms of information flow, such as MLS, are dependent upon limiting the existence of covert channels in a system.

Though covert channel analysis of any system is clearly beyond the scope of this paper, it may be possible to comment about the ability to identify and respond to covert channels in a particular system.

Mutual Suspicion The KeyKOS Factory mechanism (see Section 7.1.5) was designed to solve a problem of mutual suspicion between the owners of an application and the users of the application. This problem does not readily fit into the description of other characteristics, so we define a separate characteristic to consider a system's ability to solve this problem.

The problem is that the owner of an application wants to allow the application to be used for a fee but does not want the application's executable code to be stolen and copied. And the user of the application wishes to use the application to process sensitive data but does not want this data to be copied or passed to other users. It is assumed that the owner and the user trust the system on which the application is stored and executed and also trust the administrators of that system.

3.1.2 Least Privilege and Granularity

Granularity of MAC Attributes Due to the emphasis on MLS policies, some systems have been developed which support MAC but only support a relatively small number of distinct security attributes. For a single user workstation in an MLS environment, it may be reasonable to expect that the workstation will only need to process data at a small number of classification levels. This is probably not a reasonable assumption for servers in the same environment and it is definitely not a reasonable assumption for other MAC policy environments such as type enforcement.

Limitations of the number of MAC attributes may be due to some specific limit hardcoded into a system or through physical limits caused by segregating hardware resources among security levels or attributes. Other systems incur a significant performance penalty or place inconvenient requirements on the user when data or control passes between levels.

Protection Boundaries/Unique Identifiers Security policies often group subjects or objects based upon some kind of security attributes, so that all subjects or objects in the group are treated identically by the policy. As a simple example, within an MLS policy, all subjects (or all objects) at a particular level are treated identically.

However, other policies require policy decisions to be made based upon the identity of each individual subject or object, or can at least be implemented most naturally in this way. Examples include the High-Water Mark [29, Section 4.1], Clark-Wilson [29, Section 4.2], and ORCON [29, Section 4.5] policies. To support these policies, the security attributes associated with a subject or object should include a unique identifier.

Supporting such policies can be a difficult task for many systems because security attributes are typically inherited when new subjects or objects are created. Therefore, an important consideration in a system's ability to satisfy these policies is the difficulty of overriding these inheritance rules.

Privilege Levels/Least Privilege Traditional approaches to security too often have divided software into two classes, privileged and unprivileged, reflecting a similar distinction in processors. The most familiar example is the distinction between root and ordinary users in a Unix system.

But this binary classification of software also exists in some MLS systems. In such systems, privileged software is part of the trusted computing base (TCB) and can freely manipulate data at all classification levels. Unprivileged software is outside of the TCB and is subject to strict enforcement of the prohibition on reading from higher level objects or writing to lower level objects.

The problem with this approach is well documented by the many examples of privileged Unix applications which can be manipulated to perform privileged operations which are completely unnecessary to their intended function. It is essential that a system provide mechanisms which can limit a subject's privileges to those privileges which are actually necessary to complete its task.

Run-time Least Privilege The previous item discusses limiting the privileges granted to a subject to those privileges which may be needed at some time during the subject's existence. There are several possible ways to further limit a subject's privileges at run-time:

- A simple possibility is to grant a subject less privileges when it is started, if it is known that not all of an application's privileges are necessary during for a particular instance.
- While making a request for service, a subject could explicitly request that its privileges be limited, for instance if it is making the request on behalf of another subject.
- The operating system could enforce such limits if it can identify that a subject is acting on behalf of another subject. For instance, a system with a strong client/server connection may be able to identify when a server is making a request on behalf of some other client.[1]
- A more complex possibility is the use of privilege bracketing, whereby the subject itself determines at what time during its operation a particular privilege must be used, and turns the privilege on and off appropriately. Privilege bracketing is not supported by any of the systems considered in this report.

[1] This possibility is actively being considered for the Fluke project at the University of Utah.

3.1.3 Characteristics of Dynamic Policies

In a static security policy, the permissions granted are independent of the time when the policy is consulted. This is a very strong restriction to place on all aspects a security policy. For example, while it is reasonable to expect that Secret will always be a higher classification level than Unclassified in an MLS policy, it is unreasonable to expect that a user's clearance level will never change.

Therefore a system should support at least some simple kinds of dynamic security policies, including those which change due to explicit actions such as changing the clearance level of a user or the ACL on a file. The following items discuss dynamic policy characteristics which are potentially more complex.

History Sensitive Policies In some security policies, security decisions may depend upon permissions which have been previously granted and used. The ability of a system to satisfy such a policy requires some kind of a distinction between having a privilege and actually using it.

For instance, in the Chinese Wall security policy [4] a subject initially may have the privilege to open any of a large number of files. However, when the subject requests to open a particular file, it may immediately lose the privilege to open many of the other files.

Relinquishment Sensitive Policies Rather than being sensitive to the entire history of a system, relinquishment sensitive policies are sensitive only to those permissions which have been previously granted and are currently being used. For instance, the policy may be dependent only upon those files which are currently open for a subject.

This kind of policy is potentially simpler than a history sensitive policy because it only relies upon information about the current state of the system. However, the ability to extract this state information may not be easy, and it may actually be easier to record all information about previously granted permissions without determining whether they are current.

Examples of relinquishment sensitive policies are the Role Based Access Control (RBAC) policies of Dynamic Separation of Duty and Dynamic Membership Limits [5].

Retractive Policies The previous items consider policies in which future security decisions depend upon previously granted accesses. However, it may be necessary to actually retract previously granted accesses which are currently in use.

This is actually a very common characteristic of security policies. For instance, when a user is removed from a system or the user simply loses some privileges, it should be possible to easily revoke the user's permissions in the system. Revocation may also be done independent of any act of a system administrator. For instance intrusion detection software may revoke permissions if an intruder is suspected.

Retractive Destruction During a policy change in a retractive policy, it may not be enough to retract accesses, some subjects or objects may actually need to be destroyed. For instance, when removing a user from the system there may be a requirement that all subjects operating on behalf of the user are removed. While a subject can be effectively prevented from accessing anything by removing permissions, it may still consume resources.

In this example, it may also be sufficient to identify all of the user's subjects so they can be destroyed by an administrator.

Non-tranquility For a security policy in which security decisions are made based upon security attributes of subjects and objects, it is sometimes desirable to be able to change those attributes, i.e., for the attributes to be non-tranquil.

Complete support for non-tranquility can also require retraction of current accesses. For instance, a file may be currently open by a subject which is not allowed to open the file after a change in attributes. This is not always enforced however. For instance, when an ACL (or Unix permission bits) for a file are changed, current accesses are not necessarily recomputed.

3.1.4 Active Policy Characteristics

The previous lists of policy characteristics have generally focused on the granting of permissions. Here we list three characteristics which consider other active roles which the system may be required to take to support a security policy.

Availability Ensuring availability of system resources is a complex topic which could easily consume all of the resources in this study. It is sometimes considered to be distinct from security (as in "if the system fails to do anything, then it certainly fails to do anything insecure"), but there are potentially significant concerns if availability can be denied to specific subjects within a system. For instance, it may be possible to halt critical processing such as policy updating, audit, and intrusion detection.

We will consider any significant mechanisms which are provided by each system to address availability, recognizing that none of the systems have very sophisticated mechanisms and that a detailed consideration of availability is well beyond the scope of this report.

Accountability Accountability is an important consideration of almost all security policies. It is important to keep a sufficient record of security critical operations so that the effects of security failures, either accidental or malicious, can be determined and the system restored to a secure state. It is essential that these records remain intact during and after a security failure.

Intrusion Detection Security policies can depend upon the detection of intruders to satisfy certain requirements. Since none of the systems in this report provide any significant intrusion detection capability, it will not be discussed further.

3.2 Security Mechanisms

This section describes criteria for assessing the specific security mechanisms which may be provided by an operating system.

The distinction between this section and Section 3.1 is generally the distinction between characteristics of a policy and characteristics of a mechanism. This is not a clear distinction, however, the distinction is not important to the content of the report, only to the organization.

3.2.1 Basic Control Mechanisms

This section discusses some characteristics of the basic control mechanisms which may be provided by a system.

Decider/Enforcer Separation Perhaps the most important contributor to policy flexibility is a strong distinction between the component of the system responsible for making security policy decisions and those components of the system which enforce the decisions. Ideally, the decision making component is completely separate so that the policy can be changed simply by replacing it. In practice it is very difficult to isolate all policy dependent decision making, but there can be a wide range of variation in how close a system comes to this ideal.

Granularity of Access Control Points Access control points are the locations in the system where some access to a resource is granted or denied, i.e., where security decisions are enforced.

Some systems provide only very few access control points, each one controlling a wide range of services. In such a system, the range of policies which can be supported may be narrow since a process may need to be granted access to a large range of services just to access the one which is actually necessary.

On the other extreme, it would be possible for a system to include so many access control points that configuration of the system becomes unmanageable.

Location/Flexibility of Access Control Points A standard model of access control is to assume that all operations to be controlled can be defined as a subject requesting some operation on some object. This model is usually sufficient for a large percentage of the operations to be controlled, and therefore the access control points can be placed at the entry to a particular server where the necessary information is available.

Unfortunately, our experience on DTOS indicates that this model is insufficient for some desired controls. There are two other general models for control:

- The security decision depends upon a parameter to the request. In this case the access control point can still be located at the entry to a server.
- The security decision depends upon information which can only be determined after some processing of the request is completed.

The security mechanisms in some systems do not readily adapt to these other models of access control.

Bypassability There are at least two ways in which access control points might be bypassed. The most straightforward is if there is an entry point which enters some routine past the access control point.

A more subtle possibility which has been observed in some systems is the "time-of-check-to-time-of-use" flaw [14]. In a system exhibiting this flaw, some data used in performing the access control check can be changed after the control point is reached but before the data is actually used. For instance, suppose a process has read access to file FOO but not to file BAR. The process may be able to make a request to read file FOO, but after the access control point is successfully passed, arrange to have the parameter "FOO" changed to "BAR".

This type of flaw may exist if parameters to a request are stored in a buffer which can somehow be altered by the client process. A less obvious case occurs when a parameter is passed as some kind of a pointer, and the client is able change what is pointed at by the pointer between the time of the permission check and the time of its use.

As a specific example, consider the Unix `access()` system call, which can be used by a setuid application to ensure that it is not accessing a file which is inaccessible to a process with the real user and group IDs. Unfortunately, a file is identified to this request by a file name, and when the `access()` call is made the name can refer to a different file than when the file is actually accessed by the application.

Granularity of Access Rights Each access control point is associated with some access right (or possibly a set of rights) which the client process must hold in order to be granted access to a resource. Typical access rights include read, write and execute.

The most general mechanism will associate a distinct access right with each control point. This is not always necessary because in most systems there are multiple paths to perform an essentially identical service, so control points in each path can share a single access right. However, a system in which large collections of access control points all map to the same access right may be just as incapable of supporting certain policies as a system with a small number of access control points. A common example of this is a system in which the only access rights are effectively read and write even though there may be many control points.

3.2.2 Security Management

This section discusses criteria generally related to the acquisition and management of privileges.

Assigning Attributes Assigning security attributes to subjects and objects is a fundamental operation whenever security decisions are made based upon the attributes. Typically attributes are inherited in some way, but there still must be a way to override this or to initially assign some attributes (e.g., upon login). It is important to consider the strength of whatever mechanisms are used to assign attributes, and the controls on overriding.

There's also a possibility that attributes are not assigned explicitly but rather implicitly through related objects. This can be a significant hurdle for assurance (see Section 3.4.1).

Acquiring Capabilities For systems in which security decisions are made based upon capabilities held by a subject it is important to consider all of the ways in which a capability can be acquired. It is sometimes assumed that a capability can only be acquired if it is passed directly to a subject by another subject holding the capability. However, it is also usually possible to use capabilities currently held by a subject to gain other capabilities. In this way, a single capability may actually grant many more accesses than are indicated by the capability itself. Therefore it is important to determine all such chains through which capabilities can be acquired.

Finally, there is the possibility of guessing or forging a capability, which is likely to indicate an even more serious flaw in a capability system.

Ease of Administration An important characteristic for security mechanisms is ease in administration and configuration of the policy. More difficult administration indicates a greater likelihood of errors. In addition to specific security administration tools, there are two basic properties of a security architecture which can simplify administration:

- Grouping of subjects or objects into a collection which is controlled according to a single set of rules (e.g., all entities in the group have identical security attributes).
- Centralized security databases and separation of enforcement from decision.

As an extreme example, while an ACL mechanism could be used to implement an MLS policy, this could be quite difficult to administer because every object has an ACL associated with it and they are not necessarily available through a single "centralized" database.

Transparency To provide a complete operating system environment and a complete set of applications, a system is likely to require the reuse of code developed for other systems. Since security mechanisms are rarely compatible from one system to the next, it is desirable to isolate the impact of those mechanisms from most elements of the system, particularly from applications.

As an example of a non-transparent mechanism, some capability systems require a client to request particular access rights when requesting a capability.

While transparency by itself is not a security issue, there is an impact if the lack of transparency requires redevelopment of applications which are accepted as secure.

3.2.3 Failure Ramifications

With the complexity of modern operating systems, it is unrealistic to assume that any system is completely free of exploitable security flaws. Even if the basic system is somehow miraculously free of such flaws, the configuration of the system for a particular security policy environment is another likely source of security flaws.

Therefore it is important to assess the ramifications of such flaws. The following criteria can provide a start for performing this assessment.

Redundancy Systems which provide redundant security mechanisms, possibly instantiated at different layers within the system, are less likely to suffer catastrophic consequences as a result of a security flaw.

Root The existence of the all-powerful root user in Unix is well known. But most systems (perhaps all) have something similar, some entry point or security attribute or capability which gives total control over the system. The biggest danger with root in Unix is not its actual existence, but that root privileges must be used by many applications and therefore there are many ways to potentially gain root access. The difficulty of getting "root" access in other systems is an important consideration.

Single Failure One specific way to consider the ramifications of failures is to consider what can happen during a single failure. For instance, a single failure should not grant root access.

One problem with many capability systems is that acquiring a single capability in violation of the policy can allow a subject to gain many other capabilities in violation of the policy.

Ideally, single failures can be contained to a small portion of the system and are detectable.

3.2.4 Capabilities versus Security Attributes

One of the more contentious issues in the design of security mechanisms is whether the privileges granted to a subject should be based upon capabilities that a subject owns ("what you know") or upon security attributes assigned to the subject ("who you are"). In this section we summarize arguments made for each of these mechanisms and suggest that the two mechanisms can coexist to gain the advantages of each.

3.2.4.1 Pure Capability Systems and Information Flow Policies

The discussion of whether capability based security mechanisms can be used to satisfy MLS policies has been a lively topic, especially during the mid and late 1980's. The fundamental problem identified with "pure" capability systems is that one-way information flow can become bidirectional. Specifically, if subject A can pass information to subject B, then it can pass B a capability which grants B privilege to write back to A. In an MLS system, this means that the normal upward flow of information (from Unclassified to Secret) can be reversed, and this has obvious analogues in any information flow policy.

Kain and Landwehr [11] studied this issue systematically and created a taxonomy of capability systems to characterize those systems which can actually support MLS. It was later pointed out that this taxonomy was not sufficient to characterize all capability systems because it did not consider the possibility that information flow could be allowed without allowing any flow of capabilities. This possibility is exploited in the KeyKOS design.

Nonetheless, the taxonomy is valuable because many capability systems can be characterized within it. The taxonomy essentially suggests that there are two ways in which this reversal of information flow can be prevented:

- Whenever a capability is passed between subjects or otherwise acquired by a subject, the system must verify that the receiving subject is allowed to use the capability, according to the MLS policy.
- Whenever a capability is actually used by a subject, the system must verify that the subject is allowed to use the capability, according to the MLS policy.

Which one of these choices is preferable seems to depend upon the design and policy goals of a particular system. Some factors in favor of the first choice include the following:

- If a subject is going to be denied use of a capability, it is probably best for the subject to be denied the capability immediately rather than to continue processing and to fail only when it attempts to use the capability.
- Capabilities are likely to be acquired less often than they are used, so the security policy is accessed less often if the access control is provided when the capabilities are acquired.

Some factors in favor of the second alternative include:

- If the security policy is allowed to change, then capabilities must be checked when used unless there is another mechanism for revoking capabilities which have been acquired.
- Determining whether a capability can be used may require some interpretation of the capability by the server which issued it. In this case, performing access control when a capability is acquired requires consultation with the issuing server as well as with the security policy. On the other hand, when the capability is used it is presented to this server so it can be immediately interpreted.

3.2.4.2 Pure Attribute Systems and the Confused Deputy The Confused Deputy [10] presents a scenario which is used to argue that it is insufficient to determine permissions entirely based the assignment of security attributes to subjects and objects. The scenario can be paraphrased as follows:

> There is a server which must write application specific audit information to an audit file. The server provides output to clients through a file whose name is chosen by the client. The problem which arises is that the client may choose to receive output

through the audit file, in which case the audit records are destroyed. This cannot be prevented by mechanisms which only depend upon security attributes because those attributes must specify that the server has permission to write to the audit file.

It is possible to view this as an example of a poorly written server that should verify that it only writes audit data to the audit file. However, requiring the server to compare file identifiers for all write operations can be a significant burden, especially since in many systems it is difficult to determine if two file identifiers reference the same underlying file. Moreover, there may be many other ways in which a client can "trick" a server into using its privileges inappropriately.

Capabilities are suggested in [10] as a general solution to this problem which does not require any specific actions by the server. If the client identifies all external entities (such as files) to the server by capabilities instead of by names or other identifiers, and if the server itself uses these capabilities when performing operations on the entities, then the server will not be able to perform any operation which the client itself is prohibited from performing. In the example of the Confused Deputy, the server will not be able to write the audit file by presenting the client's capability to the audit file.

3.2.4.3 Combining Both Mechanisms

Just as it can be desirable to combine MAC and DAC policies in a single system, it can be desirable to combine capability mechanisms with mechanisms which make security decisions purely on the basis of security attributes. However, anytime mechanisms are combined there is a danger of unexpected interactions.

For instance, consider the scenario of the Confused Deputy. The capability solution requires that the client not hold any capabilities which it is not allowed to use. This implies that capabilities should be verified against the security attributes of the client when the capability is received by the client.

However, a competing scenario may be presented. In a capability system, a name server may hold capabilities to many objects which it has no need to access itself. If capabilities are verified against security attributes when used in a combined system, then the name server can be prevented from using the capabilities even though it holds the capabilities. This implies that capabilities should be verified only when used.

Note that there is another solution to the name server problem if the system supports "upgrading" of capabilities. The name server would hold capabilities which grant no permissions other than upgrade. If the name server requested a new, upgraded capability, it could be prevented from receiving that capability.

3.3 Security Features of Operating System Mechanisms

This section discusses some of the security relevant features of basic operating system mechanisms. Criteria are categorized into three sections:

- Section 3.3.1 discusses the initialization of the state of a process prior to beginning execution.
- Section 3.3.2 discusses the execution of a process and the interactions between processes through direct control and shared memory.
- Section 3.3.3 discusses interactions between processes through inter-process communication (IPC).

3.3.1 Process Initial State Integrity

Within a secure system, some processes are granted privileges which could cause a violation of the system's security policy if misused. To prevent the process from misusing such privileges, it is necessary to ensure that the process is correctly initialized.

Some aspects of the initial state which must be considered are the following:

- The layout of the address space and value of the program (instruction) counter.

  This is generally defined by some sort of an "executable" file, such as an `a.out` file. This file must be protected from the time it is created to the time of its use. Other dynamically linked executables must be similarly protected.

- The security attributes of the process, any capabilities held by the process, or similar information related to specific security mechanisms.

  Ideally, there should be some way of maintaining a relationship between this information and the executable file prior to the creation of a process.

- An application may assume certain properties about the environment in which it executes, such as where to find other services. This may be the most difficult to guarantee by specific mechanisms, so the number of such assumptions should be minimized.

- Privileges granted other processes to access the new process.

  For instance, in some systems the creator of a new process may receive capabilities granting total control over the new process.

Assuming that the integrity of all executables has been maintained up to the time of process creation, the next item to consider is where the responsibility lies for constructing the initial process state from the executables. There are two extreme models for process creation, with typical models lying somewhere between. In one extreme, the operating system creates the entire initial state of a new process with no input from the client requesting the creation. In the other extreme, the operating system creates a process with no state and gives the client process full responsibility for initializing the new process.

The first model is very useful when a process wishes to create a new process with more privileges. For instance, an untrusted shell may accept commands to create trusted processes, which it then passes to the operating system. However, this model also places a considerable burden on the operating system, so there are typically options for the new process to inherit state from the creating process. The extent to which inheritance can be controlled is relevant to considering whether a process can ever create a process with more privileges.

3.3.2 Process Execution

This section discusses some security critical aspects of the operating system features for process execution, with the main exception of interprocess communication which will be discussed in Section 3.3.3.

Process Control A process may have a significant amount of control over the basic execution state of another process. For instance, setting the program counter, blocking or unblocking, or changing the scheduling priorities.

Address Space Separation Another aspect of a process which may be controlled or observed by another process is its address space. Some systems may allow a process to read or write the address space of another process or even to allocate or deallocate memory within the address space. Strict control must be provided over these operations if a protection boundary is established between the processes.

Isolation of Executable Code One of the important aspects of the initial state integrity of a privileged process is the layout of the executable code and the assignment of the program counter. This is important so that assurance of the program from which the executable is generated will be relevant to the actual process.

But assurance relies upon a program being executed as written. If the executable code can be overwritten or the program counter can be moved so that other code is executed, it can be easy to make the process use its privileges improperly. For instance, this flaw allows stack overrun attacks to succeed.

Unfortunately, many operating systems allow all code within an address space to be executed by providing essentially two access modes, R/E and R/W/E. If R and R/W modes were provided (in addition to R/E), then there is confidence that the executables are not modified and the program counter does not jump outside of the executable range. This strong of a distinction may not be possible in operating systems which purposely modify executables during run-time, but tight control over modifications should still be possible.

3.3.3 Interprocess Communication

This section contains criteria for assessing the security features of interprocess communication (IPC) mechanisms. The criteria are stated for a message passing form of IPC though they may be generalizable to other forms of IPC.

The criteria are divided into two sections, one for criteria primarily of interest to the processes involved in the IPC and one for criteria primarily of interest to the security of the overall system.

3.3.3.1 Criteria Affecting the Communicating Processes

These criteria identify possible concerns when one of the processes involved in the IPC is trusted to satisfy some property. Note that several of these criteria can be satisfied by cryptographic protocols between the sender and receiver. However, the communications are likely to be much more efficient if they can be satisfied by operating system mechanisms instead.

Identification of Recipient There are several reasons why it may be important for the sender to precisely identify the recipient of a message it is sending, including the following:

- The sender may wish to guarantee that the message is only seen by certain recipients.
- The sender may be meeting a requirement to notify a particular recipient of some event.
- The sender may be requesting a service and requires that a specific server provide that service.

Note that it may be sufficient to identify the security attributes of the recipient.

However, the destination for a message is generally provided by some kind of identifier which may have no intrinsic meaning. To determine what the identifier represents requires consideration of the source of the identifier and how that source provided the identifier. We consider two specific examples.

In some systems, each received message contains an identifier to which a reply should be sent. When sending a reply to this identifier, the sender should consider how the identifier is supplied. The operating system may supply it with a guarantee that this identifier returns to the source of the original message. This reduces the problem to the problem of source authentication (see below). If the identifier is provided by the original message's sender, then there can in general be no guarantee of what the identifier represents.

For another example, consider a response from a name server request to resolve some name to an identifier. If the source of this response is a trusted name server, then the client can be sure that the identifier is associated with a particular name. But the value of this association must also be questioned, and so there must be consideration of how the namespace association is created.

There is clearly a potential that a very long chain of operations must be considered to determine what is represented by an identifier. Any immediate information which can be provided by a system is valuable, as are any controls which may prevent a message from reaching its destination if it is not what is expected by the sender.

Guarantee of Receipt In addition to determining the destination of a message, the sender may need to know that the message actually reached its destination. Certainly an operating system cannot guarantee that a delivered message is actually "read", so some protocol between sender and recipient may be necessary. But there is also value to providing a simple method by which the sender can ensure that a message is not lost prior to reaching the destination.

Message Integrity It is often important to guarantee that a message is actually received as sent, or if not, that some kind of notification is provided to the recipient (for instance, if the message is truncated).

Source Authentication A message recipient is often interested in determining the source of the message when deciding how to interpret or react to a message. Similar to the sender's identification of the recipient, it may be sufficient for the recipient to identify the security attributes of the sender.

Often, especially in a capability system, the system design explicitly prevents the recipient from identifying the sender. The philosophy behind this design is that the recipient should be satisfied simply to know that the sender had a capability to send a message to the recipient. However, the lack of specific identification can cause significant difficulties for assurance (see Section 3.4.1. Also, specific identification can be a valuable backup security mechanism if a process is able to gain a capability in violation of the security policy.

For all of the benefits of providing source authentication, there are however two potentially significant disadvantages if source authentication is provided on all IPC messages:

- One of the reasons for avoiding source authentication is to allow for transparent interposition between processes. For instance, this is valuable on a single system for debugging or in a distributed system to provide transparent distribution.

  Depending upon the granularity of the source identification, this feature may eliminate the possibility of transparent interposition.
- There is a general principle in cryptographic protocol design that no entity should ever apply its signature to data which the entity is unable to interpret. There is

an analogous risk here if source authentication is applied to all messages sent by a particular task, because a task may not be able to interpret the contents of all messages it sends.

For example, consider a trusted server which provides a service to write uninterpreted data to some object (e.g., a file) and to read data from the object by returning the data in a message. A client of this server, by writing particular data to that object and later reading it back, may be able to cause the server to send any message chosen by the client. If the client can further choose the destination for the server's reply, the client may be able to cause the server to send an arbitrary message to any other process. The recipient of the server's message may then act upon the message based upon the source, even though the server is completely unaware of any interpretation of the contents. This scenario is somewhat analogous to the Confused Deputy example of Section 3.2.4.2.

Therefore an option for the sender to specify anonymity (or even to pose as another sender) is desirable if source authentication is provided.

Effect on Resource Consumption A concern for both the sender and the recipient is how each can affect the resources available to the other. For instance, by sending a large message the sender may be able to cause resources (memory or processor) of the recipient to be consumed by the act of receiving a message. On the other hand, in some systems the sender may be required to donate resources to the recipient, which if uncontrolled, can cause similar problems for the sender.

3.3.3.2 Criteria Affecting System Security

These criteria are of interest to the overall security of the system because of the need for control over communication between untrusted processes.

IPC Prohibition A system must have the ability to prohibit communication, even between consenting subjects, for the system to support security policies which control the flow of information. For instance, suppose an untrusted application containing a trojan horse is introduced on a system enforcing MLS. If processes running this application are started at Secret and Unclassified, the system must be capable of prohibiting communication between the processes.

Message Content Limitations A system may be able to provide fine grained control to control messaging based not just upon the source and destination but upon the content of the message. For instance, in a capability system, the operating system could distinguish between passing capabilities and passing data, or even between different capabilities. This could be valuable for a couple of reasons:

- Section 3.2.4.1 suggests that one way to support MAC with a capability mechanism is to control capabilities as they are passed from one process to another. This can be done without forbidding all communication between the processes.
- A trusted server sending messages to clients may be able to rely upon the operating system to control the content. For example, a name server may not need to consult the security policy when responding to a request to resolve a name to a capability if it can rely upon the operating system to control whether the client is given the capability.

Message Buffering and Dynamic Policies Some IPC systems provide for message buffering in the kernel between the time that the message is sent and the time that the message is received. This can make it more difficult for the system to support dynamic security policies.

Specifically, suppose that the security policy changes between the time that the message is sent and the time it is received. The initial policy may have allowed the message to be sent but not received, while the new policy may allow the message to be received but not sent. But by changing the policy between operations, it is allowed to be sent and received. Whether this behavior is correct is difficult to determine, and most likely is dependent upon the details of the system and the old and new security policies.

3.4 Assurability Criteria

The previous three sections present criteria to assess a system's potential to support a range of security policies. However, it is not sufficient to claim that a system satisfies some security policy. There is also a need to provide evidence supporting this claim. We use the term *assurance* to refer to this evidence and the process of developing the evidence.

This section presents criteria for assessing the feasibility of providing this evidence, i.e., the assurability of a system.

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 the security properties is generally at a high level, comparable to the policy characteristics in Section 3.1.

The assurance argument often involves specification of a system at multiple levels of abstraction between the high level statement of the security properties and the actual system 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
- Code reviews
- Use of cleared personnel
- Code assertions
- Type-safe languages
- Testing
- Mathematical proofs

The criteria in this section were generated by considering mathematical proofs as the primary technique for verifying the security properties of a system. Formal mathematical methods provide the strongest guarantee of completeness in each step of an assurance argument, which is especially important when verifying security properties because a single failure can have potentially devastating consequences. This is reflected in the TCSEC [19], where the only distinction between the highest security rating (A1) and the second highest rating (B3) is the extent to which formal methods are used for system verification.[2]

However, even though these criteria were generated by considering a particular kind of assurance argument, they are not only of interest to the argument. Any aspect of a system which

[2] While formal methods provide the strongest guarantee of completeness when they are applicable, it must be emphasized that formal methods are not applicable at all levels of an assurance argument. In particular, there is little hope of applying formal methods to the actual implementation of a system.

simplifies assurance is also very likely to lessen the likelihood of an inadvertent security flaw in a system.

The assurability criteria are divided into three sections. Section 3.4.1 describes criteria specifically relating to the development of proofs relating one level of specification to the next highest level. Sections 3.4.2 and 3.4.3 describe criteria for assessing the impact on assurability from the design and implementation of a system, respectively.

3.4.1 Proof Complexity Criteria

The complexity of a proof demonstrating correspondence between two layers of specification (or between specification and requirements) is clearly proportional to the complexity of the specification. Here we consider two other factors which contribute to the complexity of a proof.

These factors assume that the properties to be proven can be stated as requirements on individual transitions (rather than sequences of transitions). For instance, properties are often defined by state invariants which require that some property of the system holds in every state. State invariants are proven by showing that each individual transition maintains the property.

- How many transitions must be considered in the proof of each property?

  There are two ways to minimize the number of transitions which need to be considered. One way is to provide strong modularization in the design, so that only a small number of requests can even potentially affect whether a property holds. The other way is to provide a strong least privilege mechanism so that only a small number of clients can make a particular request.

  Note that these are important considerations not just in the assurance of the operating system but also in the assurance of particular applications. If there are many ways in which the state of an application can be changed by external clients, then all of these cases must be considered in the assurance of the application. However, if the operating system provides strong separation between applications, then assurance of the application can focus specifically on the application itself.

- How many properties must be proven about the system?

  Within this report, this criteria will provide the most critical distinction in the assurability of the various systems. The fundamental distinguishing factor is the extent to which the security mechanisms in the system provide explicit controls over the system.

  In any assurance argument, there are certain properties which must be proven independent of the implementation. So there is a minimum number of properties which must be proven. If there is an explicit control in the system corresponding to every property, then no other properties will need to be proven, except that the controls themselves are correct. Though this is an unrealistic ideal, there is considerable variation in the amount of explicit control provided by each system.

  As an example, consider the property "an encryption process will only use cryptographic keys obtained from the key server". How difficult is it to prove that a key is actually obtained from the key server?

  Suppose the encryption process obtained the key as a return value from an RPC call. If the operating system allows the process to explicitly control the destination for an RPC, then the proof is straightforward. If the system does not provide this control, then the proof must further consider the source of the identifier through which the RPC was made, such as a name server. There is a potential for a very long chain of operations that must be considered, and each step in the chain ultimately creates another state invariant that must be proven about the system (such as "the identifier associated with the name `key server` in the name space always represents the key server").

Of course, the explicit controls must still be verified to be correct. But the complexity of such properties is likely to be significantly less than the complexity of the properties which must be proven through a long chain of implicit dependencies.

3.4.2 Design Level Criteria

Documentation/Developer Support One of the most practical concerns when generating assurance evidence for a system is the level of involvement of developers, either directly or through documentation. An assurance analysis is unlikely to be successful if the analysts must reverse engineer the system to determine the higher level design models and goals.

Identification of Privileges It should be easy to determine at any time the total set of privileges granted to any particular subject and the total set of subjects granted a particular privilege to a particular object. The security mechanisms chosen in a system can have a considerable effect on the ability to do this.

- This is a relatively simple task in a system using a centralized database to determine permissions granted based upon the security attributes of subjects and objects.
- In a system using ACLs, it is easy to determine the subjects which have access to an object but it can be very difficult to identify all of the objects to which a subject has access.
- In a capability system it can be very difficult to determine either the privileges granted to a subject or the subjects granted privilege to a particular object.

One Request/One Service Ideally, each request in a system should perform a single identifiable service and each service should only be provided by a single request. This not only makes a system easier to understand and therefore to specify, but also simplifies proofs by minimizing the number of requests which must be considered in the proof of a particular property.

Minimize Externally Visible State Assurance arguments must generally consider any information available across an interface, so data hiding contributes to assurance simplicity. Note that for some policies, most notably covert channel policies, state which is implicitly visible, for instance through error codes, is also a concern.

Unfortunately for assurability, operating system designs are moving towards making more state visible to meet flexibility goals.

Number of Services A rather crude estimate of the difficulty of assuring a system can be made just by counting the number of requests which can be made. Requests which provide multiple distinct services should be counted as multiple requests.

3.4.3 Implementation Level Criteria

This section lists a few specific criteria concerning the effect of implementation choices on assurability. Perhaps the most fundamental criteria is that code which is easier to understand and maintain is easier to assure. Good coding practices and well-structured code are essential to assurability.

Programming Language Many different aspects of programming languages can affect the assurability of code, including the following:

Abstraction level Software written in higher level languages is generally easier to assure because the programming language is closer to the level at which a practical model can be constructed and analyzed.

Data typing Strong data typing (including user defined types) allows for reliance on the compiler to catch a class of errors which would not be found in languages (such as C) with less support for data typing.

Modularization Support for software modularization, including separation between public and private elements of a module, allows the assurance analysis to rely on the compiler to guarantee that certain data is only affected by a particular limited set of routines.

Pointers Use of pointers, especially direct manipulation of pointer values, is a particularly troublesome aspect of many programming languages.

Constructor/destructor functions Constructor/destructor functions need not be negative from an assurability standpoint, but they can be if not clearly documented, due to the fact that they can be called with no obvious reference in the source code. One particularly subtle case of this is a language like C++ that allows mistyped expressions if the typing can be "corrected" by applying a constructor function to one of the data elements in the expression.

Operator overloading Similar to constructor/destructor functions, operator overloading can cause functions to be called which are not obvious in the source code, especially if common symbols are redefined for particular types of operands.

Code Generation Tools In general, tools which can be used to generate multiple instantiations of similar code can simplify assurance by eliminating the need to assure each instance of the code. Even though the code itself may be simple, repetitious code is especially prone to errors when created or reviewed because less care is likely to be taken with a repetitive process. A typical example of this kind of code generation is an interface definition language (IDL) compiler.

The disadvantage of using code generation tools is the requirement to perform some kind of assurance on the compiler itself. There is clearly a tradeoff between the amount of assurance required for the tool and the amount of assurance saved by using the tool, though this may not be easy to identify.

Complexity Between Interfaces How complex is the processing between the client and server interfaces?

In typical client/server systems, the client calls library code which is rather distant from the request processing code in the server. When the client itself needs to be assured, this means that all of the code between the programming interface used by the client and the server must be assured. If complex client side libraries are used, this can be a formidable task.

An interface definition language and stub generator can potentially simplify this assurance by creating an up front cost in assuring the compiler with the back end savings of not needing to assure the stub code individually for each request.

Code Complexity Any of the established code complexity metrics (McCabe, Halstead, etc.) can suffice as a crude metric for determining the relative assurability of implementations.

Section 4
Overview of System Assessments

The system assessments in the following sections do not follow any specific format or style, and the emphasis in each section can be quite different. While it would have been possible to simply enumerate the criteria and assess each system against each of the criteria, this approach is not taken so that the unique features of each system, and the criteria especially relevant to those features, can be the focus of each system assessment. Also, it is impossible to judge all criteria for all systems (especially the assurability criteria) without a much more detailed analysis than is permitted with the time and information available.

It is important to emphasize that the assessments of the systems in this report are simply assessments against the criteria of Section 3. None of the systems (including the DTOS prototype) were designed to satisfy all of these criteria, and the assessments do not reflect on whether a system meets the goals for which it was designed.

This report does not attempt to describe all of the mechanisms of the various systems being assessed. Descriptions of the systems have actually been avoided except when discussing some feature of a system for which a single concise reference is unavailable. References to system documentation have been included whenever possible.

Some specific explanations of differences (and similarities) in the system assessments follow:

- The assessments of Amoeba and Spring are relatively high-level because each neither system is able to support the basic goals of the policy flexibility criteria without fundamental change.
- The length of the KeyKOS and Trusted Mach assessments differs considerably, primarily because much more information is available about Trusted Mach. The DTOS assessment is also shorter than the Trusted Mach assessment, not because less information is available, but because other there are other information sources which can be easily referenced.
- Each assessment is allowed to refer back to and build upon the previous assessments. Forward references exist but are used less often.
- The KeyKOS, Trusted Mach and DTOS sections all include separate Security Assessment and Assurability Assessment sections which follow the organization of the criteria of Section 3. The purpose of this is to provide a summary for each system and for comparison of the three systems. However, even in these sections there is no attempt to cover all of the criteria for each system.

Section 5 Amoeba Assessment

In this section we describe and evaluate the basic control mechanisms provided by Amoeba, as well as examples of their use within the system. The mechanisms considered are the following:

- Kernel controls over the base RPC mechanism (section 5.1).
- Kernel controls over the group IPC mechanism (section 5.2).
- The general capability system provided by Amoeba for server control over accesses to individual objects. This system is built upon the kernel controls on the base RPC (section 5.3).
- Kernel control over user-space memory accesses. This system is based upon the general capability mechanism (section 5.4).
- Directory server controls over access to objects in the directory server. This is a specific example of the use of the general capability mechanism in a user space server. This example was chosen because of an interesting use of capabilities within the directory server and because of the directory server's visibility in the Amoeba documentation (section 5.5).

We conclude with a summary including a few comments on the feasibility of altering these mechanisms to overcome some of the identified deficiencies.

We have chosen not to evaluate Amoeba with respect to more specific criteria due to the serious deficiencies in the control mechanisms and the apparent impossibility of removing those deficiencies without fundamental changes to Amoeba, changes which would make the control mechanisms look more like the mechanisms in other systems to be studied.

5.1 Kernel Controls Over RPC

The basic mechanism for inter-process communication in Amoeba is an RPC interface provided by the kernel.[3] Using the RPC interface, client processes make blocking requests to server processes through ports.

A port in Amoeba is identified across the kernel interface by 48-bit identifiers which are different for client and server. The server uses a get-port when requesting to receive requests through a port. The get-port is chosen by the server using a standard library routine to construct a random 48-bit number. The client uses a put-port when requesting to send requests to a port. The put-port is related to the get-port through a one-way function which is also accessed through a standard library routine. For the client to obtain the put-port, it must be made available by the server, generally through a name server.

In essence, the get-port is a receive capability for the port and the put-port is a send capability for the port, though Amoeba reserves the use of the term "capability" for a different mechanism to be described in Section 5.3.

[3] Actually, Tanenbaum's book [33, page 394] hints that there is a lower-level asynchronous messaging interface, but since this is not mentioned in any of the manuals [55, 53, 54], it is not discussed in this report.

Because of the use of the one-way function, this mechanism for distinguishing put-ports and get-ports is an effective technique for separating a process's ability to send a message (as the client) from its ability to receive a message (acting as the server).[4] This is essential since it allows a server to serve clients that are less trusted than the server.

However, this mechanism has a serious fundamental flaw because the kernel is completely incapable of preventing communication between two untrusted processes. The two processes can communicate through RPC simply by agreeing to use a predetermined get-port value. Therefore Amoeba cannot support any policy which allows two untrusted processes to exist simultaneously on the system, each in a different "protection domain". For example, in an MLS system, Amoeba cannot prevent a malicious process running at SECRET from communicating information to a malicious process running at UNCLASSIFIED.

It is tempting to suggest that this problem could be solved with minimal change to the kernel if it were required that the kernel generate all get-ports, so that the malicious processes cannot agree in advance on a particular get-port value. This certainly eliminates the ability to pass capabilities before the system is operational. However, this only makes the job of agreeing on a get-port value slightly harder. Assuming that the system contains some covert channels (a reasonable assumption in any system), the "server" merely needs to transmit 48 bits through a covert channel and then the two untrusted processes can again communicate without restriction.

One way to look at this problem is that the design violates the fundamental principle that small data items (such as cryptographic keys, or in this case, capabilities) which can be used to grant access to large data items should never be held by untrusted subjects because this allows a narrow bandwidth covert channel to become effectively a very wide bandwidth channel.

Comparing Amoeba to other capability based systems, the problem is that Amoeba violates the principle that at least some capabilities must be controlled with respect to the owner, either when the owner uses the capability or when the owner receives the capability (see Section 3.2.4.1). But in Amoeba, capabilities cannot be checked when used because there is no means for a server to identify the client process, and cannot be controlled when granted since capabilities can be transferred transparently (even off-line). To pursue either path would require fundamental changes to the Amoeba kernel (see Section 5.6).

5.2 Kernel Controls over Group IPC

The Amoeba kernel also provides a group communication mechanism. Using this mechanism, groups of processes can be created and within each group, a process can send a message to all other members simultaneously. The intended use of group communication is among cooperating servers, and therefore all processes within a group are essentially treated as equals. In particular, there is no distinction between the ability to send to a group or receive as a member of the group, as both operations require knowledge only of a port (essentially a put-port) representing the group.

Since the port representing a group functions as a capability for all of the operations on the group, the security of group communication is similar to Amoeba RPC. In particular, the port representing the group can be passed as a string of bits, and cooperating untrusted processes can freely communicate by establishing a group in which they all belong.

Once again, a solution to this problem would seem to require fundamental changes within the

[4] Assuming that the one-way function is sufficiently strong, the assessment of which is outside of the scope of this study.

Amoeba kernel.

5.3 Server Capabilities

Amoeba provides library routines that servers can use to implement a capability system, on top of the kernel enforced put/get-port mechanisms, to provide for finer granularity control over services provided by the servers. The Amoeba kernel itself is one server that makes use of these mechanisms, for control over processes and segments.

It is important to recognize that Amoeba neither requires the use of these capabilities in servers nor forbids the use of other capability mechanisms in the servers. However, the mechanisms to be discussed here are generally used by the servers provided with a standard Amoeba distribution so at the very least, the assessment of the capability system is relevant to those servers, including the kernel.

A capability in Amoeba, like a put-port or get-port, is simply a number and not a distinct object type. A capability represents a particular set of access rights to a particular object. Specifically, a capability is a 128 bit quantity consisting of four fields:

- (48 bits) The put-port for the server port through which the object represented by the capability is accessed.
- (24 bits) The server's identifier for the object represented by the capability.
- (8 bits) A bit map identifying the access rights to the object which are granted by the capability. This allows up to 8 different access rights to be assigned for each object. These access rights correspond to some access controls provided by the server managing the object.
- (48 bits) The cryptographic check field, which prevents forgery of a capability even by a process holding a capability to the same object with different access rights. This is accomplished using a one-way function similar in spirit to the way in which a put-port is derived from a get-port.

Assuming that the one-way function is sufficiently strong[5], the weakest aspect of this mechanism is the fact that a capability is just a number and can again be passed in a variety of ways, including off-line. This leads to the same problems discussed in Section 5.1.

Another serious deficiency in Amoeba is simply the lack of any set of tools for use to determine what processes should receive a capability. The intended paradigm seems to be that when an object is created, the server will return a capability granting full access to the process which requested creation of the object. But now this process must decide when different access rights for the object may legitimately be granted. This should be dictated by the security policy, which ideally is not hard-coded into the process. A standard set of tools to determine how to make these decisions is necessary to have any flexibility in security policy, and even to deal with the possibility of new programs needing access to existing objects.

While such a tool set can certainly be added to the system, for example as is used by Spring to support access control lists, the tool set does need to be integrated with the Amoeba capability system and ideally the Amoeba IDL. What is notable is that the Amoeba documentation goes so far as to consider the lack of such tools for access control lists to be a feature [33, page 387].

[5] Once again, this is outside of the scope of this report.

While not requiring the use of ACLs may be legitimately considered a feature, the lack of tool support for processes that wish to use ACLs is certainly a detriment.

Another possible weakness is the limitation to 8 access rights for each object. While this granularity is certainly superior to systems which provide only "read" and "write" controls (or worse yet, a single control over all operations on an object), it is still questionable whether this is enough granularity in general. However, at the current time we have identified no specific need for more than 8 bits in Amoeba.[6] The concern is instead that maximum flexibility may be difficult to achieve with only 8 bits.

Amoeba does integrate the access control into the Amoeba IDL, a very desirable feature. The Amoeba IDL allows the specification of an access right to be checked for each request, and code to check for this access right in the capability used to gain access to the object will be generated by the IDL compiler. However, even here Amoeba falls short of full integration of the capability system into the IDL compiler, as the compiler does not generate code to verify the authenticity (cryptographic checksum) of the capability. If the developer forgets to add this code wherever needed, the access control in the automatically generated code is of little value.

5.4 Control Over Memory Accesses

The Amoeba kernel presents a very simple memory management model. Each process's address space consists of a sparse collection of mapped segments. Segments are independent of address spaces and can be mapped into zero, one, or more address spaces.

Segments are represented external to the kernel by capabilities of the type described in Section 5.3. Three distinct access rights are carried by these capabilities, read, write and delete [53, page 365].

Like many systems, Amoeba provides no support for a memory access mode to provide read access separate from execute access. As discussed in Section 3.3.2, this is a desirable feature, and it can only be provided by altering the Amoeba kernel.

Amoeba nominally supports a read-only (actually, read-and-execute-only) mode for memory access. However, it is unclear whether these distinct modes are actually available in practice. For instance, according to the Amoeba Programming Guide [53, p. 366], "... it is currently not a good idea to strip rights from segment capabilities." This makes secure use of shared memory between processes with distinct security characteristics infeasible. A mitigating factor is that Amoeba already discourages the use of shared memory (see [33, page 393]). Once again, failure to completely distinguish read permission from write permission to memory can only be remedied through alterations to the kernel itself.

5.5 Directory Server Access Control

The use of server capabilities in the Amoeba directory server provides an example of both a standard use and a creative use of capabilities. To discuss this, we first describe briefly the structure of an Amoeba directory.

A directory is a collection of entries, where each entry consists of 6 fields:

- A name used to refer to the entry.

[6] It may be that the Amoeba servers are sufficiently small in function that 8 bits are sufficient in all cases. We have not seen any servers that provide more than 8 significantly distinct operations.

- A set of capabilities. While in most capability based systems a name would refer to a single capability, multiple capabilities are stored with a name in Amoeba to provide support for replicated servers.
- Four sets of access rights (8 bit masks as in a capability). The first three of these sets are informally associated with user, group and other, respectively, while the fourth set is generally not used.

Six bits in the access rights field in a capability for a directory object are used:

- One bit to represent the right to modify the directory.
- One bit to represent the right to delete the directory.
- Four bits which form a "column mask" to indicate which set of access rights the holder of the capability is granted to objects in the directory. Only three of the bits in the mask are typically used, indicating whether the holder of the capability is considered to be owner, group, or other for objects in the directory.

Note that while there is no distinct "read" right, all operations on a directory require that at least one of the four column bits is set.

So when a client requests a set of capabilities associated with a name in a particular directory, the result is determined as follows:[7]

1. The directory server looks at the client's capability for the directory and determines whether the capability is as owner, group or other.
2. The directory server looks up the name, and selects the appropriate set of access rights (owner, group, other) as determined in the previous step.
3. The directory server creates restricted versions (if necessary) of the capabilities associated with the name, so that the restricted set of capabilities only contain the access rights determined in the previous step.
4. This set of capabilities is returned to the client.

The way in which the designers of the directory server have used the access rights in conjunction with the structure of a directory object to emulate owner/group/other protection fields demonstrates flexibility in the capability mechanism. At the same time, it must be noted that in general this scheme does not provide the flexibility of Unix protection bits. In particular, since the owner/group/other attribute is determined based upon the capability for the containing directory, it is not possible to have different owners or different groups for objects in a single directory. Moreover, the owner/group/other attribute for one directory determines exactly the attributes for each contained directory.

In support of this scheme, it should also be noted however that by naming an object in multiple locations in the directory structure it is possible to actually be more flexible in the attributes of an object, for instance so that an object can effectively have multiple owners.

The fundamental cause of the inability to support Unix permission bits is the fact that there is no facility in Amoeba for associating a user with a process. There is of course a login operation, but the result of the operation is not the association of a name with a process but rather the granting of a particular capability to the process.

[7] This is a slight simplification, but is correct enough for our purposes.

5.6 Summary

We have discussed the basic security mechanisms provided by Amoeba, and have found them to be insufficient to support many of policies considered in Section 3.1. More seriously, we have found that due to the fact that capabilities (of all three forms) are transparent to the kernel, it is impossible for the kernel to isolate untrusted processes as required for many security policies.

We can suggest a few ways to change the Amoeba kernel to allow it to support MAC policies. Since each change follows a security model to be discussed later, we make no comments here about the effectiveness of each:

- Change the capability system (and hence IPC) so that capabilities are managed by the kernel, and in particular, can only be passed through kernel operations and kernel IPC. Then provide a means of partitioning the possible capabilities which a process can receive directly as in the KeyKOS model (see Section 7).
- Add security labels to all objects in the kernel, including ports, and control all kernel operations based upon the security label of the process performing the operation and the security label of the object being acted upon. This is essentially the DTOS model (see Section 9), in which the original capability system is largely ignored.
- Some combination of the previous in which partitioning and kernel labeling are linked, for instance as in done in TMach (see Section 8).

We also discussed the Amoeba memory controls, and have found these controls to be lacking. These deficiencies are not uncommon and the fix is most likely limited in scope to kernel routines for memory management and process creation routines in libraries and/or process servers.

Finally, we discussed the use of capabilities by the Amoeba directory server. The use of access rights to distinguish owner, group and other is creative, but fails to provide the flexibility provided by Unix file permission bits due ultimately to the fact that Amoeba provides no facility for associating a user with a process. One potential means of overcoming this weakness would be to use the capability granted a process upon login as an authentication token. This approach is taken by the Spring system, and but also suffers from some problems (see Section 6).

Section 6 Spring Assessment

The final report for the program "Adding Security to Commercial Micro-kernel Based Systems" [34] presents an assessment of the security and assurability of the Spring microkernel based operating system. Rather than restate the results reported there, this section will summarize the major results.

Spring provides a general set of tools for implementing capability based security mechanisms. These mechanisms are used within the kernel and can be used essentially in an identical way by higher level servers. The capability mechanism uses kernel protected capabilities so it is impossible to gain capabilities except through the kernel, unlike Amoeba.

In general, the Spring capability mechanism is highly developed and supports some very desirable features:

- The capability system is incorporated with an access control list mechanism.

  For each object, the ACL defines those rights which may be granted to any particular principal. When a capability for an object is initially created, it is created for a particular principal and the access rights granted in the capability are limited by the principal's entry in an access control list associated with the object.

- Spring provides for revocation of access rights within the capability system.

  Capabilities record not only the granted access rights but also the principal to whom they were originally granted. If the ACL is changed to deny a previously granted access right to some principal, that access right will be removed from all capabilities created for the principal.

- Distinct access rights can be used to control each request, encouraging least privilege.

  Spring allows each server to define the access rights which it supports, and there is in general no limit on the number of distinct access rights. In the standard Spring servers, it is typical to use a different access right to control each distinct request.

- The access control mechanisms are heavily integrated into the Spring IDL, so that almost no effort is required on the part of the developer of a server to incorporate the security mechanisms into that server. This increases the confidence in the correctness of the security mechanisms and is also a boost to assurability.

However, there are still two significant shortcomings of the Spring security system:

- The capability system does not provide either of the controls suggested by Kain and Landwehr (see Section 3.2.4.1) and therefore Spring cannot support any policy which requires information to flow one-way between any pair of processes. This failure eliminates most MAC policies of interest, including MLS. This is the primary reason that Spring is given little further consideration within this report.

- The Spring authentication protocols and the mechanisms for storing authentication *credentials* are flawed and one of the flaws appears to be inherent in the design. This particular flaw makes it generally possible for any process to "steal" the credentials of another process or to replace the credentials of another process. This is discussed further in Section 8.2.1 of [34].

The program whose results are captured in [34] studied the assurability of Spring in more detail than the systems studied for this report (with the exception of DTOS). The result of that program was that it is not feasible to provide high assurance for Spring, due to complexity in several areas of the system similar to Mach 3. However, we have no reason to believe that the complexity of Spring is significantly different than any of the other systems considered in this report, with the possible exception of MK++ (because of its assurability goals, see Section 8.1.1).

Finally, [34] considered the possibility of adding DTOS-like security mechanisms to Spring in a manner similar to the way Mach 3 was enhanced for the DTOS program. It was found that the Spring system was generally quite suitable for such additions, which would serve to overcome the security mechanism shortcomings described above.

Section 7
KeyKOS Assessment

The assessment of KeyKOS is broken into the following sections:

- Section 7.1 discusses the control mechanisms provided by the KeyKOS kernel and a few of the basic object servers outside of the kernel.
- Section 7.2 discusses the KeySAFE architecture which provides specific security mechanisms in servers external to the kernel.
- Section 7.3 assesses the ability of the KeySAFE architecture to be configured to meet a range of security policies.
- Section 7.4 provides a very high level assessment of the feasibility of providing assurance evidence for KeySAFE.
- Section 7.5 provides a summary.

The KeyKOS assessment is done at a higher level than the assessments of some of the other systems considered in this report. The main reason for this is the lack of consistent details within the documentation considered for this report.[8] Nonetheless, we believe that a useful assessment can be provided at the architectural level because of the many aspects of the KeyKOS architecture that are designed with security as a major consideration.

7.1 KeyKOS Kernel

The assessment of KeyKOS begins by considering the basic system capability and IPC mechanisms, which as usual are strongly interconnected, and then consider controls over memory operations. Finally, we discuss some miscellaneous features of KeyKOS which enhance the security of the system in one way or another: meters, space banks and factories.

7.1.1 KeyKOS Capabilities and IPC

All interactions among KeyKOS processes and between those processes and the KeyKOS kernel are accomplished through invocation of capabilities known as *keys*.[9] In general, keys represent objects and access rights to those objects. Because keys are managed by the kernel and accessed indirectly by user processes, they are unforgeable and can be granted only through explicit kernel action.

[8] This lack of consistency in the details that are provided will occasionally be noted in footnotes throughout this section when we lack confidence in the strict truth of some statement. However, we do not believe that any of these details materially affect the evaluation. In many cases, the lack of consistency is probably due to changes in the system over the 15 year period spanned by the documentation.

[9] Strictly speaking, not all keys are actually capabilities, some keys known as *datakeys* simply contain data. However, datakeys could be considered to be capabilities granting read and write access to the data, so we will continue to refer to keys as capabilities.

Keys have a relatively complex structure and can represent many different kinds of objects and different access rights to the objects. However, at the highest level there are two distinctly different kinds of keys:

- *Primary* keys are used to invoke a method on some kernel object.
- *Gate* keys are used to invoke a method on an object implemented outside of the kernel.

7.1.1.1 Primary Keys and Kernel Requests

All kernel requests are made through invocation of some primary key held by the client of the request. In this section we discuss the objects and access rights which can be represented by primary keys.

Within the kernel, there are two primitive classes of objects, *pages* and *nodes*. A page is simply an abstraction of a hardware page, while a node is a collection of up to 16 keys.[10] Since primary keys represent kernel objects, it might be expected that there are only two types of primary keys, one for each primitive class of kernel object. However, this is not the case, for two reasons:

- Several kernel requests are not associated with any actual object within the kernel, for instance requests that create new objects. Usually different key types are used to represent each of these different requests as if they were methods on different types of objects.
- The kernel supports other major abstractions which are compound objects consisting of multiple nodes and pages. For each of these there is a different key type, though a key of that type actually represents a node object called the *root node* of each compound object.

The most important compound objects are domains, segments and meters. A *domain* is the potentially executable entity in KeyKOS. When a domain is actually schedulable and non-blocked, it is also called a *process*. A *segment* represents a region of virtual memory. A *meter* represents privilege to use a certain amount of processing time.

Besides representing different objects or kernel requests, primary keys can represent different access rights to the objects. For example, there are three different access rights which can be granted by a key representing a node object:

- A *node key* gives the holder the right to copy any key in the node and to remove or add keys at will.[11]
- A *fetch key* gives the holder the right to copy a key from any slot in the node, but not to actually change the contents of the node.
- A *sense key* gives the holder the right to copy a key from any slot in the node, however, the holder receives the *sensory* version of each key. A sense key also does not grant the right to change the contents of the node.

The notion of a sensory key is particularly important in KeyKOS. The purpose of sensory keys is to address the general problem that capabilities can often be used to gain other, sometimes more powerful, capabilities. For instance, the analysis of those objecting to the potential of

[10] To be strictly correct, there is more to a node than just 16 keys, as some other state information is cached in nodes, but that is not important for this report.

[11] The terminology can be rather confusing here. There are many different types of keys which represent nodes, but only one of these types is actually called a *node key*.

capability based systems to support MAC policies is often predicated on the assumption that a "read" capability can be used to gain "write" capabilities.

KeyKOS uses sensory keys to prevent this kind of access right escalation by guaranteeing that a sensory key, which is essentially a read-only key, can never be used to gain access to any keys other than additional sensory keys.

Note that for keys that represent kernel requests rather than actual kernel objects, the sensory key generally grants no access rights at all.

Actually, the separation between access rights and key types is not distinct. For instance, a key to a domain grants the holder rights to change or copy certain keys in the root node of the domain. However, the kernel does not identify a type for each node, so the kernel will attempt to treat any node which is referenced by a domain key as a root node for a domain, so in this way a domain key can be considered as a node key with a particular set of access rights.

This aspect of the KeyKOS kernel could potentially complicate its analysis by making it difficult to actually identify compound objects. To some extent this is a problem in any kernel which is not written in a completely type-safe language, since for instance there is generally no way to guarantee that a pointer references a particular type of object. The problem is more severe in KeyKOS however since it extends explicitly through the kernel interface.

7.1.1.2 Gate Keys and IPC KeyKOS IPC is accomplished through key invocation on gate keys. Each gate key represents the right to invoke a particular domain. There are two distinct types of gate keys:

- *Start* (or *entry*) keys are presented to the kernel when one domain calls another at the initiation of an IPC call. KeyKOS IPC can be blocking or non-blocking, and can be used to pass both keys and data.
- *Resume* (or *exit*) keys are presented to the kernel when one domain returns from a blocking IPC call made by another domain (or the kernel). Resume keys are created by the kernel when a blocking IPC call is made, passed to the invoked domain and consumed when the return is completed.

There are two distinct forms of KeyKOS IPC:

- A *call* causes the calling domain (client) to block awaiting a reply from the called domain (server). The kernel creates a resume key through which the server returns to the client.
- A *fork* allows the calling domain to continue to execute. No resume key is created by the kernel.

Three aspects of KeyKOS IPC deserve attention:

- IPC can be initiated by the kernel as well as by user processes. This occurs as a consequence of certain faults that can occur during processing, and results in a gate key for a *keeper* being invoked. Keepers are associated with domains, segments and meters, and can be set by any domain with appropriate access to the root node of the compound object.
- The kernel provides no mechanism for the invoked domain to determine the identity of the calling domain.

- Because of the way in which the kernel creates resume keys, it appears at first that the server domain would have confidence that as long as it sends the reply using the resume key provided with the call, that the reply is going to the client which called the server. This may not actually be the case. In an IPC fork, the client can put a key in the same location in the message where the kernel provides the resume key in an IPC call. It does not appear that the server can determine whether it was called by a fork or a call, and therefore it may not actually know whether the resume key it receives is provided by the kernel or the client.

The last two bullets just point out that there is really no meaningful authentication provided by the kernel IPC mechanisms.

7.1.2 KeyKOS Memory Controls

KeyKOS control over memory is quite straightforward. All memory is represented by page objects. For a domain to access memory, it must have a page key for the page representing the memory. Page keys have two possible sets of access rights, read and read/write. As is typical, read access to a page always implies execute access.

The kernel itself provides no specific mechanisms for controlling propagation of page keys or for treating page keys significantly different than any other capabilities.

7.1.3 Meters

KeyKOS scheduling is performed through meters, which are used to control and record the amount of CPU time granted to each domain. Meters are arranged in a tree hierarchy where each meter explicitly controls all meters in its subtree. Each domain is associated with a single meter.

From the point of view of security, meters provide a valuable service since they can be used to ensure that critical processes can receive a certain percent of overall CPU time, easing one form of denial of service attack. Meters could also be used to minimize covert channels through scheduling routines, for instance by associating all domains with a particular “label” to a single subtree of the meter hierarchy.

7.1.4 Space Banks

Space banks are not actually kernel objects, but are domains which control and record allocation of kernel resources (keys and nodes). Like meters, space banks are arranged in a tree, with the root space bank having control over all kernel resources. Each domain is associated with a single space bank which is consulted any time the domain requests kernel resources.

Like meters, space banks can be used to address availability concerns by ensuring that critical domains have sufficient kernel resources (and similarly, limiting the resources available to untrusted domains). This is a distinct contrast to some systems which allow processes to consume kernel resources until all resources are exhausted, at which time no process is able to allocate further (and the system often crashes).

Space banks can also minimize or even eliminate covert channel concerns through exhaustion of kernel resources through preallocation of those resources.

7.1.5 Factories

The KeyKOS *factory* mechanism is an infrastructure which involves several domains and protocols to build new domains whose maximum privileges (i.e., the maximum set of keys it can gain access to directly and indirectly) satisfy some known properties specific to the factory being used.

Factories were apparently conceived originally to solve a specific problem of mutual suspicion between the provider of some service (program) and a customer of that service. The provider of the service (known as the *builder*) does not want their program to be stolen or otherwise used without authorization. The customer of the service (known as the *requestor*) does not want the information which they provide to the program to be disclosed. The builder does not trust the requestor not to steal the program and the requestor does not trust that the program will not disclose its data.

This is a difficult problem to solve with a typical operating system that has a single owner for a program and all data connected with the program. The KeyKOS answer is to introduce a trusted middleman known as a factory. Both parties trust the factory because it is a simple service provided by the operating system. The purpose of the factory is to engage in a protocol to initialize a domain to an initial state in which both parties are confident that their interests have been protected.

According to [3], "understanding factories is crucial to a real understanding of KeyKOS". Unfortunately, of the documents studied for this evaluation, only the patent document[9] provides any real detail about factories. Rather than repeating the information which can be found in that document, we simply make some observations about the mechanism as recorded in the patent:

- Though described as a simple mechanism, creation of a new factory domain requires interaction of at least nine domains in addition to the KeyKOS kernel.
- The trust relationships between these domains are never explained. It is unclear which domains are meant to be trusted by the builder and which by the requestor, and for what properties the domains are to be trusted.
- There is no description of how the domains involved in creating a factory authenticate their identities to each other.

Ultimately, there is no explanation of why builder and requestor should believe that their respective needs have been met. Moreover, given the complexity of the protocols, analysis that these needs have been met is likely to be complicated.

7.2 KeySAFE

KeySAFE is an architecture for supporting mandatory access control (MAC) and discretionary access control (DAC) policies on top of the KeyKOS kernel. As a platform for supporting multilevel security (MLS) policies, KeySAFE was targeted to meet the high B-level requirements of the TCSEC [19].[12] This section contains an evaluation of KeySAFE as an architectural framework. Again, due to lack of details in the documentation considered for the report, it is impossible to evaluate the particular implementation of this architecture.

[12] According to [12],, "although a B3 rating is attainable, Key Logic initially intends to pursue a B2 level rating".

Though KeySAFE is generally described in terms of an MLS policy, its documentation also emphasizes that KeySAFE can support a wider range of policies without fundamental structural changes to the architecture. The assessment in this section will therefore focus on KeySAFE's ability to support a range of security policies in addition to the MLS and DAC policies prescribed by the TCSEC.

As discussed in Section 3.2.4.1, it has been argued that capability systems cannot support information flow policies such as MLS without enhancements to perform additional access controls whenever

- Capabilities are received by a domain, or
- Capabilities are used by a domain.

In their desire to support mandatory policies, the developers of KeySAFE reject the possibility of additional access controls when capabilities are used, based upon the following arguments:

- Consulting a separate access control policy when invoking a capability violates the fundamental purpose of a capability, which is to represent an allowed access.
- Capabilities should be used by a domain many times though they are received only once. Therefore the performance impact is less if the access control policy is consulted only when capabilities are received.[13]

Therefore the KeySAFE architecture provides additional access controls when capabilities are passed between domains. However, a key observation which is reflected in the KeySAFE architecture is that it is not actually necessary to explicitly control every transfer of capabilities between domains. For instance, if two domains share the same level in an MLS policy, the policy cannot be violated simply by passing keys between the two domains. A goal of the KeySAFE architecture is to provide separation between domains in such a way that passing of capabilities between domains with the same label can be accomplished without intervention, even intervention to verify that the labels are the same.

The KeySAFE architecture, with a particular emphasis on the combining of explicit and implicit access controls, is described in the following two sections. First, Section 7.2.1 describes how the architecture can be used to provide total isolation between domains with different security attributes. Then, Section 7.2.2 describes how the architecture expands to support interaction between domains in a completely controlled manner. Finally, Section 7.2.3 summarizes the specific features of the KeyKOS kernel which are relied upon in the KeySAFE architecture and which must exist in other capability systems to support KeySAFE.

7.2.1 Total Isolation

The fundamental aspect of the KeySAFE architecture is the partition of domains and kernel objects into a collection of *compartments*. Each compartment is isolated from the other compartments and from the KeySAFE TCB by its own *guard* domain. This basic architecture is shown in Figure 1. The four elements of the architecture are:

[13] While the purpose of this report does not include discussions of performance impacts of the various choices, it is hard to resist answering the use of performance as a justification for this decision. This argument is apparently based upon an implicit assumption that the performance cost of additional access control when capabilities are received is equivalent to the cost of additional access control when capabilities are used. There is no reason to believe that this is necessarily the case. For instance, suppose the capability represents an object in some server, and the security policy for that object is also represented in the server. Then the server may be able to very quickly consult the security policy when the capability is used, but for some other entity to consult the security policy when the capability is passed would require that entity to communicate with the server, which may be very costly.

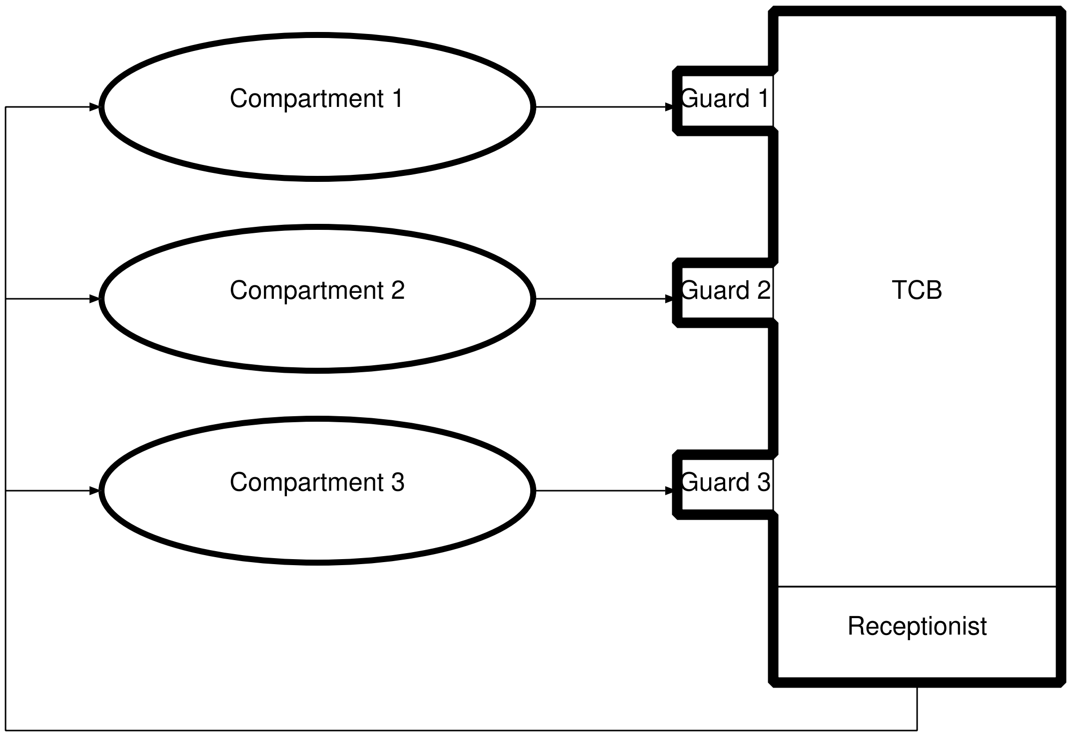

Figure 1: KeySAFE Architecture

Compartments A compartment is a collection of domains and other kernel objects which are allowed to interact freely without the mediation of the TCB.

Guards Each compartment has an associated guard domain which is the unique entry point from the compartment to the TCB. The guard records the security label for the associated compartment and is a proxy for communication from the compartment to the TCB, as described in Section 7.2.2.[14]

Receptionist The *receptionist* is responsible for processing login requests, and for connecting a successfully logged in user to the particular compartment requested by the user.

TCB The KeySAFE TCB consists of the KeyKOS kernel and some collection of external servers. The guards and the receptionist are also part of the TCB.[15]

It is important to recognize that the labeling of domains and objects in KeySAFE is implicit through the guard (and to a lesser extent, through the receptionist). There is no way to view a domain or object in isolation and determine what compartment it lies within and therefore what label is associated with it. Compartments are conceptual only and are not explicit objects. This is a considerable obstacle to overcome in any security analysis since this very fundamental relationship cannot be easily characterized in terms of the system state.

[14] It is unclear from the documentation whether there is actually a distinct guard domain for each compartment or whether there is just a distinct key for each compartment, to a single guard domain. The specific implementation is unimportant in the evaluation.

[15] The domains which make up the TCB are described on pages 14-18 of [12].

The configuration in Figure 1 suggests a state invariant which is intended to provide total isolation between compartments as long as the guards provide no response when their gate keys are invoked:

> **Total Isolation Invariant** The only key held within any compartment and representing an object outside of the compartment is a gate key to the compartment's guard domain.

This invariant only restricts keys that actually represent objects. As mentioned above, some primary keys do not represent objects but rather represent privilege to invoke a particular kernel or TCB routine. These keys are said to represent *stateless service providers* because the result of invoking such a key is independent of the system state.[16]

The remainder of this section considers how it might be demonstrated that this state invariant actually holds and is sufficient to isolate compartments. Note that this approach and the state invariant itself is not taken directly from any KeySAFE documentation, though the documentation strongly suggests this approach.

7.2.1.1 Proof of the Invariant

To prove that the invariant holds requires two steps. The first step is to show that the invariant holds in the initial state, and the second step is to show that if it holds in one state, then it holds in the next state.

For the initial state we will actually consider what happens during creation of a new compartment. A new compartment will only be created when a login request is made to a compartment which does not currently exist. In this case, the receptionist creates an initial domain for the compartment (some sort of command interpreter such as a shell) and the compartment's guard. The receptionist gives a gate key for the guard to the initial domain and then passes control to this initial domain.

So it is necessary to verify that all object keys held within a newly created compartment represent objects within that compartment (other than the guard key). It is also necessary to verify that the only key to any object within the new compartment and held outside of the compartment is the receptionist's gate key to the initial domain.[17] Verifying these statements is beyond the scope of this report, but in all likelihood they should not be difficult to verify since the receptionist controls creation of a new compartment.

The much more interesting step in the proof of the invariant is the induction step: if the invariant holds in some state, then it holds in the subsequent state.

There appear to be two ways in which the invariant could cease to hold. One way is for an object to move from one compartment to another. The second way is for a compartment to gain a key it did not previously hold and which references an object in a different compartment.

The first possibility is difficult to define since inclusion of an object in a compartment is implicitly defined through ownership of keys representing the object (and ultimately through ownership of guard keys). So all that it means for an object to move from one compartment to another is for a key to the object to become part of another compartment.

[16] Proof that so-called stateless service providers are really stateless is also a necessary part of the analysis of KeySAFE. This is likely to involve some complications. For instance, factories are given as an examples of stateless service providers, but factories are clearly not stateless. It is more likely that what is really meant is that stateless service providers are meant not to provide any "meaningful" state, and this claim would require detailed analysis to confirm.

[17] Actually, it is unclear whether this is the only key. For instance, the receptionist may well hold a domain key for the initial domain. In any case, this collection of keys must be enumerated and verified.

This case can therefore be combined with the second case so that the goal of the induction step is to prove that there is no way for a compartment to gain a key it did not previously hold and which references an object currently or previously (i.e., in the immediately preceding state) referenced by a key in a different compartment.

Since keys are managed within the kernel, a compartment can only gain keys due to some action involving the kernel. In particular, the action must involve a key invocation, possibly by the kernel itself. Therefore to prove the induction step we must consider the results of all possible key invocations.

One way to organize this analysis would be to consider the three possible invokers of a key:

- A domain within some compartment
- A domain within the TCB, including the guards and receptionist but not the kernel
- The kernel itself

And to consider four possible key types:

- A gate key to a domain within the TCB
- A gate key to a domain outside of the TCB
- A primary key to an kernel object
- A primary key to a stateless service provider

This organization leads to 12 cases to consider. A couple of these cases can be disposed of easily. For instance, if a domain within some compartment invokes a gate key to a domain outside of the TCB, both domains must be in the same compartment and no keys can be passed across compartments. However, most of the cases require inspection of all possible invocations satisfying that case. This could potentially be quite complicated and it is also possible that inspection of the various cases will lead to additional constraints being added to the state invariant in order to capture other relationships between objects.

7.2.1.2 Sufficiency of the Invariant

Ultimately, the statement of total isolation is an extension of the state invariant to claim that not only are keys not available for sharing between compartments, but neither is data. With the exception of covert channels, it appears to be true that passing of data requires a key invocation and therefore requires study of all of the 12 cases required for the proof of the invariant. Since data can be passed in many more ways than keys, the analysis of the sufficiency of the invariant is likely to be much more complex than the proof of the invariant itself.

7.2.2 Mediated Isolation

Section 7.2.1 describes how KeySAFE can provide total isolation between compartments if the guards do not allow any traffic to pass through them. However, as mentioned above, and often in the KeySAFE documentation, total isolation is not the goal. Instead, the goal is to allow controlled interactions between compartments.[18]

[18] The term *mediated isolation* actually comes from Trusted Mach, but it also used here since the basic concept is essentially the same.

To provide for such interactions, domains can request to export keys from their compartment to a name space within the TCB. The domain provides the key to be exported in a request to the TCB which is accessed through a gate key to the guard for the domain's compartment.

The guard forwards the request to the TCB, tagging the request with the label associated with this compartment. Note that each guard labels all requests the same, relying upon a requirement that the gate key for the guard is never made available outside of the compartment. This is the only source labeling on messages within KeySAFE.

When the TCB receives a request to export a key from a compartment, the user currently logged into the exporting compartment is then queried for a name to be associated with the key and for an optional access control list (ACL).[19] The TCB then stores the key at the requested location of the name space along with the ACL and the label of the exporting compartment.[20]

To retrieve a key stored in the TCB name server, a domain from another compartment makes a request through its guard, providing the name of the object and optionally a set of requested access rights. The guard, as always, simply labels the request and forwards it to the TCB. The reference monitor within the TCB now compares the request with the stored key to determine whether or not to grant the requested access. It is unclear exactly what decision logic is used, but ultimately the decision is based upon the following information:

- The type of key, including access rights and the class of the object represented by the key,
- The labels of the exporting and importing compartments,
- The ACL stored with the key in the TCB namespace, and
- The particular access rights requested.

If the reference monitor determines that the requested access can be granted, it does not simply return the key to the requesting domain. Instead, it creates a front-end object (a kind of proxy domain) and returns a gate key for the front-end object.[21] This is illustrated in Figure 2.

The front end object can potentially serve many purposes, including:

- At any time, all connections between compartments can be determined by listing all front end objects.
- A key which has been imported to a compartment can be revoked by the TCB by destroying the front end object.

[19] A couple of questions immediately come to mind. First, what if there is no user currently logged into the compartment. Does this mean that keys can't be exported? Second, why require the user to get involved at all? The only apparent benefit is to mitigate the covert channel through use of shared names. Page 12 of [12] also states that "alternatively, the reference monitor could assign the name", but no further elaboration is provided.

[20] In [12] it states:

> Once the key to a KeyKOS object is exported from a compartment and becomes a named, TCSEC object, its original key and all copies are invalidated. Future access to the object by the originating compartment must also occur through an auditable, rescindable path which must be established for that compartment in the same manner as any other "importing" compartment.

Unfortunately, there is little elaboration about how this is accomplished and how users of the existing keys are expected to react to invalidation of the key. The motivation for this action is also unclear.

[21] There's an inconsistency here which is not resolved in the documentation and which might have an effect on the assessment. The documentation states that once a key is imported to a compartment, the importing compartment makes all invocations on the key through the front end object. But suppose that the imported key is a primary key, for instance a page key. The importing compartment will attempt to use its key as a page key in kernel requests. How will the kernel react to being given a gate key (i.e., the key to the front end object) when it is expecting a page key?

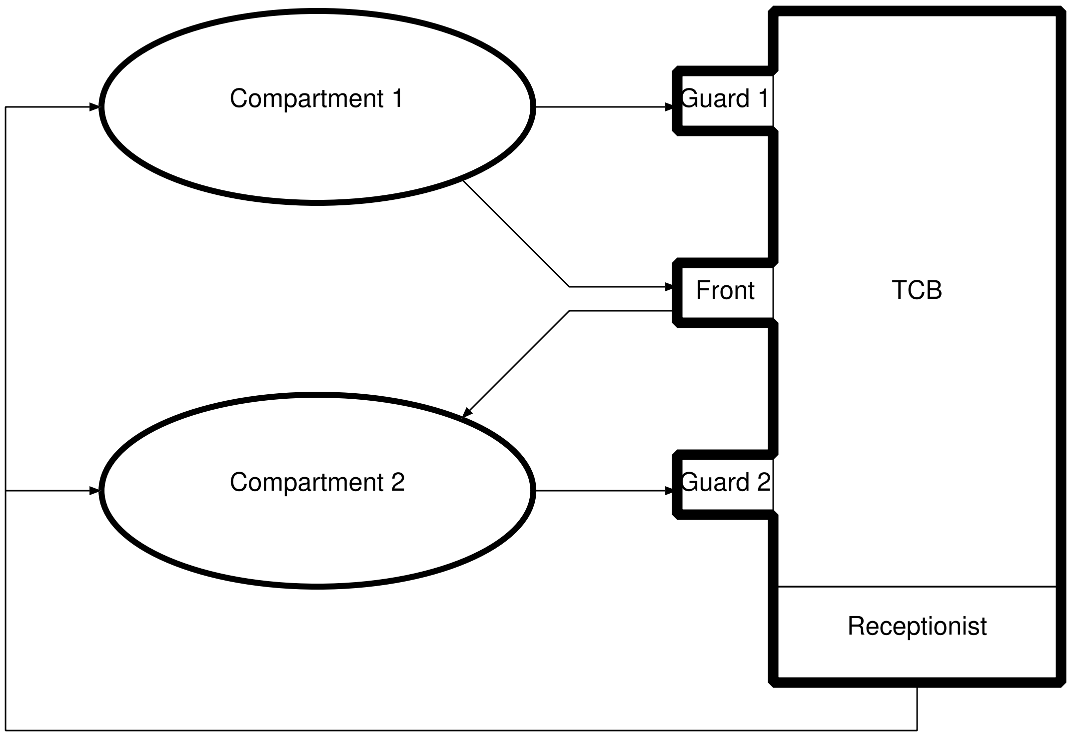

Figure 2: Front-end Objects Provide Mediated Isolation

- Auditing of interactions between compartments can be performed by querying front end objects.
- Filtering of interactions between compartments can be performed within front end objects.

The KeySAFE documentation specifically mentions all but the last of these purposes, though there are no details describing any specific support provided for audit and access revocation.

Just as Figure 1 suggests a state invariant met by the system, Figure 2 suggests an expanded version of that invariant:

> **Mediated Isolation Invariant** The only keys held within any compartment and representing an object outside of the compartment are gate keys to the compartment's guard domain and gate keys to front end objects as authorized by the TCB.

The remainder of this section considers how this invariant might be proven.

The initial state condition is less restrictive than the initial state condition for the Total Isolation Invariant, so it does not need to be considered further here.

Once again, it is the induction step that is most interesting. The additional difficulty in the induction step is that we have an additional kind of key invocation which is possible by domains within a particular compartment, invocation of a gate key to a front end object. Also, these additional keys can be passed as parameters in other key invocations.

The main concern in this architecture is the typical concern about capability based systems; that a single granted capability can be used to gain other capabilities. In KeySAFE, this is instantiated by the concern that keys can be acquired via communication through a front-end object.

One apparent means of controlling the propagation of capabilities in this way is to rely upon the front-end objects to filter out any keys which one compartment may attempt to pass to another. The KeySAFE implementation uses a different strategy, which is to prevent most types of keys from being shared in the first place. First of all, gate keys are not available for sharing between compartments (except for certain gate keys to domains within the TCB). This is essential, since gate keys provide a direct means for passing additional capabilities from one domain to another. Two particular kinds of TCB gate keys which cannot be shared are keys to front-end objects and guard keys.

Primary keys are available for sharing only if they represent stateless service providers within the kernel or "pure storage objects", such as segments.

This discussion demonstrates that a domain can only hold the following kinds of keys:

- Primary keys representing kernel objects within the domain's compartment.
- Gate keys for other domains within the compartment.
- Gate keys to the TCB as specifically allowed by the TCB. In particular, a gate key to a guard for the proper compartment and gate keys to front-end objects identified as originating with the compartment.

7.2.3 Applicability of KeySAFE Architecture to other Capability Based Systems

Since the KeySAFE mechanisms are built on top of the KeyKOS kernel without requiring changes to the kernel, it is interesting to consider whether the same basic mechanisms can be applied to other microkernels providing a capability system. This section discusses two requirements KeySAFE places on the kernel and which are of particular interest in this report because they are not met by all of the microkernels under consideration.

7.2.3.1 Distinction Between Data and Capabilities

The KeySAFE system is built around the concept of controlling the set of capabilities which can be held within any particular compartment. In particular, the intent is to ensure that all capabilities shared between compartments are mediated by the TCB. This claim is verified by demonstrating that the capabilities currently held by a compartment cannot directly be invoked to gain new capabilities. However, KeySAFE does allow capabilities in a compartment to be invoked to gain access to data without additional mediation. Therefore capabilities must be distinct from data.

It would be possible to extend the KeySAFE mechanism to provide proxying over all key invocations leading outside of a compartment. If this were done, then the proxies would filter each key being passed to ensure that it is an appropriate key to share. The same could be done with data.

There are two serious drawbacks to this approach. The first drawback is the possibility of a severe performance penalty that could result from filtering all data passed from one compartment to another. The other drawback is that it would still be necessary for the proxy to identify whenever a capability existed in the data, and depending upon how capabilities are stored as data, this may not be possible either (e.g., Amoeba).

7.2.3.2 Distinction Between Types of Capabilities

The requirement for distinction between data and capabilities is necessary for any goal of isolation. However, a practical system of mediated isolation also would seem to require some way of distinguishing between different types of capabilities. For instance, an essential requirement of KeySAFE is that gate keys to domains outside of the TCB are never available for sharing between compartments. If the TCB is unable to distinguish such capabilities, then the requirement cannot be met without forbidding sharing of all capabilities.

More generally, the ability to distinguish between types of capabilities allows for more granularity in the types of policies which are supportable. For instance, sensory keys are important because of the ability to support read-only requirements.

Compared to a basic capability system like Mach provides, this extra structure in KeyKOS is quite valuable. For instance, it is impossible (or at least very difficult) for a Mach task to determine whether a capability references a kernel object or an object in some other task. This means that any KeySAFE like architecture placed upon Mach without kernel enhancements is going to provide very limited granularity in access control and therefore will support a correspondingly limited range of policies.

7.3 Security Assessment

This section provides an assessment of the ability of KeySAFE to be configured to meet a range of security policies. The changes made to the system for each security policy are limited to the reference monitor described in Section 7.2.2. While the internal structure of the TCB is not well documented, there is no obvious limitation to the independence of the reference monitor from the rest of the TCB.

7.3.1 General Characteristics

MAC/DAC The reference monitor makes its decisions explicitly based upon MAC labels associated with each compartment and upon an discretionary ACL, so MAC and DAC are directly supported by KeySAFE.

In the specific KeySAFE implementation of MLS and DAC, the label associated with each compartment consists of a sensitivity level and user identifier. Exportation of a key from one compartment A to another compartment B is allowed only if the MLS and DAC rules are both satisfied:

MLS If compartment A and compartment B are at the same level, then the key must be a sensory key or a write key to a segment or page. If compartment B strictly dominates compartment A, then the key must be a sensory key.[22] If neither relationship holds then no key sharing is allowed.

DAC The ACL associated with the key when exported from compartment A grants access to the user associated with compartment B.

The only significant additional analysis that would need to be done to demonstrate that these rules are sufficient to meet the MLS policy is to demonstrate that sensory keys satisfy the desired property that they are truly one-way keys and cannot lead to any flow of keys or information in the "reverse" direction.

[22] Actually, not all sensory keys may be allowable. There is an implication that some sensory keys, such as sense keys for arbitrary nodes, are not available for sharing.

Information Flow-Transitivity Though the implementation of KeySAFE for MLS policies implies that the policy is transitive, the general design makes no such requirements. In fact, the inability to export a key without permission of the reference monitor provides explicit support for intransitive security policies.

As a particular example, KeySAFE has the ability to support trusted pipelines with the requirement that data can flow from compartment A to compartment B only if "filtered" through another compartment C.

Information Flow-Covert Channels The design of the KeyKOS kernel is intended to minimize the existence of covert channels. The fact that all kernel state is potentially stored in objects which implicitly belong to one or more domains could minimize storage channels through "unexpectedly" shared state. Meters and space banks provide a means for limiting channels through control of CPU time and memory allocation.

However, we are unaware of any published results of any significant search for covert channels in the kernel or any of the KeySAFE domains.

Mutual Suspicion The factory mechanism (Section 7.1.5) was implemented precisely for this reason. It is notable however that the mechanism is quite complex.

7.3.2 Least Privilege and Granularity

In general, KeyKOS provides rather fine-grained control over kernel operations through the capability system, though KeySAFE provides very coarse grained MAC mechanisms.

Granularity of MAC Attributes The KeySAFE mechanisms are most appropriate for policies which have a very limited number of distinct attributes and hence a small number of compartments. Though conceptually there is no limitation on compartments, there are some significant practical concerns:

- If there are a large number of compartments there will likely be a large number of transfers of information between compartments. For instance, a trusted pipeline passes data between domains with distinct security attributes. The proliferation of front-end objects and the indirection through them is a potential cause for performance concerns.
- In KeySAFE, new compartments are only created as a result of a login request. To implement trusted pipelines there are often many compartments which have no user session associated with them.
- The requirement that a user reauthenticate whenever changing compartments can be cumbersome. Again, this is not primarily a security concern and better mechanisms for switching between sessions should be possible.

Protection Boundaries/Unique Identifiers A protection boundary is a compartment, which may contain many domains and objects.

There is no direct support for establishing a unique compartment for each object, and this could be difficult to support because of the concerns about large numbers of MAC attributes.

Privilege Levels/Least Privilege As stated above, at the kernel level KeyKOS provides a relatively fine grained control through capabilities. Outside of the kernel, we have seen no evidence of support for similar fine-grained controls.

For example, there appears to be two levels of privilege in KeySAFE; domains are either in the TCB and entirely trusted or outside of the TCB and untrusted. Consider an attempt to create a "partially" trusted server outside of the TCB which exports its managed objects as gate keys. According to the reference monitor logic described in Section 7.2.2, the only distinction which the reference monitor will make between such gate keys is in the DAC permissions.

Run-time Least Privilege Kernel capabilities provide one means of providing run-time least privilege. Again, this is largely limited to kernel objects.

7.3.3 Characteristics of Dynamic Policies

Since all communications between compartments are through front-end objects, with small changes the TCB could be implemented to destroy front-end objects during a policy change. This provides support for relinquishment sensitive and retractive policies, though it does not directly support aborting any operations which are in progress.

The implicit labeling of objects within a compartment makes it difficult or impossible to support two desirable features of dynamic policies:

- It is not possible to locate all kernel objects with a particular label and destroy them.
- It is not possible to relabel a single object out of a compartment. Any relabeling must be done on the entire compartment. Again, with the practical limitations on the number of compartments, this is a significant weakness.

7.3.4 Active Policy Characteristics

Availability Meters and space banks provide support for ensuring that CPU and memory resources are not monopolized.

Accountability The use of front-end objects creates a bottleneck between compartments which is potentially very useful for policies which require some kind of content filtering or audit of data and keys passed between compartments.

However, there is also a serious limitation in the ability to audit in KeySAFE, due to the fact that labeling of TCB requests is possible only if made through a guard or front-end object. The kernel has no way of placing a label on a request made directly through a primary key, so auditing of kernel requests is not possible without adding indirection to kernel requests as well.

7.3.5 Failure Ramifications

Another problem caused by the failure to perform access control over all requests is that a single security failure can multiply without detection. For instance, an improperly granted capability may be used to gain other capabilities, and the system has no way to detect that any of the capabilities have been used incorrectly.

7.4 Assurability Assessment

We have two significant concerns about the assurability of KeySAFE. Both arise primarily from the use of implicit labeling and the failure to provide access controls over all operations.

- The difficulty of the proof of the invariants in Sections 7.2.1 and 7.2.2.
- The inability to identify all of the privileges granted in the system in any particular state. While the reference monitor has the ability to easily identify all front-end objects, this does not identify all capabilities.

7.5 Summary

The developers of KeyKOS and KeySAFE were obviously committed to security as one of the fundamental goals of the system. Moreover, the interest in a range of policies is unique to the systems considered in this report, with the exception of DTOS. The proposed solutions are quite creative, and may have in fact pushed the limits of a capability system to satisfy MAC policies about as far as it can be pushed. KeySAFE has certainly pushed the limits farther than some had thought possible.

Nonetheless, we still have strong reservations about security and assurance for KeySAFE, ultimately due to the lack of explicit labeling of objects. As already discussed, this lack severely complicates analysis of the Total Isolation Invariant. It also can be expected to complicate analysis of other properties as well. For instance, the factory mechanism is a particularly graphic example of the complexities that need to be added to solve a problem that can be quite easily solved if explicit labeling is provided.

Finally, there is little defense-in-depth in KeySAFE. There are many examples in the system where a single implementation error can lead to dramatic failures in security, with no way to detect the error through auditing and often ways for one error to propagate and lead to an entire sequence of security failures. For instance, a single key propagation in violation of the security policy can lead to an entire sequence of such violations, with no possibility of an audit trail.

Section 8 Trusted Mach Assessment

Trusted Mach (TMach) is a multi-level secure operating system consisting of several trusted servers running on top of a modified version of the MK++ kernel. TMach was designed from the start to provide multi-level security, and is intended to be evaluated at the B3 level of the TCSEC [19].

Our discussion of TMach begins by considering the MK++ kernel and then the TMach servers which comprise the security architecture.

8.1 MK++ Kernel

The MK++ kernel[23] was developed to provide the familiar Mach interface with a highly layered, object oriented design. Some changes were made to the interface when necessary to support the security requirements of the TMach project, but these do not change the basic Mach flavor of the interface.

As is typical of many features provided by microkernels, MK++ provides access control mechanisms while relying upon external servers to define the policy being enforced by these mechanisms. In the following sections we discuss five mechanisms which can contribute substantially to the security of a system built from MK++. We also discuss the layering of the kernel. Though not a security mechanism, the layering is intended to increase confidence in the correctness of the kernel and hence in the security mechanisms in particular.

8.1.1 Layering in MK++

The internal design of the kernel is heavily layered, apparently driven by the system architecture requirements at the B3 level of the TCSEC. According to the MK++ Kernel High Level Design [23], the kernel is broken into 13 subsystems organized into 11 layers (two layers consist of two subsystems each). While we do not have access to source code and therefore cannot evaluate how rigorously this layering is actually enforced, it is quite clear from the design documentation that the layering and structuring of the kernel has been given considerable attention.

What is less clear is whether the layered design will simplify assurance as much as it could have. Assurance is generally aided more by dividing a system into distinct modules at the interface level, creating "vertical slices" through the system rather than horizontal layers. When there are required interactions between these modules, layering then serves the purpose of simplifying the analysis of these interactions, and in particular of removing circularities in the requirements of each module.

However, layering by itself is of unclear value, and taken to extremes it may in fact do more harm than good since it may force logically distinct modules to share a common layering architecture. We do not have any evidence to suggest that this occurs in MK++, but a design

[23] Though the TMach kernel is a modified version of the MK++ kernel, the MK++ documentation [21, 22, 23] is still applicable and therefore this section refers to the MK++ kernel and TMach kernel interchangeably.

which modularized the kernel at the interface based upon the different object classes would have provided much clearer assurance benefits.

8.1.2 Capabilities

The primary protection mechanism in MK++ is the capability system. MK++ capabilities are maintained by the kernel, are unforgeable and can be passed between tasks only through the kernel.

The MK++ kernel interface provides operations on several object types. Some object types are associated with multiple types of capabilities, each representing different allowed accesses to an object of that type. With the exception of a capability to receive from a message queue, there may be multiple copies of any capability.[24]

In two special cases discussed in Sections 8.1.5 and 8.1.6, the underlying objects themselves maintain further access control information and therefore can be considered to refine the capability system further.

The means of protecting capabilities vary with the type of object that a capability represents. Table 1 lists all kernel object types which are protected by capabilities. For each type, the table lists the following:

- The number of objects of this type which can exist on a particular host.
  - One. There are several different types of capabilities to the host, each representing a different collection of interfaces.
  - Some. This means that there is a fixed number of such objects, determined by hardware.
  - Many. This means that there can be many such objects, created and destroyed while the system is operational, and limited only by memory resources.
- The different capabilities associated with the object type. For several types of objects, there are distinct *control* and *name* capabilities. A control capability for an object generally provides total control over the object, whereas a name capability usually only allows the holder to read some attributes of the object.

  A blank indicates that there is only one type of capability for each object of a particular type.
- Some indication of how each type of capability can be obtained, and hence, how they must be protected. This column indicates how the kernel will explicitly grant a capability to any of these object types. In addition, any task which has a capability can explicitly make that capability available to other tasks. And any task which has a capability for a second task can request that the kernel provide a copy of any capability held by the second task.
  - Initial task. The initial task is given a host bootstrap capability. By presenting this capability to the kernel, additional capabilities can be gained. Any capability marked this way in the table can be received only through capabilities directly traceable back to the initial task.

    Figure 3 depicts how each of these capabilities is obtained leading from the host bootstrap capability.

[24] Note that throughout this chapter we will make no distinction between send and send-once rights to a message queue. There is arguably a security benefit with send-once rights, but in the fundamental architecture they are indistinguishable.

With the exception of ledgers, all of these objects exist regardless of the number of capabilities for each object. Ledger objects are created by invoking a request using the host control capability.

- Initial task and others. This applies to one type of capability, the name capability for a ledger object. This capability is returned when a new ledger object is created, and is therefore protected from the initial task like the ledger control capability. However, this capability will also be given to any task drawing resources from the ledger.
- Creator. The creator of an alarm and or message queue object obtains a capability to the object when the object is created.
- Creator and others. The creator of a task or thread object obtains a capability to the object when the object is created. In addition, the kernel will also provide capabilities for these objects to clients holding related capabilities.

  In particular, a task capability will be given to a client holding a control capability for the processor set on which the task executes, or to a client holding a capability for a thread executing within the task. A thread capability will be given to a client holding a capability for the task in which the thread executes.
- Memory manager. The kernel associates each memory cache object with a particular message queue known as the *memory object port*. The kernel will grant a capability for a memory cache object to any task which has a capability for this message queue. It is expected that the memory manager will protect capabilities for the memory object port.
- Unprotected. The kernel provides these capabilities to any task with essentially no restrictions.

Object type	Quantity	Capability	Protected By
Host	One	Bootstrap	Initial task
Host	One	Paging	Initial task
Host	One	Security	Initial task
Host	One	Device Master	Initial task
Host	One	Control	Initial task
Host	One	Name	Unprotected
Clock	Some	Control	Initial task
Clock	Some	Name	Unprotected
Processor set	Some	Control	Initial task
Processor set	Some	Name	Unprotected
Device	Some	—	Initial task (also see Section 8.1.5)
Ledger	Many	Control	Initial task
Ledger	Many	Name	Initial task and others
Alarm	Many	—	Creator
Message queue	Many	Receive	Creator
Message queue	Many	Send	Creator
Task	Many	—	Creator and others
Thread	Many	—	Creator and others
Memory cache	Many	—	Memory manager

Table 1: Protection of MK++ Object Capabilities

The fact that capabilities are expected to be protected in at least six different ways is a significant concern. Analysis of a system built upon MK++ will likely require at least this many

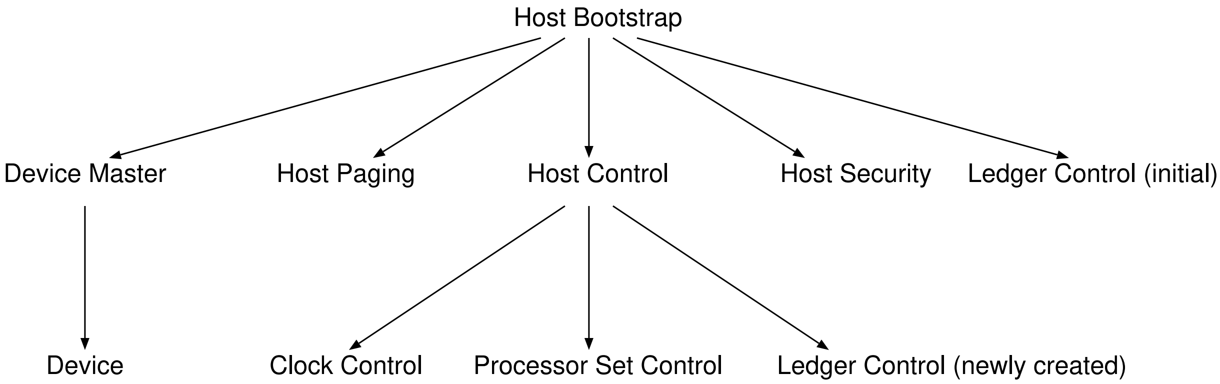

Figure 3: TMach Capabilities Protected by the Initial Task

different cases when demonstrating that the capabilities are in fact protected as required by that system. In the case of TMach in particular, the capabilities protected by the "initial task" are in fact protected by three different servers, further increasing the complexity of the analysis.

8.1.3 Security Identifiers

MK++ assigns a security ID to all tasks to provide an unforgeable universal identifier. The kernel itself does not interpret the security ID, although in some cases (described in Sections 8.1.5 and 8.1.6) it does compare security IDs. Security IDs are not required to be unique and therefore do not necessarily uniquely identify tasks. Security IDs can be used by higher level entities to represent whatever information they choose about a task.

The purpose of the security ID is to provide a form of source authentication on messages, which are tagged with the security ID of the task which sent the message.[25] This can be used instead of relying upon complicated (and possibly unreliable) authentication protocols, as long as the security ID provides all of the authentication information required by the message recipient.

Assignment of the security ID to a task is dependent upon the way in which a task is created. If the task is created using the standard Mach `task_create` interface, the security ID of the child task is inherited from the parent task. MK++ also adds a new interface, `task_create_security_id`, through which a client can specify a particular security ID to assign to the new task. Privilege to use this request is granted to any client holding the host security capability. Once assigned, the security ID of a task cannot be changed.

Security IDs solve what can be a difficult problem of authentication. The way in which they are used does however raise a couple of issues:

- Transparent interposition on interfaces is much more difficult, because the interposer must have the same security ID as the task on which it is interposing. One limitation therefore is that a single task cannot transparently interpose on tasks with different security IDs.[26]

[25] Messages sent by the kernel are tagged with a well-known constant, `KERNEL_SECURITY_ID`.

[26] In [1], the authors mention the possibility of privileged tasks having the ability to specify their identity when sending a message. This apparently is not however part of MK++ or TMach but part of proposed extensions for the Triad program. This privilege minimizes the impact on interposition but also removes the unspoofability of the source authentication.

- There is a general principle in cryptographic protocol design that no entity should ever apply its signature to data which the entity is unable to interpret. There is an analogous risk here, because the security ID is applied to all messages sent by a particular task, and a task may not be able to interpret the contents of all messages it sends.

  For example, consider a server which provides a service to write uninterpreted data to some object (e.g., a file) and to read data from the object by returning the data in a message. A client of this server, by writing particular data to that object and later reading it back, may be able to cause the server to send any message chosen by the client. Moreover, since replies to client requests are sent to a message queue specified by the client, the client can direct the message to any message queue for which the client has a capability.

  Without the sender security ID attached to the message, this is not an issue, because the client could certainly construct the reply message itself and send it using any capability it has. But with the addition of the security ID, the client is no longer able to construct an identical message by itself. So for instance, if the recipient of the reply message interprets the message as a request for which the server is privileged but the client is not, then the client may be able to perform an operation which it would otherwise be unprivileged to perform.

  The only apparent mechanism provided to avoid this risk is the use of the *message ID* field of a message header. The intended use of this field is to uniquely identify how the contents of a message are to be interpreted, generally by indicating that the message is a particular request or a reply from a particular request. This use is not mandated however, and it is unclear whether message IDs are used this way throughout TMach and whether uniqueness of message IDs is enforced across all servers.[27]

  Another possible means of minimizing this risk would be to allow a task to choose whether its security ID should be tagged to each message that it sends. When sending server replies or other uninterpreted data, a task could elect to refuse to provide authentication. The kernel could be similarly configured. The difficulty with this solution is finding a general way to decide when a message should be “signed” and when it should be anonymous.

8.1.4 Resource Limitations

MK++ provides two mechanisms for limiting the resources available to some task.

The most significant mechanism is provided by ledgers, kernel objects which control consumption of memory resources. There are four values associated with a ledger:

- The number of bytes of wired kernel memory available through the ledger.
- The percentage of total wired kernel memory available through the ledger.
- The number of bytes of swap space available through the ledger.
- The percentage of total swap space available through the ledger.

Each task is associated with a particular ledger upon creation, and all resources consumed by the task are counted against the resources available on that ledger.[28] It appears that message

[27] In Mach 3, the reply to a kernel message is given a message ID that is incremented by 100 from the message ID on the request. This actually gives a client task the ability to get the kernel to send a message with virtually any message ID. However, since most message IDs are invalid in kernel requests, the body of these reply messages will usually contain error codes. Still, it is one case where the client even has the ability to affect the message ID.

[28] The only exception is that message queues can be created from different ledgers using `ledger_create_port` and providing the name capability for a particular ledger.

queues are the only kernel objects other than tasks to be associated with ledgers. Ledgers can potentially provide the two benefits identified for KeyKOS space banks (see Section 7.1.4), minimizing attacks on availability and minimizing resource exhaustion covert channels.

However, there are a couple of potentially significant shortcomings of the ledger model:

- It is unclear which ledger is charged for all of the various ways in which kernel memory can be allocated. For instance, consider a message on some message queue. If charged against the ledger for that message queue, then it may be possible for a task to use up resources that should belong to another, possibly more trusted task.
- It is unclear how the ledger mechanism actually limits resource usage, and the information that is available is not particularly encouraging. For instance, page 257 of the Kernel Interfaces document [22] says the following:

  > The kernel sporadically checks all of the system ledgers. If the resource consumption associated with a ledger exceeds the specified entitlement and the ratio of system resources consumed exceeds the specified ratio, then the ledger is considered for destruction, destroying all objects bound to it.

  Compared to KeyKOS space banks, this is a particularly violent approach. Space banks are used to limit resource usage when resources are requested. If there are not enough remaining resources assigned to a particular space bank, then the request fails. However, existing clients can continue to execute as long as they do not require additional resources.[29]

Note that because of these problems, ledgers are not made available in TMach (see Section 8.2.4.2).

The other mechanism which can be used to provide resource usage limitations is the ability of a task to assign a limit on the amount of out-of-line memory and capabilities which can be received through a particular message queue. The amount of in-line memory is also restricted by a parameter to `mach_msg`. This ability is valuable because it prevents a sending task from consuming resources available to the receiving task. It does not appear to have any affect on the issue discussed above relating to the ledger charged for a particular message, since the limitations are only enforced when the message is actually received, not while it is in a message queue.

8.1.5 Device Control

Kernel devices are accessed through device capabilities. Device capabilities are created by the request `device_open`, which requires a device master capability. Inputs to the `device_open` call includes the following:

- The name of the device.
- A security ID which identifies the client which will be allowed to use the newly created device capability. This can be wildcarded to match all security IDs.
- A set of access rights to be granted to this client.

[29] While on the topic of comparing MK++ resource controls to KeyKOS, it is notable that there is no MK++ analogue to KeyKOS meters.

Each method on a device requires one or more of five possible access rights. Whenever one of these methods is called using a particular device capability, the kernel verifies that the task making the request has the same security ID provided when the capability was created and that the required access rights for the call were granted at that time.

The way in which this mechanism is used in TMach is discussed further in Section 8.2.4.3.

8.1.6 Virtual Memory Control

MK++ provides the usual separation of virtual address spaces and setting of page table bits. At the kernel interface, MK++ supports any combination of read, write, and execute privileges, however there is no indication that any implementation maintains a distinction greater than read and read/write at the hardware level.

Setting of page table bits is dependent upon the way in which memory is allocated. For anonymously allocated memory the bits are set completely at the will of the client allocating the memory.[30]

The setting of page table bits for mapped memory is much more interesting, and is the topic of the remainder of this section.

Allocation of mapped memory involves the kernel, an external memory manager task, and some client task in whose address space the mapping is done. There are three types of objects which are important in the determination of the correct permission bits.

- A *memory cache control object* represents a particular region of virtual memory. The kernel provides a capability for this object only to the memory manager, however, this is to protect the memory manager and not the kernel. A memory manager concerned about security should not pass its capability to any other task.
- An *abstract memory object* is a message queue providing the memory manager's representation of some memory. The memory manager holds the receive right, and passes the kernel a send right. A memory manager concerned about security should ensure that the send right is only made available to the kernel. The kernel itself will not pass its send right to any task.
- A *memory object representative* is a message queue used strictly as an unforgeable identifier to represent a particular security ID and the permissions to some memory granted to tasks with that security ID. The memory manager holds the receive right to this message queue, but send rights need not be protected at all.

Accessing memory provided by an external memory manager begins with some client task invoking a `vm_map` call on some task (possibly other than the client). One parameter to this request is a send right for a memory object representative.

When this memory is first accessed, the kernel and external memory manager engage in a protocol to initialize all of the necessary data structures. Of particular interest in this protocol is that the memory manager reports the security ID and permissions associated with the memory object representative provided in the `vm_map` request.

The protocol is detailed on pages 36 through 40 of [21], and there is no need to repeat the description here. A few observations about this protocol follow.

[30] It's not clear what is done for memory allocated anonymously as a side-effect of another operation, such as receiving a message. Presumably the default is read and write permission.

- In the Mach 3 VM system, there are certain virtual copy optimizations which can effectively result in a memory cache object moving from one task's address space to another. If such operations occur in MK++, then it would be necessary for the memory object representative to be consulted with regard to the new task.

  For example, in Mach 3 if one task having a memory object mapped at some region in its address space were to send out-of-line memory including this region to another task, and then deallocate that region, the second task would actually be given the mapped version of the memory. In this example, if the memory was a mapped file, then the second task would be able to modify the file itself rather than just a local copy of the file.

  If this behavior were provided by MK++, then the kernel would need to verify the security ID of the second task against the memory object representative when the memory cache control object was "transferred" to the second task. However, it appears that MK++ does not provide this same behavior. For example, out-of-line memory transfers always provide the second task with anonymously allocated memory, even if the sender immediately deallocates it.

- Another concern similar to the previous is the inheritance of memory from the parent task to the child task during task creation. The child task can inherit a mapped memory object from the parent even if no associated memory object representative exists matching the child's security ID. The kernel relies upon trusted user level servers to provide sufficient separation to prevent this from happening except when specifically allowable.

- The protocol relies upon the use of the host name capability to authenticate the kernel to the memory manager. With the addition of security ID labeling on messages, an alternative means of authentication would be for the memory manager to simply verify the security ID on the message. There is no discussion of why this mechanism is not used here.

  One difference between the two authentication mechanisms is that a task's view of the host name capability can be changed using a capability for the task. This makes the authentication protocol potentially spoofable. However, it also means that transparent interposition is possible.

- It is worthwhile to note that while the permission checking mechanism here appears to be similar to the standard access control model in TMach (see Section 8.2.2), there is a fundamental difference: the security ID for the memory object representative is not required to match the task making the `vm_map` request, but rather to match the task in whose address space the memory is mapped.

- MK++ provides two requests which the memory manager can use to revoke previously granted permissions.

  - `memory_object_lock_request` can be used to revoke an access right for all task's mapping a particular page.
  - `memory_object_revoke` can be used to revoke all access through any particular memory object representative.

  It seems that it would be useful for a particular access right to be revoked based upon the memory object representative, but this granularity of revocation is not available.

8.2 Trusted Mach Server Level

The TMach operating system consists of a collection of user space servers running on the MK++ kernel. With the few exceptions of policy enforcement in the kernel as noted in Section 8.1, the TMach security policy is completely defined and enforced at this server level.

The TMach security architecture is described in [1], which includes a figure on page 87 illustrating the major elements of the TMach server TCB. The basic division of duties is between the Root Name Server which makes security decisions and all of the various trusted item managers which enforce those decisions. However, as we shall see there are also special cases which deviate from this model.

At a high level, the TMach security architecture has the flavor of a cross between KeySAFE and Spring. Like KeySAFE, some interactions between tasks are completely unmediated and rely upon an isolation invariant. But while KeySAFE fails to provide a general set of tools for trusted servers, TMach provides an extensive set of such tools including an indirection mechanism for associating a particular client's access rights to server objects. This indirection mechanism uses what are known as *agents*, which are similar to Spring's front objects.

The organization of this section is similar to the organization of the KeySAFE assessment in Section 7.2. The most significant difference is a much more complete discussion of the workings of the TCB servers. While this is directly due to the existence of much more complete information about the TMach servers, the amount of documentation also corresponds to the relative maturity of the two architectures.

- Section 8.2.1 discusses how the basic isolation goal is met by TMach, and a state invariant that the system meets to achieve this goal.
- Section 8.2.2 discusses the standard access control model provided by the TMach TCB.
- Section 8.2.3 discusses the extensibility of this model, and in particular the ability to incorporate servers with limited trust into the model.
- Section 8.2.4 discusses some servers which do not fit exactly into the standard access control model or which have a special role in TMach because of their relationship with kernel access control mechanisms.
- Section 8.2.5 defines a state invariant which defines how the system can meet isolation goals while allowing mediation through the TCB.
- Section 8.2.6 contains an explicit comparison of the TMach and KeySAFE security architectures.

8.2.1 Total Isolation

The TMach architecture is intended to allow total isolation between tasks with distinct security IDs in a manner similar to KeySAFE. This section describes the fundamental architecture support for isolation and the ability to demonstrate that an isolation policy is satisfied.

In the TMach architecture, all tasks with the identical security ID are logically collected into a *subject*. A TMach subject is analogous to a KeySAFE compartment with two important distinctions:

- Since labeling of tasks is explicit in TMach, membership of a task in a particular subject is explicit, unlike KeySAFE compartments.

- KeySAFE compartments are considered to consist of domains and related kernel objects. In TMach, the term subject is generally not used to include kernel objects associated with the subject's tasks.

While TMach documentation does not generally use the term subject to include kernel objects, this does not remove the need to associate some kernel objects with particular subjects. In particular, of the 18 object types identified in Table 1, five object types are specifically related to a particular subject:

- Alarms are associated with a subject through the task which created the alarm.
- Message queues are associated with a subject through the task which holds the receive right for the message queue.
- Memory cache control objects are associated with a subject through the task (acting as a memory manager) which created the object.
- Task objects are associated with a subject according to the task's security ID.
- Thread objects are associated with a subject through the task in which the thread executes.

One basic weakness of the TMach architecture is the lack of labeling on kernel objects so that even though the relationship between tasks and subjects is explicit, the relationship between these five kinds of kernel objects and subjects remains generally implicit as in KeySAFE.

Taking into account that new subjects are created with a capability to a directory in the root namespace, the isolation policy is dependent upon the following state invariant:

> **Total Isolation Invariant (TMach)** A non-TCB subject's capabilities are limited to the following:
>
> - The three types of capabilities identified as unprotected in Table 1.[31]
> - Capabilities for kernel objects of the five types discussed above, such that the objects "belong" to the subject.
> - Ledger name capabilities for the ledgers from which tasks in the subject draw their resources.
> - A capability for a message queue representing the subject's root directory in the namespace.

It is important to recognize that this invariant is not stated in any of the TMach documentation reviewed, even indirectly. Nonetheless, it appears that proof of such an invariant is a necessary first step towards proving any security properties requiring some amount of isolation between subjects.

8.2.1.1 Proof of the Invariant

As usual, the first step is to prove that the invariant holds in all possible initial states. This should be the easier step in the proof, and will not be discussed further.

To prove that the invariant holds across all state transitions requires consideration of any transition which results in a task being granted a capability it did not previously hold. Since capabilities are maintained by the kernel, such transitions must be the result of some kernel

[31] Note that these are roughly comparable to the "stateless service providers" of KeySAFE.

operation, either called explicitly by some task (not necessarily the task receiving the capability) or performed asynchronously by the kernel.

The case of a “direct” transfer of a capability from one task to another is simple to dispose of because the invariant provides no way for one task to directly pass a capability to a task in another subject. The difficulties in proving the invariant arise from the many ways in which the kernel stores capabilities which it later makes available to tasks as a result of a kernel request or asynchronous kernel operation. To give some flavor of the different kinds of cases which can occur, consider the following examples:

- The requests `task_set_special_port` and `task_get_special_port` allow tasks to set or get capabilities in special kernel storage associated with a particular task.
- During the memory management protocol the kernel makes memory object representative and memory cache object capabilities available to a task acting as a memory manager.
- If certain kinds of faults occur during thread processing, the kernel will send a message including capabilities to the thread and its containing task using a capability provided earlier to the kernel.

In order to reason about all of these different ways in which the kernel stores and releases capabilities, the invariant will need to be augmented for each special case.

The resulting invariant will become quite long, though no single statement in the invariant should be overly complex. Still, the number of conditions in the invariant could make analysis of it prone to error even if no single condition is difficult to analyze.

Note in particular that at this point in the analysis, task labeling accomplishes very little because the kernel ignores the labels except as described in Sections 8.1.5 and 8.1.6.

8.2.1.2 Sufficiency of the Invariant

As with KeySAFE, to show the sufficiency of the invariant we must show that if the invariant holds, then different subjects cannot share capabilities or data. A significant complication in TMach is that not all data within a task’s address space is represented by individual capabilities as in KeyKOS. This will require more additions to the invariant, in order to provide a connection between all memory regions in a task’s address space, the memory managers backing the memory and any tasks with which the memory is shared. These could possibly be the most complex statements needed in the state invariant.

Beyond that, proving sufficiency of the invariant is essentially a search for covert channels, which is a difficult task for any system. Of particular interest in that analysis are shared capabilities, including the three types of unprotected global capabilities and ledger name capabilities.

8.2.2 Standard Access Control Model

This section defines the standard model of access control over *security objects*, the labeled objects within the root namespace. The Root Name Server (RNS) makes decisions about a particular client task’s access to a security object, which are then passed to the object’s item manager for enforcement.

This access control model is described in detail in three sections:

- Section 8.2.2.1 defines the basic state information maintained by the servers involved.

- Section 8.2.2.2 defines the standard security decision logic.
- Section 8.2.2.3 describes the five fundamental states involved in security decision making and enforcement.

These sections contain a considerable amount of description of the mechanisms involved with relatively little discussion of the merits of these mechanisms. The main reason for including the descriptions is because other documentation of these mechanisms are at either a much higher level of abstraction and therefore do not provide the necessary detail, or provide a great deal of extraneous detail (the interface documents in particular).[32]

8.2.2.1 State Information

Figure 4 shows the state information which is necessary to describe the TMach access control model.

For every object type managed by an item manager, the item manager has a receive right for a message queue on which it receives requests on the item manager interface. It also has a mapping which defines the rights required for each function exported on objects of that type.

The state of a security object is maintained in cooperation by two servers, an item manager and the RNS.

The item manager component of an object consists of the following:

- *Private data*, which is used as an identifier for the object internal to the item manager.
- A send right used to send messages on the item control interface, for communication with the RNS about the object.
- Some content which is type specific and defines the "real" state of the object. Since this is specific to each object type, it is irrelevant to the general model.
- A set of *agents* representing open client connections to the item manager component of the object. Agents are described further below.

The RNS component of an object consists of the following:

- Private data which is uninterpreted by the RNS but is used to identify this object in communications with the item manager.
- A receive right for a message queue through which the RNS receives item control interface requests, for communication with the item manager about the object.
- A *type identifier*, which is a global identifier used to identify the type of the object.
- The security level assigned to the object. Once assigned, this value never changes.
- The ACL assigned to the object. The ACL is allowed to change.
- A set of *agents* representing open client connections to the RNS component of the object.

When a client has a component of an object open for access, the corresponding server (item manager or RNS) has an agent internal to the server through which all of the client's accesses must pass. The state of an agent includes the following:

[32] Since this section was first written, an updated version of the Trusted Mach Philosophy of Protection [35] was released which contains a discussion at a level similar to that which is provided here.

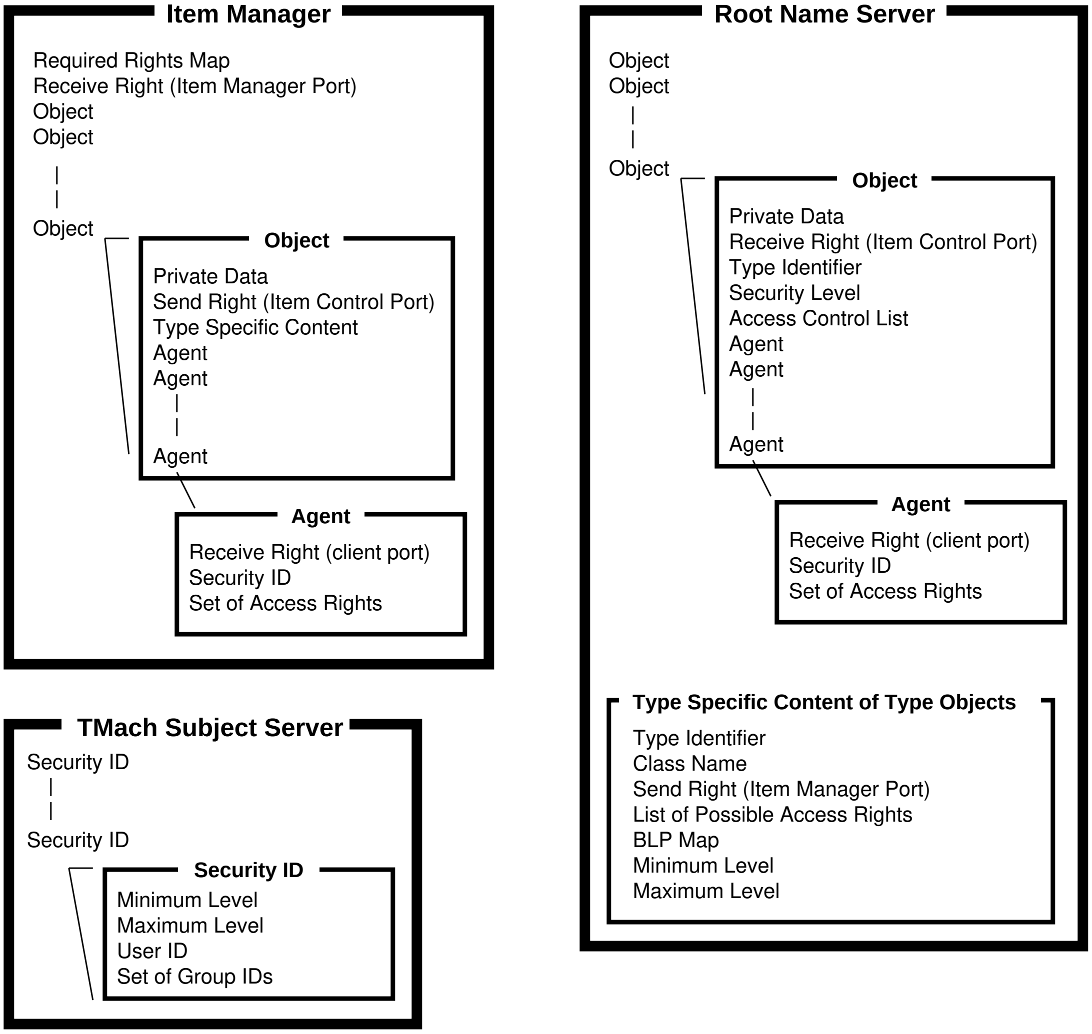

Figure 4: State Information for Access Control

- A receive right for a message queue through which a client accesses this particular component of the object.
- The security ID which must belong to any client attempting to gain access through the agent.
- A set of access rights which have been granted to a client with this security ID.

Note that the item manager maintains no knowledge of the security level or ACL of an object, and is therefore incapable of making any security decisions based upon this information. The only information about the security policy which is maintained in the item manager is in the agents associated with an object, and as we shall see, the item manager receives this information directly from the RNS.

The RNS itself is an item manager for three types of objects, directories, symlinks, and type objects. Objects of these types still have distinct item manager and RNS components, managed by different subsystems within the RNS.

A type object is of particular importance in the access control model because it defines aspects of the model which are the same for all objects of that type. In particular, the content (in the item manager component) of a type object includes the following:

- A unique *type identifier*.
- A *class name* which clients of objects of this type will use to reference the objects.
- A send right to a message queue through which the RNS will make calls to the item manager interface for objects having this type identifier.
- A list of all possible access rights for objects having this type identifier.
- A *Bell-LaPadula (BLP) map* which associates each of these access rights with exactly one of the four BLP permissions (see Section 8.2.2.2.1).
- The minimum level for objects of this type.
- The maximum level for objects of this type.

Note that since there is at most one receive right for any message queue, this means that all objects of a particular type are managed by the same item manager. However, there may be multiple type identifiers each having the same class name so it is possible to have multiple item managers serving objects of a particular class name.

Finally, the access control model requires the identification of particular security attributes with the security ID maintained for each task in the kernel. The TMach Subject Server provides the interface which translates a security ID into the following attributes:

- A range of security levels defined by a minimum and a maximum. The maximum level must dominate the minimum.
- A user ID.
- A set of group IDs.

The set of security attributes associated with a security ID are not allowed to change, except that security IDs are not persistent between boots of the system.

Security levels in TMach are represented by a classification level and a category set. This representation will generally be ignored throughout the report. The mappings of levels, user IDs and group IDs to string is maintained within the TMach Authentication Server, but for the most part these mappings are not important to the report.

8.2.2.2 Standard Permission Checking Logic The standard access control decision in TMach is to decide whether a client task is granted a particular access right to an object. A decision to grant an access requires independent MAC and DAC tests to be passed.

8.2.2.2.1 Mandatory Access Control MAC permission checking is nominally performed by considering the range of levels on a client subject and the level of an object being accessed. However, since operations on an object are also operations on the item manager for the object, MAC permission checking also involves the level range of the item manager through the range of levels associated with the type object for the object being accessed.

MAC decisions are not made directly based upon any access right defined by an item manager. Instead, there are four *Bell LaPadula permissions* which can be granted between a client task and an object. The BLP map associated with each type object defines which BLP permission is required for each access right. In this way, the specific interpretation of the BLP permissions are defined by each item manager. the following describes a general interpretation of each of the four BLP permissions. It also defines exactly when each permission is allowed (the details of which can be found on pages 6 and 7 of [47].

- `BLP_NONE` permission is used to control operations on an object which do not so much affect the object itself as the item manager for the object. This permission is granted whenever the range of levels associated with the client intersects the range of levels allowed for objects with the same type identifier as the accessed object (i.e., the range of levels associated with the type object).[33]
- `BLP_READ` permission is used to control operations that read some aspect of an object. This permission is granted whenever `BLP_NONE` is granted and in addition, the maximum level of the client dominates the level of the accessed object.
- `BLP_APPEND` permission is used to control operations that write some aspect of an object without simultaneously reading the object. This permission is granted whenever `BLP_NONE` is granted and in addition, the level of the accessed object dominates the minimum level of the client.
- `BLP_WRITE` permission is used to control operations that simultaneously read and write some aspect of an object. This permission is granted whenever both `BLP_READ` and `BLP_APPEND` are granted.

8.2.2.2.2 Discretionary Access Control DAC permission checking is done by comparing the ACL on an object with the user and group IDs of the client task. The specific structure of ACLs and the algorithm for making access computations based upon ACLs is rather complex and is described in detail on pages 13 through 18 of [49]. Rather than repeating the description found there, we just make a few observations:

- TMach ACLs can specify permissions as either granted or denied. However, it is possible for a permission to be denied explicitly in the ACL but still be granted. For instance, this occurs if a permission is denied in a wildcarded entry but granted in an entry specific to a particular user ID. Once understood, this logic does not present any particular problems, however it would not be surprising for a user to assume that any permission explicitly denied in one entry will be denied regardless of the other entries.[34]

[33] The matrix on page 7 of the RNS interface doesn't say it this way, however, it is easy to show that if levels are defined as in TMach by a "base level" from a totally ordered set and a collection of categories, then indeed the two ranges must intersect in at least one level.

[34] Spring ACL entries can similarly explicitly deny or grant access. However, the logic for computing access rights from an ACL is such that if a permission is denied through one entry, then no amount of granting in other entries can override the denial.

- Perhaps the most unusual aspect of TMach DAC computations is that the logic supports simultaneous computation of a set of access rights, with the property that access rights which would be granted individually may actually be denied when computed as a set (the converse does not happen).

  It is unclear what benefit is gained by this complication in the logic, and in addition to the usual disadvantages brought on by any unnecessary complications, it has one particularly detrimental side effect: there is no unique maximum set of access rights granted by an ACL. This is a problem because other parts of the system depend upon a maximum set of access rights (see Section 8.2.2.3.2).

- TMach uses the DAC mechanism to support least privilege among the TCB servers by assigning each server a unique user ID and defining ACLs which distinguish between these user IDs.

- TMach uses the DAC mechanism as the primary way to prohibit non-TCB servers from performing "privileged" tasks. For instance, permission `NSR_RECREATE` on the initial type object controls the `ns_recreate_type_secure` operation, which if invoked by an untrusted task could allow the task to completely subvert the security of the system.

The last two items in this list emphasize that the DAC mechanisms are actually being used to enforced mandatory requirements. Therefore integrity of the DAC mechanisms in TMach is every bit as important as integrity of the MAC mechanisms, and both mechanisms must be analyzed with the same care. This is not unusual for trusted systems, though all too often DAC mechanisms are at least informally considered to be secondary to MAC.

8.2.2.3 State Transitions

This section describes the five basic transitions which embody the standard access control model in TMach.

1. A task accessing a previously opened object.
2. A task opening an object by resolving a name in the namespace.
3. A task creating a new object in the namespace.
4. A task creating a new type object.
5. A task registering as an item manager for a type object.

The order in which these transitions are described is chosen to allow the description of each transition to build upon the previous as much as possible:

- All of the transitions involve accessing a previously opened object, so all other transitions build on transition 1.
- Creation of a new object also opens the new object, so transitions 3 and 4 build on transition 2.
- Creation of a new type object is essentially just a special case of creating a new object, so transition 4 builds on transition 3.

However, conceptually the order in which these transitions would take place for a particular object is 4-5-3-2-1.

8.2.2.3.1 Accessing an Open Object A client accesses an open object by sending to a message queue whose receive right is held by an agent for the object. The client typically holds send rights for agents in both the item manager and the RNS and must send to the appropriate one depending upon the particular function being called. Before providing any service to the client, the agent enforces the security decisions recorded in the agent as follows:

- Verifies that the security ID of the client (as appended to the client's message by the kernel) matches the security ID as recorded with the agent.
- Verifies that the required access rights for the function being called are among the set of access rights recorded with the agent.

This operation is the basic security policy enforcement operation in TMach. The agent makes no security decisions itself, it merely enforces decisions made previously and recorded with the agent. The code which performs this policy enforcement is encompassed in the Class Library, which is used by all trusted servers. The existence of this library should significantly increase assurance in the enforcement operation since the operation only need be analyzed once for all servers.

8.2.2.3.2 Opening An Object A client can open an object in the name space by invoking `ns_resolve` on a directory which the client already has open.

As inputs to `ns_resolve`, the client provides:

- A string describing the path which it wishes to resolve, relative to the invoked directory.
- A set of access rights which the client requires for the resolved object. This list of access rights must be a subset of the access rights defined for the type of object being resolved. Alternatively, the client can specify `NSR_ALLOWED` to request the maximum set of access rights granted to the client.

The ability to ask for the maximum set of access rights is interesting, since as described in Section 8.2.2.2.2, DAC computations are such that there may not be a unique maximum set of access rights. It is unclear how this discrepancy is resolved in the implementation.

The RNS enforces the security policy as described in Section 8.2.2.3.1 by verifying that the client task has the required right `NSR_LOOKUP` for the directory object through which the call is invoked.

In addition, the RNS performs the following security policy decision making and enforcement:

- As the path is resolved, the RNS consults the MAC and DAC policies to decide if the client can be granted `NSR_LOOKUP` to each of the directories along the path. If not, the request fails.[35]
- After the path is completely resolved, the RNS consults the MAC policy to decide if the client can be granted at least `BLP_NONE` permission to the resolved object. If not, the request fails.
- The RNS consults the MAC and DAC policies to decide if the client can be granted the set of requested access rights to the resolved object. If not, the request fails. If so, these decisions will be passed to agents as described below for further enforcement.

[35] There are some differences here if the path includes symbolic links, but that is not important at this level of analysis.

If all of these permission checks (and any functional checks) are successful, then the RNS creates an agent to represent the client's access to the namespace component of the object. This agent is associated with the client's security ID, the set of requested (and now granted) access rights, and the receive right for a newly created message queue.

A similar agent must also be initialized in the item manager to control the item manager component of the resolved object. To accomplish this, the RNS invokes the `im_access` request on the send right for the item manager (as recorded in the type object for the resolved object's type). The RNS passes the following to the item manager with this request:

- The security ID of the client
- The set of access rights granted to the client.
- The receive right to another newly created message queue which will be used for client requests to the item manager. For now, the RNS keeps a send right to this message queue.
- The private data which identifies the resolved object to the item manager (this is stored in the namespace component of the object).
- A send right for the item control port for the resolved object (the receive right for this port is part of the namespace component of the object).

The item manager is expected to use the provided data and capabilities to initialize a new agent for the object, however, the `im_access` call is asynchronous and the RNS does not wait for a response.

Finally, the RNS returns from the `ns_resolve` call, returning to the client send rights for both of the newly created agent ports.

8.2.2.3.3 Creating a New Named Object TMach presents a unique model for adding items to the name space. In a typical multiserver operating system, a client issues object creation requests directly to the object server. If the client or server wishes to make the new object visible in the name space, it makes a separate request to bind the object to some location in the name space.

In TMach, objects can be bound to the name space only if the request to create the object is made through the RNS. While there is nothing to stop servers from providing an interface to create objects directly at the request of a client, there is no means for such an object to be later added to the root name space, i.e., to become a security object.

Though the documentation does not seem to discuss the rationale behind the object creation model, the apparent reason is to avoid requiring item managers to initialize agents independent of the RNS. For the managers themselves it is much easier to rely upon the RNS to simply provide the exact information needed to initialize an agent. The RNS is then responsible for maintaining all object labels, even for objects which are persistent. In fact, item managers have no need to ever know anything about the labels on the objects which they manage, which appears to be a rather unique feature of TMach.[36]

The request to create a new named security object is `ns_create_secure`, which is invoked on a directory which the client has open.

As inputs to `ns_create_secure`, the client provides:

[36] There is at least one functional disadvantage to this mechanism, which is that a labeled object can exist at only one location in the name space, except through symbolic links.

- The name for the new object in the invoked directory.
- The type identifier of the new object.
- The ACL for the new object.
- The security level for the new object (which can default to the maximum level of the client).
- A set of access rights which the client wishes to receive for the new object, just like in the call `ns_resolve`. These are specified because this request not only creates the object but also opens it for the client.

The RNS enforces the security policy as described in Section 8.2.2.3.1 by verifying that the client task has the required right `NSR_INSERT` for the directory object through which the call is invoked.

In addition, the RNS performs the following security policy decision making and enforcement:

- The RNS consults the MAC and DAC policies to decide if the client can be granted `NSR_CREATE` to the type object for the requested type identifier. If not, the request fails.
- The RNS verifies that the MAC and DAC policies grant the client `NSR_ADMIN` to the object to be created (according to the ACL provided as a parameter). If not, the request fails.
- The RNS verifies four special MAC conditions. If any of these conditions do not hold, the request fails. There are no corresponding DAC controls.
  - The security level for the new object must dominate the level of the invoked directory.[37]
  - The security level for the new object must dominate the minimum level of the client.[38]
  - The security level for the new object must lie between the minimum and maximum levels for the object's type.
  - The level of the invoked directory must lie between the minimum and maximum levels for the object's type.
- The RNS consults the MAC and DAC policies to decide if the client can be granted the set of requested access rights to the resolved object. If not, the request fails. If so, these decisions will be passed to agents.

If all of these permission checks (and any functional checks) are successful, then the RNS:

- Creates a message queue to be an item control port for the new object.
- Creates a second message queue and an agent for the client's access to the namespace component of the object, as it does for `ns_resolve`.
- Creates a third message queue that will be held by the item manager's agent. The RNS invokes the request `im_create` on the send right for the item manager port, as found with the type object for the new object's type. With this request the RNS passes the following information to the item manager:

[37] This is different from the AIM model, which requires that all objects be at the same level as the directory in which they are contained, except for other directory objects. This is normally done to minimize covert channels as much as possible while still maintaining useful function.

[38] There is no mention of a corresponding check for the maximum level of the client to dominate the level of the new object. Apparently this is considered a legal write upward in level.

- The security ID of the client
- The set of access rights granted to the client.
- The receive right to this third newly created message queue.
- A send right for the newly created item control port.

The `im_create` call is asynchronous, and it does not appear that the RNS waits for any response from the item manager before proceeding.

- The RNS returns from the `ns_create_secure` call, returning to the client send rights for both of the newly created agent ports.

As a result of the `im_create` call, the item manager is expected to invoke `ictl_write_cache` on the item control port to provide the RNS with the private data associated with the object. Note that the object itself within the item manager has no state content until the client initializes it through the agent port returned to the client.

8.2.2.3.4 Creating a new Type Since a type is just a particular class of object, creating a type is very similar to creating any other object. The differences are primarily due to the fact that types are managed by the RNS, and the call to create a type also initializes the content of the type object.

A new type is created by invoking `ns_create_type_secure` on a directory which the client has open.

As inputs to this call, the client provides the following data to initialize the namespace component of the object. This information is identical to the information provided with `ns_create_secure` except that there is no need to provide the type identifier, since that is explicit in the name of the request:

- The name for the new object in the invoked directory.
- The ACL for the new object.
- The security level for the new object (which can default to the maximum level of the client).
- A set of access rights which the client wishes to receive for the new object.

The client also provides the following additional data inputs to initialize the item manager component of the new type object:

- The class name.
- The list of all possible access rights for objects of this type.
- The BLP map.
- Minimum and maximum security levels for objects of this type.

The RNS enforces the security policy as described in Section 8.2.2.3.1 by verifying that the client task has the required right `NSR_INSERT` for the directory object through which the call is invoked.

In addition, the RNS performs the following security policy decision making and enforcement:

- The RNS consults the MAC and DAC policies to decide if the client can be granted NSR_CREATE to the type object for the class name "type". If not, the request fails.
- The RNS verifies that the MAC and DAC policies grant the client NSR_ADMIN to the object to be created (according to the ACL provided as a parameter). If not, the request fails.
- The RNS verifies six special MAC conditions, four of which are analogous to those in ns_create_secure. If any of these conditions do not hold, the request fails. There are no corresponding DAC controls.
  - The security level for the new object must dominate the level of the invoked directory.
  - The security level for the new object must dominate the minimum level of the client.
  - The security level for the new object must lie between the minimum and maximum levels for the type "type".
  - The level of the invoked directory must lie between the minimum and maximum levels for the type "type".
  - The maximum level for the new type must dominate the minimum level.
  - The minimum level for the new type must dominate the security level for the type object itself (another parameter).
- The RNS consults the MAC and DAC policies to decide if the client can be granted the set of requested access rights to the resolved object. If not, the request fails. If so, these decisions will be passed to agents.

  There is one very important special case in this permission checking. The MAC permission check for the access right NSR_MANAGE (referenced in Section 8.2.2.3.5) is not performed against any of the BLP permissions. Instead, it is granted only if the client's security level range contains the type's range and the type's security level.
- There are also some integrity checks on the BLP map to ensure that access rights enforced by the RNS (for the namespace component of an object) map to the correct BLP permissions. These checks are described on page 62 of the Name Service Interface [47].

If all of these permission checks (and any functional checks) are successful, then the RNS initializes agents for both components of the new object as done in ns_resolve and ns_create_secure. The RNS initializes all of the specific aspects of a type object with the data provided by the client and a new uniquely generated type identifier. The only aspect of the type object not initialized at this time is the send right for an item manager port.

The RNS returns from the ns_create_type_secure call, returning to the client send rights for both of the newly created agent ports and the type identifier associated with the new object.

8.2.2.3.5 Registering as an Item Manager When a new type is created, the one thing it is missing is an item manager for items of that particular type. The item manager registers by invoking ns_register_item_mgr on the type object.

This call provides a single input parameter, which is a send right which will be used for item manager requests on this object type. There are no outputs from the request (other than error codes).

The RNS enforces the security policy as described in Section 8.2.2.3.1 by verifying that the client task has the required right NSR_MANAGE for the directory object through which the call is invoked.

8.2.3 Untrusted Item Managers

The discussion of the standard permission model in Section 8.2.2 assumes that the RNS and the item manager are truly cooperating. One particularly nice feature of TMach is the easy extensibility which is provided by the way in which new item managers can be added to the system by simply adding new type objects. However, this flexibility also leaves open the possibility of untrusted item managers.

This section discusses the implications of an untrusted task acting as an item manager, and the possibility of such a task attempting to manage items outside of the task's level range.

TMach is expected to support untrusted item managers who still wish to make use of the namespace services of the RNS. To understand the implications, we consider what happens if an untrusted single level task acts as an item manager. The extension to a multilevel task is relatively straightforward.

For a task to become an item manager, it must register as in Section 8.2.2.3.5. The only security check during this operation is to verify that the task holds `NSR_MANAGE` to the type object. As explained in Section 8.2.2.3.4, this requires that the range of levels associated with the task encompasses the range of levels associated with the type object. Therefore the range of levels associated with the type object must consist of just a single level matching the level of the task.

Next suppose there is an attempt to create an object of this type. As in Section 8.2.2.3.3, the level of the new object must again be the same level as the task. Since object levels cannot be changed, this means that all objects managed by the untrusted item manager are at the same level as the item manager.

Now suppose a client wishes to resolve some object managed by this untrusted item manager. Because `BLP_NONE` permission is required regardless of the access rights requested, the level range of the client must include the range of the untrusted item manager.

The only place in Section 8.2.2 where the item manager is expected to enforce security decisions itself is upon object invocation as described in Section 8.2.2.3.1. If the untrusted item manager fails to enforce these permissions, it can allow for essentially unrestricted communication between any of its clients. For single-level clients, this is not a violation of the MLS policy since they must all be at the same level. Multilevel clients of the untrusted item manager must contain the level of the item manager and must be trusted to consider all interactions to be at that level. It could be argued that there is a potential violation of the DAC policy, but as long as this is understood, this is not a serious concern. And in fact, it is well-understood in the documentation.

Finally, note that although the item manager may not enforce any DAC policy on its component of the object, the RNS still enforces its DAC policy on the namespace component.

8.2.4 Special Servers

The previous section describes the standard permission checking model in TMach. Several special cases were described in that section with respect to access right computations in the RNS.[39] There are also special cases of this model in a few of the TMach servers. This section describes those special cases and also the TMach servers which interact directly with kernel security policy enforcement on devices and virtual memory.

[39] There are also some special cases in the RNS which were not described there.

8.2.4.1 Subject Server The TMach Subject Server (TMSS) is an important part of the TMach security architecture for several reasons:

- It maintains the mapping between task security IDs and the actual security attributes, as described in Section 8.2.2.1.
- It is the only task which holds the host security capability.
- By holding the host security capability, the TMSS controls all creation of new subjects (i.e., of tasks with new security IDs).[40]
- As part of the creation of new subjects, the TMSS makes some access control decisions independent of the RNS. It is apparently the only server to make decisions in addition to enforcing the decisions of the RNS.

The TMSS presents two main interfaces to clients. One interface allows clients to retrieve the security attributes of a security ID. Requests on this interface are controlled using the standard access control model of Section 8.2.2, with the TMSS enforcing the security decisions of the RNS.

The other interface allows clients to create new subjects, which is the topic of the remainder of this section.

Subjects could potentially be created through the RNS call `ns_create_secure` like other security objects. However, there are at least three important distinctions between subjects and security objects:

- Subjects do not have a representation in the RNS name space.
- The security attributes of subjects are more than a single security level.
- The permission checks that are necessary when creating a subject go beyond those required for standard object creation, and the RNS only recognizes special cases in permission checking when dealing with objects for which the RNS is item manager.

Instead, there is a single subject server object in the name space, which clients resolve like any other object. Subjects can then be created by invoking a call such as `create_task_subj` on this object.

As inputs to `create_task_subj`, the client provides:

- The minimum and maximum levels for the new subject.
- The user ID for the subject.
- A set of group IDs for the subject.

The TMSS enforces RNS provided security decisions as described in Section 8.2.2.3.1 by verifying that the client task has the required right `NSR_WRITE` for the subject server object.

In addition, the RNS consults the "layering database" to determine if the client is allowed to create a subject with the requested user ID. The layering database is effectively a list of user ID pairs identifying that a client with the first user ID is allowed to create a subject with the

[40] The other request requiring a capability for the host security capability is `host_security_id_list`, which is briefly discussed in Section 8.3.4.

second user ID. Complete contents of this database are given in [51]. In particular, clients outside of the TCB and trusted shells are never granted permission to create a new subject.

The RNS also verifies that the range of levels requested for the new subject is a subrange of the level range of the client. Presumably it also verifies that the requested maximum level dominates the requested minimum level, though this is not explicitly mentioned.

Other conditions on the security attributes of the new subject should also be confirmed, such as:

- Is the maximum level of the subject one for which the user is cleared?
- Is the set of group IDs appropriate for the user?[41]

The TMSS itself does not verify these conditions, rather it trusts its clients (such as the Trusted Shell) to have verified them.

For both of these conditions, it is also unclear what happens if the relationship changes while a task exists, for instance if a user is removed from a particular group.

8.2.4.2 Host Control Server

The Host Control Server holds several of the capabilities identified as protected by the initial task in Figure 3. In particular, it is the only task to hold host control, clock control and processor set control capabilities.

The Host Control Server directly exports several kernel requests which require these capabilities, by creating objects in the RNS namespace to represent the capabilities. The Host Control Server never gives away any of these protected capabilities, it simply acts as a mediator between clients and the kernel, enforcing the decisions of the RNS as in the standard model of Section 8.2.2.

There are several interesting aspects of the host control server:

- The host control capability is required to create ledgers. Because the Host Control Server does not export any interface for creating ledgers, ledgers cannot be used in TMach. This is done because they are not seen as valuable due to the shortcomings discussed in Section 8.1.4.
- Several kernel requests available through processor set control capabilities are not exported by the Host Control Server, including all requests dealing with scheduling policies, `thread_wire` and `processor_set_statistics`. By forbidding all access to these requests, the Host Control Server is effectively eliminating from the kernel interface (after bootstrap is complete) operations that are considered unnecessary. An alternate approach would have been to provide these requests through the Host Control Server but use the ACL on the server object to forbid all use of the requests. Eliminating them altogether makes analysis simpler since it must only ensure that the processor set control capabilities are protected.
- In Mach 3, there was a kernel request `vm_statistics` which could be made by any task simply by invoking the request on its capability to itself. This request returns quite a bit of information about current memory usage, and could potentially be a rich source of

[41] It is unclear what exactly is meant by "appropriate" in this condition. In some systems it might be expected that the requirement is for the user to be a member of each group. However, since TMach DAC calculations may cause a task to lose access rights for being a member of a group, the requirement in TMach might be that the set of group IDs is the complete set of groups to which the user belongs.

covert channels. In MK++, the equivalent request, `host_statistics`, is only available through the host control capability. At first glance, the purpose of this change appeared to be to limit its use to those tasks which really require such information.

However, according to the BLP Map and ACLs recorded in [43], any client task is permitted to make this request through the host control server. No rationale for this decision is given, though one can speculate that it is to allow untrusted debuggers to access the request, or perhaps untrusted operating system personalities.

8.2.4.3 Device Server

As described in Section 8.1.5, the TMach kernel is prepared to enforce security decisions on device capabilities much like an item manager. What is missing is the connection between device capabilities and the RNS namespace, and it is this connection that the TMach Device Server provides. The Device Server is the only task holding a capability to the kernel's device master object.

The Device Server acts as an item manager for "named raw device" objects, which represent kernel devices in the RNS namespace. For a client to receive a capability for a kernel device, it must first resolve the corresponding named raw device object and then invoke `dev_get_kdevice_port` on the resolved object.

The client provides no parameters for this call. The Device Server essentially just forwards this call to the kernel through the device master capability, including the security ID of the client and the list of access rights granted to the client by the RNS. The kernel's response is forwarded directly back to the client.

The only complication at all in this operation is that the Device Server does not rely upon the Class Library to perform permission checking on the `dev_get_kdevice_port` call. The reason for this is that the kernel allows a client to get a device capability if any one access right is granted out of a possible five access rights. The Class Library cannot handle this kind of "OR'ing" of access rights, so the Device Server must perform this check outside of the Class Library. Note that this is just a different enforcement model, the Device Server itself makes no security decisions.

Once the client receives the capability for the kernel device, it need not communicate again with the Device Server.

8.2.4.4 File Server

The TMach File Server acts as an external memory manager in the protocol described in Section 8.1.6. There is a high level similarity between the actions of the file server and those of the device server, since both relay security decisions from the RNS to the kernel. However, the mechanisms for doing so are completely different.

A client opens a file for access through the RNS, like any other security object. Part of the RNS processing when opening an object is to create a message queue whose receive right is passed to the item manager and a send right passed to the client. This message queue acts as the memory object representative described in Section 8.1.6, and the security ID and access rights returned to the File Server by the RNS are associated with the representative. The client does not invoke this send right directly, but rather passes is to the kernel as a parameter to `vm_map`.

Further processing is described in Section 8.1.6. Perhaps the most interesting aspect of this is that to perform typical reading and writing of a file, the client never contacts the File Server directly.

8.2.5 Mediated Isolation Invariant

In the KeySAFE operating system, it was relatively simple to extend the Total Isolation Invariant to a state invariant which applied when considering the capabilities granted through the mediation of the TCB (see Section 7.2.2). However, this is not nearly as simple to do for TMach.

The essential aspect of KeySAFE which made this possible was that a key was allowed to be shared between compartments only if the key could not be used to gain further keys (except perhaps through the TCB). No such requirement holds in TMach, and the TMach TCB will indeed grant capabilities which the TCB cannot possibly interpret (e.g., capabilities placed in the namespace by untrusted item managers).

In an attempt to derive a Mediated Isolation Invariant for TMach, we distinguish three cases in which the TCB will grant capabilities to a client outside of the TCB:

- When the client opens (or creates) a security object in the root namespace, as described in Sections 8.2.2.3.2 through 8.2.2.3.4, and the item manager for the object is within the TCB.
- When the client opens (or creates) a security object in the root namespace and the item manager is not within the TCB.
- Other. There are at least two other cases in which TCB servers grant capabilities to clients outside of the TCB, such as granting of capabilities to devices by the device server, and the three kinds of capabilities which are passed to item managers by the RNS. Determination of all other cases would require a much more detailed analysis of all of the TCB interfaces.

The second case above is the one which creates the most significant difference between KeySAFE and TMach. As discussed in Section 8.2.3, the use of an untrusted item manager makes possible unlimited transfer of capabilities between subjects at the same level, regardless of user or group IDs. Therefore the invariant must be weakened to only specify the capabilities held by subjects based upon their level.

Since many of the requests provided by TCB servers return capabilities to the caller, identification of all examples of the third case requires a determination of exactly which of these requests are actually available to tasks outside of the TCB. Availability of TCB requests is typically limited by ACLs based upon the user IDs associated with the standard TCB servers. For the purpose of this discussion, the TCB consists not only of tasks identified as TCB by special user IDs but also includes any task which has a range of more than one level.

Now we are ready to state the invariant:

> **Mediated Isolation Invariant (TMach)** A non-TCB subject's capabilities are limited to the following:
>
> - The three types of capabilities identified as unprotected in Table 1.
> - Capabilities for kernel objects of the five types identified in the Total Isolation Invariant, such that the objects "belong" to the subject or another subject having the same level.
> - Ledger name capabilities for the ledgers from which tasks at the same level as the subject draw their resources.
> - A capability for a message queue representing a directory object in the root namespace.

- Capabilities granted to the subject or another subject at the same level by opening or creating an object in the RNS namespace.
- Capabilities granted to the subject or another subject at the same level through some “other” action of the TCB.

To prove the invariant holds requires analysis of all TCB requests available outside of the TCB and which can result in passing capabilities. It is certain that the invariant will need to be expanded like the Total Isolation Invariant, to include properties of capabilities held by the TCB. The “other” case will also need to be explicitly expanded upon into all specific cases.

While this should be a manageable task, it is by no means straightforward and certainly could be prone to error due to the number of requests that need to be analyzed.

8.2.6 Comparison Of KeySAFE With TMach

It is worthwhile to summarize some of the more important distinctions between KeySAFE and TMach.

- MK++ supports labeling of tasks, while KeySAFE explicitly denies the necessity of doing so. While KeySAFE demonstrates that this is not necessary, task labeling is an important step towards simplifying the analysis and hence increasing confidence in the correctness of the security mechanisms.
- There is much more documentation available for the TMach TCB than for the KeySAFE TCB. In fact, there is only about 30 pages of information explicitly about the KeySAFE additions to KeyKOS, while there are at least 500 pages of information available on TMach TCB servers. Based upon the somewhat vague contents of the KeySAFE documentation, it is easy to conclude that the TMach TCB design has received proportionally more attention.
- The TMach design is explicitly extensible to allow the addition of new item managers with no change to the existing TCB. There is no indication that any extensibility property holds for the KeySAFE TCB.
- The use of the TMach Class Library and other libraries simplifies assurance by encouraging reuse.
- KeySAFE only allows a capability to be shared between compartments if it cannot be used to gain access to other capabilities. TMach makes no such restriction. As a result, once a capability is shared between TMach subjects there is no way to control sharing of other capabilities.
- When a capability is shared between compartments in KeySAFE, the TCB must become involved in all invocations of the key. This is potentially a performance hit for which there is nothing comparable in TMach.
- KeyKOS protects memory directly through capabilities. MK++ has a special security mechanism for protecting virtual memory. In general, the KeyKOS kernel has a simpler, more consistent control philosophy than MK++.
- KeyKOS provides space banks and meters to provide resource availability guarantees. The only comparable mechanism in MK++ is the ledger mechanism, which appears to be much less useful.

8.3 Security Assessment

This section provides an assessment of the ability of TMach to be configured to meet a range of security policies. Since TMach has no single component responsible for making security decisions, we must first identify the elements of the system which may need to be reimplemented for different policies. The MLS/DAC policy currently implemented by TMach includes policy dependencies in the following system elements:

- The interface to create new security objects requires identification of a level and ACL for the new object (though the client side libraries will choose a default level and ACL).
- The interface to create a new subject requires identification of the level range, user ID and group IDs for the subject.
- The interface to create a new type object requires identification of a level range and a BLP map.
- The databases of security information maintained by the Authentication Server are all completely dependent upon the semantics of the policy being enforced.
- Security decision making in the TMSS when a new subject is created is dependent upon the use of user IDs when consulting the layering database.
- The standard security decision making in the RNS, and the many special cases in the RNS all depend upon the MAC and DAC attributes.

This shows that the policy is localized to three servers, the RNS, TMSS and Authentication Server, and through interfaces to them. Moreover, only clients which need to create a security object or subject with particular security attributes have any need to be aware of the security policy. Particularly notable is that item managers do not have any need to be aware of the labels of objects which they manage, a property which is perhaps unique to TMach. However, item managers (or whatever tasks create the object types) must be aware at least of the range of levels and BLP map for their object types.

The straightforward generalization of these policy dependencies are as follows:

- Interfaces for creation of security objects or subjects must specify security attributes from the particular policy being enforced.
- The interface to create a new type object must somehow specify a set of security attributes which can be assigned to objects of that type. While the BLP map may appear to be unique to multilevel policies, read/write distinctions in security policies are common so it is likely that the BLP map will be useful in general.
- The single security decision made by the TMSS can be represented through a table indicating which subjects can create other subjects.
- The most difficult abstraction is all of the special case permission checking in the RNS. The RNS needs to have a distinct component to make all security decisions, sometimes based upon attributes of several different objects, such as when creating a new object, which involves the security attributes of the client, the requested attributes of the new object, the attributes of the object type, and the attributes of the directory in which the object is named.

Accomplishing all of these abstractions, especially in the RNS, is likely to take some work. However, as mentioned above, the effort is localized to just a few servers.

8.3.1 General Characteristics

MAC/DAC In the generalization just described, TMach provides support for MAC and DAC policies.

To consider the strength of this support, we look at the MLS implementation in particular. The main issue that must be considered is whether the Mediated Isolation Invariant suffices to prevent downward flow of information. We have not attempted to understand all of the special cases of MAC permission checking within the RNS, but within the standard permission checking model we only have one significant concern: access rights which are mapped to `BLP_NONE` do not provide the item manager with a way to distinguish between clients at different levels.

Recall that `BLP_NONE` permission is apparently intended to be used for accesses to the item manager rather than a particular object. Therefore allowing clients at any level in the range spanned by the item manager to communicate with the item manager is appropriate. But it does not automatically follow that all accesses to the item manager should be allowed independent of the level of the client.

Consider for example an item manager with a range of levels between MIN and MAX (not necessarily system low or system high). Single level subjects with any level between MIN and MAX can act as clients of this item manager, and receive `BLP_NONE` permission. Requests from two subjects operating on behalf of the same user ID, one at MIN and the other at MAX receive exactly the same set of access rights among those mapped to `BLP_NONE`, and are therefore indistinguishable by the item manager. Therefore any information "written" through an operation requiring `BLP_NONE` permission must be treated by the item manager as MAX data, and any information "read" by such an operation must be known by the item manager to be MIN.

But it appears to be typical to map many access rights which write and/or read data to `BLP_NONE`. A simple example is the requirement that for all types, `NSR_GETATTR` must be mapped to `BLP_NONE`. The request `ns_get_default_attributes` returns certain basic information about the namespace component of an object. The request can return "content related" and "non-content related" information. To return "non-content related" information, the client must be granted `NSR_GETATTR`.

According to the analysis above, any information "read" by this request must be at the level MIN. But at least some of these values are set during a `ns_create_secure` operation, and that operation can be invoked by a MAX client. This shows that the MAX client can write data visible to the MIN client.

Whether this example is common or an isolated instance is unclear, as no systematic search for such examples has been performed. In any case, the inability of an item manager to distinguish between the levels of clients accessing item manager information appears to be a serious limitation.

A lesser concern about the ability of the system to meet MAC policies is the fact that DAC mechanisms are actually relied upon to provide much of the critical permission checking in TMach. Each TMach server is given a distinct user ID, and most of the access rights for truly privileged operations are actually mapped to `BLP_NONE`, with the ACL relied upon to limit the operation to particular TCB servers.

Some of these operations are extremely critical. For instance, DAC provides the only meaningful control over `ns_recreate_type_secure` operations, which can effectively rewrite portions of the security policy, including the BLP map for an object type.

It is not that unusual for DAC mechanisms to be used in this way in trusted systems. It is extremely important to avoid the temptation of minimizing the importance of those mechanisms, as it is common to consider DAC to be of lesser importance.

Concerning the ability of the system to meet the DAC policy, there is little to add beyond the comments already made in Section 8.2.2.2.2. The Mediated Isolation Invariant points out explicitly that outside of the TCB, there is no way to guarantee that capabilities do not propagate between subjects at the same level. The use of agents for additional permission checking means that capabilities for security objects cannot be used outside of the subject for which they were issued, however, the standard DAC trojan horse concerns apply to TMach as to other secure systems.

Finally, we consider the Mediated Isolation Invariant in terms of a general policy. Assuming that the security policy maintains the distinctions among TCB components and in particular provides the same restrictions on the interfaces available outside of the TCB, the invariant can be easily extended to other policies. In particular, the only policy dependence in the statement of the invariant is due to the ability to share capabilities among subjects at the same level. In some policies, there may be no analogue to this feature, but more generally it would be possible to have equivalence classes of subjects among which capabilities could be shared.

Information Flow-Transitivity Among subjects at the same level (or more generally, in the same equivalence class), there is a transitivity relationship through untrusted item managers. Through the TCB itself there is no requirement of transitivity, though the use of the BLP map encourages transitive policies.

Information Flow-Covert Channels While detailed discussion of covert channels is clearly outside of the scope of this report, we do have two general concerns about the difficulty of removing and detecting covert channels in TMach:

- It appears that the kernel design has made a specific attempt to remove some covert channels by requiring protected capabilities for requests that return potentially sensitive information. For instance, the `host_statistics` request mentioned in Section 8.2.4.2. However, high resolution clocks are still available through unprotected capabilities. In general, the kernel design still does not appear to reflect a serious concern for covert channel elimination.
- Even in a kernel designed to minimize covert channels, covert channels invariably remain. It is therefore important to have the ability to detect the possible use of remaining covert channels. It is certainly possible that such mechanisms have been added to the MK++ kernel, but there is no mention of such mechanisms in any of the kernel documentation under consideration.

  Moreover, the lack of labeling on kernel objects may make detection of covert channels more difficult.

These concerns about the kernel are based upon our understanding of the MK++ kernel and the impression that it is unlikely that the kernel would contain sufficient covert channel mitigation, given the potentially competing requirements placed on the kernel by its developers at OSF. However, the alterations to this kernel made as part of the TMach project may well have addressed these concerns.

Mutual Suspicion It is unclear whether TMach can support the solution proposed by KeyKOS. However, because of the explicit labeling of tasks and the use of memory object representatives, a simpler solution such as that proposed for DTOS (see Section 9.4.1) should be possible.

8.3.2 Least Privilege and Granularity

Granularity of MAC Attributes There appear to be no practical limits on the number of distinct security IDs which can be assigned.

Protection Boundaries/Unique Identifiers The protection boundary in TMach is the set of all tasks with the same security ID. In fact, this set of tasks is referred to as a single subject in [1]. There is no direct support for assigning unique security IDs so this must be provided through some protocol agreed upon by all relevant clients.

Privilege Levels/Least Privilege With the use of the BLP map, TMach essentially requires that all access rights are treated as read or write permission. This can be a considerable limitation on least privilege. However, this is the only significant limitation on least privilege.

Run-time Least Privilege Capabilities can provide for this, however, the server level capabilities provided through agents do not naturally support delegation because they are associated with a particular security ID.

8.3.3 Characteristics of Dynamic Policies

History Sensitive Policies Those elements of the system responsible for making security decisions can certainly maintain information about previous decisions if necessary, though there is no existing mechanism for doing so or for notifications between these elements.

Relinquishment Sensitive Policies The use of agents in trusted servers (and similar controls in the kernel) provides a way to identify all permissions which are possibly in use over those objects. It does not provide a way to identify the permissions that are actually in use or those related to implicitly labeled kernel objects.

Retractive Policies If previously granted access rights to an object need to be invalidated, the `ns_revoke` request must be invoked upon the object's namespace component. The request causes the RNS to invalidate all agents associated with the object and to invoke the `im_revoke` request on the item manager. The item manager is expected to respond by revoking all of its agents to the object.

Further requests to a revoked agent will result in an error message to the client. The error message informs the client that it must reopen the object, requesting again the necessary set of access rights.

This is a rather crude approach since it invalidates all accesses to an object by all clients, when there may only be a single minor change in policy. On the other hand, the simplicity is very nice for analysis.

This mechanism only affects future operations which require these permission checks, not those currently being performed. It also has no effect on those kernel objects which are not protected through these mechanisms.

Retractive Destruction There is no direct support for destroying all objects with a particular label. Like KeyKOS, this is especially difficult in the kernel because some of the objects (though not all) are only implicitly labeled.

Non-tranquility The security ID of a task, and the mapping from security ID to a set of security attributes, are tranquil. The security attributes of an object can change however. In the current implementation, only the ACL is allowed to change, not the object's level.

Perhaps most unusual is that the attributes associated with a type object, including the range of levels and the BLP map, are allowed to change. It is unclear why this feature has been provided, but it clearly can lead to dramatic changes in the policy being enforced.[42]

8.3.4 Active Policy Characteristics

Availability TMach (the MK++ kernel in particular) provides very little support for policies intended to guarantee availability of system resources.

Accountability TMach includes an Audit Server which acts as a repository for audit records.

Other than the TMSS, there is little mention of the use of the audit server in any of the TMach documentation. The TMSS is responsible for audit of destruction of subjects. However, the way in which this is done is unusual.

The problem is that subjects are destroyed by kernel operations, and yet the kernel provides no audit facilities. What the kernel does provide is a request, `host_security_id_list`, which is invoked using the host security capability and returns a list of all security IDs currently in use by the kernel. The TMSS periodically queries the kernel using this request to see if a subject no longer exists.

By doing this, the TMSS can determine that a subject has been destroyed. It has no information about how the subject was destroyed, or about the last task associated with the subject, and therefore there would seem to be very little benefit in auditing the subject's destruction. The only apparent reason to do so is to satisfy a requirement of the TCSEC.

It would be tempting to use this as an argument for placing audit facilities in the kernel. But while the kernel may be able to identify more information about the destruction of the last task within a subject, the kernel also lacks all of the information which would be necessary to make the audit record truly useful. For instance, the kernel has no context for what the task represents, such as a shell command.

This example is a general problem with audit in microkernel based systems. Even more generally, for any system generating a concise and meaningful audit trail is a very difficult problem, and there is no indication that TMach has solved this problem.

8.3.5 Failure Ramifications

Like KeyKOS, TMach suffers from the possibility that a single capability which is received in violation of the policy can lead to many other capabilities being received. The problem is not quite as serious in TMach as in KeyKOS, because of explicit labeling and controls over some kernel objects.

To elaborate, we consider the potential consequences of a malicious task gaining one of the protected capabilities listed in Table 1.

- If a malicious task is able to gain a capability to any of the five protected host capabilities or the control capability for a processor set, the task can effectively eliminate all security controls implemented in the entire system through simple requests.

[42] The current TMach databases are configured so that this feature can only be invoked while the system is in maintenance mode.

- A control capability to a clock can be used reset the clock. If the clock is being used for security critical functions, such as time-stamping of critical messages, this can be a valuable weapon. However, by itself we do not know of any way in which the capability can be used directly in an attack.
- A device capability is of little use to a malicious task, precisely because the kernel does provide a second layer of defense on device access, as described in Section 8.1.5.
- A ledger capability is mainly of use for covert channels and denial of service attacks. It is unclear how either capability to a ledger can be used directly in an attack.
- An alarm capability appears to be of use only for covert channels, and only once.
- A message queue capability can be used in various ways. If two malicious tasks operating with distinct security IDs gain capabilities to the same message queue, they can freely share information and capabilities without any future kernel intervention. A send capability to a message queue of a server can potentially be used in a denial of service attack against the server and clients of the server.
- If a malicious task is able to gain a capability for a second task (or a thread in the second task), the malicious task can read the entire state of the second task, and even more significantly, can entirely change the state of the second task so that it executes any program desired. If the second task is any of the privileged servers, the entire security of the system can be effectively defeated by altering the behavior of the server.
- If a malicious task is able to gain a capability for a memory cache object, it can read and write that memory, as well as destroy it. It can also interfere with the actual memory manager, potentially providing a denial of service attack. The value of either attack depends upon the way in which the memory is used and the criticality of the memory manager.

8.4 Assurability Assessment

Like KeyKOS, there are two significant assurability concerns with TMach, the difficulty of proving the invariants and the impossibility of determining whether a system is even in a secure state.

8.5 Summary

Ultimately our conclusions about TMach are quite similar to the conclusions about KeySAFE. While the architecture is certainly capable of satisfying MAC/DAC policies, and with some changes, a broader range of policies, we have serious reservations about the system. The principal reason for these reservations is the difficulty of demonstrating the basic state invariants about the location of capabilities in the system and the potential consequences if the invariants fail to hold.

The complexity in the analysis of the invariants indicates similar complexity in the system. Implementation of a complex system is likely to contain some flaws, and it is therefore important to consider the consequences of even a single failure in the system. In TMach, these consequences are potentially severe, primarily because there are almost no backup security mechanisms controlling kernel operations.

Section 9
DTOS Assessment

This section contains an assessment of the DTOS prototype kernel against the criteria of Section 3. This prototype is a version of the Mach 3 kernel from Carnegie Mellon University with enhancements to provide flexible security mechanisms. The prototype enforces security decisions made by an external component, a security server, which executes as a separate Mach task.

This assessment is organized somewhat differently than the assessments of KeyKOS and Trusted Mach, due to the emphasis on kernel mechanisms for security.

- Section 9.1 briefly describes Mach 3 through comparisons with the MK++ kernel.
- Section 9.2 describes the basic security architecture of a DTOS system with an emphasis on the differences between DTOS and the KeySAFE/TMach approaches and on the separation between security decision making and security enforcement.
- Section 9.3 describes the specific enhancements made to the Mach 3 kernel to support this architecture.
- Section 9.4 contains an assessment of the DTOS prototype against the security policy flexibility criteria of Section 3.
- Section 9.5 contains an assessment of the DTOS prototype against the assurability criteria of Section 3.
- Section 9.6 summarizes the findings of this section.

9.1 Mach 3 Kernel

The DTOS prototype is derived from the Mach 3 kernel developed at Carnegie Mellon University. This kernel is similar to the OSF implementation of Mach 3, which is a predecessor of the MK++ kernel used in Trusted Mach. Though the basic services provided by Mach 3 and MK++ are quite similar, the security features of Mach 3 are deficient to MK++ in many ways.

This section provides a direct comparison of Mach 3 to MK++, organized parallel to Section 8.1.

Layering The MK++ design emphasizes layering and modularity. There is no similar emphasis in Mach 3.

Capabilities The basic capability system in Mach 3 is essentially the same as MK++, though the collection of kernel objects is slightly different.

Security Identifiers Mach 3 makes no use of security identifiers.

Resource Limitations Mach 3 provides neither of the MK++ mechanisms for controlling resource consumption.

Device Control In Mach 3, a task holding a device capability can request any method on a device. There is no way to assign different access rights to a device capability as is possible in MK++.

Virtual Memory Control The Mach 3 kernel provides none of the flexibility in virtual memory control provided by MK++.

In both Mach 3 and MK++, a task holding a capability for an abstract memory object can impersonate the kernel in the paging protocol and therefore the capability should not be granted to untrusted (or less trusted) tasks. In MK++, this is addressed by introducing a memory object representative object, which is a capability providing a controllable indirect reference to the abstract memory object. Mach 3 has nothing analogous to the memory object representative.

9.2 Architecture

A key distinction between the DTOS security architecture and that of KeySAFE and Trusted Mach is the amount of explicit security functionality within the kernel itself. The approach taken by KeySAFE and TMach is to limit the kernel's security responsibility to providing separation which is reflected in the "Total Isolation Invariants". In particular, the kernel in both systems provides no explicit object labeling. And other than the device and virtual memory controls in TMach, neither kernel takes any explicit action to enforce a specific security policy.

The DTOS approach is quite different. The DTOS kernel provides explicit labeling of all kernel objects and controls all accesses according to security decisions made based upon those labels. While the underlying Mach 3 kernel supports a Total Isolation Invariant, the invariant is not relied upon as a basic security mechanism.

Though a considerable amount of the responsibility for security policy enforcement is placed upon the DTOS kernel, the kernel remains independent of any particular security policy. The security decisions which the kernel enforces are made by an external security server which is queried by the kernel.

The DTOS architecture encourages user space servers to maintain a similar relationship to the security policy. In particular, each server can provide labeling and policy enforcement over all objects it implements while relying upon the security server for security decisions. Other than a small demonstration, no such server was developed under the DTOS program, though the government sponsor for DTOS has developed a security enhanced version of the Lites Unix server which follows this model.

9.3 DTOS Kernel Enhancements

The main purpose of DTOS security enhancements to the Mach 3 kernel is to provide enforcement of security server decisions over all kernel operations. The kernel does not ask the security server for a decision based on the actual identity of the client task making a request or an object being accessed. 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.

The specific enhancements made to the Mach 3 kernel to support this model are discussed in the remainder of this section.

- 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 9.3.1.
- 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 9.3.2.
- The kernel uses security decisions from the security server to define the hardware based memory access controls. This is discussed further in Section 9.3.3.
- Extensions were added to the messaging interface to allow clients the ability to set and get the SIDs associated with a message. This is discussed further in Section 9.3.4.
- An interface was added to set or get the capability identifying the security server. This is discussed further in Section 9.3.5.
- An interface to coordinate security enforcement with the security server was added. This is discussed further in Section 9.3.6.

9.3.1 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 (data 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 defaulted based upon the SID of some related object or the client creating the object. 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 an object is generally not allowed to change. The one exception to this rule is that a task SID is allowed to change through a new kernel interface. As a result of the change of a task SID, the SID of threads associated with the task also change. However, the kernel does not provide for retraction of previously granted accesses when the SID of a task changes (see Section 3.1.3), so non-tranquility is not fully supported.

9.3.2 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. Section 6 of the DTOS Lessons Learned Report [31] discusses more completely the motivation for this emphasis on services.

There are almost 200 distinct services controlled by the DTOS kernel, with essentially one access right per service. This allows for extremely fine grained control over the privileges granted to an application, though it can be difficult to configure a security policy to fully take advantage of this feature.

9.3.3 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. Therefore, like all of the other systems in this report, DTOS is ultimately unable to separate read privilege from execute privilege.

9.3.4 IPC Extensions

The Mach IPC interface was extended to provide two additional services which address the "Identification of Recipient" and "Source Authentication" criteria of Section 3.3.3.1:

Identification of Recipient When sending a message in Mach, the sending task identifies the port to which the message is sent but has no way of knowing or controlling which task receives the message. A DTOS extension allows the sender to specify the particular SID that must belong to a task receiving the message. If a task with a different SID holds the receive right to the port to which the message is sent and attempts to receive the message, the message is destroyed. The sender is not notified, which is consistent with Mach semantics which guarantee message enqueueing but cannot make any guarantee about message receipt.

This feature can make it more difficult to provide transparent interposition on interfaces, so it can be overridden if allowed by the policy. To be specific, the policy may allow a task to receive messages through a particular port even if the task has a SID other than the SID specified by the sender.

Comparing these mechanisms with the criteria of Section 3.3.3.1, some deficiencies can still be noted:

- The sender is not notified when a message cannot be received. This is fundamental to the buffering of messages in Mach and the resulting loss of any connection between the message and the sender. A messaging system providing a stronger client/server (e.g., RPC) model would be necessary to eliminate this problem.

- The sender can only specify a single SID, rather than some collection of SIDs. This could be addressed quite simply by increasing the flexibility of the override mechanism with a permission check between the recipient attempting to override and the SID specified by the sender.
- The sender must identify the SID, resulting in an possible loss of policy flexibility. This could be mitigated by allowing the sender to query the security server, so it is not a fundamental problem with the model.

Source Authentication When receiving a message in Mach, the recipient task has no way of knowing which task sent the message. The recipient only knows that the sender held a capability allowing it to send a message to the port through which the message was received. A DTOS extension provides the recipient with the SID of the sending task.

To avoid the potential problems with this feature identified in Section 3.3.3.1, the sender may specify a particular SID which should be attached to the message instead of its own SID. By choosing the SID of another task, the sender may perform transparent interposition or may choose to exercise least privilege over its actions by acting on behalf of a different SID. If the policy provides for this, the SID could also reflect "anonymous" messaging. The security policy is consulted to verify that the sender is allowed to specify a particular SID.[43]

9.3.5 Security Server Identification

Because the security server is external to the DTOS kernel, the kernel must have some mechanism by which the security server is identified. The security server is identified like any other service, by a capability representing a particular port. This is initially set during bootstrap processing.

The DTOS kernel allows a client to request the capability which identifies the security server. This is necessary for other tasks which must interact with the security policy in some way. Permission to access this request is controlled by the security policy.

It is also possible to change this capability, so that the kernel sends queries for policy decisions to a different port. This must obviously be very tightly controlled or there is no guarantee that the kernel is talking to the "correct" security server.

9.3.6 Security Server Interface and Caching

As described so far, the interface between the security server and the kernel simply provides security decisions based upon the SIDs of entities involved in some controlled service. However, to avoid requiring a call to the security server at every access control point, the kernel caches some of the security decisions received from the security server.

Unfortunately, caching of the decisions makes it impossible to support a security policy in which previously granted permissions may be revoked, so other mechanisms were added to provide the security server with some control over caching to gain some support for dynamic security policies. These mechanisms are described briefly here. For more details, see Section 3.4 of the DTOS Lessons Learned Report [31].

[43] Actually, in the current implementation a sender allowed to specify any SID is allowed to specify all SIDs, but this is not as general as it should be.

Cache Flushing The security server can request that the kernel's entire cache be flushed, or that permissions associated with particular SIDs be flushed.

Cache Control Bit Along with each security decision, the security server returns a cache control bit which, if set, indicates that the particular security decision should not be cached in the kernel.

Timeout Values The security server can return a timeout value with each security decision to indicate a limit on the amount of time that the decision can remain in the kernel's cache.

Audit Notification Bit If a security decision is cached, then the security server will not be able to audit every "use" of a security decision. Therefore, along with each security decision the security server returns an audit notification bit. If this bit is set, the kernel will audit the use of the security decision.

These features can make it easier to support certain kinds of dynamic policies, though there are still significant shortcomings as will be discussed in Section 9.4.3.

9.4 Security Assessment

9.4.1 General Characteristics

Mandatory Access Control (MAC) Because security decisions are made by the security server, it is possible to control those decisions in any way. Security attributes are assigned based upon default rules except as explicitly allowed by the security policy. Therefore MAC policies can be supported by DTOS.

Discretionary Access Control (DAC) A security server could be implemented to provide users with explicit control over the security policy as one way to support DAC. DTOS does not directly support an ACL mechanism over kernel objects, however, kernel objects are rarely visible to the user so this is not a significant weakness. Higher level servers can be used to provide ACLs over the objects (such as files) which are typically visible to users.

Information Flow-Transitivity The DTOS kernel makes no assumptions about transitivity, and security servers can be implemented to support transitive or intransitive flow rules.

Information Flow-Covert Channels No significant formal search for covert channels in the DTOS kernel has been performed. The simple audit of security decisions provided by DTOS is most likely insufficient to identify attempts to exploit potential covert channels.

Mutual Suspicion The mutual suspicion problem is very easy to address using a MAC policy such as type enforcement with DTOS. The policy must associate a particular object SID A (a "type" in the terminology of type enforcement) with the executable code for the application being protected and a particular task SID B (a "domain" in the terminology of type enforcement) with a task executing that code. The security policy must be implemented to ensure the following:

- The creator of the application requires that permission to read or execute an object with SID A is only granted to tasks with SID B.
- To ensure that a task with SID B executes correctly, the creator requires that permission to create a task with SID B is only granted to a trusted task creator. This is required because the DTOS kernel cannot ensure that a task is initialized properly.
- The user of the application requires that tasks with SID A only have write permission to objects acceptable to the user.

9.4.2 Least Privilege and Granularity

Granularity of MAC Attributes The only practical limitations on the number of distinct SIDs which can be supported by DTOS arise from the size of the cache of security decisions and performance concerns if most decisions cannot fit into the cache.

Protection Boundaries/Unique Identifiers Like Trusted Mach, the natural protection boundary for DTOS is a collection of tasks with identical security identifiers. There are a few exceptions however. When a task is requesting an operation to be performed on a task or thread, different access rights can be assigned dependent upon whether the task or thread being operated upon is separate from the client. This provides some additional least privilege among tasks with identical SIDs.

Like Trusted Mach, the DTOS kernel provides no mechanism to assign unique SIDs to each task. If this is to be performed, it is the responsibility of the client task requesting creation of a new task through some application-level mechanism agreed upon by all such clients.

Privilege Levels/Least Privilege As mentioned in Section 9.3.2, DTOS supports extremely fine grained least privilege control. With few exceptions, each kernel request is controlled by a distinct access right. This is a strong distinction with the granularity of least privilege controls provided over Mach and TMach kernel operations. For instance, a task holding a capability to a task port in Mach (or TMach) has permission to over 60 separate kernel requests, so the security policy has much less flexibility in how permissions are assigned.

Run-time Least Privilege The DTOS kernel provides two mechanisms for implementing "run-time" least privilege.

- When a task is created, the creating task can specify the SID of the new task. This can be used to create a task with less privileges than may be generally necessary to support an application.
- When a message is sent, the sender can specify a particular sender SID, and this SID may restrict the permissions that the sender would have otherwise.[44]

9.4.3 Characteristics of Dynamic Policies

History Sensitive Policies The audit notifications provided by the DTOS kernel can be used so that the security server can track all permissions which are actually used. If the security server requires immediate notification, it set the cache control bit so that it will be queried whenever a permission is needed.

The only weakness in this is that a permission may be requested but not actually "used". Therefore the security server should not make any assumption about the use of a permission if it would cause an increase in the permissions grantable by the policy. For example, if a user requests permission to change roles in an RBAC policy, the security server should not assume that the change is actually made without some other verification.

Relinquishment Sensitive Policies The DTOS kernel provides no direct support for informing the security server when a previously granted permission is no longer being used. While it may be possible to provide this through indirect means, the limitations discussed below under "Retractive Policies" still apply.

[44] This was actually never fully implemented in the kernel. In particular, if the message is actually a kernel request, the kernel still performs all permission checks based upon the actual SID rather than the sender SID.

Retractive Policies The flush operation and the cache control bits provide a way to ensure that previously granted permissions will not be granted in the future. However, there is no mechanism to abort an operation which is incomplete during a policy change and which is not allowed after a change in security policy. This is the most significant weakness in DTOS' support for dynamic policies, though it is largely due to the design of the underlying Mach 3 kernel.

One particular example of this difficulty is discussed in Section 3.3.3.2 under the heading "Message Buffering and Dynamic Policies".

Retractive Destruction The DTOS kernel provides no support for directly destroying tasks or objects based upon the SID. It is possible to perform such an operation indirectly by removing all permissions associated with a particular SID, then finding all objects with the SID and then destroying them, though this requires some sensitive permissions to be granted to the client which is performing these operations.

Non-tranquility DTOS support non-tranquility directly by changing the SIDs of tasks (and threads), with the limitations noted in Section 9.3.1. Non-tranquility can also be implemented by changing the mapping between SIDs and security attributes in the security server, though this simultaneously changes the attributes of all objects with a particular SID.

9.4.4 Active Policy Characteristics

Availability No enhancements have been added to DTOS specifically to address availability concerns. The fine grained least privilege controls can address some availability concerns, and may actually be necessary to address all concerns, but this is certainly not sufficient because of the fundamental shortcomings of Mach.

Accountability The only audit mechanism provided by the DTOS kernel is the audit of security decisions described in Section 9.3.6. The kernel provides no guarantee that these records are maintained after they are received by the audit server.[45]

9.4.5 Security Management

None of the systems in this report have very well developed security administration tools. However, because of the extreme fine-grained control possible in DTOS, good tools are especially necessary to properly configure a security policy for DTOS.

9.4.6 Failure Ramifications

One of the strengths of the DTOS approach is that the ramifications of a single failure are generally quite limited. If the security policy is misconfigured so that some permission is granted inappropriately, only the service granted by that permission can be accessed. This is a considerable improvement over a system like KeyKOS or TMach in which a single errant capability can potentially be used to gain access to many different services.

In many cases, DTOS actually presents multiple layers of security, so a single failure may not allow any services to be accessed. For instance, capabilities are controlled both when received and when used.

[45] Additional audit mechanisms are being added to DTOS under a different program, Adaptive Security Policy Experiences. No documentation of these mechanisms is available at this time.

However, there are still two single points of failure for the entire security system. The kernel is dependent upon the correct identification of the default pager for management of kernel memory and of the security server. If another task can replace or impersonate either of these servers, then all kernel controls are effectively nullified. However, this can only be done through explicit kernel requests and permission to grant these requests can be very tightly controlled (or not granted at all).

9.4.7 Capabilities and Mandatory Access Control

It is interesting to note that DTOS provides both of the controls which are suggested as ways to support MAC policies in a capability system. Permission to hold a capability is checked when a capability is first gained by some task, and permission to use a capability is checked whenever the capability is used. There is even an additional control when a capability is sent in a message to verify that the sender is allowed to pass the capability.

9.5 Assurability Assessment

One of the results of the DTOS program is that it is not possible to build a highly-assured system from the CMU Mach 3 code base (see Section 5.2 of the DTOS Lessons Learned Report [31]). Therefore this section considers assurability of the DTOS security architecture, and in particular, a comparison with the KeySAFE and TMach approaches.

Two features of the DTOS security mechanisms lead to significant assurability benefits compared with the other approaches:

- DTOS provides explicit labeling over all kernel objects, while kernel objects in the other systems are labeled implicitly through their relationship with some task.
- DTOS provides a single form of access control over all objects, while KeySAFE relies upon several different aspects of the capability mechanism and TMach relies upon the capability mechanism and additional explicit controls added for virtual memory and devices.

One example of how these increase assurability is the simplicity of the solution to the "Mutual Suspicion" problem (see Section 3.1.1) in DTOS compared to the complexity of the solution embodied in the KeyKOS factory mechanism. Assurability is always improved with simplicity.

Another advantage of this model is that it is quite simple to immediately determine the maximum set of privileges granted to any task in the system.

However, perhaps the best example of the assurability differences is to consider the "Total Isolation Invariant" that must be proven for both KeySAFE and TMach. These invariants describe all of the capabilities that can be held by a task (or tasks) which is to be isolated from the rest of the tasks in the system. To state and prove these invariants requires a considerable amount of analysis of every kernel request to recognize all of the ways in which capabilities can be passed by one task or acquired by another task, and all of the ways in which one capability can be used to gain other capabilities.

On the other hand, the DTOS security mechanisms explicitly control which capabilities can be held by any task. They further explicitly control exactly how each capability can be used (i.e., which requests can be made using a capability). Analysis of this is primarily limited to the security mechanisms themselves and yet the results of the analysis are much stronger.

It is interesting to note that since TMach provides explicit controls over server based objects, it does have a similar assurability advantage over KeySAFE at the server level.

9.6 Summary

DTOS provides explicit security labeling of kernel objects and controls kernel services based upon these labels through a single security mechanism. There are three significant security and assurability advantages of the DTOS approach compared to any of the other systems considered in this report:

- The DTOS approach allows for a much simpler assurance argument identifying the maximum set of capabilities usable by any particular task.
- The DTOS approach provides the most flexibility in the security policy. This is not surprising since the same range of policy flexibility was not an explicit goal of the other systems, except possibly KeyKOS.[46]
- The DTOS approach provides the strongest form of least privilege and isolation of security failures.

The only significant shortcoming of DTOS compared with the other systems in this report is the lack a flexible underlying capability system for providing "run-time" least privilege, such as in KeyKOS or Spring. This is actually less a weakness of DTOS than it is a weakness of Mach 3. The approach taken to incorporate security into Mach 3 to construct the DTOS prototype could be just as easily used to incorporate security into other similar kernels, including Spring, KeyKOS and MK++. If a system with a more flexible underlying capability mechanism were used, then this approach would result in a system with all of the benefits of DTOS and with the flexible run-time least privilege mechanism of the underlying system.

DTOS does have several shortcomings which are shared with the other systems in this report. These will be discussed more in Section 10.

[46]The KeyKOS documentation which we have asserts that policy flexibility is a goal, but without any sense of the extent to which it was really considered.

Section 10 Conclusions

This report 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.

The five system assessments are presented in an order that indicates roughly how well the systems meet the criteria, in increasing order: Amoeba, Spring, KeyKOS, Trusted Mach, and DTOS. It is important to recognize that this is not an analysis of how well any particular system meets the design goals for that system; in fact, none of the systems were designed to meet all of the criteria in this report.

Furthermore, since the basic categories from which the criteria were derived represented goals for the DTOS prototype, it is not surprising that DTOS meets more of the criteria than the other systems which were considered. It is perhaps more surprising that several of the criteria were not met by DTOS. In many cases, this is because the specific criteria were developed well after the DTOS prototype was designed, and experiences with the system were a strong influence on the criteria. In other cases, such as audit, DTOS was never planned to meet the criteria, though this was a known deficiency in the original design. In still other cases, the difficulties are inherent the underlying Mach 3 kernel rather than the DTOS security mechanisms.

We conclude this report with a summary of the criteria which are not satisfied by any system considered in this report, organized by the categories of Section 3:

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 [28] discusses some of the open issues in this area (these issues are also summarized in Section 11 of the DTOS Lessons Learned Report [31]).

Least Privilege and Granularity The only significant shortcoming in this category is that none of the systems support (at least in a natural implementation) MAC policies which depend upon unique identifiers on all entities, or all entities of some type (e.g., all files). The difficulty with implementing mandatory access control and providing unique labels is that labels are usually assigned through default rules.

Characteristics of 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.

Active Policy Characteristics 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 accountability (for security) 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.

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.

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 *11*

Notes

11.1 Acronyms

ACL Access Control List

AIL Amoeba Interface Language

AIM Access Isolation Mechanism

API Application Program Interface

BLP Bell LaPadula

CCA Covert Channel Analysis

CMU Carnegie Mellon University

COTS Commercial Off The Shelf

DAC Discretionary Access Control

DTOS Distributed Trusted Operating System

FSPM Formal Security Policy Model

FTLS Formal Top Level Specification

HAL Hardware Abstraction Layer

IDL Interface Definition Language

IPC InterProcess Communication

MAC Mandatory Access Control

MIG Mach Interface Generator

MLS MultiLevel Secure

OSF Open Software Foundation

RNS Root Name Server (TMach)

RPC Remote Procedure Call

SID Security Identifier

TCB Trusted Computing Base

TCSEC Trusted Computer System Evaluation Criteria

TMach Trusted Mach

TMSS Trusted Mach Subject Server

VM Virtual Memory

Appendix A
Bibliography

[1] 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.

[2] 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.

[3] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992.

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

[5] David 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.

[6] Bill Frantz. KeyKOS - a secure, high-performance environment for S/370. In *Proceedings of SHARE 70*, pages 465–471. SHARE Inc., Chicago, February 1988.

[7] Jeff Graham, April 1997. Email message.

[8] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

[9] Norman Hardy. Computer security system. U.S. Patent 4,584,639, April 1986.

[10] Norman Hardy. The confused deputy. *Operating Systems Review*, 22(4):36–38, October 1988.

[11] R. Y. Kain and C. E. Landwehr. On access checking in capability-based systems. In *IEEE Symposium on Security and Privacy*, pages 95–100, Oakland, CA, April 1986.

[12] Key Logic, Inc. Introduction to KeySAFE. Key Logic Document SEC009.

[13] Key Logic, Inc. KeyKOS principles of operation. Key Logic Document KL002.

[14] Carl E. Landwehr et al. A taxonomy of computer program security flaws, with examples. *ACM Computing Surveys*, 26(3), September 1994.

[15] Keith Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, November 1992.

[16] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, May 1993.

[17] 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.

[18] James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. An overview of the Spring system. In *A Spring Collection*. Sun Microsystems, Inc., 1994.

[19] 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.

[20] 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.

[21] Open Software Foundation, Inc. *MK++ Kernel Executive Summary*, November 1995.

[22] Open Software Foundation, Inc. *MK++ Kernel Interfaces*, November 1995.

[23] Open Software Foundation, Inc. *MK++ Kernel High Level Design*, January 1996.

[24] Secure Computing Corporation. Adding Security to Commercial Microkernel-Based Systems. CDTOS CDRL A001, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1996.

[25] Secure Computing Corporation. DTOS Formal Security Policy Model. DTOS CDRL A004, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1996.

[26] Secure Computing Corporation. DTOS Formal Top-Level Specification. DTOS CDRL A005, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1996.

[27] 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.

[28] Secure Computing Corporation. DTOS Covert Channel Analysis Plan. DTOS CDRL A017, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, May 1997.

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

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

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

[32] SunSoft. *A Spring Collection*. Sun Microsystems, Inc., 1994.

[33] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Inc., 1995.

[34] 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.

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

[36] Trusted Information Systems, Inc. *Trusted Mach Class Library Executive Summary*, November 1995.

[37] Trusted Information Systems, Inc. *Trusted Mach Class Library Interface*, November 1995.

[38] Trusted Information Systems, Inc. *Trusted Mach Device Server Executive Summary*, December 1995.

[39] Trusted Information Systems, Inc. *Trusted Mach Device Service Interface*, December 1995.

[40] Trusted Information Systems, Inc. *Trusted Mach File Server Executive Summary*, November 1995.

[41] Trusted Information Systems, Inc. *Trusted Mach File Service Interface*, November 1995.

[42] Trusted Information Systems, Inc. *Trusted Mach Host Control Server Executive Summary*, November 1995.

[43] Trusted Information Systems, Inc. *Trusted Mach Host Control Service Interface*, November 1995.

[44] Trusted Information Systems, Inc. *Trusted Mach Item Control Service Interface*, December 1995.

[45] Trusted Information Systems, Inc. *Trusted Mach Item Manager Library Interface*, December 1995.

[46] Trusted Information Systems, Inc. *Trusted Mach Item Manager Service Interface*, December 1995.

[47] Trusted Information Systems, Inc. *Trusted Mach Name Service Interface*, November 1995.

[48] Trusted Information Systems, Inc. *Trusted Mach Root Name Server Executive Summary*, November 1995.

[49] Trusted Information Systems, Inc. *Trusted Mach Security Features User's Guide*, November 1995.

[50] Trusted Information Systems, Inc. *Trusted Mach Subject Server Executive Summary*, November 1995.

[51] Trusted Information Systems, Inc. *Trusted Mach Subject Service Interface*, November 1995.

[52] Trusted Information Systems, Inc. *Trusted Mach System Architecture*, October 1995.

[53] Vrije Universiteit en Stichting Mathematisch Centrum. *The Amoeba Reference Manual—Programming Guide*, 1994.

[54] Vrije Universiteit en Stichting Mathematisch Centrum. *The Amoeba Reference Manual—System Administration Guide*, 1994.

[55] Vrije Universiteit en Stichting Mathematisch Centrum. *The Amoeba Reference Manual—User Guide*, 1994.