Part Number 00-0930896A001 Rev B

Version Date 17 February 1999

Assurance in the Fluke Microkernel Formal Security Policy Model

CONTRACT NO. MDA904-97-C-3047
CDRL SEQUENCE NO. A003

Prepared for:
Maryland Procurement Office

Prepared by:

Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113

Authenticated by ______________________ Approved by ______________________

(Contracting Agency) (Contractor)

Date ______________________ Date ______________________

Assurance in the Fluke Microkernel Formal Security Policy Model

Secure Computing Corporation

Abstract

This document presents the formal model of the security control requirements for the Fluke microkernel. More generally, it provides a security model for the Fluke and Flask systems, discussing the overall security architecture, security requirements on the different elements of the architecture and the range of policies supportable within the architecture.

Part Number	00-0930896A001 Rev B
Created	2 July 1997
Revised	17 February 1999
Done for	Maryland Procurement Office
Distribution	Secure Computing and U.S. Government
CM	`/projects/cmt/rcs/fluke/docs/fspm/RCS/fspm.vdd,v 1.14` `17 February 1999`

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

Contents

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

2 Security Architecture **3**
2.1 Fluke . . . 3
2.1.1 The Recursive Virtual Machine Architecture . . . 4
2.1.2 Requirements for Total Isolation of Virtual Machines . . . 6
2.1.3 Security Features . . . 8
2.1.4 Analysis of Requirements for Total Isolation . . . 12
2.1.5 Allowing Controlled Interactions Between Virtual Machines . . . 13
2.2 Flask . . . 13
2.2.1 Overview . . . 14
2.2.2 Flask Integration with the Fluke Microkernel . . . 20
2.2.3 Flask Additions to Fluke Object Managers . . . 23
2.2.4 Flask and the Recursive Virtual Machine Architecture . . . 26
2.2.5 Revisiting the Requirements for Total Isolation . . . 27

3 Microkernel Control Policy **28**
3.1 Microkernel State Model . . . 28
3.1.1 Common Object State . . . 29
3.1.2 Threads . . . 31
3.1.3 Tasks . . . 32
3.1.4 Regions . . . 33
3.1.5 Mappings . . . 34
3.1.6 Ports . . . 35
3.1.7 Port Sets . . . 35
3.1.8 Mutexes . . . 35
3.1.9 Condition Variables . . . 36
3.1.10 References . . . 36
3.1.11 Flask Security Data . . . 37
3.1.12 Physical Memory Management . . . 40
3.1.13 IPC Connections . . . 40
3.1.14 Kernel Requests . . . 41
3.1.15 Summary . . . 42
3.2 Microkernel Services and Requirements . . . 44
3.2.1 Overview . . . 44
3.2.2 Common Object State . . . 49
3.2.3 Threads . . . 52
3.2.4 Tasks . . . 58
3.2.5 Regions . . . 60
3.2.6 Mappings . . . 63
3.2.7 Ports . . . 66
3.2.8 Port Sets . . . 68
3.2.9 Mutexes . . . 68

3.2.10 Condition Variables 70
3.2.11 References 72
3.2.12 Security 74
3.2.13 Memory 75
3.2.14 IPC 78
3.2.15 Outcalls 80
3.2.16 Request Invocation 81
3.2.17 Service Requirements Summary 82

A Bibliography 84

B Implementation Notes 85

B.1 State Model Correspondence 85
B.1.1 Common Object State 86
B.1.2 Threads 86
B.1.3 Tasks 87
B.1.4 Regions 87
B.1.5 Mappings 88
B.1.6 Ports 88
B.1.7 Port Sets 88
B.1.8 Mutexes 88
B.1.9 Condition Variables 89
B.1.10 References 89
B.1.11 Security 89
B.1.12 Kernel Requests 90
B.1.13 Physical Memory Management 90
B.1.14 IPC Connections 90
B.1.15 Summary 91
B.2 Microkernel State Visbility 91
B.2.1 Common Object State Visibility 91
B.2.2 Thread State Visibility 91
B.2.3 Task State Visibility 92
B.2.4 Region State Visibility 92
B.2.5 Mapping State Visibility 92
B.2.6 Port State Visibility 92
B.2.7 Mutex State Visibility 92
B.2.8 Condition Variable State Visibility 92
B.2.9 Security Data Visibility 93
B.2.10 Memory State Visibility 93
B.2.11 IPC State Visibility 93
B.3 Request Mapping 93
B.3.1 Common Object Transformation Services 93
B.3.2 Thread Transformation Services 94
B.3.3 Task Transformation Services 95
B.3.4 Region Transformation Services 95
B.3.5 Mapping Transformation Services 96
B.3.6 Port Transformation Services 96
B.3.7 Mutex Transformation Services 96
B.3.8 Condition Variable Transformation Services 97
B.3.9 Reference Transformation Services 97
B.3.10 Security Data Transformation Services 97

B.3.11 Memory Transformation Services 97
B.3.12 IPC Transformation Services 98

C Requirements Summary 99

List of Figures

1 An Example of a Recursive Virtual Machine Tree 4
2 Interfaces in a Recursive Virtual Machine 5
3 Links Between Kernel Objects 10
4 Flask Controls over Inter-object Links 22

List of Tables

1	Requirements on Standard Object Services	48
2	Requirements on Common Object Services	50
3	Requirements on Thread Services	56
4	Requirements on Task Services	59
5	Requirements on Region Services	62
6	Requirements on Mapping Services	65
7	Requirements on Port Services	67
8	Requirements on Port Set Services	68
9	Requirements on Mutex Services	69
10	Requirements on Condition Variable Services	71
11	Requirements on Reference Services	73
12	Requirements on Memory Services	77
13	Requirements on IPC Services	79
14	Requirements on Common Object Services	99
15	Requirements on Thread Services	99
16	Requirements on Task Services	100
17	Requirements on Region Services	100
18	Requirements on Mapping Services	100
19	Requirements on Port Services	100
20	Requirements on Port Set Services	100
21	Requirements on Mutex Services	101
22	Requirements on Condition Variable Services	101
23	Requirements on Reference Services	101
24	Requirements on Memory Services	101
25	Requirements on IPC Services	102

List of PVS Theories

address_types 30
sid 30
sob 30
thread 32
task 32
memprot 33
region 33
mapping 34
port 35
mutex 35
cond_var 36
ref 37
security_data 37
cache 39
kernel_request 41
kernel_state 42
common_services 49
common_service_rqmts 50
thread_services 53
thread_service_rqmts 55
task_services 58
task_service_rqmts 59
region_services 60
region_service_rqmts 62
mapping_services 63
mapping_service_rqmts 64
port_services 66
port_service_rqmts 67
mutex_services 69
mutex_service_rqmts 69
cond_var_services 71
cond_var_service_rqmts 72
ref_services 72
ref_service_rqmts 73
security_services 74
memory_services 76
memory_service_rqmts 77
ipc_services 78
ipc_service_rqmts 79
invocation_services 81
invocation_service_rqmts 81
service_rqmts 82

Section 1 Scope

1.1 Identification

This document contains the Formal Security Policy Model, CDRL A003, for the Assurance in the Fluke Microkernel program, contract MDA904-97-C-3047.

1.2 Program Overview

The goal of the Assurance in the Fluke Microkernel program is to support security in the Fluke operating system project at the University of Utah by applying known assurance techniques to determine the security and assurability of key elements of the Fluke technology, and by performing research to extend assurance techniques to address unique aspects of Fluke. The program is especially concerned with the Flask implementation of Fluke.

Flask is a successor to the DTOS [10] and Synergy [8] programs, under which security enhanced versions of the Mach 3 microkernel and the Lites Unix server were prototyped. The prototype has been used as a research platform for secure applications projects and as a basis for assurance research under the DTOS program. Flask is a similar enhancement to the Fluke system, providing support for mandatory access control policies with a flexible design that separates security decisions into a component distinct from security enforcement.

1.3 Document Overview

This document lays the groundwork for performing analysis of the Flask system by identifying the security requirements which it is intended to satisfy. It begins with a description of the Flask security architecture, with emphasis on the microkernel's role in the architecture but also discussing other components of a complete Flask system. The document then presents the control policy for the Flask microkernel.

The security policy is formalized in the PVS specification language [7]. This document is also produced in a version which does not contain the PVS specification [12]. The two versions are otherwise identical and are generated from the same source text.

The document is structured as follows:

- Section 1, **Scope**, defines the scope and this overview of the document.
- Section 2, **Security Architecture**, provides an overview of the security architectures of Fluke and Flask.
- Section 3, **Microkernel Control Policy**, provides a model of the state of the microkernel and defines its control policy.
- Appendix A, **Bibliography**, provides citations for all referenced documents.

- Appendix B, **Implementation Notes**, describes the correspondence between the model presented in Section 3 and the implementation.
- Appendix C, **Requirements Summary**, contains copies of all of the tables which define the control requirements in Section 3, providing a single concise reference.

Editorial Note:
Throughout this document, occasional mention to the "current implementation" of Fluke will be made. This refers to a code snapshot dated August 25, 1998.

Section 2 Security Architecture

This section provides an overview of the security architectures of Fluke and Flask.

A necessary first step towards meeting the goals of the Fluke Assurance program is to understand the security objectives which the system is intended to satisfy, the requirements necessary to meet these objectives and the features which are intended to be used to meet the requirements. The main purpose of this section is to document these aspects of the system, both because this information has not previously been collected into one document and also to ensure a common understanding among the three teams (i.e., at the University of Utah, NSA, and Secure Computing) involved in the Flask project.

This section also provides some initial analysis of the ability to reason about the system's ability to meet these basic requirements. This analysis will be expanded upon (possibly in other documents) as the program progresses and ultimately will be summarized in the program's Final Report [11].

This section first discusses Fluke independent of Flask, and then discusses the Flask system. This organization serves the purpose of clearly identifying those features added in the Flask implementation and the specific advantages that these additions provide.

2.1 Fluke

The Fluke security model is based upon a Recursive Virtual Machine (RVM) architecture [5, 4], which provides a tree of nested virtual machines in which processes can execute. The essential security property of this model is that processes can be isolated by locating them in non-overlapping virtual machines.

This section describes this architecture, identifies the security responsibilities of the various components in this architecture, and discusses how the system can be analyzed to assure that the components satisfy their responsibilities. It is organized as follows:

- Section 2.1.1 provides an overview of the RVM architecture.
- The basic objective of this architecture is the ability to totally isolate processes executing in disjoint virtual machines. Section 2.1.2 identifies, at a very high level, the security requirements required to meet this objective.
- Section 2.1.3 describes the security features of the components of the RVM architecture and how they contribute to the requirements.
- Section 2.1.4 considers the feasibility of analyzing the system's ability to use the security features describes in Section 2.1.3 to meet the security requirements identified in Section 2.1.2.
- A more realistic objective of a real security policy is to control the interactions between processes rather than to isolate them completely. Section 2.1.5 discusses how the requirements and analysis are affected by the possibility of allowing some interactions between processes in disjoint virtual machines.

2.1.1 The Recursive Virtual Machine Architecture

The RVM model can be represented by a tree in which every process in the system exists as a node in the tree.[1] The root node represents the processor hardware and the Fluke microkernel. The root node and interior nodes are known as *nesters*, which provide the Fluke virtual machine interface to their descendents (further away from the root). An example of such a tree is shown in Figure 1.

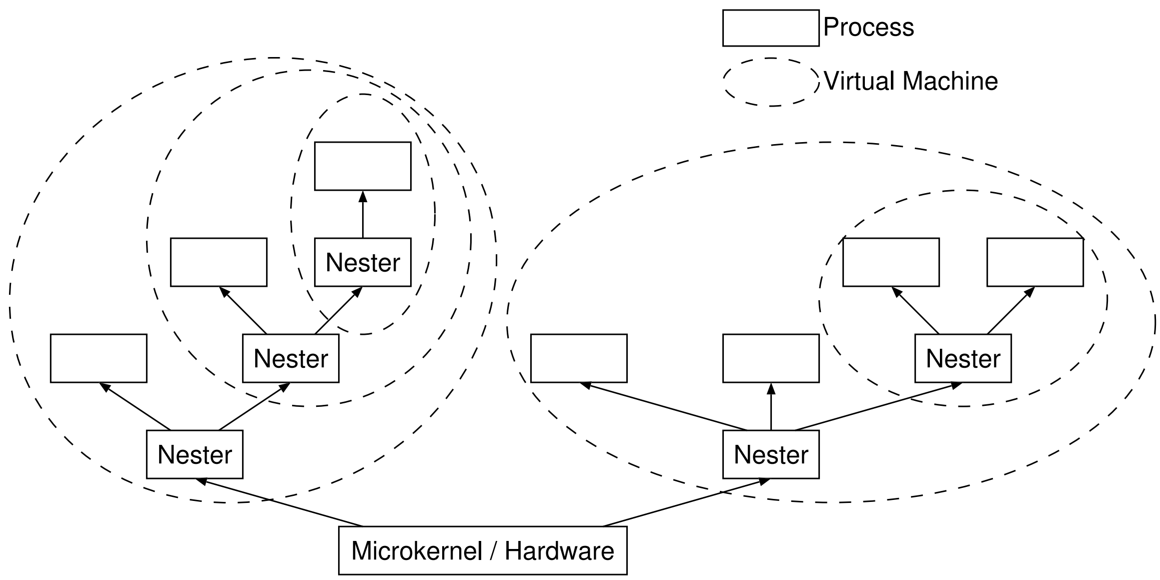

Figure 1: An Example of a Recursive Virtual Machine Tree

The Fluke virtual machine interface is divided into three layers:

- The hardware interface,
- The Fluke microkernel (system call) interface, and
- The Fluke Common Protocols (CP) interface.

Unlike typical virtual machine architectures, the Fluke RVM architecture does not require the entire virtual machine interface to be reimplemented at each level, i.e., by each nester. Nesters only reimplement the CP interface, and often, each nester only reimplements a portion of the CP interface. This lessens the performance degradation typically associated with virtual machine architectures while retaining many of their benefits.

The CP interface is actually a collection of several distinct interfaces, including a parent interface, memory allocation interface, file system interface, process interface, and others. Every nester supports the parent interface, which its children call to find the services provided by each of the other CP interfaces. The nester can allow its children to access these other interfaces from lower level nesters in the RVM tree, or it may reimplement them itself.

[1] Throughout Section 2, a "process" should be interpreted conceptually, not as a specific entity. In particular, "process" may refer to something which consists of multiple threads and address spaces.

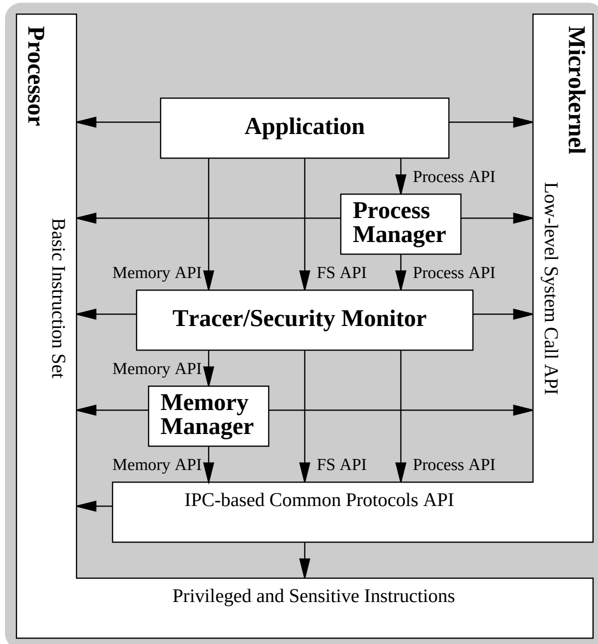

Figure 2: Interfaces in a Recursive Virtual Machine

Figure 2[2] details one possible path through an RVM tree, beginning at the hardware and microkernel levels and continuing through several nesters to the level of an application at a leaf of the tree. The Figure demonstrates the following:

- All processes have access to the basic instruction set of the processor, while only the microkernel, which executes in supervisor mode, has access to privileged processor instructions.
- All processes have access to the Fluke microkernel interface, referred to as the “Low-level System Call API” in the figure.
- The microkernel (through the kernel server) also provides the lowest level implementation of the CP interface, referred to as the “IPC-based Common Protocols API”.
- The figure shows three of the CP interfaces, memory allocation, file system and process. It does not show the parent interface directly, though it is implied by the layering that each process uses the parent interface from the next lowest level.
- The Memory Manager and Process Manager nesters each reimplement only part of the CP interfaces, allowing their descendents to receive some services from lower levels. For example, even though the Application is a child of the Process Manager, it receives its memory and file system services from the Tracer/Security Monitor nester. The Tracer/Security Monitor itself reimplements all of the CP interfaces.

This architecture presents a flexible security model in which a nester can control the extent to which its descendents can interact with lower level nesters. However, as Figure 2 demonstrates, there are opportunities for interaction between processes through the hardware and

[2] This figure was taken directly from [4].

microkernel interfaces as well as through those CP interfaces which are not reimplemented by a nester. The remainder of Section 2.1 focuses on the ability to control these interactions and to completely eliminate them when required.

2.1.2 Requirements for Total Isolation of Virtual Machines

The fundamental security objective of the RVM architecture is that nesters should have complete control over the virtual machine environment accessible to their descendents and over the interactions between their descendents and processes in other virtual machines.

At a minimum, if this objective is to be met, then the architecture must be capable of preventing interactions which bypass the nesters which create the virtual machines. In other words, the architecture must be able to totally isolate disjoint virtual machines.

This objective of total isolation is a convenient starting point for analyzing the security architecture. In this section, we refine the objective into high-level requirements on the components of the architecture which provide each of the three interface layers in the Fluke virtual machine.

2.1.2.1 Requirements on the Processor Interface Since all processes can directly access the basic (unprivileged) processor instruction set, the objective of total isolation requires that the processor prevent processes in disjoint virtual machines from communicating through these instructions. Since Fluke is intended to be implemented on a range of processor hardware, there can be no specific discussion of the hardware. However, the hardware is generally expected to meet this requirement by guaranteeing that processes with disjoint processor states[3] are incapable of communicating through unprivileged instructions.

This requirement on the processor hardware can be stated as follows:

> If there is no shared processor state between processes in disjoint virtual machines, then it is impossible for any particular pair of processes in disjoint virtual machines to communicate through unprivileged hardware instructions.

Since the processor is unaware of the concept of a virtual machine, it is entirely the responsibility of higher level components of the architecture to guarantee that there is no shared processor state between processes in disjoint virtual machines.

2.1.2.2 Requirements on the Microkernel System Call Interface Since all processes can directly access the system call interface to the microkernel, the objective of total isolation requires that the microkernel prevent processes in disjoint virtual machines from communicating through system calls. The microkernel is also responsible for maintaining the processor state of processes, so it has the additional requirement of ensuring disjoint processor states for processes in disjoint virtual machines.

The microkernel relies upon a capability mechanism to meet these requirements. The services provided by the microkernel interface can only be accessed by a client holding the proper capabilities to kernel objects. If two processes do not have a capability to the same kernel object, they cannot interact.

The requirement on the microkernel is therefore:

[3] The term "processor state" is used here to refer to the state of a process which is loaded or unloaded during a context switch, such as register values and page tables. It appears that this is called the "context switch state" (`csw_state`) in the Fluke code. The precise definition of "disjoint processor states" is processor dependent, but it certainly implies that there is no shared memory.

> If there are no kernel objects with capabilities shared between disjoint virtual machines, then there is no shared processor state between processes in disjoint virtual machines and it is impossible for any particular pair of processes in disjoint virtual machines to communicate through microkernel system calls.

Like the processor hardware, the microkernel has no concept of a virtual machine. Therefore it is entirely the responsibility of higher level components of the architecture to guarantee that there are no shared capabilities between disjoint virtual machines.

2.1.2.3 Requirements on Nesters

The requirements on the hardware and microkernel system call interfaces provide that no communications between disjoint virtual machines can occur through those interfaces, so long as no capabilities are shared between virtual machines. The nesters are left with the responsibility of ensuring that capabilities are separated and also that communications can be prevented between virtual machines through the interfaces provided by the nesters.

These responsibilities are complicated by the fact that a nester spans virtual machines.[4] It holds capabilities for the virtual machine created by its parent and for the virtual machine which it creates. Furthermore, the children of a nester may themselves be nesters, and a nester has access to all capabilities held by any of its descendents. Since capabilities held by a process are accessible to its ancestors, requirements on a nester can only define how the nester communicates with its siblings and descendents.

We distinguish three types of capabilities held by a nester, and define different requirements for each type:

- Capabilities which the nester receives through interactions within its parent's virtual machine.

  A nester must not pass these capabilities to its descendents.[5]

- Capabilities which the nester creates for use in its own virtual machine. These capabilities may be created by making microkernel requests or possibly requests to the parent.

  A nester must not pass these capabilities to its siblings, or to descendents other than its immediate children.

- Capabilities which are created by its descendents for their own use, possibly as part of a descendent's virtual machine.

  A nester must not pass these capabilities to its siblings, or to any descendents other than the descendent which created the capability.

These three requirements address the first responsibility of the nester, to isolate capabilities between virtual machines.

The nester also has the responsibility of preventing communication through the interfaces which it provides to its children. To meet this responsibility, the data handled by a nester must be effectively segregated into the same three categories as capabilities, and the requirements on the communication of data is analogous to the requirements on passing capabilities. There is no need to explicitly restate the requirements with these additions.

[4] Perhaps it would be better to show the boundary of a virtual machine in Figure 1 as passing directly through a nester rather than including the nester.

[5] Clearly, this requirement is violated when a nester chooses not to implement one of the CP interfaces. But in that case the nester is not choosing a policy of total isolation, so it is not relevant to this discussion.

It is an essential feature of the architecture that a nester can only affect its own branch of the virtual machine tree. If a nester fails to meet its requirements, the most serious effect that can occur is that all of the descendents of the nester will effectively be siblings of the nester in the virtual machine tree. This feature indicates a trust hierarchy among nesters, where a nester must trust its ancestors but has no need to trust its descendents.

2.1.3 Security Features

This section discusses some of the features of the Fluke components which are provided to help the system meet the requirements just presented. Most of the features discussed here are part of the lower-level system call layer of the microkernel. Processor features are not described since the system is intended to be processor independent. Higher level features (other than segments and mempools) are not described since there do not appear to be any specific features of nesters which are used to meet their requirements.

The specific features to be discussed are the following:

- Section 2.1.3.1 describes how the microkernel defines the memory protections to be enforced by the processor hardware.
- Section 2.1.3.2 describes the lack of externally visible global state in the microkernel.
- Section 2.1.3.3 describes the microkernel capability system.
- Section 2.1.3.4 describes controls over access to hardware resources.

2.1.3.1 Controlling Memory Accesses

The microkernel is responsible for setting the permission bits on memory, which are then enforced by the hardware. This section provides a high-level description of how the microkernel determines the correct bits to set.

There are four possible permission bits at the software level, read, write, execute and create. The create permission controls whether a kernel object can be created at a particular address, and is enforced purely at the software level (write permission is also required to create an object because it results in writing to the memory). The other three bits are passed to the hardware level, with one important exception – on some architectures there is no hardware equivalent of the execute bit.[6] This is unfortunate because when read permission implies execute permission, it is more difficult to control attacks that involve overwriting executable code or changing the program counter to execute arbitrary code, e.g., by overwriting the stack.

To describe how the software protection bits are set we must discuss how an address space is populated with memory.

In general, an address space receives memory through mapping from another address space. Memory mapping proceeds in two stages. First, a region object is created to export a range of memory from the source address space, and then a mapping object is created to import memory from the region object to the destination address space. Protection bits are assigned during each of these operations. The protection bits associated with the region object define the maximum set of permissions that the source will allow the destination to receive, while the protection bits associated with the mapping object define the permissions actually required by the destination.

[6] It is common to fail to enforce execute permission distinct from read permission at the hardware level, for instance, many processor architectures do not provide a separate execute bit on individual pages (only on segments).

The permissions which are actually granted to the mapped memory in the destination address space are no more than the intersection of the protection bits in the region and mapping objects. However, the permissions are further limited by the protection bits in the region object from which the source address space mapped the memory, and so on back to the initial source of the memory. Note that the mapping object used by the source address space has no affect on the permissions in the destination address space.[7]

What is missing in the previous discussion is the "original source" of memory. The only source of newly allocated memory is the microkernel itself (through the kernel server CP interface). Memory is allocated by requesting the creation of a segment. The kernel server responds to a request for a new segment by creating a new task, and returns a region exporting the desired amount of memory from that task, with all permissions allowed.

2.1.3.2 No Externally Visible Global State

The Fluke virtual machine model makes it essential that there be no global state exported from the microkernel. For instance, there are no universal object identifiers or resource usage statistics.

2.1.3.3 Microkernel Capability System

This section describes the capability system for controlling system calls. Most of the discussion is independent of any particular implementation, though some implementation details are provided at the end of the discussion.

All Fluke system calls have one or more capabilities as input parameters, and the capabilities must be of the correct type for the request to be honored.[8] For each kernel object, there are three types of capabilities, a write capability, a read capability, and a reference capability.[9][10]

A write capability is effectively all powerful, because read and reference capabilities can be created from a write capability. However, read capabilities and reference capabilities are generally incomparable, as each is used to control different requests and neither can be created from the other.

The following is a list of the kinds of system calls provided by the microkernel and the capabilities required by each. This list is not intended to be complete (though it is intended to be accurate), rather it is intended to provide some sense of the way in which capabilities are used and some of the complications that can arise.

- Requests which modify the state of a kernel object require at least a write capability to the object. Other capabilities may be required depending upon how the state is modified. The state for each object is separated into simple state (data) and inter-object links. Modification to simple state only requires a write capability to an object. However, modifying a link from one object (the source) to another (the target) requires both a write capability to the source object and a reference capability to the new target object.

[7] This is perfectly consistent with the intention of region and mapping objects. For instance, an address space that is granted write permission to memory through a region object can export the memory writeable even if it does not choose to actually map the memory with the write permission turned on. This is potentially useful for supporting least privilege.

[8] There are a few exceptions to this rule. No capability is required for certain requests for which the client thread is operating on itself or the address space in which it executes.

[9] There is one exception to this rule. A reference object only has write and read capabilities.

[10] The Fluke team may reserve the term capability for reference objects, and would discuss what we call read and write capabilities as read and write permissions to a memory address. We avoid the connection between capabilities and memory permissions for now in order to simplify the discussion and to emphasize what is basically a capability model.

While this appears rather straightforward, it is actually the case that the target object is sometimes directly affected by this linking operation. There are three notable cases of this:

- A thread is added to a task by setting a link from the thread to the task.
- Memory is mapped into a task by setting a link from a mapping object to the task.
- A port is added to a port set by setting a link from a port to the port set.

Therefore an operation which is effectively a write operation on an object can sometimes only require a reference capability for the object.

- Requests which observe the state of a kernel object generally require a read capability to the object. Note that reading the state of an object returns a reference capability for any other objects linked into that object.

  There are however a couple of special cases. As part of their simple state, all objects have a hash value and a type value, and these values can be read using a reference capability to the object. Also, through linking operations a reference capability is sometimes sufficient to observe something about an object. The most important example of this is that a reference capability to a task can be used to export any memory from that task.

- The standard control over IPC operations is that the client must have a reference capability to a port and the server must have a read capability to the port's port set. Neither client or server is required to hold a capability to the other. However, a reliable IPC connection can be made by setting links in both the client and server, without any port or port set.

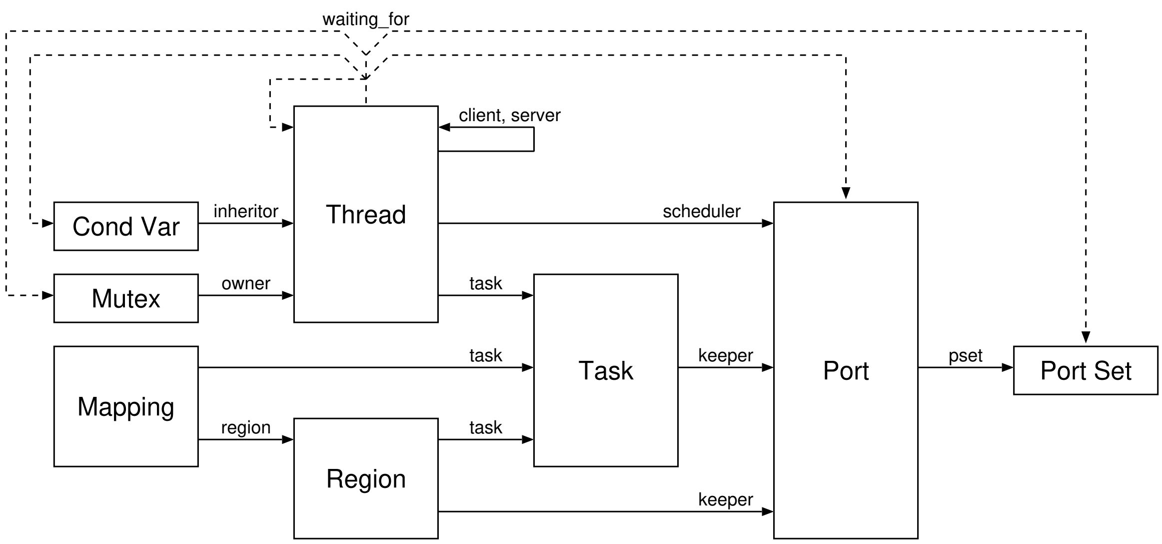

Figure 3: Links Between Kernel Objects

Figure 3 displays all of the links which are maintained between kernel objects. The name associated with each link corresponds to the name used in the kernel API document [2]. Each link is modified or observed through an operation on the object that the link points from, i.e., the object at the tail of the arrow.

This figure illustrates the following about the capability system:

- Suppose a client holds a write capability to one object and a reference capability for a second object, where the type of the second object is pointed at by an arrow from the first object type. Then the client can change that link of the first object to point to the second object.
- Suppose a client holds a read capability to one object. Then it can obtain a reference capability for any object pointed to by that object.
- Suppose a client holds a reference capability to one object. The client can create an object of a type that points to the first object type, and link the two objects. As discussed above, in some cases this has the effect of actually changing the state of the first object.
- The waiting_for links, shown as broken arrows in the figure, are read-only, and are only changed as a side effect of other kernel operations.
- Since no (writeable) links point to the object types on the far left of the figure (condition variables, mutexes and mappings), reference capabilities to those object types are useful only for reading type and hash value of an object.
- Since no links originate from the object types on the far right of the figure (port sets), read and write capabilities to these objects types cannot affect links. However, they are still useful for reading and writing the state of the object (though ports sets have no state), and write capabilities can still be used for creating reference capabilities.

To evaluate the security provided by a capability system it is as important to understand how capabilities are obtained as to understand when capabilities are required to obtain some service. Capabilities for Fluke microkernel objects can be obtained in all of the ways listed below. A key point is that obtaining capabilities requires some interaction with the microkernel; a capability cannot be obtained simply as a sequence of bits within a address space.

- System calls to get the state of a kernel object return reference capabilities to all objects pointed at by that object (as in Figure 3).
- A request to create a new kernel object returns a write capability to the newly created object.
- A reference capability can be created from a write capability.
- Any client thread can obtain a write capability to itself.
- IPC messages from one process to another can contain reference capabilities.
- Capabilities can be mapped into one address space from another through memory mapping.

2.1.3.3.1 Implementation Notes A kernel object is identified across the kernel interface by a location in a virtual address space. What we have called write and read capabilities to an object are actually implemented as write and read permission bits to this location.

A reference is actually a distinct type of kernel object. The only state associated with a reference object is a link to another kernel object. What we have called a reference capability is actually implemented as read permission to the location by which the reference object is identified across the kernel interface.

A reference object can be linked to any type of kernel object other than another reference object. This is the reason that there are only two capabilities for a reference object, read and write.

The other permission bits, execute and create, are essentially irrelevant to the capability system, except to the extent that the create permission can prevent the creation of an object at a particular address.

Capabilities are verified during the same operation where they are resolved to the relevant object, which makes it easy to verify that the capabilities are actually always checked.

2.1.3.4 Controlling Shared Hardware Resources

Fluke is designed with an intent to provide a hierarchical model for allocating processor time and physical memory. This model can minimize covert channels through resource allocation and prevent processes in one virtual machine from denying resources to a process in another virtual machine.

However, the current implementation only provides controls over the amount of physical memory allocated for use by each user space task. There is no hierarchical scheduling model and no control over the amount of the kernel's physical memory which is allocated for kernel object's created by each task.

2.1.4 Analysis of Requirements for Total Isolation

Section 2.1.2 identified high-level requirements necessary to meet the objective of total isolation. These include one requirement on the processor hardware, one requirement on the microkernel implementation of system calls, and several requirements on nesters. In this section, we discuss how the system design could be analyzed in its ability to meet the requirements.

Analysis of the processor hardware is processor dependent and will not be considered further.

Analysis of the microkernel and nester requirements ultimately comes down to three steps:

- Analyzing the flow of capabilities across the interfaces provided by the microkernel and nesters.
- Analyzing the flow of data across the interfaces provided by the microkernel and nesters.
- Identifying the practical effects of any flaws that may have occurred in the analysis or in the implementation of the design which is analyzed.

The first difficulty that is encountered when analyzing the flow of capabilities is the need for identifying precisely the requirements about where capabilities can be held. This is especially important because of the fact that nesters span multiple virtual machines.

However, once this is done the Fluke microkernel is simple enough that it should be easy to gain confidence that capabilities are not leaked between virtual machines through the microkernel. The nesters are more complex and it is unclear how difficult it would be to completely analyze capability flow through the nesters.

Analysis of data flow is much more complex because of the subtle (covert) ways in which data can often be communicated. Complete information flow analysis is difficult for any system which provides a rich programming interface, and this is most likely true of Fluke as well.

However, even though it is much more difficult to analyze the flow of data than the flow of capabilities, this does not suggest that the difficulty of analyzing a system is dependent only upon the difficulty of analyzing data flow. The practical effects of flaws (or omissions) in the analysis of capability flow can be much more serious. If it is indeed possible for a capability to be shared between virtual machines, then there may be an unlimited potential for data sharing between the virtual machines. On the other hand, covert channels for data flow which are difficult to detect may be similarly difficult to exploit. Moreover, the exploitation of a covert channel may be detectable if it requires "unusual" behavior, while using a capability to pass data would not be identified as unusual behavior.

In summary, our main concern for the requirements analysis is the potential difficulty of analyzing both capability and data flow through nesters. If analysis of capability flow through nesters does prove to be complex, then concerns about flaws in the analysis (or implementation) are also significant.

2.1.5 Allowing Controlled Interactions Between Virtual Machines

The requirements presented in Section 2.1.2 and analyzed in Section 2.1.4 considered the objective of total isolation between processes in disjoint virtual machines. While this is a necessary first step, the ultimate objective is to allow for some interactions between such processes, but only under controlled circumstances. The difficulty is to allow some sharing of capabilities or data while controlling the extent of the communications and the propagation of capabilities.

For instance, a nester can make the choice to reimplement a CP interface or to simply pass its service port reference to its children. However, beyond this very crude choice, there appears to be no general mechanism for allowing certain interactions between virtual machines while disallowing others. This is one of the purposes of the Flask additions discussed in Section 2.2.

2.2 Flask

This section presents the security architecture for Flask.

Flask is a collection of mechanisms for providing access control in Fluke based upon explicit security attributes assigned to objects in the system. Flask consists primarily of enhancements to object servers (including the microkernel[11]) for providing access control over the services they provide.

The primary purpose of introducing these mechanisms into Fluke is to provide a general framework for defining and enforcing security policies. Recall that while Fluke provides a security architecture, it provides no mechanisms for configuring an implementation for a specific security policy. Flask also provides direct control over services through explicit permission checks rather than indirect control through capabilities. This direct control makes it much easier to assure that mandatory security policies are being correctly enforced by the system.

Some other reasons for introducing Flask include:

- Flask can be used to control interactions within a virtual machine, potentially diminishing the need for disjoint virtual machines and the overhead that this can incur.

[11]Throughout this section, the microkernel will be included in the category "object server". This reflects the fact that many of the services provided by the microkernel are essentially operations on a particular object. However, the microkernel is certainly more than an object server, and some of the services it provides cannot be directly attributed to any particular object (e.g., IPC).

- The overhead of Flask should be considerably less than that required by a security monitor such as suggested in [4], since Flask adds permission checking to existing layers rather than requiring the addition of new layers.

Of course, there are potential negative aspects of Flask, especially the performance impact of the additional mechanisms and the need to alter existing software. The Flask design is intended to minimize these impacts, though they cannot be eliminated entirely.

- Section 2.2.1 provides an overview of the key elements of the architecture, independent of the implementation on Fluke.
- Section 2.2.2 describes the specific adaptations made to the Fluke microkernel to support Flask.
- Section 2.2.3 discusses some of the adaptations made (or planned to be made) to Fluke object managers to support Flask.
- Section 2.2.4 describes how the architecture is adapted to take advantage of the Fluke recursive virtual machine architecture.
- Section 2.2.5 considers how the Flask enhancements simplify the requirements for a total isolation policy as was discussed in Section 2.1.4.

2.2.1 Overview

The Flask security architecture describes the interactions between a subsystem which enforces security policy decisions and a subsystem which makes those decisions, and the requirements on the components within each subsystem. The primary goal of the architecture is to provide for flexibility in the security policy by ensuring that these subsystems always have a consistent view of policy decisions regardless of how those decisions are made or how they may change over time.

The architecture currently provides for three kinds of policy decisions:

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

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

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

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

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

- The architecture should anticipate the performance impact of all operations that it requires and take steps to minimize those impacts whenever possible.
- The architecture should anticipate the need to verify the correctness of mechanisms which implement it, and should allow for implementation through relatively simple, isolated mechanisms whenever possible.

The architecture specifies interfaces and requirements on two main components, while assuming that the underlying system provides separation between these components and mechanisms for communicating between them:

- Components which enforce the security policy decisions are referred to as *object managers*, based upon the origins of the architecture in a client/server model.
- Components which provide security decisions to the object managers are referred to as *security servers*. The decision making subsystem may include other components such as administrative interfaces and policy databases, but the interfaces among these are policy dependent and are therefore not addressed by the architecture.

The Flask security architecture provides the following for object managers:

- Interfaces for retrieving security, labeling and polyinstantiation decisions from a security server.
- An *access vector cache* (AVC) module that allows the object manager to cache security decisions in order to avoid any performance costs associated with security server communications.
- The ability to register to receive notifications of changes to the security policy.

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

- All objects involved in security decisions must be labeled. The mechanism for assigning labels to new objects must be clearly defined, which may involve the use of the labeling interfaces provided by a security server.
- The object manager must define and implement a control policy which defines how security decisions are used to control the services which the object manager provides.
- The object manager must determine which policy changes are relevant and define handling routines which are called in response to policy changes.
- For all uses of polyinstantiation, the object manager must define the mechanism by which the proper instantiation of a resource is chosen, possibly using the interfaces provided by a security server.

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

Each of these is discussed further below. Further details of the interfaces between the AVC and security server can be found in the Formal Top Level Specification [13].[12]

2.2.1.1 Object Labeling

Security policy decisions are made based upon a set of security attributes, referred to as a *security context*, associated with subjects and objects in the system. A fundamental issue in the system architecture is how this association is maintained. The simplest solution would be to define a policy independent datatype (e.g., a null-terminated string) which is part of the data associated with every subject and object in the system. However, the use of a single datatype for all labeling introduces some conflicts among the various possible uses of labels. For example:

- To associate labels with subjects and objects, labels may need to be included within existing data structures (e.g., network packets). It is unrealistic to expect that a single datatype would be appropriate in all instances.
- For security administration, the most convenient form of a label is a highly-descriptive string.
- For high performance when looking up security decisions, it is desirable to use labels which can be quickly translated into hash values for table lookups.

The Flask architecture addresses these conflicts by providing two policy-independent datatypes for labeling:[13]

- A security context is a variable-length string which can be interpreted by any application (or user) with an understanding of the security policy.
- A *security identifier* (SID) is a fixed size value which can be interpreted only by the security server which maps SIDs into security contexts. This mapping cannot be assumed to be consistent across executions (reboots) of the security server.[14] SIDs are used to identify labels across all interfaces between a policy-independent object manager and a security server.

These two datatypes are not sufficient to satisfy all of the conflicting requirements for labels identified above. Other labeling schemes local to particular object managers or applications may also be needed (see Sections 2.2.3.2 and 2.2.3.3 for examples).

A label must be assigned to an object when it is created. Typically[15] one of the following models is used for assigning a label to the new object:

[12] It is expected that the interfaces described in the FTLS will continue to undergo changes, and that these changes will be summarized in the Final Report [11].

[13] It has been suggested that there should also be a policy dependent datatype corresponding to the security context which can be used for communications among the security server, applications which are aware of the particular security policy embodied in the security server, and security administrators. The value of this extension cannot be determined without further experience with the architecture.

[14] In reality, during system startup there may be a need to hardcode certain SID values into the system for subjects and objects which come into existence prior to the security server.

[15] Here "typically" means that all existing cases in the Flask implementation follow one of these models, though these are certainly not the only possible models.

- The new object's label may be copied from the client requesting the creation of the object or from some other related object. This is a typical choice when the object is private to the client or is in some way essential to the existence of the client or other related object (e.g., allocating new memory within a process).
- The new object's label may be specified by the client as part of the request to create the object.
- The security server may be consulted to provide a labeling decision. The inputs to that decision include the class of object being created and the SIDs of the client and any related objects. The security policy can then specify how the security contexts which these SIDs represent are combined into a security context (and SID) for the new object. For instance, the security context of a new file is determined (by default) by combining aspects of the security context of the client requesting creation of the file and the directory in which the file is created.

In any case, the security policy is typically consulted to ensure that the client is allowed to create an object with the chosen SID.

For some object types, a client may request to change an object's SID after it has been initially assigned. The object manager's control policy will typically require that the security policy be consulted when a relabeling is requested to verify three permissions: the client must be allowed to relabel objects having the old SID, the client must be allowed to label objects with the new SID, and objects with the old SID must be able to transition to the new SID.

2.2.1.2 Client and Object Manager Identification To support the basic access control and labeling operations, the Flask architecture requires that the system include some mechanism for providing an object manager with the SID of a client making a request.

This feature should also provide the client with an override mechanism so that it can specify an *effective SID* by which it will be identified to the object manager. This allows the client to limit its privileges, for instance when making a request on behalf of another subject. It can also be used to support transparent interposition, such as through a network IPC server connecting processes within a distributed system [6]. Finally, it can provide for complete anonymity of the client if required (though most likely with an accompanying loss of privileges).

Just as it is necessary for the recipient of a request to be able to identify the client, it may be desirable for a client to identify the recipient before sending a request. While absolute identification may not be possible (the ultimate recipient may not even exist when a message is sent), the client should at least be provided with a means of restricting the set of potential recipients, for instance by specifying a particular SID for the recipient. As with the client, the server should also be provided with a means of specifying an effective SID by which it will be identified to the client.

2.2.1.3 Control Policy Each object manager has a control policy which specifies precisely the security controls provided by the object manager.

A typical statement within a control policy is a requirement that the client of an operation be granted a particular permission to an object accessed by the operation. However, the control policy is not limited to such statements and can include virtually any kind of security requirement. The only limitation is that security requirements must be independent of the security policy, so for instance, a requirement which relies upon the value of a particular SID is not allowed.

For the Flask architecture, the important statements within the control policy are those which require a security decision, i.e., those which require that a particular permission be granted between two objects. We distinguish two forms of such requirements:

- A *state transition requirement* identifies a permission which must be granted in order to allow an operation to proceed during some state of the system (i.e., for a state transition to occur).
- A *state invariant requirement* identifies a permission which must be allowed through all states in which a particular condition (invariant) holds.

2.2.1.4 Retrieving and Caching Security Decisions

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

The caching mechanism in Flask provides for much more than simply caching individual security decisions. The *access vector cache* (AVC) module which can be incorporated into any object manager provides for coordination of the policy between object manager and security server. There are two basic operations which provide this coordination, requests to the security server for policy decisions and requests from the security server to enact a policy change. We discuss the first of these here and will discuss policy changes below.

When the object manager requires a security decision, it queries the AVC, providing the SIDs of the objects involved and the permission(s) required.[16] If the required decisions are not found within the AVC, the AVC makes a request to the security server, forwarding the SIDs of the objects and the permission(s) required.

To improve efficiency, the security server can respond with decisions for more permissions than those requested. In particular, it will respond with the current state of the security policy for a set of permissions known as an *access vector*. An access vector is a collection of related permissions for the pair of SIDs provided in the request. For instance, all file access permissions are grouped into a single access vector.

For each permission in an access vector there are three possible responses from the security server. Either the permission is granted, or it is denied, or the decision is deferred. Deferring decisions is necessary for some kinds of policies with dynamic conflict of duty constraints, such as the piecemeal Clark-Wilson policy [9], though the decision for a requested permission should not be deferred.

To represent these three possible values for each permission, the return parameters from the security server include two bit vectors (each bit corresponds to a permission), the *decided* vector and the *allowed* vector. The decided vector indicates whether the decision is provided or deferred. If a decision is provided, the allowed vector indicates whether the permission is granted or denied. This response is stored within the AVC to eliminate future requests to the security server whenever the object manager requests a decision which has already been provided.

2.2.1.5 Changes to Security Decisions

Changes to the security policy must be propagated to the object manager for several reasons. The most obvious reason is that the AVC may include

[16] A reader knowing the details of Flask will recognize that the “security class” parameters to such a request are missing from this description. For current purposes, these parameters can be considered to be part of the permission.

a cached decision which is incorrect after the policy change. The object manager itself may be affected by a policy change in the case of a state invariant requirement which is no longer satisfied after the policy change. A policy change can even affect a state transition requirement if there is any operation outstanding for which the permission check has been completed but not the operation itself.

The Flask security architecture provides three kinds of interfaces which are used by the security server to propagate security policy changes to the object manager:

- The security server can request removal of a policy decision from the AVC. This would typically be used only if the decision previously denied a permission which is now granted.
- The security server can request revocation of a previously granted permission. The AVC will not only remove the permission from the cache but may also call the object manager through a callback interface as discussed below.
- The security server can request that a previously granted permission be changed to a deferred permission, so it can be recomputed by the security server. The AVC will mark the permission as deferred and may call the object manager through a callback interface as discussed below.

To support dynamic security policies, the object manager provides three kinds of callback routines which are called by the AVC in response to requests from the security server.[17]

- The revocation callback routine is called to notify the object manager that a particular permission has been revoked. The object manager may need to abort operations in progress to satisfy state transition requirements or adjust its state to satisfy state invariant requirements.
- The recompute callback routine is called to notify the object manager that the security server has requested that a particular permission be treated as deferred. If this permission is required in the current state of the object manager, it must request that the permission be recomputed.
- The query callback routine is called to simply query the object manager whether a particular permission (or set of permissions) is currently retained by the object manager for any reason.

This support for changing policy decisions introduces the need for coordination among the various requests which provide security decisions and notify of changes in those decisions. A serialization protocol maintained by the AVC and security server is provided to address this need, as described in the FTLS [13].

2.2.1.6 Polyinstantiation Support

An object manager's control policy can specify how a client is or is not allowed to access some object or other managed resource. However, there are some fixed resources which a client cannot operate without, but yet a security policy may need to restrict the sharing of the resource among clients.

[17] While the descriptions here agree conceptually with the current implementation, there are some subtle differences. The ideal set of callback routines is still unclear.

One solution to this problem is to polyinstantiate the resource and partition clients into sets which can share the same instantiation of the resource.[18] The Flask architecture supports polyinstantiation by providing an interface by which the security server can identify which version of an instantiated resource can be accessed by a particular client. Both the client and the version are identified by SIDs.

2.2.2 Flask Integration with the Fluke Microkernel

The Fluke microkernel plays two roles within the Flask architecture outlined above. It provides process separation and IPC services, and it is an object manager providing control over the objects which it manages.

We describe the role of the Fluke microkernel and the enhancements made to it to support the Flask architecture in four subsections which parallel the organization of Section 2.2.1. Two subsections from Section 2.2.1 are not represented here: the subsection on caching is independent of the object manager and the section on polyinstantiation is not included since the microkernel makes no use of the polyinstantiation interfaces.

2.2.2.1 Object Labeling The SID of a memory segment (see Sections 2.1.3.1 and 2.2.3.1) defaults to the SID of the client requesting its creation. The client can alternatively request a specific SID, which is then subject to the control policy. The SID of a segment is not allowed to change after the segment is created.

The SIDs of all other kernel objects are derived from the SID of a segment. Each kernel object is created at a particular location in an address space, and the SID of the kernel object is set equal to the SID of the segment to which that location maps. As we shall see later when describing the control policy, this approach allows us to leverage off of the existing Fluke capability model.

This approach does however create a potential loss of labeling flexibility. While a client can label an object with any desired SID by creating a segment with that SID (assuming this is allowed by the security policy) and then creating the object in that segment, this may waste significant memory since a segment is at least one page in length while each kernel object requires only a few words of memory.

Though a kernel object can be moved within an address space, its SID is not allowed to change. The SID of the segment to which an object is moved must be the same as the SID of the segment from which it is moved.

2.2.2.2 Client and Object Manager Identification Flask enhances the Fluke IPC mechanisms to allow a client and server engaged in IPC to identify the SID of the other, and to specify an effective SID by which each will be identified.

Fluke provides three forms of IPC. In the simplest two forms, one-way and idempotent, the client identifies the server by specifying a port to which a message is sent. The SID of this port is the client's view of the SID of the server. Since this port is ultimately provided by the server, the server can specify an effective SID by creating a port with that SID.

[18] The other common solution to this problem is to partition the resource itself, as is often done with hardware resources in multilevel secure (MLS) systems. The Flask architecture currently provides no direct support for partitioning, though the research team headed by Dr. Tom Keefe at Penn State University is investigating the partitioning of hardware (disk and CPU) resources in the Flask model as part of an MLS database project.

The server listens for requests on this port, and when a request is received, the server can request to be notified of the SID of the client. However, the client has an opportunity to specify an effective SID when the message is sent.

The third form, reliable IPC, can be more complex. In the simplest case, a reliable IPC connection is established using the same basic mechanisms as one-way and idempotent IPC, and client and server identification is provided in the same way.

However, reliable IPC connections can also be established (or re-established) by explicitly setting the states of the client and server threads. In this case there is no port by which the client can identify the server and no initial request through which the server can identify the client. Instead, the state of each thread is expanded to include the effective SID by which it is known in a reliable IPC connection and the effective SID of its peer in that connection. For a connection to be established by setting the state, the view of the effective SIDs from each side of the connection must agree.

2.2.2.3 Control Policy The control policy for the Flask microkernel is generally a simple extension to the Fluke control policy. Flask does not require controls over any Fluke operations that are not controlled by the Fluke capability mechanisms, it simply adds to those controls so that they can be more directly associated with the security policy.

The basic Flask model is to control the use of memory capabilities (i.e., read, write, create and execute) through permissions controlling whether a task can hold such capabilities and to control the use of reference capabilities through permissions checked at the time that a reference capability is used.

Each of these are discussed here, followed by a discussion of controls over operations that are provided in Flask but not Fluke.

2.2.2.3.1 Flask Controls Over Memory Capabilities As described in Section 2.1.3, Fluke supports four memory access modes: read, write, execute and create. The first two of these modes also effectively serve as capabilities to (existing) kernel objects.

Fluke provides control over the propagation of these memory access modes when memory is mapped from one address space to another. Flask provides additional direct security policy control over their propagation by associating a Flask permission with each of the four access modes. During the initial attempt to access mapped memory, the microkernel verifies that the security policy explicitly grants permission for each requested access mode based upon the SID of the address space and the SID of the memory segment. It is generally sufficient to verify these permissions upon the initial access since the SID of a segment is not allowed to change. The only exception is if the policy itself changes, which will be discussed in Section 2.2.2.4.

2.2.2.3.2 Flask Controls Over Reference Capabilities A different approach is taken to control reference capabilities under Flask. One reason is that there is no simple way to leverage off of the Fluke mechanisms as in the case for read and write capabilities.

A more compelling reason for taking a different approach is that reference capabilities are used in many different ways, unlike read and write capabilities which basically control the same types of operations on each object. For instance a reference to a task can be used to map memory into a task or export memory from a task, operations which have very different effects on the task. Thus Flask will support finer-grained control than Fluke.

Section 2.1.3.3 identifies three types of operations that are controlled by reference capabilities. We describe each of these and any additional Flask controls over these operations:

- A reference capability to an object is required to establish a link to that object from another one. All possible links are shown in Figure 3 on page 10. Flask adds additional requirements between the two linked objects in several cases. These controls, shown in Figure 4, fall into two categories:

  - In those cases for which the two objects are considered to be essentially equivalent for the purpose of security, it is required that the SIDs of the two objects be the same.
  - In other cases, a distinct permission is added to control each link.

  As a special case, the client and server links from a thread are controlled indirectly through the IPC permissions described below.

  Also, in a few cases no additional Flask control is added.

  Specific rationale for each of these decisions is provided in the relevant portion of the control policy in Section 3.

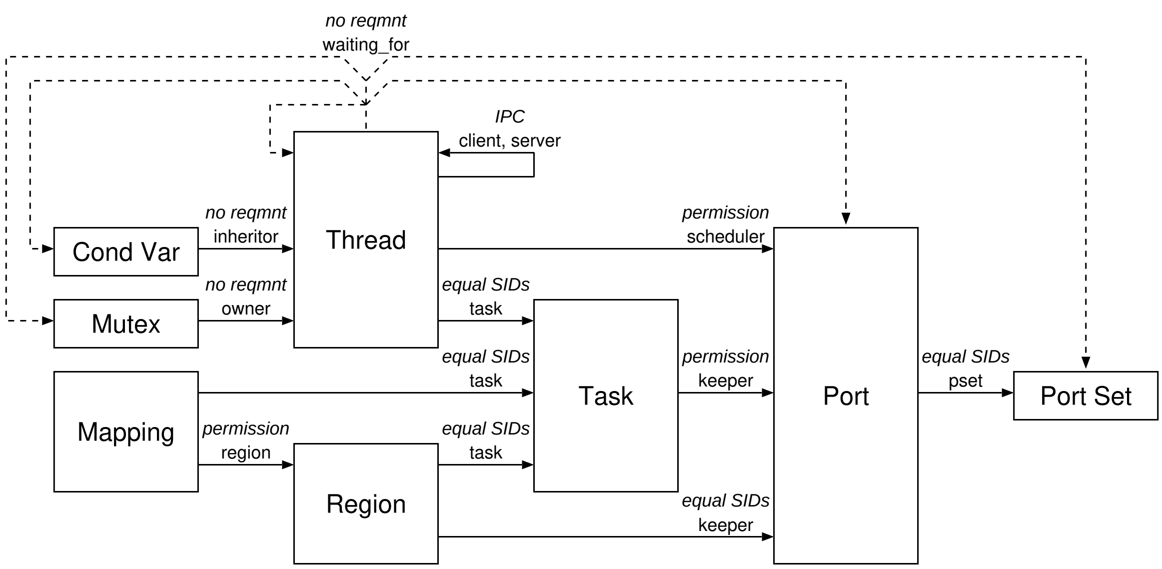

Figure 4: Flask Controls over Inter-object Links

- A reference capability to an object can be used to obtain the hash value or type of an object. Flask adds no additional controls over these operations, as will be discussed in Section 3.2.11.

- A reference capability to a port is required in order to send an IPC message to that port. Flask adds three distinct permissions to control sending messages under each of the three forms of IPC.[19]

[19] One reason for separate permission bits is because one-way IPC can be used as a one-way communication channel, while idempotent and reliable IPC are bidirectional. There is no similarly substantial reason for using separate bits for idempotent and reliable IPC, but there is essentially no cost in doing so either.

2.2.2.3.3 Controls Over Flask Specific Operations

There are two Flask specific operations whose controls have not been addressed previously.

There is a request `fluke_ref_sid` that allows a thread holding a reference capability for an object to obtain the SID of that object, analogous to the Fluke requests which return the hash or type of the object. Similar to those requests, no additional Flask requirement is added to this request.

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

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

2.2.2.4 Changes to Security Decisions

The Flask microkernel supports the revocation and recomputation of all Flask permissions. There are three kinds of permissions, memory, link and IPC, and the mechanisms for addressing revocation of each are discussed here.

If a memory permission from an address space (task) SID to a segment SID is being revoked, the Flask microkernel locates all memory translations that map memory from a segment with the given SID to an address space with the given SID, and updates the translation protections accordingly. The microkernel can rely upon the atomic properties of kernel requests [3] to ensure that requests in progress at the time of the revocation either finish in a timely manner or are restarted after the revocation is complete.

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

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

2.2.3 Flask Additions to Fluke Object Managers

2.2.3.1 Virtual Memory Manager

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

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

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

2.2.3.2 File Server

The Flask file server labels four types of objects: file systems, directories, files and file description objects. We briefly discuss labeling and control policies over each. The most interesting aspects of the file server are the need for persistent labels, the ability to relabel files and directories, and the use of polyinstantiation. The file server currently does not provide any callbacks for reacting to policy changes nor has any design for implementing them been determined.

File system labels are defined when the file system is created. Alternatively, a label can be assigned to an unlabeled file system under the control of an authorized user. The label on a file system is used to control operations such as mounting and unmounting and represents the aggregate label of all files within the file system.

Files and directories are labeled according to the request of the creator (subject to the control policy). If no specific SID is requested, the SID is assigned by querying the security server to compute the SID from a combination of the client SID and the SID of the directory in which the new object is being created.

Since file systems, directories and files are persistent objects, their labels must also be persistent, and hence SIDs cannot be used as labels. Instead, each file system includes a table which identifies the security context of the file system and every directory and file in the file system. For internal use, these labels are mapped to SIDs by a query to the security server.

In contrast to Unix access controls, the Flask file server defines a permission for each service that observes or modifies the state of a file or directory. For example, whereas Unix permits a process to invoke `stat` or `unlink` on a file purely on the basis of the process' access to the file's parent directory, the Flask file server verifies that the client has *Getattr* or *Unlink* permission to the file in addition to the directory based permissions. The Flask file server also supports fine-grained distinctions among services, such as separate *Write* and *Append* permissions for files and separate *Add_Name* and *Remove_Name* permissions for directories.

The Flask file server allows a client to request that a file or directory be relabeled. If the request is allowed by the security policy, steps must be taken to ensure that the control policy is satisfied after the change to the new SID. For instance, any state invariant requirements involving the file must be recomputed, such as the storing of permissions to the file within a file description object. Operations in progress involving the file may need to be aborted if disallowed according to the new SID. These actions are similar to the actions that would need to be taken during a change in the security policy.[20]

Certain file names are effectively fixed resources because applications widely expect them to be used in a particular way. Examples include the `/tmp` directory and configuration files for network data, user data and applications. In some policies it may be appropriate for these files and directories to be shared among all clients, while in others polyinstantiation may be appropriate. *Security Union Directories* (SUDs) have been proposed to provide polyinstantiation when required by the security policy. To a typical client, a SUD looks no different than any other directory, but internal to the file server there are many different directories with the same name. The correct directory for each client to access is determined through the security server's polyinstantiation interface. A client which is aware of the existence of a SUD and has the appropriate permissions may be able to access more than one of the instantiations through an extended interface. At the current time, SUDs remain a proposal and have not been implemented.

File description objects are labeled as an extension of a client, not of the file or directory itself, since it contains information about the client's current access to the file or directory. Thus by default, a file description object is labeled with the SID of the client which created it (usually by making an `open` call). Like most objects, the client can also specify a particular SID if desired.

The file description object contains the permissions which a client with the SID of the file description object is granted to the file or directory. Therefore if a client with that same SID uses the object to access the file or directory, there is no need to query the policy for permissions. However, if a client with a different SID attempts such an access, the control policy specifies that the client must not only have the appropriate access to the file or directory but also to the file description object itself.

2.2.3.3 Network Server

The Flask network server is currently under development. This section describes its expected implementation. A much more detailed description of an earlier version of this design can be found in [1].

The network server recognizes labels on six types of objects, though only three of these are completely internal to the network server. These six objects types are the routing table (this is a single object), network interfaces (physical), nodes (throughout the network), packets (incoming or outgoing), sockets and port names. Of all of these object types, the only labels actually assigned during execution of the network server are those for packets and sockets.

Internally, all labeling is done using SIDs. However, since the label of a packet must be transmitted across the network with the packet, the label must have a common interpretation at both nodes and SIDs cannot be used. Moreover, the variable length of security contexts rules out their use. Instead, a table of *network SIDs* is built by the two nodes and separate mappings between SIDs and network SIDs are used on each side to translate for both incoming and outgoing packets.

The assignment of a socket SID is dependent upon the manner in which it is created. The SID of a socket created directly by a local client defaults to the SID of that process unless a different SID is specified by the client. The method for assigning the SID to a socket created as a result of a new connection being accepted on an existing socket is determined by the listening process, with three choices: the SID of the socket on which the process is listening, the SID of the remote socket from which the connection is requested, or a specific SID requested explicitly by the process. As always, a client's request to assign a particular label is under the control of

[20] This discussion of relabeling reflects a design which has not yet been implemented.

the security policy.

The SID of an outgoing packet defaults to the SID of the socket. For a non-stream socket, the client can specify a particular SID for each packet, under the control of the security policy.

The control policy for the network server is quite complicated because of the number of objects involved. The basic operations of sending or receiving a packet are subject to requirements that involve the SIDs of a client process, the socket, the packet, the network interface and the remote node. For instance, an incoming packet has a label associated with it, but the local network server may not be able to trust the label. It must make that decision based upon the SIDs of the packet, the network interface and the remote node. Identification of the remote node and of the label attached to an incoming packet may themselves be dependent upon cryptographic mechanisms and protocols such as IPSEC and ISAKMP.

The network server controls are layered to match the network protocol layering architecture, so the controls over high level IPC services resolves to a collection of controls at each layer. For example, control over the transmission of a packet is provided through controls at the socket layer involving the SIDs of the client process, packet and socket and controls at the network layer involving the SIDs of the packet, network interface and remote node.

Since the port name space is a single fixed resource, the network server may polyinstantiate it, for instance, to allow a single network server to distribute traffic among multiple ftp servers in isolated information domains. One implementation of this called *Security Union Port Spaces* (SUPS) is described in [1].

2.2.3.4 Process Manager

Relative to the file and network servers, the Flask enhancements to the process manager are quite simple. Processes are the only labeled object type within the process manager, the controls over processes are straightforward and there is no support for polyinstantiation.

The only unique aspect of the process manager is the relabeling of processes when a file is executed. A new process comes into existence through a *fork* call which simply copies an existing process, so the new process inherits the SID of the existing process. The more interesting step in the life of a new process is when it calls the *exec* routine, at which time the process manager calls the security server to compute a new SID for the process from the current SID of the process and the SID of the file to be executed (unless a specific SID has been requested).

During an *exec* there are also controls over which elements of the process state can be inherited, with sanitization if necessary.

2.2.4 Flask and the Recursive Virtual Machine Architecture

In Sections 2.2.2 and 2.2.3 we have described how the Fluke microkernel and object managers have been modified to integrate into the policy enforcement subsystem of the Flask security architecture. In the current implementation, the policy decision making subsystem consists of a single security server which services requests from the microkernel and all object managers.

However, ideally the interfaces provided by the security server could be considered to be a part of the common protocols interface within the Fluke recursive virtual machine architecture, so that each virtual machine can define its own policy over the objects existing within that virtual machine. There is unfortunately a significant obstacle to accomplishing this: the security policy involving objects created within the virtual machine by lower level nesters or the microkernel is enforced by the lower level nester or the microkernel, and hence the SIDs of those objects must be meaningful at those lower levels.

Ultimately, this problem arises from the fact that the entire virtual machine interface in Fluke is not reimplemented (or at least interposed upon) at every level. Therefore there is no opportunity for each virtual machine layer to translate a SID into something understandable at the next lower level. Instead, the security servers must interact to do this themselves.

The interfaces and requirements for these interactions among security servers is currently under study. The intention of this study is to identify mechanisms which are appropriate to the Fluke recursive virtual machine model and are also of value independent of Fluke in providing a general framework for extending security policies and dividing the responsibility of policy decision making among security servers.

2.2.5 Revisiting the Requirements for Total Isolation

One of the benefits of Flask is to simplify the analysis of the Fluke requirements for total isolation.

Section 2.1.4 identifies three steps in the analysis of the requirements. Flask simplifies the first and third of these steps.

- The analysis of capability flow in a Flask system still requires identifying precisely how capabilities are divided between the virtual machines. Kernel objects are then given a SID which identifies the correct virtual machine.

  However, there is no need to identify the flow of capabilities in Flask, since the Flask mechanisms provide control over the use of capabilities based upon their SID. The Flask mechanisms must be analyzed, but these are intended to be quite simple and most of the analysis is performed once regardless of the complexity of any particular object server.

- The effect of flaws in the capability flow analysis (or implementation) are also minimized by Flask. First of all, the addition of another mechanism decreases the likelihood of failure. Perhaps more significantly, Flask minimizes the “domino effect” that can occur if a capability is incorrectly granted. In Fluke, a process which receives one capability incorrectly may be able to use that capability to gain other capabilities, while this is generally not true in Flask.

Section 3

Microkernel Control Policy

This section contains the control policy for the Fluke (and Flask) microkernel. It defines the services which are controlled and the conditions under which each service is allowed to occur. Logically, the policy consists of three parts:

- A high-level definition of the state maintained by the kernel.

  This defines the kernel entities which are controlled, and describes the attributes associated with each type of entity. It also describes those elements of the kernel state where permissions are stored and any other state which is necessary to describe the kernel access controls.

- The definition of the services controlled by the kernel.

  There are two distinct types of services defined in the control policy:

  - *Transformation services* are those services which can be defined as a change in the state of the kernel without referring to any particular kernel request.
  - *Invocation services* are those services which are defined as the execution of a particular kernel request. Invocation services generally (but not always) reflect kernel requests which observe elements of the kernel state, and therefore the services are not identified as a change in kernel state.

- The control requirements on the kernel.

  Associated with each service are one or more requirements which define conditions which must be met for the service to be performed. Generally, the requirements indicate permissions that must be held by the entities performing or otherwise involved with the service. However, in some cases the requirements define other relationships that can hold. For instance, a thread is allowed to perform certain operations on itself without verifying that any permissions are held.

The policy is presented in two sections:

- Section 3.1 defines the state model.
- Section 3.2 defines each controlled service and the requirements associated with it.

Note that this policy does not contain any requirements constraining the initial state of the system. At a minimum, any requirement in the policy which controls the value of an element of the state, rather than just the ability to observe or modify the element, must have a corresponding initial state requirement. However, no systematic attempt has been made to identify other initial state requirements.

3.1 Microkernel State Model

The state model for the microkernel is presented in the following sections:

- Section 3.1.1 defines the state common to all kernel objects.
- Sections 3.1.2 through 3.1.10 define the state specific to each type of kernel object. These follow closely the description of each of the object types as provided in version 2.1 of the Fluke API [2]. The simple state of each object type, containing data, is described first, followed by the links which connect an object to another object.

  In a few cases, object state which is not directly visible outside of the kernel (according to version 2.1 of the API) has been included in the model. These cases are pointed out explicitly, with an explanation of why it is necessary to include the information in the state model.
- Section 3.1.11 defines the Flask security data and access vector cache.
- Section 3.1.12 defines the microkernel state used for physical memory management.
- Section 3.1.13 defines the microkernel state maintained for established IPC connections.
- Section 3.1.14 defines a general form for Fluke kernel requests. This is necessary to define the requirements on invocation services.
- Section 3.1.15 defines the combined state of the microkernel.

3.1.1 Common Object State

Underlying all Fluke objects is a supervisor object (sob) which contains state common to all object types. A supervisor object is named by its address in the kernel's address space. The state associated with a supervisor object includes the following:

- *hash* — An integer value which can be used by user space clients to search for the object related information (in the client) using library provided hashing routines.
- *sob_type* — This defines the type of the object of which the sob is a part. It can take any of the following values:
  - *FLUKE_THREAD*
  - *FLUKE_TASK*
  - *FLUKE_REGION*
  - *FLUKE_MAPPING*
  - *FLUKE_PORT*
  - *FLUKE_PSET*
  - *FLUKE_MUTEX*
  - *FLUKE_COND*
  - *FLUKE_REF*
- *sid* — The security identifier associated with the object, used in permission checking.
- *u_pa* — The physical address at which the user component of the object is located. Though this value is not directly visible across the kernel interface, it is included in the model to provide a connection to the virtual addresses which are used to identify objects across the kernel interface. This connection is made explicit in Section 3.1.12.

Throughout this model, we will refer to three kinds of memory addresses:

- A *SOB_ADDRESS* represents the kernel's address for a kernel object.

- A *VIRTUAL_ADDRESS* represents a location in a virtual address space.
- A *PHYSICAL_ADDRESS* represents an address in physical memory.

THEORY *address_types*

```
address_types: THEORY

  BEGIN

  SOB_ADDRESS : TYPE+
  VIRTUAL_ADDRESS : TYPE+
  PHYSICAL_ADDRESS : TYPE+

  END address_types
```

THEORY *sid*

```
sid : THEORY

  BEGIN

    SID : TYPE+

  END sid
```

THEORY *sob*

```
sob : THEORY

  BEGIN

  IMPORTING address_types
  IMPORTING sid

    FLUKE_SOB_TYPE : TYPE = {
        FLUKE_THREAD,
        FLUKE_TASK,
        FLUKE_REGION,
        FLUKE_MAPPING,
        FLUKE_PORT,
        FLUKE_PSET,
        FLUKE_MUTEX,
        FLUKE_COND,
        FLUKE_REF
        }

    SOB : TYPE = [#
        hash          : nat,
        sob_type      : FLUKE_SOB_TYPE,
        sid           : SID,
        u_pa          : PHYSICAL_ADDRESS
        #]
```

END *sob*

3.1.2 Threads

The simple state of a thread object contains the following data:

Editorial Note:
This differs quite a bit from the description in the API, since it includes machine dependent state not described in the API.

- *thread_va* — The virtual address in the associated task (see *task_link* below) at which the thread object must be mapped in order to run.
- *sched_val* — A scheduler defined thread identifier.
- *flags* — Various machine dependent flags defining the execution state of the thread.
- *exc_state* — A machine dependent structure containing exception state information.
- *exc_handlers* — A machine dependent structure containing entrypoints for exception handlers.
- *ipc_state* — A machine dependent structure containing information about existing IPC connections.
- *effective_client_sid* — The effective SID of this thread for a current reliable IPC connection in which this thread is the client.
- *effective_server_sid* — The effective SID of this thread for a current reliable IPC connection in which this thread is the server.
- *client_effective_client_sid* — The effective SID of the client thread for a current reliable IPC connection in which this thread is the server.
- *server_effective_server_sid* — The effective SID of the server thread for a current reliable IPC connection in which this thread is the client.

A thread object contains the following links to other supervisor objects:

- *task_link* — The address of the task object to which the thread belongs.
- *scheduler_link* — The address of the port object through which the thread receives CPU time.
- *waiting_for_link* — If the thread is in a waiting state, this is the address of the object that the thread is waiting for. See the Fluke API [2] for more details.
- *client_link* — The address of the thread object which is used by IPC requests to identify the client of this thread in an existing reliable IPC connection.
- *server_link* — The address of the thread object which is used by IPC requests to identify the server for this thread in an existing reliable IPC connection.

THEORY *thread*

```
thread : THEORY

  BEGIN

  IMPORTING address_types
  IMPORTING sid

    THREAD_SCHED_VAL : TYPE+
    THREAD_FLAGS : TYPE+
    THREAD_EXC_STATE : TYPE+
    THREAD_EXC_HANDLERS : TYPE+
    THREAD_IPC_STATE : TYPE+

    THREAD : TYPE = [#
        thread_va                      : VIRTUAL_ADDRESS,
        sched_val                      : THREAD_SCHED_VAL,
        flags                          : THREAD_FLAGS,
        exc_state                      : THREAD_EXC_STATE,
        exc_handlers                   : THREAD_EXC_HANDLERS,
        ipc_state                      : THREAD_IPC_STATE,
        effective_client_sid           : SID,
        effective_server_sid           : SID,
        client_effective_client_sid    : SID,
        server_effective_server_sid    : SID,
        task_link                      : SOB_ADDRESS,
        scheduler_link                 : SOB_ADDRESS,
        waiting_for_link               : SOB_ADDRESS,
        client_link                    : SOB_ADDRESS,
        server_link                    : SOB_ADDRESS
        #]

  END thread
```

3.1.3 Tasks

There is no simple state associated with a task object.

A task object contains the following links to other supervisor objects:

- *keeper_link* — The address of the port object to which messages are sent when certain exceptions are encountered by threads executing in the task object.

THEORY *task*

```
task : THEORY

  BEGIN

  IMPORTING address_types

  TASK : TYPE = [#
        keeper_link        : SOB_ADDRESS
        #]

  END task
```

3.1.4 Regions

A region object represents an exportable contiguous range of memory within a task.

The simple state of a region object contains the following data:

- *start_va* — The starting virtual address of the contiguous range of memory which the region object represents.
- *size* — The size of the memory range.
- *prot* — The maximum set of permissions which can be granted to memory exported through this region object. This can be any combination of the following four values:
  - *FLUKE_PROT_READ* — Grants read permission.
  - *FLUKE_PROT_WRITE* — Grants write permission.
  - *FLUKE_PROT_EXEC* — Grants execute permission.
  - *FLUKE_PROT_CREATE_OB* — Grants permission to create a Fluke object in this region of memory.

A region object contains the following links to other supervisor objects:

- *task_link* — The address of the task object from which the region is exporting memory.
- *keeper_link* — The address of the port to which a message is sent when an exception occurs during an attempted access of memory mapped from this region through a mapping object into another task.

THEORY *memprot*

```
memprot : THEORY

  BEGIN

    MEMORY_PROTECTION : TYPE = {
        FLUKE_PROT_READ,
        FLUKE_PROT_WRITE,
        FLUKE_PROT_EXEC,
        FLUKE_PROT_CREATE_OB
        }

    MEMORY_PROTECTIONS : TYPE = setof[MEMORY_PROTECTION]

  END memprot
```

THEORY *region*

```
region : THEORY

  BEGIN

  IMPORTING address_types
```

```
IMPORTING memprot

  REGION : TYPE = [#
      start_va        : VIRTUAL_ADDRESS,
      size            : nat,
      prot            : MEMORY_PROTECTIONS,
      task_link       : SOB_ADDRESS,
      keeper_link     : SOB_ADDRESS
      #]

END region
```

3.1.5 Mappings

A mapping object imports a range of memory from a region object to a task.

The simple state of a mapping object contains the following data:

- *region_off* — The offset in the region object where the imported memory begins.
- *start_va* — The virtual address in the task object where the imported memory begins.
- *size* — The size of the range of memory imported by this mapping.
- *prot* — The desired access permissions for the imported memory. The actual accesses granted are dependent upon the permissions in the region from which the memory is imported and the permissions granted by the Flask security server (see Section 3.2.13).

A mapping object contains the following links to other supervisor objects:

- *region_link* — The address of the region object from which the mapping is importing memory.
- *task_link* — The address of the task object to which the memory is imported.

THEORY *mapping*

```
mapping : THEORY

  BEGIN

  IMPORTING address_types
  IMPORTING memprot

  MAPPING : TYPE = [#
      region_off      : nat,
      start_va        : VIRTUAL_ADDRESS,
      size            : nat,
      prot            : MEMORY_PROTECTIONS,
      region_link     : SOB_ADDRESS,
      task_link       : SOB_ADDRESS
      #]

  END mapping
```

3.1.6 Ports

The simple state of a port object contains the following data:

- *alias* — A value passed to the server when it receives an IPC invocation through this port.

A port object contains the following links to other supervisor objects:

- *pset_link* — The address of the port set to which this port belongs.

THEORY *port*

```
port : THEORY

  BEGIN

  IMPORTING address_types

    PORT_ALIAS : TYPE

    PORT : TYPE = [#
        alias            : PORT_ALIAS,
        pset_link        : SOB_ADDRESS
        #]

  END port
```

3.1.7 Port Sets

Except for common state, there is no state, simple or links, associated with a port set object.

3.1.8 Mutexes

The simple state of a mutex object contains the following data:

- *flags* — Describes attributes of the mutex object. See the Fluke API [2] for more details.
- *locked* — A flag indicating true if the mutex object is currently locked.

A mutex object contains the following links to other supervisor objects:

- *owner_link* — The address of the thread object that currently holds a lock on the mutex.

THEORY *mutex*

```
mutex : THEORY

  BEGIN

  IMPORTING address_types
```

```
    MUTEX_FLAGS : TYPE+

    MUTEX : TYPE = [#
        flags          : MUTEX_FLAGS,
        locked         : bool,
        owner_link     : SOB_ADDRESS
        #]

  END mutex
```

3.1.9 Condition Variables

The simple state of a condition variable object contains the following data:

- *flags* — Describes attributes of the condition variable object. See the Fluke API [2] for more details.

A condition variable object contains the following links to other supervisor objects:

- *inheritor_link* — The address of the thread object targeted for priority inheritance.

A condition variable object also includes the following internal state:

- *waiting_threads* — A list of addresses of thread objects which are waiting on the condition variable. This list is not directly visible across the kernel interface. It is included in the model to provide a means for controlling access to a condition variable through a transformation service rather than an invocation service.

THEORY *cond_var*

```
cond_var : THEORY

  BEGIN

  IMPORTING address_types

    COND_FLAGS : TYPE+

    COND_VAR : TYPE = [#
        flags            : COND_FLAGS,
        inheritor_link   : SOB_ADDRESS,
        waiting_threads  : setof[SOB_ADDRESS]
        #]

  END cond_var
```

3.1.10 References

There is no simple state associated with a reference object.

A reference object contains the following links to other supervisor objects:

- *object_link* — The address of the supervisor object represented by the reference object.

THEORY *ref*

```
ref : THEORY

  BEGIN

  IMPORTING address_types

    REF : TYPE = [#
        object_link          :  SOB_ADDRESS
        #]

  END ref
```

3.1.11 Flask Security Data

This section describes the data maintained by the kernel to support the Flask security mechanisms, in addition to SIDs which were introduced in Section 3.1.1.

Every object has an associated security class. All kernel objects are associated with the class *SECCLASS_SUBJECT*.

Most kernel services are controlled by requiring that a particular permission has been granted. The following lists all permissions used to control services on kernel objects:

- *READ_PERM*
- *WRITE_PERM*
- *EXEC_PERM*
- *CREATE_OB_PERM*
- *SPECIFY_CLIENT_PERM*
- *SPECIFY_SERVER_PERM*
- *CONNECT_PERM*
- *CALL_PERM*
- *SEND_PERM*
- *THREAD_SCHEDULER_PERM*
- *TASK_KEEPER_PERM*
- *MAP_PERM*

The permissions *READ_PERM*, *WRITE_PERM*, *EXEC_PERM* and *CREATE_OB_PERM*correspond directly to the four Fluke memory protections described in Section 3.1.4.

An *ACCESS_VECTOR* is a set of permissions.

THEORY *security_data*

```
security_data : THEORY

  BEGIN
```

```
  IMPORTING memprot

  SECURITY_CLASS : TYPE+
  PERMISSION : TYPE+

%% The following classes are used in the kernel:
  SECCLASS_SUBJECT : SECURITY_CLASS

%% The following permissions are used in the kernel:
  READ_PERM, WRITE_PERM, EXEC_PERM, CREATE_OB_PERM, SPECIFY_CLIENT_PERM,
  SPECIFY_SERVER_PERM, CONNECT_PERM, CALL_PERM, SEND_PERM, THREAD_SCHEDULER_PERM,
  TASK_KEEPER_PERM, MAP_PERM : PERMISSION

  prot : VAR MEMORY_PROTECTION

  prot_to_perm(prot) : PERMISSION =
    CASES prot OF
        FLUKE_PROT_READ      : READ_PERM,
        FLUKE_PROT_WRITE     : WRITE_PERM,
        FLUKE_PROT_EXEC      : EXEC_PERM,
        FLUKE_PROT_CREATE_OB : CREATE_OB_PERM
    ENDCASES

  ACCESS_VECTOR : TYPE = setof[PERMISSION]

  END security_data
```

The kernel caches access vectors in a cache consisting of a collection of nodes. Each node describes some permissions which have been granted (or not granted) by the security server for particular source and target objects. Specifically, each contains the following information:

- *sclass* — The source object's security class.
- *ssid* — The source object's sid.
- *tclass* — The target object's security class.
- *tsid* — The target object's sid.
- *decided* — An access vector containing the set of all permissions for which this node includes a decision.
- *allowed* — An access vector containing the set of all permissions which this node identifies as having been granted by the security server; any permissions in *decided* but not in *allowed* have been specifically disallowed by the security server.

The cache is accessed to determine if a requested permission is allowed between a source and target object. The function *cache_allows* takes the security classes and sids of the two objects and the requested permission and returns a flag indicating whether the permission is allowed. Specifically, the flag is set to TRUE when there is a node in the cache such that both of the following hold:

- The *sclass*, *ssid*, *tclass* and *tsid* in the node are the same as for the objects of interest, and
- The requested permission is in both the *decided* and *allowed* access vectors in the node.

Editorial Note:
Note that with this definition, it is actually possible that a permission could be disallowed by some node but actually allowed by the cache if the same permission were allowed in a different node. The implementation avoids this problem because there is actually at most one node referring to each (sclass, ssid, tclass, tsid) tuple. However, there doesn't seem to be any reason to place this requirement on the model at this time.

THEORY *cache*

```
cache : THEORY

  BEGIN

  IMPORTING sid
  IMPORTING security_data

  AVC_NODE : TYPE =
     [# sclass   :  SECURITY_CLASS,
        ssid     :  SID,
        tclass   :  SECURITY_CLASS,
        tsid     :  SID,
        decided  :  ACCESS_VECTOR,
        allowed  :  ACCESS_VECTOR
        #]

  node : VAR AVC_NODE
  scls, tcls : VAR SECURITY_CLASS
  ssid, tsid : VAR SID
  perm : VAR PERMISSION

  node_allows(node, scls, ssid, tcls, tsid, perm) : bool =
        sclass(node) = scls
        AND ssid(node) = ssid
        AND tclass(node) = tcls
        AND tsid(node) = tsid
        AND member(perm, decided(node))
        AND member(perm, allowed(node))

  AVC_NODE_ADDRESS : TYPE+

  CACHE : TYPE =
     [# existing_nodes        : setof[AVC_NODE_ADDRESS],
        node_data             : [(existing_nodes) -> AVC_NODE]
        #]

  cache : VAR CACHE
  node_addr : VAR AVC_NODE_ADDRESS

  cache_allows(cache, scls, ssid, tcls, tsid, perm) : bool =
    EXISTS node_addr :
      member(node_addr, existing_nodes(cache))
      AND
      node_allows(node_data(cache)(node_addr), scls, ssid, tcls, tsid, perm)

  END cache
```

3.1.12 Physical Memory Management

We model the kernel's physical memory management by the following elements:

- *paged_in* — The set of virtual addresses that are paged in for each task.
- *phys_addr* — For each paged-in address, the physical address to which it is mapped.
- *mem_prot* — For each paged-in address for a task, the memory protections (among *FLUKE_PROT_READ*, *FLUKE_PROT_WRITE*, *FLUKE_PROT_EXEC* and *FLUKE_PROT_CREATE_OB*) which threads in the task are granted to the address.
- *phys_addr_sid* — The SID associated with each physical address.
- *contents* — The contents of each physical address.

In the requirements, *mem_prot* is accessed through three functions:

- The function *memory_allows_va*(*thread*, *va*, *prot*) returns TRUE if *thread* has the requested access *prot* to the virtual address *va* (through the thread's *task_link*).
- The function *memory_allows_pa*(*thread*, *pa*, *prot*) returns TRUE if *thread* has the requested access *prot* to the physical address *pa*.
- The function *memory_allows_sob*(*thread*, *sob*, *prot*) returns TRUE if *thread* has the requested access *prot* to the kernel object *sob*, according to starting address for the object.[21]

The formal specification of these elements are provided in Section 3.1.15.

3.1.13 IPC Connections

One of the primary functions of the microkernel is to provide IPC services. Though the Fluke API provides no direct visibility of state relating to IPC connections, providing control over IPC services is critical to Flask and therefore it is useful to provide some kind of model of IPC connections.

Fluke provides three forms of IPC, one-way, idempotent and reliable. At the level of the API (and of this model), a one-way IPC transfer is essentially an atomic operation which leaves no record in the kernel state. Therefore, our model contains no state relating to one-way IPC.

Idempotent IPC consists of separate calls from a client and replies from a server. There is clearly a record of an idempotent IPC call in the kernel state because the server does not identify a particular client when replying. Because of this, our model should ultimately consider idempotent IPC. However this has been left out, primarily because we have chosen to focus on reliable IPC which is more complex.

For reliable IPC, we add to the kernel state model a single function, *active_connections*, which identifies all of the (*client*, *server*) pairs which have an established reliable IPC connection between them. According to the Fluke API, the set of established reliable IPC connections can be determined by inspecting the *client_link* and *server_link* values of all threads. However, in Flask these links only identify the maximum set of established connections, as other requirements must hold for the connection to be complete.

The formal specification of the *active_connections* function is provided in Section 3.1.15.

[21]Though not modeled, alignment restrictions ensure that the same protections hold for the entire block of memory used for each kernel object.

3.1.14 Kernel Requests

In order to model invocation services (which were defined in the introduction to Section 3), it is necessary to define a model of kernel requests and the invocation of a kernel request.

The type *KERNEL_OP* denotes the set of operations that may be requested of the kernel. This includes the Fluke requests defined in the API and the machine instructions which read, write and execute memory (*read_op*, *write_op*and *exec_op*).

A kernel request consists of the following information:

- *subject* — The thread making the request.
- *op* — The *KERNEL_OP* requested.
- *primary_va* — The *VIRTUAL_ADDRESS* of the fluke object upon which the request is made, or the address being accessed by a *read_op*, *write_op*or *exec_op*.
- *input_vas* - The set of virtual addresses supplied as input parameters (which therefore must be readable by the *subject*.[22]

The set of operations for which the *primary_va* is an input parameter (and hence must be readable by *subject*) will be denoted by *read_ops*.

Finally, to model invocation services we introduce the set *validated_requests* of requests which have satisfied all invocation service requirements and may therefore be processed (subject to any transformation service requirements). This set is modeled in the following section.

THEORY *kernel_request*

```
kernel_request : THEORY

  BEGIN

  IMPORTING address_types

  KERNEL_OP : TYPE+

  read_op, write_op, exec_op : KERNEL_OP

  read_ops : setof[KERNEL_OP]
%% to be complete we really should enumerate the set read_ops, but
%% that would be tedious and provides little benefit

  KERNEL_REQUEST : TYPE =
     [# subject          : SOB_ADDRESS,
        op               : KERNEL_OP,
        primary_va       : VIRTUAL_ADDRESS,
        input_vas        : setof[VIRTUAL_ADDRESS]
        #]

  END kernel_request
```

[22]We do not bother to model a corresponding set of output parameters, since writing to memory can be controlled more directly through transformation services.

3.1.15 Summary

The previous sections define all of the elements of the state of the microkernel. In summary, the microkernel consists of:

- A set *existing_objects* which contains the addresses (*SOB_ADDRESS*) of all existing kernel objects.
- A function *sob_data*, which takes the address of an existing object and returns the object's common object state, described in Section 3.1.1.
- For each of the specific object types described in Sections 3.1.2 through 3.1.10, there are functions analogous to *existing_objects* and *sob_data*. For instance, the function *thread_data* returns a *THREAD* structure for every address in *existing_threads*.

  The set *existing_objects* is the union of the existing objects of each specific object type.
- A *cache* of access vectors, as described in Section 3.1.11.
- The elements of physical memory management described in Section 3.1.12.
- The set of *active_connections* described in Section 3.1.13.
- The set of *validated_requests* described in Section 3.1.14.

THEORY *kernel_state*

```
kernel_state : THEORY

  BEGIN

  IMPORTING sob
  IMPORTING thread
  IMPORTING task
  IMPORTING region
  IMPORTING mapping
  IMPORTING port
  IMPORTING mutex
  IMPORTING cond_var
  IMPORTING ref
  IMPORTING cache
  IMPORTING kernel_request

    BASE_KERNEL_STATE : TYPE = [#
        existing_objects       : setof[SOB_ADDRESS],
        sob_data               : [(existing_objects)->SOB],

        existing_threads       : setof[SOB_ADDRESS],
        thread_data            : [(existing_threads)->THREAD],

        existing_tasks         : setof[SOB_ADDRESS],
        task_data              : [(existing_tasks)->TASK],

        existing_regions       : setof[SOB_ADDRESS],
        region_data            : [(existing_regions)->REGION],

        existing_mappings      : setof[SOB_ADDRESS],
        mapping_data           : [(existing_mappings)->MAPPING],
```

```
        existing_ports              : setof[SOB_ADDRESS],
        port_data                   : [(existing_ports)->PORT],

        existing_psets              : setof[SOB_ADDRESS],
%% no pset_data function since there is no type specific data associated with a port set

        existing_mutexes            : setof[SOB_ADDRESS],
        mutex_data                  : [(existing_mutexes)->MUTEX],

        existing_cond_vars          : setof[SOB_ADDRESS],
        cond_data                   : [(existing_cond_vars)->COND_VAR],

        existing_refs               : setof[SOB_ADDRESS],
        ref_data                    : [(existing_refs)->REF],

        cache                       : CACHE,

        paged_in                    : setof[[(existing_tasks), VIRTUAL_ADDRESS]],
        phys_addr                   : [(paged_in) -> PHYSICAL_ADDRESS],
        mem_prot                    : [(paged_in) -> MEMORY_PROTECTIONS],

        phys_addr_sid               : [PHYSICAL_ADDRESS -> SID],
        contents                    : [PHYSICAL_ADDRESS -> nat],

        active_connections          : setof[[(existing_threads), (existing_threads)]],

        validated_requests          : setof[KERNEL_REQUEST]
        #]


%% invariants and assertions about state (currently just one)

  ks : VAR BASE_KERNEL_STATE

  kernel_obj_union(ks) : bool =
    FORALL (sob_addr : SOB_ADDRESS) :
      (existing_objects(ks)(sob_addr)
        IFF (existing_regions(ks)(sob_addr)
                OR existing_mappings(ks)(sob_addr)
                OR existing_ports(ks)(sob_addr)
                OR existing_psets(ks)(sob_addr)
                OR existing_cond_vars(ks)(sob_addr)
                OR existing_mutexes(ks)(sob_addr)
                OR existing_refs(ks)(sob_addr)
                OR existing_tasks(ks)(sob_addr)
                OR existing_threads(ks)(sob_addr)))

  kernel_state(ks) : bool =
    kernel_obj_union(ks)

  KERNEL_STATE : TYPE = (kernel_state)


%% finally we define the memory protection access functions described earlier

  st : VAR KERNEL_STATE
  thread_addr : VAR SOB_ADDRESS
  va : VAR VIRTUAL_ADDRESS
  pa : VAR PHYSICAL_ADDRESS
  prot : VAR MEMORY_PROTECTION

  memory_allows_va(st, thread_addr, va, prot) : bool =
    member(thread_addr, existing_threads(st))
    AND LET task_addr = task_link(thread_data(st)(thread_addr)) IN
```

```
      member(task_addr, existing_tasks(st))
  AND member((task_addr, va), paged_in(st))
  AND member(prot, mem_prot(st)(task_addr, va))

memory_allows_pa(st, thread_addr, pa, prot) : bool =
  EXISTS va : (
    memory_allows_va(st, thread_addr, va, prot)
    AND pa = phys_addr(st)(task_link(thread_data(st)(thread_addr)), va)
    )

memory_allows_sob(st, thread_addr, (sob_addr : (existing_objects(st))), prot) : bool =
  memory_allows_pa(st, thread_addr, u_pa(sob_data(st)(sob_addr)), prot)

%% Ideally, we would restrict the domain of memory_allows_va to those
%% threads which exist and have an existing task as their task_link.
%% But this would require some messy PVS here and everywhere that the
%% function is called (e.g., to ensure that cur_thread meets these
%% restrictions).

END kernel_state
```

3.2 Microkernel Services and Requirements

This section defines the services which are controlled by Flask and the requirements that must be met for each service to be performed. It is organized as follows:

- Section 3.2.1 describes the general approach used to define the services and requirements in the following sections. This section is necessary background to understand the statements of the services and requirements.
- Sections 3.2.2 through 3.2.13 define the services and requirements for accessing the elements of the kernel state described in Sections 3.1.1 through 3.1.12.
- Section 3.2.14 defines the services and requirements controlling IPC initiated by threads outside of the microkernel.
- Section 3.2.15 defines the services and requirements controlling IPC initiated by the microkernel.

3.2.1 Overview

The services and requirements for each portion of the kernel state are presented in the three subsections:

- Definition of transformation services.
- Definition of invocation services.
- Definition of the requirements which must be met for any of the services to be provided.

In addition, there is occasionally a fourth section which discusses outstanding issues.

This overview is organized in the same three subsections.

3.2.1.1 Transformation Services

Transformation services are those services which can be defined as a change in the state of the kernel without referring to any particular kernel request, i.e., a change to any element of the kernel state other than the set of *validated_requests*.

Our basic model is to define a single transformation service for changing each individual element of the kernel state. However, this model is almost never followed exactly, generally for some mix of the following reasons:

- Often it is desirable to define separate transformation services depending upon how an element of the kernel state changes, so that different changes can be controlled by different permissions.

  For instance, the function *sob_data* is an element of the state of the kernel which takes as input the address of an existing object and returns the data associated with that object. While it would be possible to control the service of changing this function without any consideration of how it is changed, that would not be very useful. It is more useful to control changes to the function's value for a particular object.

  As a less drastic example, the set of *existing_objects* can change by either adding or removing an object, and each of these is defined as a separate service (see Section 3.2.2.1).

- Elements of the kernel state are not completely independent: a change to one element of the kernel state can have a side effect of changing another element of the kernel state. In some of these cases, changes to an element caused by side effects are not considered part of the service of changing that element.

  For instance, the *sob_data* function can change as a side effect of changing the set of *existing_objects*. In this case, the services defined as changes in *sob_data* factor out changes which arise simply because of changes to *existing_objects*.

3.2.1.1.1 Transformation Services for Kernel Objects

The state of the base kernel object types are defined by functions **_data* which take as input the address of an existing object of the appropriate type and return the state of the object as defined in Section 3.1. There are three types of transformation services which are generally defined for kernel objects. These services are described here in some detail, using port objects as an example. The corresponding service definitions in Sections 3.2.2 through 3.2.11 will be much briefer.

The state of a port object is captured by the *port_data* function in the kernel state. This function takes as input the address of an existing port and returns the state of the port object. The basic model for transformation services would suggest that a single transformation service should be defined to capture changes to the function *port_data*.

As discussed above, it makes more sense to define services identifying changes to the value of *port_data* for a particular port. This can change in three ways:

- The port can be added to the domain of *port_data*
- The port can be removed from the domain of *port_data*
- The port can remain in the domain of *port_data* but the value of the function can change.

The first two are side effects of object creation and destruction. Since creation and destruction are defined as separate services (see Section 3.2.2.1), these side effects can be factored out of the service for changing the state of a port. So it seems that a very simple service definition is sufficient: a change to the port state occurs whenever *port_addr* is an existing port which is not destroyed and *port_data*(*port_addr*) is changed.

Unfortunately, there is still a problem that the value of the *pset_link* can change as a side effect of the destruction of the object to which it points. Our approach to addressing this is to define one service for changes to the simple state and a separate service for each link.

For instance, the service for changing the simple state of a port object is defined as follows:

> (*Port_Set_Simple_State*) This service is provided if the simple state of an existing port object is modified during a transition, other than by destruction of the port object.

Or more precisely:

> This service is provided if there is an address *port_addr* which is in the set of *existing_ports* in both the initial and final states of a transition, and the *alias* value associated with *port_addr* is modified during the transition.

The general name of this service is *Object_Set_Simple_State*, where "Object" is one of the fluke object types.

The service for changing the port set link from a port object factors out the possibility of destruction of the port set, and is defined as follows:

> (*Port_Set_Pset*) This service is provided if the *pset_link* from a port object is modified during a transition, other than by destruction of the port object or destruction of the object pointed at by the *pset_link* in the initial state of the transition.

Or more precisely:

> This service is provided if there is an address *port_addr* such that:
>
> - *port_addr* is in the set of *existing_ports* in both the initial and final states of a transition,
> - the *pset_link* associated with *port_addr* is modified during the transition, and
> - the port set object (if any) identified by *pset_link* in the initial state of the transition is not destroyed during the transition.

The general name of this service is *Object_Set_Link*, where "Object" is one of the fluke object types and "Link" is the name of a link from that object type.

The *Object_Set_Link* service only identifies changes to the link of an existing object. As mentioned in Section 2.2.2.3, in several cases the control policy also controls the value of the link. There are several possible ways in which this control can be recorded in the control policy:

- By defining a state invariant requirement relating the microkernel state to the state of the security policy. This may be the ideal solution, but it requires that the control policy address the security policy simultaneous with the microkernel, and we have otherwise attempted to avoid this complexity.
- By defining a control on the value of the link when it is actually accessed rather than at all times. For some links, including all three links which point to a port object, this

is a perfectly reasonable approach because it is the use of the link in determining the destination for a message from the kernel which is important. However, for other links which define an ongoing relationships, such as the association between a thread and a task, this is not appropriate.

- By defining requirements on state transitions which assign a new value to a link. This is weaker than stating an invariant because it does not address the value of any links in the initial state of the system and it does not address how a security policy change can affect the invariant.

We have chosen the last of these options, with the understanding the there are also requirements on the links in the initial state of the system and a requirement that the value of a link must be reset if a required permission is revoked during a policy change.

Thus we control the value of inter-object links by stating requirements on services like the following:

> (*Port_Associate_Pset*) This service is provided if a *pset_link* is established between a port object and a port set object.

Or more precisely:[23]

> This service is provided if there are addresses *port_addr* and *pset_addr* such that in the final state of a transition
>
> - *port_addr* is in *existing_ports*,
> - *pset_addr* is in *existing_psets*, and
> - the *pset_link* from *port_addr* points to *pset_addr*,
>
> whereas at least one of these three conditions did not hold in the initial state of the transition.

The general name of this service is *Object_Associate_Link*.

3.2.1.2 Invocation Services

Invocation services are those services which are defined by adding a particular kernel request to the set of *validated_requests*. Invocation services generally (but not always) reflect kernel services which observe elements of the kernel state, and therefore cannot be easily defined by detecting changes in the kernel state.

For each object type, the "Invocation Services" subsection simply lists those requests for which invocation services are defined, and the parameters to those requests which are needed in the definition of the requirements. The names of the parameters are taken from the Fluke API [2].

3.2.1.3 Requirements

The requirements which must be satisfied for a service to be provided are presented in a four column table:

- The first column identifies a service, either the name of a transformation service or the name of a request controlled through an invocation service. In each table, the transformation services are listed first, separated from the invocation services by a double-line.

[23] There is a more subtle case that is also addressed by the more general definition of *Port_Associate_Pset*. Suppose *pset_link* currently points to an address with no object, and a port set is later created at that address. Then a link is "established" even though there is no explicit change to the state of the port and the *Port_Set_Pset* services does not occur. However, this example is also not possible in the current implementation.

- The second and third columns identify the source and target entities for a permission listed in the fourth column. When "subject" appears in these columns, it indicates the thread for whom the service is being performed. The columns are empty if the requirement in the fourth column is something other than a permission check.

  For transformation services, entities listed in these columns are identified in the definition of the service.

  For invocation services, entities listed in these columns refer to parameters to the request or other values derived from the parameters.

- The fourth column identifies the permission that must be held between the entities in the second and third columns, or if those columns are blank, some other condition that must be satisfied. When multiple requirements are listed for a single service, all requirements must be satisfied unless they are specified by an "OR" (in which case there is no separating line between the requirements).

  If this column indicates a Fluke memory protection (such as FLUKE_PROT_READ), the requirement is that the task in the second column is granted that permission to the virtual address or object indicated in the third column, according to the *memory_allows* and *memory_allows_sob* functions defined in Section 3.1.12. These requirements are the capability requirements described in Section 2.1.3.3.

  Any entry in this column beginning with "FLASK:" is a requirement in the Flask microkernel but not the Fluke microkernel. This typically indicates a Flask permission that must be held between the entities in the second and third columns, though it may indicate some other condition on the state.

The formalized PVS statements of these requirements indicate that a permission in the fourth column must be held in the initial state of each transition, with no such requirement on the final state of the transition. This is another way in which revocation of permissions is not fully addressed by the formal policy statements.

It is important to recognize that invocation services are sometimes defined for requests which also perform a transformation service. In these cases, the requirements listed in the table specifically for the request only identify some of the requirements for the request to be performed. The requirements on the transformation services must always be satisfied.

The most common examples of this are requests that change the state of an object by providing a new object link. Such a request provides a transformation service by changing the link but also provides an invocation service because the input reference parameter must be readable.

In Section 3.2.1.1.1, three standard services are described for kernel objects, *Object_Set_Simple_State*, *Object_Set_Link* and *Object_Associate_Link*. The standard requirements for these services are shown in Table 1, where *sob_addr* is the address of the object being accessed, and *target_addr* is the address of an object being linked to *sob_addr*.

Transformation Service	Source	Target	Requirement
Object_Set_Simple_State	cur_thread	sob_addr	FLUKE_PROT_WRITE
Object_Set_Link	cur_thread	sob_addr	FLUKE_PROT_WRITE
Object_Associate_Link	sob_addr	target_addr	FLASK: OBJECT_LINK_PERM OR FLASK: sid(sob_addr) = sid(target_addr)

Table 1: Requirements on Standard Object Services

The two FLUKE_PROT_WRITE permissions in the table are Fluke capability checks as discussed in Section 2.1.3.3. The Flask permission OBJECT_LINK_PERM is the typical name of a permission controlling a value of a link. However, as discussed in Section 2.2.2.3, this permission is often replaced with a requirement that the initial object and the linked object have the same SID, as indicated in the table.

Any Flask requirements other than those requiring an OBJECT_LINK_PERM permission will be discussed in the text with the rationale for the requirement.

The requirements sections each end with a formal statement of the requirements on transformation services. The formalization of the invocation services is provided in Section 3.2.16.2.

3.2.2 Common Object State

3.2.2.1 Common Object Transformation Services We define two separate transformation services for changes to the set of *existing_objects*, one for adding an object to the set (*Object_Create* and one for removing an object from the set (*Object_Destroy*).

Transformation Service Definition *Object_Create(sob_addr)* This service is provided if a new supervisor object is created. Specifically, if there is an address *sob_addr* which is added to the set of *existing_objects*.

Transformation Service Definition *Object_Destroy(sob_addr)* This service is provided if an existing supervisor object is destroyed. Specifically, if there is an address *sob_addr* which is removed from the set of *existing_objects*.

The common state of an object could be controlled through a single service analogous to the *Object_Set_Simple_State* service described in Section 3.2.1.1.1. This will not be done however, since three of the four elements of the state (*hash*, *sob_type* and *sid*) are not allowed to change. Therefore we define distinct services for changing each of the elements of the common state.

Transformation Service Definition *Object_Set_Hash(sob_addr)* This service is provided if the *hash* value for an existing object is modified during a transition, other than by destruction of the object.

Transformation Service Definition *Object_Set_Type(sob_addr)* This service is provided if the *sob_type* of an existing object is modified during a transition, other than by destruction of the object.

Transformation Service Definition *Object_Set_Sid(sob_addr)* This service is provided if the *sid* of an existing object is modified during a transition, other than by destruction of the object.

Transformation Service Definition *Object_Set_U_Pa(sob_addr)* This service is provided if the *u_pa* of an existing object is modified during a transition, other than by destruction of the object.

THEORY *common_services*

---

common_services : **THEORY**

BEGIN

IMPORTING *kernel_state*

st1, *st2* : **VAR** *KERNEL_STATE*
sob_addr : **VAR** *SOB_ADDRESS*

```
Object_Create(st1, st2, sob_addr) : bool =
  member(sob_addr, existing_objects(st2))
  AND NOT member(sob_addr, existing_objects(st1))

Object_Destroy(st1, st2, sob_addr) : bool =
  member(sob_addr, existing_objects(st1))
  AND NOT member(sob_addr, existing_objects(st2))

Object_Set_Hash(st1, st2, sob_addr) : bool =
  existing_objects(st1)(sob_addr) AND existing_objects(st2)(sob_addr)
  AND
  hash(sob_data(st1)(sob_addr)) /= hash(sob_data(st2)(sob_addr))

Object_Set_Type(st1, st2, sob_addr) : bool =
  existing_objects(st1)(sob_addr) AND existing_objects(st2)(sob_addr)
  AND
  sob_type(sob_data(st1)(sob_addr)) /= sob_type(sob_data(st2)(sob_addr))

Object_Set_Sid(st1, st2, sob_addr) : bool =
  existing_objects(st1)(sob_addr) AND existing_objects(st2)(sob_addr)
  AND
  sid(sob_data(st1)(sob_addr)) /= sid(sob_data(st2)(sob_addr))

Object_Set_U_Pa(st1, st2, sob_addr) : bool =
  existing_objects(st1)(sob_addr) AND existing_objects(st2)(sob_addr)
  AND
  u_pa(sob_data(st1)(sob_addr)) /= u_pa(sob_data(st2)(sob_addr))

END common_services
```

3.2.2.2 Common Object Invocation Services

All kernel requests are specific to a particular object type, so there are no invocation services on the common object state.

3.2.2.3 Common Object Requirements

Table 2 defines the requirements on the common object services.

Transformation Service	Source	Target	Requirement
Object_Create	cur_thread	sob_addr	FLUKE_PROT_WRITE
	cur_thread	sob_addr	FLUKE_PROT_CREATE_OB
	—	—	FLASK: sid(sob_addr) = phys_addr_sid(u_pa)
Object_Destroy	cur_thread	sob_addr	FLUKE_PROT_WRITE
Object_Set_Hash	—	—	PROHIBITED
Object_Set_Type	—	—	PROHIBITED
Object_Set_Sid	—	—	FLASK: PROHIBITED
Object_Set_U_Pa	cur_thread	old_pa	FLUKE_PROT_WRITE
	cur_thread	new_pa	FLUKE_PROT_WRITE
	cur_thread	new_pa	FLUKE_PROT_CREATE_OB
	—	—	FLASK: phys_addr_sid(old_pa) = phys_addr_sid(new_pa)

Table 2: Requirements on Common Object Services

The non-standard Flask requirements on the *Object_Create* and *Object_Set_U_Pa* are discussed below in Section 3.2.2.4.

THEORY *common_service_rqmts*

```
common_service_rqmts : THEORY

  BEGIN

  IMPORTING common_services

  st1, st2 : VAR KERNEL_STATE
  cur_thread : VAR SOB_ADDRESS
  sob_addr : VAR SOB_ADDRESS

  Object_Create_rqmts(st1, st2, cur_thread) : bool =
    FORALL sob_addr :
      Object_Create(st1, st2, sob_addr)
        IMPLIES (
          LET pa = u_pa(sob_data(st2)(sob_addr)) IN
          memory_allows_pa(st1, cur_thread, pa, FLUKE_PROT_WRITE)
          AND
          memory_allows_pa(st1, cur_thread, pa, FLUKE_PROT_CREATE_OB)
          AND
          sid(sob_data(st2)(sob_addr)) = phys_addr_sid(st2)(pa)
        )

  Object_Destroy_rqmts(st1, st2, cur_thread) : bool =
    FORALL sob_addr :
      Object_Destroy(st1, st2, sob_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, sob_addr, FLUKE_PROT_WRITE)

  Object_Set_Hash_rqmts(st1, st2) : bool =
    FORALL sob_addr :
      NOT Object_Set_Hash(st1, st2, sob_addr)

  Object_Set_Type_rqmts(st1, st2) : bool =
    FORALL sob_addr :
      NOT Object_Set_Type(st1, st2, sob_addr)

  Object_Set_Sid_rqmts(st1, st2) : bool =
    FORALL sob_addr :
      NOT Object_Set_Sid(st1, st2, sob_addr)

  Object_Set_U_Pa_rqmts(st1, st2, cur_thread) : bool =
    FORALL sob_addr :
      Object_Set_U_Pa(st1, st2, sob_addr)
        IMPLIES (
          LET old_pa = u_pa(sob_data(st1)(sob_addr)),
              new_pa = u_pa(sob_data(st2)(sob_addr)) IN
          memory_allows_pa(st1, cur_thread, old_pa, FLUKE_PROT_WRITE)
          AND
          memory_allows_pa(st1, cur_thread, new_pa, FLUKE_PROT_WRITE)
          AND
          memory_allows_pa(st1, cur_thread, new_pa, FLUKE_PROT_CREATE_OB)
          AND
          phys_addr_sid(st1)(old_pa) = phys_addr_sid(st2)(new_pa)
        )

  common_service_rqmts(st1, st2, cur_thread) : bool =
    Object_Create_rqmts(st1, st2, cur_thread)
    AND Object_Destroy_rqmts(st1, st2, cur_thread)
    AND Object_Set_Hash_rqmts(st1, st2)
    AND Object_Set_Type_rqmts(st1, st2)
    AND Object_Set_Sid_rqmts(st1, st2)
    AND Object_Set_U_Pa_rqmts(st1, st2, cur_thread)
```

END *common_service_rqmts*

3.2.2.4 Common Object State Issues

- One of the fundamental properties of Flask (see Section 2.2.2.1) is that the SID of a kernel object is equal to the SID of the physical memory where the user space component of the object is located, i.e., *u_pa*.

  This property can be stated formally as:

  $sid(sob\_data(st)(sob\_addr)) = phys\_addr\_sid(st)(u\_pa(sob\_data(st)(sob\_addr)))$

  Because we have chosen to state requirements on state transitions rather than state invariants, this property is decomposed into four requirements:

  – The property must hold when an object is created. This is the Flask requirement on the *Object_Create* service (though this does not address objects existing in the kernel's initial state).

  – The *sid* of an object cannot change. This is the requirement on the *Object_Set_Sid* service.

  – When the *u_pa* of an object changes, the new address must have the same sid as the old address. This is the Flask requirement on the *Object_Set_U_Pa* service.

  – When the sid of a physical address changes, there must not be any object with that address as its *u_pa* value. This is the Flask requirement on the service *Memory_Set_Sid*, in Section 3.2.13.

3.2.3 Threads

3.2.3.1 Thread Transformation Services

The following transformation services are defined for thread objects.

Note that while there are five object links from a thread object, there are *Set_Link* services for only four of the links and *Associate_Link* services for only two of the links.

There is no *Set_Link* or *Associate_Link* service for the *waiting_for_link* since it cannot be changed except as a side effect of other processing.

The values of *client_link* and *server_link* are not directly controlled, since as described in Section 2.2.2.3, the IPC control policy relates each side of the connection through the effective SID values.

The values of the *effective_client_sid* and *effective_server_sid* are controlled when set for the purpose of satisfying the IPC policy described in Section 2.2.2.3. The values of *client_effective_client_sid* and *server_effective_server_sid* are not controlled directly when they are set. However, for a reliable IPC connection to be established the the value of the client's *server_effective_server_sid* must agree with the server's *effective_server_sid* and the value of the server's *client_effective_client_sid* must agree with the client's *effective_client_sid* (see Section 3.2.14.3). Hence, these values are indirectly controlled through a permission against the SID of the peer when a reliable IPC connection is established.

Transformation Service Definition *Thread_Set_Simple_State(thread_addr)* This service is provided if the simple state of an existing thread object is modified during a transition, other than by destruction of the thread object.

Transformation Service Definition *Thread_Set_Task(thread_addr)* This service is provided if the *task_link* from a thread object is modified during a transition, other than by destruction of the thread object or destruction of the object pointed at by the *task_link* in the initial state of the transition.

Transformation Service Definition *Thread_Set_Scheduler(thread_addr)* This service is provided if the *scheduler_link* from a thread object is modified during a transition, other than by destruction of the thread object or destruction of the object pointed at by the *scheduler_link* in the initial state of the transition.

Transformation Service Definition *Thread_Set_Client(thread_addr)* This service is provided if the *client_link* from a thread object is modified during a transition, other than by destruction of the thread object or destruction of the object pointed at by the *client_link* in the initial state of the transition.

Transformation Service Definition *Thread_Set_Server(thread_addr)* This service is provided if the *server_link* from a thread object is modified during a transition, other than by destruction of the thread object or destruction of the object pointed at by the *server_link* in the initial state of the transition.

Transformation Service Definition *Thread_Associate_Eff_Client(thread_addr, esid)* This service is provided if the *effective_client_sid* for the thread is set to a new value.

Transformation Service Definition *Thread_Associate_Eff_Server(thread_addr, esid)* This service is provided if the *effective_server_sid* for the thread is set to a new value.

Transformation Service Definition *Thread_Associate_Task(thread_addr, task_addr)* This service is provided if a *task_link* is established between a thread object and a task object.

Transformation Service Definition *Thread_Associate_Scheduler(thread_addr, port_addr)* This service is provided if a *scheduler_link* is established between a thread object and a port object.

THEORY *thread_services*

```
thread_services    : THEORY

  BEGIN

  IMPORTING kernel_state
  IMPORTING common_services

  st, st1, st2 : VAR KERNEL_STATE
  thread_addr : VAR SOB_ADDRESS
  task_addr, port_addr : VAR SOB_ADDRESS
  esid : VAR SID


  Thread_Set_Simple_State(st1, st2, thread_addr) : bool =
    member(thread_addr, existing_threads(st1)) AND member(thread_addr, existing_threads(st2))
    AND LET initial = thread_data(st1)(thread_addr),
            final = thread_data(st2)(thread_addr) IN
        (
            thread_va(initial) /= thread_va(final)
         OR sched_val(initial) /= sched_val(final)
         OR flags(initial) /= flags(final)
         OR exc_state(initial) /= exc_state(final)
         OR exc_handlers(initial) /= exc_handlers(final)
         OR ipc_state(initial) /= ipc_state(final)
```

```
      OR effective_client_sid(initial) /= effective_client_sid(final)
      OR effective_server_sid(initial) /= effective_server_sid(final)
      OR client_effective_client_sid(initial) /= client_effective_client_sid(final)
      OR server_effective_server_sid(initial) /= server_effective_server_sid(final)
     )

Thread_Set_Task(st1, st2, thread_addr) : bool =
  member(thread_addr, existing_threads(st1)) AND member(thread_addr, existing_threads(st2))
  AND
  task_link(thread_data(st1)(thread_addr)) /= task_link(thread_data(st2)(thread_addr))
  AND
  NOT Object_Destroy(st1, st2, task_link(thread_data(st1)(thread_addr)))

Thread_Set_Scheduler(st1, st2, thread_addr) : bool =
  member(thread_addr, existing_threads(st1)) AND member(thread_addr, existing_threads(st2))
  AND
  scheduler_link(thread_data(st1)(thread_addr)) /= scheduler_link(thread_data(st2)(thread_addr))
  AND
  NOT Object_Destroy(st1, st2, scheduler_link(thread_data(st1)(thread_addr)))

Thread_Set_Client(st1, st2, thread_addr) : bool =
  member(thread_addr, existing_threads(st1)) AND member(thread_addr, existing_threads(st2))
  AND
  client_link(thread_data(st1)(thread_addr)) /= client_link(thread_data(st2)(thread_addr))
  AND
  NOT Object_Destroy(st1, st2, client_link(thread_data(st1)(thread_addr)))

Thread_Set_Server(st1, st2, thread_addr) : bool =
  member(thread_addr, existing_threads(st1)) AND member(thread_addr, existing_threads(st2))
  AND
  server_link(thread_data(st1)(thread_addr)) /= server_link(thread_data(st2)(thread_addr))
  AND
  NOT Object_Destroy(st1, st2, server_link(thread_data(st1)(thread_addr)))

thread_eff_client_value(st, thread_addr, esid) : bool =
  member(thread_addr, existing_threads(st))
  AND effective_client_sid(thread_data(st)(thread_addr)) = esid

Thread_Associate_Eff_Client(st1, st2, thread_addr, esid) : bool =
  NOT thread_eff_client_value(st1, thread_addr, esid)
  AND thread_eff_client_value(st2, thread_addr, esid)

thread_eff_server_value(st, thread_addr, esid) : bool =
  member(thread_addr, existing_threads(st))
  AND effective_server_sid(thread_data(st)(thread_addr)) = esid

Thread_Associate_Eff_Server(st1, st2, thread_addr, esid) : bool =
  NOT thread_eff_server_value(st1, thread_addr, esid)
  AND thread_eff_server_value(st2, thread_addr, esid)

thread_task_linked(st, thread_addr, task_addr) : bool =
  member(thread_addr, existing_threads(st))
  AND member(task_addr, existing_tasks(st))
  AND task_link(thread_data(st)(thread_addr)) = task_addr

Thread_Associate_Task(st1, st2, thread_addr, task_addr) : bool =
  NOT thread_task_linked(st1, thread_addr, task_addr)
```

```
    AND thread_task_linked(st2, thread_addr, task_addr)

  thread_scheduler_linked(st, thread_addr, port_addr) : bool =
    member(thread_addr, existing_threads(st))
    AND member(port_addr, existing_ports(st))
    AND scheduler_link(thread_data(st)(thread_addr)) = port_addr

  Thread_Associate_Scheduler(st1, st2, thread_addr, port_addr) : bool =
    NOT thread_scheduler_linked(st1, thread_addr, port_addr)
    AND thread_scheduler_linked(st2, thread_addr, port_addr)

  END thread_services
```

3.2.3.2 Thread Invocation Services

Invocation services are defined for the following thread requests.

- `fluke_thread_get_client(cur_thread)`
- `fluke_thread_get_client_secure(cur_thread)`
- `fluke_thread_set_client(client_ref)`
- `fluke_thread_set_client_secure(client_ref)`,
- `fluke_thread_get_server(cur_thread)`
- `fluke_thread_get_server_secure(cur_thread)`
- `fluke_thread_set_server(server_ref)`
- `fluke_thread_set_server_secure(server_ref)`,
- `fluke_thread_get_state(thread)`
- `fluke_thread_set_state(task_ref, scheduler_ref, waiting_for_ref, client_ref, server_ref)`
- `fluke_thread_set_state_secure(task_ref, scheduler_ref, waiting_for_ref, client_ref, server_ref)`
- `fluke_thread_get_saved_state(cur_thread)`

3.2.3.3 Thread Requirements

Table 3 defines the requirements on the thread services.

The *Thread_Associate_Task* service requires SID equality because a thread has complete access to the task with which it is associated.

THEORY *thread_service_rqmts*

```
thread_service_rqmts : THEORY

  BEGIN

  IMPORTING thread_services

  st1, st2 : VAR KERNEL_STATE
  cur_thread : VAR SOB_ADDRESS
  thread_addr, task_addr, port_addr : VAR SOB_ADDRESS
  esid : VAR SID

  Thread_Set_Simple_State_rqmts(st1, st2, cur_thread) : bool =
    FORALL thread_addr :
      Thread_Set_Simple_State(st1, st2, thread_addr)
        IMPLIES (
```

Transformation Service	Source	Target	Requirement
Thread_Set_Simple_State	cur_thread	thread_addr	FLUKE_PROT_WRITE OR cur_thread = thread_addr
Thread_Set_Task	cur_thread	thread_addr	FLUKE_PROT_WRITE
Thread_Set_Scheduler	cur_thread	thread_addr	FLUKE_PROT_WRITE
Thread_Set_Client	cur_thread	thread_addr	FLUKE_PROT_WRITE OR cur_thread = thread_addr
Thread_Set_Server	cur_thread	thread_addr	FLUKE_PROT_WRITE OR cur_thread = thread_addr
Thread_Associate_Eff_Client	thread_addr	esid	FLASK: SPECIFY_CLIENT_PERM OR FLASK: sid(thread_addr) = esid
Thread_Associate_Eff_Server	thread_addr	esid	FLASK: SPECIFY_SERVER_PERM OR FLASK: sid(thread_addr) = esid
Thread_Associate_Task	—	—	FLASK: sid(thread_addr) = sid(task_addr)
Thread_Associate_Scheduler	thread_addr	port_addr	FLASK: THREAD_SCHEDULER_PERM

Invocation Service	Source	Target	Requirement
fluke_thread_get_client(_secure)	cur_thread	thread	cur_thread = thread
fluke_thread_set_client(_secure)	cur_thread	client_ref	FLUKE_PROT_READ
fluke_thread_get_server(_secure)	cur_thread	thread	cur_thread = thread
fluke_thread_set_server(_secure)	cur_thread	server_ref	FLUKE_PROT_READ
fluke_thread_get_state(_secure)	cur_thread	thread	FLUKE_PROT_WRITE
fluke_thread_set_state(_secure)	cur_thread	task_ref	FLUKE_PROT_READ
	cur_thread	scheduler_ref	FLUKE_PROT_READ
	cur_thread	waiting_for_ref	FLUKE_PROT_READ
	cur_thread	client_ref	FLUKE_PROT_READ
	cur_thread	server_ref	FLUKE_PROT_READ
fluke_thread_get_saved_state	cur_thread	thread	cur_thread = thread

Table 3: Requirements on Thread Services

```
        memory_allows_sob(st1, cur_thread, thread_addr, FLUKE_PROT_WRITE)
        OR
        cur_thread = thread_addr
      )

Thread_Set_Task_rqmts(st1, st2, cur_thread) : bool =
  FORALL thread_addr :
    Thread_Set_Task(st1, st2, thread_addr)
      IMPLIES
        memory_allows_sob(st1, cur_thread, thread_addr, FLUKE_PROT_WRITE)

Thread_Set_Scheduler_rqmts(st1, st2, cur_thread) : bool =
  FORALL thread_addr :
    Thread_Set_Scheduler(st1, st2, thread_addr)
      IMPLIES
        memory_allows_sob(st1, cur_thread, thread_addr, FLUKE_PROT_WRITE)

Thread_Set_Client_rqmts(st1, st2, cur_thread) : bool =
  FORALL thread_addr :
    Thread_Set_Client(st1, st2, thread_addr)
      IMPLIES (
        memory_allows_sob(st1, cur_thread, thread_addr, FLUKE_PROT_WRITE)
        OR
        cur_thread = thread_addr
      )

Thread_Set_Server_rqmts(st1, st2, cur_thread) : bool =
```

```
    FORALL thread_addr :
      Thread_Set_Server(st1, st2, thread_addr)
        IMPLIES (
          memory_allows_sob(st1, cur_thread, thread_addr, FLUKE_PROT_WRITE)
          OR
          cur_thread = thread_addr
        )

  Thread_Associate_Eff_Client_rqmts(st1, st2) : bool =
    FORALL thread_addr, esid :
      Thread_Associate_Eff_Client(st1, st2, thread_addr, esid)
        IMPLIES (
          cache_allows(cache(st1),
                  SECCLASS_SUBJECT, sid(sob_data(st2)(thread_addr)),
                  SECCLASS_SUBJECT, esid,
                  SPECIFY_CLIENT_PERM)
          OR
          esid = sid(sob_data(st2)(thread_addr))
        )

  Thread_Associate_Eff_Server_rqmts(st1, st2) : bool =
    FORALL thread_addr, esid :
      Thread_Associate_Eff_Server(st1, st2, thread_addr, esid)
        IMPLIES (
          cache_allows(cache(st1),
                  SECCLASS_SUBJECT, sid(sob_data(st2)(thread_addr)),
                  SECCLASS_SUBJECT, esid,
                  SPECIFY_SERVER_PERM)
          OR
          esid = sid(sob_data(st2)(thread_addr))
        )

  Thread_Associate_Task_rqmts(st1, st2) : bool =
    FORALL thread_addr, task_addr :
      Thread_Associate_Task(st1, st2, thread_addr, task_addr)
        IMPLIES
          sid(sob_data(st2)(thread_addr)) = sid(sob_data(st2)(task_addr))

  Thread_Associate_Scheduler_rqmts(st1, st2) : bool =
    FORALL thread_addr, port_addr :
      Thread_Associate_Scheduler(st1, st2, thread_addr, port_addr)
        IMPLIES
          cache_allows(cache(st1),
                  SECCLASS_SUBJECT, sid(sob_data(st2)(thread_addr)),
                  SECCLASS_SUBJECT, sid(sob_data(st2)(port_addr)),
                  THREAD_SCHEDULER_PERM)

  thread_service_rqmts(st1, st2, cur_thread) : bool =
    Thread_Set_Simple_State_rqmts(st1, st2, cur_thread)
    AND Thread_Set_Task_rqmts(st1, st2, cur_thread)
    AND Thread_Set_Scheduler_rqmts(st1, st2, cur_thread)
    AND Thread_Set_Client_rqmts(st1, st2, cur_thread)
    AND Thread_Set_Server_rqmts(st1, st2, cur_thread)
    AND Thread_Associate_Eff_Client_rqmts(st1, st2)
    AND Thread_Associate_Eff_Server_rqmts(st1, st2)
    AND Thread_Associate_Task_rqmts(st1, st2)
    AND Thread_Associate_Scheduler_rqmts(st1, st2)

  END thread_service_rqmts
```

3.2.3.4 Thread Issues

- The list of requests which may potentially provide the service *Thread_Set_Simple_State* is exceptionally long. For most most of these requests, the thread state is changed as a side effect of a request performed on another thread or object. The two most common examples of this are IPC requests made by a thread which is connected to or attempting to connect to another thread, and changes to the state of an object that a thread is waiting for.

  Ultimately, this should be addressed by partitioning the thread state into smaller units and expanding the requirements to address these side effects. For instance, a client of a server thread should be allowed to change the *client_effective_client_sid* element of the server thread's state.

  However, we do not expect to perform this partitioning as part of the current program. First of all, it would take a considerable amount of effort to determine precisely all of the ways in which the thread state is changed as a side effect of other operations, and this may be highly implementation dependent. Moreover, there do not appear to be any fundamental questions that would be answered by undertaking such an effort.

- There is a similar issue with the *waiting_for_link*, but in that case we have avoided stating any requirement at all. A better solution would be to state a requirement on a *Thread_Set_Waiting_For* and actually enumerate those situations in which the link can be changed.

3.2.4 Tasks

3.2.4.1 Task Transformation Services The following transformation services are defined for task objects.

Transformation Service Definition *Task_Set_Keeper(task_addr)* This service is provided if the *keeper_link* from a task object is modified during a transition, other than by destruction of the task object or destruction of the object pointed at by the *keeper_link* in the initial state of the transition.

Transformation Service Definition *Task_Associate_Keeper(task_addr, port_addr)* This service is provided if a *keeper_link* is established between a task object and a port object.

THEORY *task_services*

```
task_services : THEORY

  BEGIN

  IMPORTING kernel_state
  IMPORTING common_services

  st, st1, st2 : VAR KERNEL_STATE
  task_addr : VAR SOB_ADDRESS
  port_addr : VAR SOB_ADDRESS


  Task_Set_Keeper(st1, st2, task_addr) : bool =
    member(task_addr, existing_tasks(st1)) AND member(task_addr, existing_tasks(st2))
    AND
```

```
    keeper_link(task_data(st1)(task_addr))  /=  keeper_link(task_data(st2)(task_addr))
    AND
    NOT  Object_Destroy(st1,  st2,  keeper_link(task_data(st1)(task_addr)))


  task_keeper_linked(st,  task_addr,  port_addr)  :  bool  =
    member(task_addr,  existing_tasks(st))
    AND  member(port_addr,  existing_ports(st))
    AND  keeper_link(task_data(st)(task_addr))  =  port_addr

  Task_Associate_Keeper(st1,  st2,  task_addr,  port_addr)  :  bool  =
    NOT  task_keeper_linked(st1,  task_addr,  port_addr)
    AND  task_keeper_linked(st2,  task_addr,  port_addr)

  END  task_services
```

3.2.4.2 Task Invocation Services

Invocation services are defined for the following task requests.

- `fluke_task_get_state(task)`
- `fluke_task_set_state(keeper_ref)`

3.2.4.3 Task Requirements

Table 4 defines the requirements on the task services.

Transformation Service	Source	Target	Requirement
Task_Set_Keeper	cur_thread	task_addr	FLUKE_PROT_WRITE
Task_Associate_Keeper	task_addr	port_addr	FLASK: TASK_KEEPER_PERM
Invocation Service	**Source**	**Target**	**Requirement**
fluke_task_get_state	cur_thread	task	FLUKE_PROT_READ
fluke_task_set_state	cur_thread	keeper_ref	FLUKE_PROT_READ

Table 4: Requirements on Task Services

THEORY *task_service_rqmts*

```
task_service_rqmnts  :  THEORY

  BEGIN

  IMPORTING  task_services

  st1,  st2  :  VAR  KERNEL_STATE
  cur_thread  :  VAR  SOB_ADDRESS
  task_addr  :  VAR  SOB_ADDRESS
  port_addr  :  VAR  SOB_ADDRESS

  Task_Set_Keeper_rqmts(st1,  st2,  cur_thread)  :  bool  =
    FORALL  task_addr  :
      Task_Set_Keeper(st1,  st2,  task_addr)
        IMPLIES
          memory_allows_sob(st1,  cur_thread,  task_addr,  FLUKE_PROT_WRITE)

  Task_Associate_Keeper_rqmts(st1,  st2)  :  bool  =
    FORALL  task_addr,  port_addr  :
```

```
    Task_Associate_Keeper(st1, st2, task_addr, port_addr)
      IMPLIES
        cache_allows(cache(st1),
                SECCLASS_SUBJECT, sid(sob_data(st2)(task_addr)),
                SECCLASS_SUBJECT, sid(sob_data(st2)(port_addr)),
                TASK_KEEPER_PERM)

  task_service_rqmts(st1, st2, cur_thread) : bool =
    Task_Set_Keeper_rqmts(st1, st2, cur_thread)
    AND Task_Associate_Keeper_rqmts(st1, st2)

  END task_service_rqmnts
```

3.2.5 Regions

3.2.5.1 Region Transformation Services The following transformation services are defined for region objects.

Transformation Service Definition *Region_Set_Simple_State(region_addr)* This service is provided if the simple state of an existing region object is modified during a transition, other than by destruction of the region object.

Transformation Service Definition *Region_Set_Task(region_addr)* This service is provided if the *task_link* from a region object is modified during a transition, other than by destruction of the region object or destruction of the object pointed at by the *task_link* in the initial state of the transition.

Transformation Service Definition *Region_Set_Keeper(region_addr)* This service is provided if the *keeper_link* from a region object is modified during a transition, other than by destruction of the region object or destruction of the object pointed at by the *keeper_link* in the initial state of the transition.

Transformation Service Definition *Region_Associate_Task(region_addr, task_addr)* This service is provided if a *task_link* is established between a region object and a task object.

Transformation Service Definition *Region_Associate_Keeper(region_addr, port_addr)* This service is provided if a *keeper_link* is established between a region object and a port object.

THEORY *region_services*

```
region_services : THEORY

  BEGIN

  IMPORTING kernel_state
  IMPORTING common_services

  st, st1, st2 : VAR KERNEL_STATE
  region_addr : VAR SOB_ADDRESS
  task_addr, port_addr : VAR SOB_ADDRESS


  Region_Set_Simple_State(st1, st2, region_addr) : bool =
    member(region_addr, existing_regions(st1)) AND member(region_addr, existing_regions(st2))
    AND (
         start_va(region_data(st1)(region_addr)) /= start_va(region_data(st2)(region_addr))
         OR size(region_data(st1)(region_addr)) /= size(region_data(st2)(region_addr))
         OR prot(region_data(st1)(region_addr)) /= prot(region_data(st2)(region_addr))
```

```
    )

Region_Set_Task(st1, st2, region_addr) : bool =
  member(region_addr, existing_regions(st1)) AND member(region_addr, existing_regions(st2))
  AND
  task_link(region_data(st1)(region_addr)) /= task_link(region_data(st2)(region_addr))
  AND
  NOT Object_Destroy(st1, st2, task_link(region_data(st1)(region_addr)))

Region_Set_Keeper(st1, st2, region_addr) : bool =
  member(region_addr, existing_regions(st1)) AND member(region_addr, existing_regions(st2))
  AND
  keeper_link(region_data(st1)(region_addr)) /= keeper_link(region_data(st2)(region_addr))
  AND
  NOT Object_Destroy(st1, st2, keeper_link(region_data(st1)(region_addr)))

region_task_linked(st, region_addr, task_addr) : bool =
  member(region_addr, existing_regions(st))
  AND member(task_addr, existing_tasks(st))
  AND task_link(region_data(st)(region_addr)) = task_addr

Region_Associate_Task(st1, st2, region_addr, task_addr) : bool =
  NOT region_task_linked(st1, region_addr, task_addr)
  AND region_task_linked(st2, region_addr, task_addr)

region_keeper_linked(st, region_addr, port_addr) : bool =
  member(region_addr, existing_regions(st))
  AND member(port_addr, existing_ports(st))
  AND keeper_link(region_data(st)(region_addr)) = port_addr

Region_Associate_Keeper(st1, st2, region_addr, port_addr) : bool =
  NOT region_keeper_linked(st1, region_addr, port_addr)
  AND region_keeper_linked(st2, region_addr, port_addr)

END region_services
```

3.2.5.2 Region Invocation Services

Invocation services are defined for the following region requests.

- `fluke_region_get_state(region)`
- `fluke_region_set_state(task_ref, keeper_ref)`
- `fluke_region_search(region)`

3.2.5.3 Region Requirements

Table 5 defines the requirements on the region services.

The *Region_Associate_Task* service requires SID equality because a region linked to a task can export any memory from the task with all of the privileges granted to that task.

The SID equality requirement for the *Region_Associate_Keeper* service is based more upon intended use. The memory manager which receives messages from the keeper port must have full access to the task from which the region exports memory. Since there is no known use for labeling this task any different than the memory manager, the keeper port, region and task are all expected to have the same SID as the memory manager.

Transformation Service	Source	Target	Requirement
Region_Set_Simple_State	cur_thread	region_addr	FLUKE_PROT_WRITE
Region_Set_Task	cur_thread	region_addr	FLUKE_PROT_WRITE
Region_Set_Keeper	cur_thread	region_addr	FLUKE_PROT_WRITE
Region_Associate_Task	—	—	FLASK: sid(region_addr) = sid(task_addr)
Region_Associate_Keeper	—	—	FLASK: sid(region_addr) = sid(port_addr)
Invocation Service	**Source**	**Target**	**Requirement**
fluke_region_get_state	cur_thread	region	FLUKE_PROT_READ
fluke_region_set_state	cur_thread	task_ref	FLUKE_PROT_READ
	cur_thread	keeper_ref	FLUKE_PROT_READ
fluke_region_search	cur_thread	region	FLUKE_PROT_READ
			Also need FLUKE_PROT_READ to all of the memory actually searched

Table 5: Requirements on Region Services

THEORY *region_service_rqmts*

```
region_service_rqmts : THEORY

  BEGIN

  IMPORTING region_services

  st1, st2 : VAR KERNEL_STATE
  cur_thread : VAR SOB_ADDRESS
  region_addr : VAR SOB_ADDRESS
  task_addr, port_addr : VAR SOB_ADDRESS

  Region_Set_Simple_State_rqmts(st1, st2, cur_thread) : bool =
    FORALL region_addr :
      Region_Set_Simple_State(st1, st2, region_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, region_addr, FLUKE_PROT_WRITE)

  Region_Set_Task_rqmts(st1, st2, cur_thread) : bool =
    FORALL region_addr :
      Region_Set_Task(st1, st2, region_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, region_addr, FLUKE_PROT_WRITE)

  Region_Set_Keeper_rqmts(st1, st2, cur_thread) : bool =
    FORALL region_addr :
      Region_Set_Keeper(st1, st2, region_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, region_addr, FLUKE_PROT_WRITE)

  Region_Associate_Task_rqmts(st1, st2) : bool =
    FORALL region_addr, task_addr :
      Region_Associate_Task(st1, st2, region_addr, task_addr)
        IMPLIES
          sid(sob_data(st2)(region_addr)) = sid(sob_data(st2)(task_addr))

  Region_Associate_Keeper_rqmts(st1, st2) : bool =
    FORALL region_addr, port_addr :
      Region_Associate_Keeper(st1, st2, region_addr, port_addr)
        IMPLIES
          sid(sob_data(st2)(region_addr)) = sid(sob_data(st2)(port_addr))
```

```
region_service_rqmts(st1, st2, cur_thread) : bool =
   Region_Set_Simple_State_rqmts(st1, st2, cur_thread)
   AND Region_Set_Task_rqmts(st1, st2, cur_thread)
   AND Region_Set_Keeper_rqmts(st1, st2, cur_thread)
   AND Region_Associate_Task_rqmts(st1, st2)
   AND Region_Associate_Keeper_rqmts(st1, st2)

END region_service_rqmts
```

3.2.5.4 Region Issues According to the Fluke API [2], `fluke_region_search` can cause "repairs" to objects with invalid state. This suggests that it would be more appropriate to require *FLUKE_PROT_WRITE* permission to execute that request.

3.2.6 Mappings

3.2.6.1 Mapping Transformation Services The following transformation services are defined for mapping objects.

Transformation Service Definition *Mapping_Set_Simple_State(mapping_addr)* This service is provided if the simple state of an existing mapping object is modified during a transition, other than by destruction of the mapping object.

Transformation Service Definition *Mapping_Set_Region(mapping_addr)* This service is provided if the *region_link* from a mapping object is modified during a transition, other than by destruction of the mapping object or destruction of the object pointed at by the *region_link* in the initial state of the transition.

Transformation Service Definition *Mapping_Set_Task(mapping_addr)* This service is provided if the *task_link* from a mapping object is modified during a transition, other than by destruction of the mapping object or destruction of the object pointed at by the *task_link* in the initial state of the transition.

Transformation Service Definition *Mapping_Associate_Region(mapping_addr, region_addr)* This service is provided if a *region_link* is established between a mapping object and a region object.

Transformation Service Definition *Mapping_Associate_Task(mapping_addr, task_addr)* This service is provided if a *task_link* is established between a mapping object and a task object.

THEORY *mapping_services*

```
mapping_services : THEORY

  BEGIN

  IMPORTING kernel_state
  IMPORTING common_services

  st, st1, st2 : VAR KERNEL_STATE
  mapping_addr : VAR SOB_ADDRESS
  region_addr, task_addr : VAR SOB_ADDRESS
```

```
Mapping_Set_Simple_State(st1, st2, mapping_addr) : bool =
  member(mapping_addr, existing_mappings(st1)) AND member(mapping_addr, existing_mappings(st2))
  AND (
       region_off(mapping_data(st1)(mapping_addr)) /= region_off(mapping_data(st2)(mapping_addr))
       OR start_va(mapping_data(st1)(mapping_addr)) /= start_va(mapping_data(st2)(mapping_addr))
       OR size(mapping_data(st1)(mapping_addr)) /= size(mapping_data(st2)(mapping_addr))
       OR prot(mapping_data(st1)(mapping_addr)) /= prot(mapping_data(st2)(mapping_addr))
      )


Mapping_Set_Region(st1, st2, mapping_addr) : bool =
  member(mapping_addr, existing_mappings(st1)) AND member(mapping_addr, existing_mappings(st2))
  AND
  region_link(mapping_data(st1)(mapping_addr)) /= region_link(mapping_data(st2)(mapping_addr))
  AND
  NOT Object_Destroy(st1, st2, region_link(mapping_data(st1)(mapping_addr)))


Mapping_Set_Task(st1, st2, mapping_addr) : bool =
  member(mapping_addr, existing_mappings(st1)) AND member(mapping_addr, existing_mappings(st2))
  AND
  task_link(mapping_data(st1)(mapping_addr)) /= task_link(mapping_data(st2)(mapping_addr))
  AND
  NOT Object_Destroy(st1, st2, task_link(mapping_data(st1)(mapping_addr)))


mapping_region_linked(st, mapping_addr, region_addr) : bool =
  member(mapping_addr, existing_mappings(st))
  AND member(region_addr, existing_regions(st))
  AND region_link(mapping_data(st)(mapping_addr)) = region_addr

Mapping_Associate_Region(st1, st2, mapping_addr, region_addr) : bool =
  NOT mapping_region_linked(st1, mapping_addr, region_addr)
  AND mapping_region_linked(st2, mapping_addr, region_addr)


mapping_task_linked(st, mapping_addr, task_addr) : bool =
  member(mapping_addr, existing_mappings(st))
  AND member(task_addr, existing_tasks(st))
  AND task_link(mapping_data(st)(mapping_addr)) = task_addr

Mapping_Associate_Task(st1, st2, mapping_addr, task_addr) : bool =
  NOT mapping_task_linked(st1, mapping_addr, task_addr)
  AND mapping_task_linked(st2, mapping_addr, task_addr)

END mapping_services
```

3.2.6.2 Mapping Invocation Services

Invocation services are defined for the following mapping requests.

- `fluke_mapping_get_state(mapping)`
- `fluke_mapping_set_state(region_ref, task_ref)`

3.2.6.3 Mapping Requirements

Table 6 defines the requirements on the mapping services.

The *Mapping_Associate_Task* service requires SID equality because a mapping linked to a task can import arbitrary memory into the task.

THEORY *mapping_service_rqmts*

Transformation Service	Source	Target	Requirement
Mapping_Set_Simple_State	cur_thread	mapping_addr	FLUKE_PROT_WRITE
Mapping_Set_Region	cur_thread	mapping_addr	FLUKE_PROT_WRITE
Mapping_Set_Task	cur_thread	mapping_addr	FLUKE_PROT_WRITE
Mapping_Associate_Region	mapping_addr	region_addr	FLASK: MAP_PERM
Mapping_Associate_Task	—	—	FLASK: sid(mapping_addr) = sid(task_addr)
Invocation Service	**Source**	**Target**	**Requirement**
fluke_mapping_get_state	cur_thread	mapping	FLUKE_PROT_READ
fluke_mapping_set_state	cur_thread	region_ref	FLUKE_PROT_READ
	cur_thread	task_ref	FLUKE_PROT_READ

Table 6: Requirements on Mapping Services

```
mapping_service_rqmts : THEORY

  BEGIN

  IMPORTING mapping_services

  st1, st2 : VAR KERNEL_STATE
  cur_thread : VAR SOB_ADDRESS
  mapping_addr : VAR SOB_ADDRESS
  region_addr, task_addr : VAR SOB_ADDRESS

  Mapping_Set_Simple_State_rqmts(st1, st2, cur_thread) : bool =
    FORALL mapping_addr :
      Mapping_Set_Simple_State(st1, st2, mapping_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, mapping_addr, FLUKE_PROT_WRITE)

  Mapping_Set_Region_rqmts(st1, st2, cur_thread) : bool =
    FORALL mapping_addr :
      Mapping_Set_Region(st1, st2, mapping_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, mapping_addr, FLUKE_PROT_WRITE)

  Mapping_Set_Task_rqmts(st1, st2, cur_thread) : bool =
    FORALL mapping_addr :
      Mapping_Set_Task(st1, st2, mapping_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, mapping_addr, FLUKE_PROT_WRITE)

  Mapping_Associate_Region_rqmts(st1, st2) : bool =
    FORALL mapping_addr, region_addr :
      Mapping_Associate_Region(st1, st2, mapping_addr, region_addr)
        IMPLIES
          cache_allows(cache(st1),
                   SECCLASS_SUBJECT, sid(sob_data(st2)(mapping_addr)),
                   SECCLASS_SUBJECT, sid(sob_data(st2)(region_addr)),
                   MAP_PERM)

  Mapping_Associate_Task_rqmts(st1, st2) : bool =
    FORALL mapping_addr, task_addr :
      Mapping_Associate_Task(st1, st2, mapping_addr, task_addr)
        IMPLIES
          sid(sob_data(st2)(mapping_addr)) = sid(sob_data(st2)(task_addr))

  mapping_service_rqmts(st1, st2, cur_thread) : bool =
    Mapping_Set_Simple_State_rqmts(st1, st2, cur_thread)
```

```
    AND Mapping_Set_Region_rqmts(st1, st2, cur_thread)
    AND Mapping_Set_Task_rqmts(st1, st2, cur_thread)
    AND Mapping_Associate_Region_rqmts(st1, st2)
    AND Mapping_Associate_Task_rqmts(st1, st2)

  END mapping_service_rqmts
```

3.2.7 Ports

3.2.7.1 Port Transformation Services The following transformation services are defined for port objects.

Transformation Service Definition *Port_Set_Simple_State(port_addr)* This service is provided if the simple state of an existing port object is modified during a transition, other than by destruction of the port object.

Transformation Service Definition *Port_Set_Pset(port_addr)* This service is provided if the *pset_link* from a port object is modified during a transition, other than by destruction of the port object or destruction of the object pointed at by the *pset_link* in the initial state of the transition.

Transformation Service Definition *Port_Associate_Pset(port_addr, pset_addr)* This service is provided if a *pset_link* is established between a port object and a pset object.

THEORY *port_services*

```
port_services : THEORY

  BEGIN

  IMPORTING kernel_state
  IMPORTING common_services

  st, st1, st2 : VAR KERNEL_STATE
  port_addr : VAR SOB_ADDRESS
  pset_addr : VAR SOB_ADDRESS


  Port_Set_Simple_State(st1, st2, port_addr) : bool =
    member(port_addr, existing_ports(st1)) AND member(port_addr, existing_ports(st2))
    AND
    alias(port_data(st1)(port_addr)) /= alias(port_data(st2)(port_addr))


  Port_Set_Pset(st1, st2, port_addr) : bool =
    member(port_addr, existing_ports(st1)) AND member(port_addr, existing_ports(st2))
    AND
    pset_link(port_data(st1)(port_addr)) /= pset_link(port_data(st2)(port_addr))
    AND
    NOT Object_Destroy(st1, st2, pset_link(port_data(st1)(port_addr)))


  port_pset_linked(st, port_addr, pset_addr) : bool =
    member(port_addr, existing_ports(st))
    AND member(pset_addr, existing_psets(st))
    AND pset_link(port_data(st)(port_addr)) = pset_addr

  Port_Associate_Pset(st1, st2, port_addr, pset_addr) : bool =
```

```
    NOT port_pset_linked(st1, port_addr, pset_addr)
    AND port_pset_linked(st2, port_addr, pset_addr)

  END port_services
```

3.2.7.2 Port Invocation Services

Invocation services are defined for the following port requests.

- `fluke_port_get_state(port)`
- `fluke_port_set_state(pset_ref)`

3.2.7.3 Port Requirements

Table 7 defines the requirements on the port services.

Transformation Service	Source	Target	Requirement
Port_Set_Simple_State	cur_thread	port_addr	FLUKE_PROT_WRITE
Port_Set_Pset	cur_thread	port_addr	FLUKE_PROT_WRITE
Port_Associate_Pset	—	—	FLASK: sid(port_addr) = sid(pset_addr)
Invocation Service	**Source**	**Target**	**Requirement**
fluke_port_get_state	cur_thread	port	FLUKE_PROT_READ
fluke_port_set_state	cur_thread	pset_ref	FLUKE_PROT_READ

Table 7: Requirements on Port Services

The *Port_Associate_Pset* service requires SID equality because the port and pset both represent the server which receives requests on the port.

THEORY *port_service_rqmts*

```
port_service_rqmts : THEORY

  BEGIN

  IMPORTING port_services

  st1, st2 : VAR KERNEL_STATE
  cur_thread : VAR SOB_ADDRESS
  port_addr : VAR SOB_ADDRESS
  pset_addr : VAR SOB_ADDRESS

  Port_Set_Simple_State_rqmts(st1, st2, cur_thread) : bool =
    FORALL port_addr :
      Port_Set_Simple_State(st1, st2, port_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, port_addr, FLUKE_PROT_WRITE)

  Port_Set_Pset_rqmts(st1, st2, cur_thread) : bool =
    FORALL port_addr :
      Port_Set_Pset(st1, st2, port_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, port_addr, FLUKE_PROT_WRITE)

  Port_Associate_Pset_rqmts(st1, st2) : bool =
    FORALL port_addr, pset_addr :
      Port_Associate_Pset(st1, st2, port_addr, pset_addr)
```

```
    IMPLIES
      sid(sob_data(st2)(port_addr)) = sid(sob_data(st2)(pset_addr))

port_service_rqmts(st1, st2, cur_thread) : bool =
  Port_Set_Simple_State_rqmts(st1, st2, cur_thread)
  AND Port_Set_Pset_rqmts(st1, st2, cur_thread)
  AND Port_Associate_Pset_rqmts(st1, st2)

END port_service_rqmts
```

3.2.8 Port Sets

3.2.8.1 Port Set Transformation Services Since port sets do not have any state associated with them (beyond the common object state), there are no transformation services defined for port sets.

3.2.8.2 Port Set Invocation Services Invocation services are defined for the following port set requests.

- `fluke_pset_get_state(pset)`

3.2.8.3 Port Set Requirements Table 8 defines the requirements on the port set services.

Invocation Service	Source	Target	Requirement
fluke_pset_get_state	cur_thread	pset	FLUKE_PROT_READ

Table 8: Requirements on Port Set Services

3.2.9 Mutexes

3.2.9.1 Mutex Transformation Services The following transformation services are defined for mutex objects.

Transformation Service Definition *Mutex_Set_Simple_State(mutex_addr)* This service is provided if the simple state of an existing mutex object is modified during a transition, other than by destruction of the mutex object.

Transformation Service Definition *Mutex_Set_Owner(mutex_addr)* This service is provided if the *owner_link* from a mutex object is modified during a transition, other than by destruction of the mutex object or destruction of the object pointed at by the *owner_link* in the initial state of the transition.

Unlike most other links, there is no service defined to control the value of the *owner_link*. The only function of this value is to record which thread can release the lock on the mutex. If it is set to link to an unexpected thread, then the only impact is that the mutex will never be released. However, any thread with permission to set the value of *owner_link* could accomplish this by simply locking the mutex itself and never releasing it.

By not defining any requirements on the value of *owner_link*, we also leave open the possibility of a user-space implementation of mutex objects since all other requirements on mutex services are Fluke memory access requirements.

THEORY *mutex_services*

```
mutex_services : THEORY

  BEGIN

  IMPORTING kernel_state
  IMPORTING common_services

  st1, st2 : VAR KERNEL_STATE
  mutex_addr : VAR SOB_ADDRESS

  Mutex_Set_Simple_State(st1, st2, mutex_addr) : bool =
    member(mutex_addr, existing_mutexes(st1)) AND member(mutex_addr, existing_mutexes(st2))
    AND (
         flags(mutex_data(st1)(mutex_addr)) /= flags(mutex_data(st2)(mutex_addr))
         OR locked(mutex_data(st1)(mutex_addr)) /= locked(mutex_data(st2)(mutex_addr))
        )

  Mutex_Set_Owner(st1, st2, mutex_addr) : bool =
    member(mutex_addr, existing_mutexes(st1)) AND member(mutex_addr, existing_mutexes(st2))
    AND
    owner_link(mutex_data(st1)(mutex_addr)) /= owner_link(mutex_data(st2)(mutex_addr))
    AND
    NOT Object_Destroy(st1, st2, owner_link(mutex_data(st1)(mutex_addr)))

  END mutex_services
```

3.2.9.2 Mutex Invocation Services

Invocation services are defined for the following mutex requests.

- `fluke_mutex_get_state(mutex)`
- `fluke_mutex_set_state(owner_ref)`
- `fluke_mutex_trylock(mutex)`

3.2.9.3 Mutex Requirements

Table 9 defines the requirements on the mutex services.

Transformation Service	Source	Target	Requirement
Mutex_Set_Simple_State	cur_thread	mutex_addr	FLUKE_PROT_WRITE
Mutex_Set_Owner	cur_thread	mutex_addr	FLUKE_PROT_WRITE
Invocation Service	**Source**	**Target**	**Requirement**
fluke_mutex_get_state	cur_thread	mutex	FLUKE_PROT_READ
fluke_mutex_set_state	cur_thread	owner_ref	FLUKE_PROT_READ
fluke_mutex_trylock	cur_thread	mutex	FLUKE_PROT_WRITE

Table 9: Requirements on Mutex Services

THEORY *mutex_service_rqmts*

```
mutex_service_rqmts : THEORY

  BEGIN
```

```
IMPORTING mutex_services

st1, st2 : VAR KERNEL_STATE
cur_thread : VAR SOB_ADDRESS
mutex_addr : VAR SOB_ADDRESS

Mutex_Set_Simple_State_rqmts(st1, st2, cur_thread) : bool =
  FORALL mutex_addr :
    Mutex_Set_Simple_State(st1, st2, mutex_addr)
      IMPLIES
        memory_allows_sob(st1, cur_thread, mutex_addr, FLUKE_PROT_WRITE)

Mutex_Set_Owner_rqmts(st1, st2, cur_thread) : bool =
  FORALL mutex_addr :
    Mutex_Set_Owner(st1, st2, mutex_addr)
      IMPLIES
        memory_allows_sob(st1, cur_thread, mutex_addr, FLUKE_PROT_WRITE)

mutex_service_rqmts(st1, st2, cur_thread) : bool =
  Mutex_Set_Simple_State_rqmts(st1, st2, cur_thread)
  AND Mutex_Set_Owner_rqmts(st1, st2, cur_thread)

END mutex_service_rqmts
```

3.2.10 Condition Variables

3.2.10.1 Condition Variable Transformation Services The following transformation services are defined for condition variable objects.

Transformation Service Definition *Cond_Set_Simple_State(cond_addr)* This service is provided if the simple state of an existing cond_var object is modified during a transition, other than by destruction of the cond_var object.

Transformation Service Definition *Cond_Set_Inheritor(cond_addr)* This service is provided if the *inheritor_link* from a cond_var object is modified during a transition, other than by destruction of the cond_var object or destruction of the object pointed at by the *inheritor_link* in the initial state of the transition.

Unlike most other links, there is no service defined to control the value of the *inheritor_link*. While the value of this link is significant when inheritance scheduling is in place, condition variables are intended to be used among fully cooperating threads.

By not defining any requirements on the value of *inheritor_link*, we also leave open the possibility of a user-space implementation of condition variable objects since all other requirements on condition variable services are Fluke memory access requirements.

We define a service to control changes to the set of *waiting_threads* analogous to the *Object_Set_Link*.

Transformation Service Definition *Cond_Set_Waiting_Threads(cond_addr)* This service is provided if the set of *waiting_threads* for a condition variable object is modified during a transition, other than by destruction of the condition variable object or by removal of destroyed threads.

Editorial Note:
The PVS for the Cond_Set_Waiting_Threads does not model this service exactly since it does not factor out the side effect of removing destroyed threads.

THEORY *cond_var_services*

```
cond_var_services : THEORY

  BEGIN

  IMPORTING kernel_state
  IMPORTING common_services

  st, st1, st2 : VAR KERNEL_STATE
  cond_addr : VAR SOB_ADDRESS
  thread_addr : VAR SOB_ADDRESS


  Cond_Set_Simple_State(st1, st2, cond_addr) : bool =
    member(cond_addr, existing_cond_vars(st1)) AND member(cond_addr, existing_cond_vars(st2))
    AND
    flags(cond_data(st1)(cond_addr)) /= flags(cond_data(st2)(cond_addr))


  Cond_Set_Inheritor(st1, st2, cond_addr) : bool =
    member(cond_addr, existing_cond_vars(st1)) AND member(cond_addr, existing_cond_vars(st2))
    AND
    inheritor_link(cond_data(st1)(cond_addr)) /= inheritor_link(cond_data(st2)(cond_addr))
    AND
    NOT Object_Destroy(st1, st2, inheritor_link(cond_data(st1)(cond_addr)))


  Cond_Set_Waiting_Threads(st1, st2, cond_addr) : bool =
    member(cond_addr, existing_cond_vars(st1)) AND member(cond_addr, existing_cond_vars(st2))
    AND
    waiting_threads(cond_data(st1)(cond_addr)) /= waiting_threads(cond_data(st2)(cond_addr))

  END cond_var_services
```

3.2.10.2 Condition Variable Invocation Services

Invocation services are defined for the following condition variable requests.

- `fluke_cond_get_state(cond)`
- `fluke_cond_set_state(inheritor_ref)`

3.2.10.3 Condition Variable Requirements

Table 10 defines the requirements on the condition variable services.

Transformation Service	Source	Target	Requirement
Cond_Set_Simple_State	cur_thread	cond_addr	FLUKE_PROT_WRITE
Cond_Set_Inheritor	cur_thread	cond_addr	FLUKE_PROT_WRITE
Cond_Set_Waiting_Threads	cur_thread	cond_addr	FLUKE_PROT_WRITE
Invocation Service	**Source**	**Target**	**Requirement**
fluke_cond_get_state	cur_thread	task	FLUKE_PROT_READ
fluke_cond_set_state	cur_thread	inheritor_ref	FLUKE_PROT_READ

Table 10: Requirements on Condition Variable Services

THEORY *cond_var_service_rqmts*

```
cond_var_service_rqmts : THEORY

  BEGIN

  IMPORTING cond_var_services

  st1, st2 : VAR KERNEL_STATE
  cur_thread : VAR SOB_ADDRESS
  cond_addr : VAR SOB_ADDRESS
  thread_addr : VAR SOB_ADDRESS

  Cond_Set_Simple_State_rqmts(st1, st2, cur_thread) : bool =
    FORALL cond_addr :
      Cond_Set_Simple_State(st1, st2, cond_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, cond_addr, FLUKE_PROT_WRITE)

  Cond_Set_Inheritor_rqmts(st1, st2, cur_thread) : bool =
    FORALL cond_addr :
      Cond_Set_Inheritor(st1, st2, cond_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, cond_addr, FLUKE_PROT_WRITE)

  Cond_Set_Waiting_Threads_rqmts(st1, st2, cur_thread) : bool =
    FORALL cond_addr :
      Cond_Set_Waiting_Threads(st1, st2, cond_addr)
        IMPLIES
          memory_allows_sob(st1, cur_thread, cond_addr, FLUKE_PROT_WRITE)

  cond_service_rqmts(st1, st2, cur_thread) : bool =
    Cond_Set_Simple_State_rqmts(st1, st2, cur_thread)
    AND Cond_Set_Inheritor_rqmts(st1, st2, cur_thread)
    AND Cond_Set_Waiting_Threads_rqmts(st1, st2, cur_thread)

  END cond_var_service_rqmts
```

3.2.11 References

3.2.11.1 Reference Transformation Services

The following transformation service is defined for reference objects.

Transformation Service Definition *Ref_Set_Link(ref_addr)* This service is provided if the *object_link* from a reference object is modified during a transition, other than by destruction of the reference object or destruction of the object pointed at by the *object_link* in the initial state of the transition.

THEORY *ref_services*

```
ref_services : THEORY

  BEGIN

  IMPORTING kernel_state
  IMPORTING common_services
```

```
st1, st2 : VAR KERNEL_STATE
ref_addr : VAR SOB_ADDRESS

Ref_Set_Link(st1, st2, ref_addr) : bool =
  member(ref_addr, existing_refs(st1)) AND member(ref_addr, existing_refs(st2))
  AND
  object_link(ref_data(st1)(ref_addr)) /= object_link(ref_data(st2)(ref_addr))
  AND
  NOT Object_Destroy(st1, st2, object_link(ref_data(st1)(ref_addr)))

END ref_services
```

3.2.11.2 Reference Invocation Services

Invocation services are defined for the following requests.

- `fluke_ref_check(ref)`
- `fluke_ref_compare(ref1, ref2)`
- `fluke_ref_copy(src_ref)`
- `fluke_ref_hash(ref)`
- `fluke_ref_sid(ref)`
- `fluke_ref_type(ref)`
- `fluke_`*`object`*`_reference(object, new_object_ref)`[24]

3.2.11.3 Reference Requirements

Table 11 defines the requirements on the reference services.

Transformation Service	Source	Target	Requirement
Ref_Set_Link	cur_thread	ref_addr	FLUKE_PROT_WRITE
Invocation Service	**Source**	**Target**	**Requirement**
fluke_ref_check	cur_thread	ref	FLUKE_PROT_READ
fluke_ref_compare	cur_thread	ref1	FLUKE_PROT_READ
	cur_thread	ref2	FLUKE_PROT_READ
fluke_ref_copy	cur_thread	src_ref	FLUKE_PROT_READ
fluke_ref_hash	cur_thread	ref	FLUKE_PROT_READ
fluke_ref_sid	cur_thread	ref	FLUKE_PROT_READ
fluke_ref_type	cur_thread	ref	FLUKE_PROT_READ
fluke_object_reference	cur_thread	object	FLUKE_PROT_WRITE

Table 11: Requirements on Reference Services

THEORY *ref_service_rqmts*

```
ref_service_rqmts : THEORY

  BEGIN

  IMPORTING ref_services

  st1, st2 : VAR KERNEL_STATE
  cur_thread : VAR SOB_ADDRESS
```

[24] There are requests of this form for each of the Fluke object types.

```
ref_addr : VAR SOB_ADDRESS

Ref_Set_Link_rqmts(st1, st2, cur_thread) : bool =
  FORALL ref_addr :
    Ref_Set_Link(st1, st2, ref_addr)
      IMPLIES
        memory_allows_sob(st1, cur_thread, ref_addr, FLUKE_PROT_WRITE)

ref_service_rqmts(st1, st2, cur_thread) : bool =
  Ref_Set_Link_rqmts(st1, st2, cur_thread)

END ref_service_rqmts
```

3.2.11.4 Reference Issues

- The requirement on *Ref_Set_Link* is not met during transfer of references during IPC operations. Since the transfer occurs while the sender is the subject, the requirements indicate that the sender must have WRITE permission to any reference objects in the recipients `ipc_params` structure. But it is actually the recipient which should have WRITE permission.

  It is unclear how to state the requirement to match this without introducing a special case for IPC.

- The lack of any direct Flask controls between a client of a `fluke_ref_hash`, `fluke_ref_sid` or `fluke_ref_type` call and the actual object whose state is returned (not the reference object) violates the general principles of Flask, and makes it difficult to argue that any particular client cannot view these elements of the state of an object. In particular, this leaves open a possible covert channel, though we believe that it is likely to be insignificant since the values returned by these calls cannot be changed after an object is created.

3.2.12 Security

3.2.12.1 Security Transformation Services

The following security transformation services are defined.

Transformation Service Definition *Increase_Cache_Allows* This service is provided by a transition if a permission is not allowed by the cache from source s to target t in the initial state of the transition, but is allowed in the final state.

We do not control the removal of a permission from the cache because this can be a side effect of any operation (if the cache has limited size and old values are removed). There is certainly a potential covert channel here but we do not believe that it is significant.

THEORY *security_services*

```
security_services : THEORY

  BEGIN

  IMPORTING kernel_state

  st1, st2 : VAR KERNEL_STATE
```

```
sclass, tclass : VAR SECURITY_CLASS
ssid, tsid : VAR SID
perm : VAR PERMISSION

Increase_Cache_Allows(st1, st2) : bool =
  EXISTS sclass, ssid, tclass, tsid, perm :
    NOT cache_allows(cache(st1), sclass, ssid, tclass, tsid, perm)
    AND cache_allows(cache(st2), sclass, ssid, tclass, tsid, perm)

END security_services
```

3.2.12.2 Security Invocation Services

There are no security invocation services.

3.2.12.3 Requirements

No requirements are stated at this time.

In the implementation when the *Increase_Cache_Allows* service is performed via the kernel server interface, *ADD_PERM* is checked. However, when a kernel client operating in the kernel sends a request to the security server, it uses idempotent IPC. The security server calls **ipc_reply** to send the response. While operating in kernel mode to send the reply the security server updates the cache. Since this mechanism relies upon the security server to ensure that no other thread can receive and then reply to security requests, this will complicate the assurance argument.

> *Editorial Note:*
> There should also be initial state requirements for the contents of the cache, even though the requirements relating to changes in the cache would control only which threads can change the cache, not the actual permissions which are added.

3.2.13 Memory

3.2.13.1 Memory Transformation Services

There are five elements in the memory model. We do not define any services to directly control the value of *paged_in* or *phys_addr*, since these are primarily determined by external memory managers and are indirectly controlled through the mapping and region services.

Memory protection rights can be taken away in many ways through manipulation of mappings and regions. Since this is only a concern for denial of service attacks, we do not define a separate service. However, granting additional rights is controlled directly by Flask as well as indirectly through the Fluke mechanisms.

The following service is defined in terms of both the *mem_prot* and *phys_addr* functions. This is appropriate because the protections are actually defined from a particular virtual address in a particular task to a particular location in physical memory. This definition is also essential to meeting the fundamental requirement as is discussed below in Section 3.2.13.3.

Transformation Service Definition *Memory_Add_Protection(task_addr, va, prot)* This service is provided by a transition if in the final state, a memory protection *prot* is granted for a virtual address *va* within a task *task_addr*, while this does not hold in the initial state of the transition.

We define separate services for controlling each of the remaining two elements of the memory model.

Transformation Service Definition *Memory_Set_Sid(pa)* This service is provided if the SID associated with a physical address is modified.

Transformation Service Definition *Memory_Write(pa)* This service is provided if the contents at a physical address are modified.

THEORY *memory_services*

```
memory_services : THEORY

  BEGIN

  IMPORTING kernel_state

  st, st1, st2 : VAR KERNEL_STATE
  prot : VAR MEMORY_PROTECTION
  task_addr : VAR SOB_ADDRESS
  va : VAR VIRTUAL_ADDRESS
  pa : VAR PHYSICAL_ADDRESS


  memory_protection_allows(st, task_addr, va, prot) : bool =
    member(task_addr, existing_tasks(st))
    AND member((task_addr, va), paged_in(st))
    AND member(prot, mem_prot(st)(task_addr, va))

  Memory_Add_Protection(st1, st2, task_addr, va, prot): bool =
    NOT memory_protection_allows(st1, task_addr, va, prot)
    AND memory_protection_allows(st2, task_addr, va, prot)


  Memory_Set_Sid(st1, st2, pa) : bool =
    phys_addr_sid(st1)(pa) /= phys_addr_sid(st2)(pa)


  Memory_Write(st1, st2, pa) : bool =
    contents(st1)(pa) /= contents(st2)(pa)


  END memory_services
```

3.2.13.2 Memory Invocation Services

Invocation services are defined to represent reading and executing at a virtual memory location.

- `read_op(va)`
- `exec_op(va)`

3.2.13.3 Memory Requirements

Table 12 defines the requirements on the memory services.

The first requirement on the service *Memory_Set_Sid* is necessary to satisfy the state invariant that the SID of a kernel object is always equal to the SID of the physical address where its user component is located. See Section 3.2.2.4 for further discussion.

The second requirement on the service *Memory_Set_Sid* is provided for a similar reason. It ensures that memory protections aren't propagated if the SID of a memory location is changed.

Transformation Service	**Source**	**Target**	**Requirement**
Memory_Add_Protection	task_addr	va	FLASK: prot_to_perm(prot)
Memory_Set_Sid	—	—	FLASK: No existing objects have this physical address
	—	—	FLASK: Nothing is paged in at this physical address
Memory_Write	cur_thread	pa	FLUKE_PROT_WRITE
Invocation Service	**Source**	**Target**	**Requirement**
read_op	cur_thread	va	FLUKE_PROT_READ
exec_op	cur_thread	va	FLUKE_PROT_EXEC

Table 12: Requirements on Memory Services

It would seem natural to place some other kind of requirement on the SID of physical memory, for instance a relationship between the SID of the memory location and the SID of any task that maps a virtual address to that location. This is unnecessary, since the service *Memory_Add_Protection* controls the access that a task has to any mapped memory.

THEORY *memory_service_rqmts*

```
memory_service_rqmts : THEORY

  BEGIN

  IMPORTING memory_services

  st1, st2 : VAR KERNEL_STATE
  cur_thread : VAR SOB_ADDRESS
  prot : VAR MEMORY_PROTECTION
  task_addr : VAR SOB_ADDRESS
  va : VAR VIRTUAL_ADDRESS
  pa : VAR PHYSICAL_ADDRESS

  Memory_Add_Protection_rqmts(st1, st2) : bool =
    FORALL prot, task_addr, va :
      Memory_Add_Protection(st1, st2, task_addr, va, prot)
        IMPLIES
          cache_allows(cache(st1),
                SECCLASS_SUBJECT, sid(sob_data(st2)(task_addr)),
                SECCLASS_SUBJECT, phys_addr_sid(st2)(phys_addr(st2)(task_addr, va)),
                prot_to_perm(prot))

  Memory_Set_Sid_rqmts(st1, st2) : bool =
    FORALL pa :
      Memory_Set_Sid(st1, st2, pa)
        IMPLIES (
          FORALL (sob_addr : (existing_objects(st1))) :
            pa /= u_pa(sob_data(st1)(sob_addr))
          AND
          FORALL (taskva : (paged_in(st1))) :
            pa /= phys_addr(st1)(taskva)
        )

  Memory_Write_rqmts(st1, st2, cur_thread) : bool =
    FORALL pa :
      Memory_Write(st1, st2, pa)
        IMPLIES
```

```
      memory_allows_pa(st1, cur_thread, pa, FLUKE_PROT_WRITE)

  memory_service_rqmts(st1, st2, cur_thread) : bool =
    Memory_Add_Protection_rqmts(st1, st2)
    AND Memory_Set_Sid_rqmts(st1, st2)
    AND Memory_Write_rqmts(st1, st2, cur_thread)

  END memory_service_rqmts
```

3.2.13.4 Memory Issues

- The requirements on the *Memory_Add_Protection* service do not incorporate any of the Fluke requirements, i.e., there is no mention of the mapping object or any ancestor region objects. It would be more satisfying to be able to include these "discretionary" controls in the policy requirements as well.

3.2.14 IPC

3.2.14.1 IPC Transformation Services We define a single IPC service, the establishment of a reliable IPC connection. We do not define services for one-way or idempotent IPC since (as discussed in Section 3.1.13) there are no elements of the model relating to these forms of IPC.

Transformation Service Definition *IPC_Add_Connection(client, server)* This service is provided if a reliable IPC connection is established between two threads.

This service is similar to an *Object_Associate_Link* service. It is provided by a transition if in the final state of the transition, there are existing threads *client* and *server* such that (*client*, *server*) is in the set of *active_connections*, when this was not true in the initial state of the transition.

THEORY *ipc_services*

```
ipc_services : THEORY

  BEGIN

  IMPORTING kernel_state

  st, st1, st2 : VAR KERNEL_STATE
  client, server : VAR SOB_ADDRESS

  connected(st, client, server) : bool =
    member(client, existing_threads(st))
    AND member(server, existing_threads(st))
    AND member((client, server), active_connections(st))

  IPC_Add_Connection(st1, st2, client, server) : bool =
    NOT connected(st1, client, server)
    AND connected(st2, client, server)

  END ipc_services
```

3.2.14.2 IPC Invocation Services

Invocation services are defined for the following IPC requests.

This list of requests is not complete, however, all other requests are ultimately a variation or combination of these.

- `fluke_ipc_send(destination)`
- `fluke_ipc_send_secure(destination, client_sid)`
- `fluke_ipc_call(destination)`
- `fluke_ipc_call_secure(destination, client_sid)`
- `fluke_ipc_client_connect_send(destination)`
- `fluke_ipc_client_connect_send_secure(destination)`
- `fluke_ipc_wait_receive(pset)`
- `fluke_ipc_wait_receive_secure(pset)`

3.2.14.3 IPC Requirements

Table 13 defines the requirements on the IPC services.

In the table, `port_addr` is the port object to which the destination reference is linked.

These requirements and the requirements on the services *Thread_Associate_Effective_Client* and *Thread_Associate_Effective_Server* comprise the IPC control policy described in Section 2.2.2.3.

Transformation Service	Source	Target	Requirement
IPC_Add_Connection	—	—	client = client_link(server)
	—	—	server = server_link(client)
	eff_client_sid	eff_server_sid	FLASK: CONNECT_PERM
	FLASK: eff_client_sid(client) = client_eff_client_sid(server)		
	FLASK: eff_server_sid(server) = server_eff_server_sid(client)		
Invocation Service	**Source**	**Target**	**Requirement**
fluke_ipc_send	cur_thread	destination	FLUKE_PROT_READ
	cur_thread	port_addr	FLASK: SEND_PERM
fluke_ipc_send_secure	cur_thread	destination	FLUKE_PROT_READ
	cur_thread	client_sid	FLASK: SPECIFY_CLIENT_PERM
	client_sid	port_addr	FLASK: SEND_PERM
fluke_ipc_call	cur_thread	destination	FLUKE_PROT_READ
	cur_thread	port_addr	FLASK: CALL_PERM
fluke_ipc_call_secure	cur_thread	destination	FLUKE_PROT_READ
	cur_thread	client_sid	FLASK: SPECIFY_CLIENT_PERM
	client_sid	port_addr	FLASK: CALL_PERM
fluke_ipc_client_connect_send(_secure)	cur_thread	destination	FLUKE_PROT_READ
fluke_ipc_wait_receive(_secure)	cur_thread	pset	FLUKE_PROT_READ

Table 13: Requirements on IPC Services

THEORY *ipc_service_rqmts*

ipc_service_rqmts : **THEORY**

BEGIN

IMPORTING *ipc_services*

```
st1, st2 : VAR KERNEL_STATE
client, server : VAR SOB_ADDRESS

IPC_Add_Connection_rqmts(st1, st2) : bool =
  FORALL client, server :
    IPC_Add_Connection(st1, st2, client, server)
      IMPLIES (
        client = client_link(thread_data(st2)(server))
        AND
        server = server_link(thread_data(st2)(client))
        AND
        cache_allows(cache(st1),
              SECCLASS_SUBJECT, effective_client_sid(thread_data(st2)(client)),
              SECCLASS_SUBJECT, effective_server_sid(thread_data(st2)(server)),
              CONNECT_PERM)
        AND
        effective_client_sid(thread_data(st2)(client))
          = client_effective_client_sid(thread_data(st2)(server))
        AND
        effective_server_sid(thread_data(st2)(server))
          = server_effective_server_sid(thread_data(st2)(client))
      )

ipc_service_rqmts(st1, st2) : bool =
  IPC_Add_Connection_rqmts(st1, st2)

END ipc_service_rqmts
```

3.2.14.4 IPC Issues

- The requirements in Table 13 do not include any requirements on transferring the contents of a message.

  The table should include requirements that the sender have READ permission to all memory and references being sent and the recipient have WRITE permission to the locations where these are copied. Note that the API doesn't specify which side of an IPC connection is the "subject" when data is transferred, so we can not just rely upon the generic requirements to handle these requirements.

3.2.15 Outcalls

There are three situations in which the kernel initiates IPC

- Page faults
- User exceptions
- Security faults

There is no need for explicit permission checking on these outcalls. The destination for page faults and user exceptions are controlled by controlling the *keeper_link* for regions and tasks. The destination port for security faults is initialized during system startup and cannot be changed.[25]

[25] Note also that there is currently a kernel request which returns the value of the security port. We have not included that within the model because this may change if a suitable extension policy model is found and because there aren't any significant security ramifications from a request which returns a fixed value.

3.2.16 Request Invocation

3.2.16.1 Request Invocation Services Invocation services can be defined by the addition of a request to the set of *validated_requests*.

Transformation Service Definition *Initiate_Request(request)* This service is provided if a request is added to the set *validated_requests*.

THEORY *invocation_services*

```
invocation_services : THEORY

  BEGIN

  IMPORTING kernel_state

  st1, st2 : VAR KERNEL_STATE
  request : VAR KERNEL_REQUEST

  Initiate_Request(st1, st2, request) : bool =
     NOT member(request, validated_requests(st1))
     AND member(request, validated_requests(st2))

  END invocation_services
```

3.2.16.2 Request Invocation Requirements The requirements on invocation services are recorded in Tables 3 through 13.

The formalization of the invocation requirements which is presented below is incomplete in that it only addresses the most common invocation requirements, those which require *FLUKE_PROT_READ* or *FLUKE_PROT_EXEC*. In the invocation service tables we find that the following requests all have non-standard invocation requirements which are not reflected in the formalization. Though it would be possible to include special cases for each of these requirements, there appears to be little value in doing so:

- fluke_thread_get_client
- fluke_thread_get_client_secure
- fluke_thread_get_server
- fluke_thread_get_server_secure
- fluke_thread_get_state
- fluke_thread_get_state_secure
- fluke_thread_get_saved_state
- fluke_region_search
- fluke_mutex_trylock
- fluke_*object*_reference (for all object types)
- various fluke_ipc_* calls

THEORY *invocation_service_rqmts*

```
invocation_service_rqmts : THEORY

  BEGIN
```

```
  IMPORTING invocation_services

  st, st1, st2 : VAR KERNEL_STATE
  request : VAR KERNEL_REQUEST
  va : VAR VIRTUAL_ADDRESS

  read_ops_rqmt(st, request) : bool =
    member(op(request), read_ops)
      IMPLIES
        memory_allows_va(st, subject(request), primary_va(request), FLUKE_PROT_READ)

  exec_op_rqmt(st, request) : bool =
    op(request) = exec_op
      IMPLIES
        memory_allows_va(st, subject(request), primary_va(request), FLUKE_PROT_EXEC)

  input_vas_rqmt(st, request) : bool =
    FORALL va :
      member(va, input_vas(request))
        IMPLIES
          memory_allows_va(st, subject(request), va, FLUKE_PROT_READ)

  invocation_service_rqmts(st1, st2) : bool =
    FORALL request :
      Initiate_Request(st1, st2, request)
        IMPLIES (
          read_ops_rqmt(st1, request)
          AND exec_op_rqmt(st1, request)
          AND input_vas_rqmt(st1, request)
        )

  END invocation_service_rqmts
```

3.2.17 Service Requirements Summary

The set of allowed system transitions is modeled by *allowed_transitions*. It states that the transition must satisfy all of the requirements in the preceding sections.

THEORY *service_rqmts*

```
service_rqmts : THEORY

  BEGIN

  IMPORTING common_service_rqmts
  IMPORTING thread_service_rqmts
  IMPORTING task_service_rqmnts
  IMPORTING region_service_rqmts
  IMPORTING mapping_service_rqmts
  IMPORTING port_service_rqmts
  IMPORTING mutex_service_rqmts
  IMPORTING cond_var_service_rqmts
  IMPORTING ref_service_rqmts
  IMPORTING memory_service_rqmts
  IMPORTING ipc_service_rqmts
  IMPORTING invocation_service_rqmts

  st1, st2 : VAR KERNEL_STATE
```

```
cur_thread : VAR SOB_ADDRESS

allowed_transitions(st1, st2, cur_thread) : bool =
   common_service_rqmts(st1, st2, cur_thread)
   AND thread_service_rqmts(st1, st2, cur_thread)
   AND task_service_rqmts(st1, st2, cur_thread)
   AND region_service_rqmts(st1, st2, cur_thread)
   AND mapping_service_rqmts(st1, st2, cur_thread)
   AND port_service_rqmts(st1, st2, cur_thread)
   AND mutex_service_rqmts(st1, st2, cur_thread)
   AND cond_service_rqmts(st1, st2, cur_thread)
   AND ref_service_rqmts(st1, st2, cur_thread)
   AND memory_service_rqmts(st1, st2, cur_thread)
   AND ipc_service_rqmts(st1, st2)
   AND invocation_service_rqmts(st1, st2)

END service_rqmts
```

Appendix *A*
Bibliography

[1] Ajay Chitturi. Implementing mandatory network security in a policy flexible system. Master's thesis, University of Utah, 1998.

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

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

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

[5] Jay Lepreau, Bryan Ford, and Mike Hibler. The persistent relevance of the local operating system to global applications. In *Proceedings of the ACM SIGOPS European Workshop*, September 1996.

[6] 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.

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

[8] O. Sami Saydjari, S. Jeffrey Turner, D. Elmo Peele, John F. Farrell, Peter A. Loscocco, William Kutz, and Gregory L. Bock. Synergy: A distributed, microkernel-based security architecture. Technical report, INFOSEC Research and Technology, R231, November 1993.

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

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

[11] Secure Computing Corporation. Assurance in the Fluke Microkernel: Final Report. CDRL A002, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, TBD TBD.

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

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

Appendix *B*

Implementation Notes

This section contains notes describing the correspondence between the model presented in Section 3 and the current implementation.

These notes are organized into three sections:

- Section B.1 describes the correspondence between the state model and the data structures appearing in the code.
- Section B.2 describes the visibility of the elements of the state model across the kernel interface. In particular, it lists the requests which return the values of each element of the state.
- Section B.3 describes the relationship between microkernel requests and the services defined in Section 3.2. In particular, for each service it lists all requests which may cause that service to be performed.

B.1 State Model Correspondence

This section contains a correspondence between the microkernel state model and the current Fluke implementation. It is organized parallel to Section 3.1. Note that this lists all of the elements of the formal model, and therefore includes some elements which do not appear in the non-pvs version of this document.

The correspondence is presented in tables, each with three columns:

- The first column contains identifiers from the formal model, listed in the order in which they appear in the PVS specification. Where the model contains a list of constant values of some type, this column does not list all of those values. Instead it lists the single value CONSTANTS.

  Identifiers in the first column are indented when they are elements of a record whose type immediately precedes them.

- The second column contains the identifier from the code which corresponds to the identifier from the model in the first column. In some cases, several identifiers from the code are represented by a single identifier in the model.

  Identifiers in this column are indented when they are elements of an immediately preceding structure.

  Often, “SEE BELOW” will be listed in the second column, indicating that the correspondence is not direct and a more lengthy explanation is provided below the table. In a few cases, “NOT IMPLEMENTED” will be listed in the second column, indicating that something in the model is not part of the current implementation. “PREDICATE” may also appear in this column to indicate that the identifier of the first column is a predicate, not an element of the state, and generally has no direct analogue in the implementation.

- The third column indicates the name of the file where the identifier in the second column is declared. Except when noted by an asterisk, these files are named relative to the `fluke/kernel` directory. Files in the oskit are not identified.

B.1.1 Common Object State

```
Model                   Code                          File
--------------------    ---------------------         --------
SOB_ADDRESS             SEE BELOW
VIRTUAL_ADDRESS         oskit_addr_t                  * Oskit
PHYSICAL_ADDRESS        oskit_addr_t                  * Oskit
SID                     security_id_t                 * fluke/flask/security.prefix
FLUKE_SOB_TYPE          fluke_type_t                  * fluke/fluke/ob.h
CONSTANTS                                             * fluke/fluke/ob.h
SOB                     s_ob                          s_ob.h
     hash                    u_hash                   s_ob.h
     sob_type                type                     s_ob.h
     sid                     sid                      s_ob.h
     u_pa                    u_pa                     s_ob.h
```

- *SOB_ADDRESS* corresponds to the address of the kernel portion of a sob in kernel memory, i.e., a pointer to an object of type `s_ob`.

B.1.2 Threads

```
Model                   Code                          File
--------------------    ---------------------         --------
THREAD_SCHED_VAL        NOT IMPLEMENTED
THREAD_FLAGS            SEE BELOW
THREAD_EXC_STATE        SEE BELOW
THREAD_EXC_HANDLERS     SEE BELOW
THREAD_IPC_STATE        SEE BELOW
THREAD                  s_thread                      s_thread.h
     thread_va               thread_va                s_thread.h
     sched_val               NOT IMPLEMENTED
     flags                   gflags                   s_thread.h
                             lflags                   s_thread.h
                             ipc_state.flags          ipc.h
     exc_state               exc_state.eax            x86/exception.h
                             exc_state.ebx            x86/exception.h
                             exc_state.ecx            x86/exception.h
                             exc_state.edx            x86/exception.h
                             exc_state.esi            x86/exception.h
                             exc_state.edi            x86/exception.h
                             exc_state.ebp            x86/exception.h
                             exc_state.esp            x86/exception.h
                             exc_state.eip            x86/exception.h
                             exc_state.eflags         x86/exception.h
                             md.saved_eax             x86/s_thread.h
                             md.saved_ecx             x86/s_thread.h
                             md.saved_edx             x86/s_thread.h
```

	md.saved_eip	x86/s_thread.h
	md.saved_eflags	x86/s_thread.h
exception_handlers	md.trap_handler	x86/s_thread.h
	md.interrupt_handler	x86/s_thread.h
	md.client_alert_handler	x86/s_thread.h
	md.server_alert_handler	x86/s_thread.h
ipc_state	ipc_state.md.min_msg	x86/ipc.h
	ipc_state.server_pset_uva	ipc.h
	ipc_state.server_esp	ipc.h
	ipc_state.server_oneway_eip	ipc.h
	ipc_state.server_idempotent_eip	ipc.h
	ipc_state.server_reliable_eip	ipc.h
effective_client_sid	ipc_state.effective_client_sid	ipc.h
effective_server_sid	ipc_state.effective_server_sid	ipc.h
client_effective_client_sid	ipc_state.client_effective_client_sid	ipc.h
server_effective_server_sid	ipc_state.server_effective_server_sid	ipc.h
task_link	task_state.task_link	s_task.h
scheduler_link	sched_state.sched_link	SEE BELOW
waiting_for_link	NOT IMPLEMENTED	
client_link	ipc_state.client_link	ipc.h
server_link	ipc_state.server_link	ipc.h

- The correspondence for the types *THREAD_FLAGS*, *THREAD_EXC_STATE*, *THREAD_EXC_HANDLERS* and *THREAD_IPC_STATE* are described by the corresponding elements of the *THREAD* record.
- The location of the `sched_link` value in the code is dependent upon the scheduling policy being implemented, and may not be used at all for some policies.

B.1.3 Tasks

Model	Code	File
TASK	s_task	s_task.h
keeper_link	keeper	s_task.h

B.1.4 Regions

Model	Code	File
MEMORY_PROTECTION	SEE BELOW	
CONSTANTS	SEE BELOW	
MEMORY_PROTECTIONS	fluke_prot_t	
REGION	s_region	s_region.h
start_va	start_va	s_region.h
size	size	s_region.h
prot	prot	s_region.h
task_link	task	s_region.h
keeper_link	keeper	s_region.h

- The identifiers for memory protections correspond to identifiers in the code which are apparently included in files generated by Flick.

The *MEMORY_PROTECTION* and *MEMORY_PROTECTIONS* types are distinct in the model because PVS cannot consider sets without first declaring a type for individual elements of the set. In essence, the *FLUKE_PROT_** constants in the model are elements of the type *MEMORY_PROTECTION* while the same constants in the code are singleton sets, i.e., they correspond to elements of the type *MEMORY_PROTECTIONS*.

Editorial Note:
This may be considered to be a flaw in the model, though it seems more natural to talk about *FLUKE_PROT_READ* as an element of a set of protections rather than as a subset of a set of protections. On the other hand, it is more natural to talk about *FLUKE_PROT_READ_WRITE* as a subset.

B.1.5 Mappings

```
Model                   Code                              File
--------------------    ---------------------             --------
MAPPING                 s_mapping                         s_mapping.h
     region_off              region_off                   s_mapping.h
     start_va                start_va                     s_mapping.h
     size                    size                         s_mapping.h
     prot                    prot                         s_mapping.h
     region_link             region                       s_mapping.h
     task_link               task                         s_mapping.h
```

B.1.6 Ports

```
Model                   Code                              File
--------------------    ---------------------             --------
PORT_ALIAS              fluke_alias_t                     * fluke/fluke/ipc.h
PORT                    s_port                            s_port.h
     alias                   alias                        s_port.h
     pset_link               pset_link                    s_port.h
```

B.1.7 Port Sets

There is no state specific to a port set object.

B.1.8 Mutexes

```
Model                   Code                              File
--------------------    ---------------------             --------
MUTEX_FLAGS             oskit_u32_t                       * Oskit
MUTEX                   s_mutex                           s_mutex.h
     flags                   flags                        s_mutex.h
     locked                  locked                       s_mutex.h
     owner_link              owner                        s_mutex.h
```

B.1.9 Condition Variables

```
Model                   Code                              File
--------------------    ---------------------             --------
COND_FLAGS              oskit_u32_t                       * Oskit
COND_VAR                s_cond                            s_cond.h
     flags                   flags                        s_cond.h
     inheritor_link          inheritor_link               s_cond.h
     waiting_threads         cond.queue                   cond.h
```

B.1.10 References

```
Model                   Code                              File
--------------------    ---------------------             --------
REF                     s_ref                             s_ref.h
     object_link             link                         s_ref.h
```

B.1.11 Security

For basic security Types:

```
Model                   Code                              File
--------------------    ---------------------             --------
SECURITY_CLASS          security_class_t                  * fluke/flask/security.prefix
CONSTANTS                                                 * fluke/flask/security_classes
PERMISSION              SEE BELOW
CONSTANTS               SEE BELOW                         * fluke/flask/*.{common,av}
prot_to_perm            SEE BELOW
ACCESS_VECTOR           access_vector_t                   * fluke/flask/security.prefix
```

- *PERMISSION*, the constants, and *ACCESS_VECTOR* have the same kind of relationship as the *MEMORY_PROTECTIONS* discussed above.
- The function *prot_to_perm* represents the identity function between constants. See for instance the assertions in `fault.c`.

For the cache:

```
Model                   Code                              File
--------------------    ---------------------             --------
AVC_NODE                avc_node                          avc.c
     sclass                  ae.sclass                    avc.h
     ssid                    ae.ssid                      avc.h
     tclass                  ae.tclass                    avc.h
     tsid                    ae.tsid                      avc.h
     decided                 ae.decided                   avc.h
     allowed                 ae.allowed                   avc.h
node_allows             PREDICATE
AVC_NODE_ADDRESS        SEE BELOW
CACHE                   avc_cache_t                       avc.c
     existing_nodes     SEE BELOW
     node_data          SEE BELOW
cache_allows            PREDICATE
```

- *AVC_NODE_ADDRESS* is analogous to *SOB_ADDRESS*, and can be interpreted as a pointer to an `avc_node`.
- *existing_nodes* are the nodes reachable by recursively following links from the nodes in the `slots` array of an `avc_cache_t` structure.
- *node_data* does not have any direct counterpart in C. It is the operator which resolves a pointer to a node (*AVC_NODE*) to the node itself (*AVC_NODE_DATA*). This is analogous to the functions such as *sob_data* in the kernel state, below.

B.1.12 Kernel Requests

Nothing in this section has any direct relationship to anything in the implementation.

B.1.13 Physical Memory Management

The physical memory management model is not a direct model of the implementation. However, ultimately the frag structures in the kernel contain most of the information in the model, and so we will relate the model to the frag structures.

- *paged_in* — A (*task_addr*, *va*) pair is in the *paged_in* set if there exists a frag such that:
  - The mapping `frag.mapping` is mapped into *task_addr*.
  - *va* lies between the beginning and ending virtual addresses for the frag, which are
    - `frag.mapping.start_va` + `frag.offset`
    - `frag.mapping.start_va` + `frag.offset` + `frag.size` -1
- *phys_addr*(*task_addr*, *va*) — `frag.start_pa` indicates the starting physical address in a frag. *phys_addr(task_addr,va)* indicates the physical address which corresponds to *va* in the frag associated with (*task_addr*, *va*).
- *mem_prot*(*task_addr*, *va*) — This is the `frag.prot` value from the frag associated with (*task_addr*, *va*).
- *phys_addr_sid*(*pa*) — The SID of a physical address is ultimately the `segment_sid` field from the task in the memory graph which is closest to the kernel. This SID is also copied into the `segment_sid` field of all frags which point at that physical address. In the current implementation, this value never changes once a physical address is initially associated with a segment task.
- *contents* — This is simply the contents of a particular location of physical memory.

B.1.14 IPC Connections

In the terminology of the current Fluke implementation, a pair (*client*, *server*) is in the set *active_connections* if there is an unpickled connection between the two threads. Specifically, this means that the `ipc_state.server` field in the client's thread structure points at server thread and the `ipc_state.client` field in the server's thread structure points at the client thread. It is important to recognize that these are different than the `ipc_state.client_link` and `ipc_state.server_link` fields, which are part of the model of the thread state.

B.1.15 Summary

```
Model                   Code                              File
--------------------    ---------------------             --------
BASE_KERNEL_STATE       SEE BELOW
     existing_objects   SEE BELOW
     sob_data           SEE BELOW
     existing_*         SEE BELOW
     *_data             SEE BELOW
     cache              avc_cache                         avc.c
kernel_obj_union        PREDICATE
kernel_state            SEE BELOW
KERNEL_STATE            SEE BELOW
```

- *BASE_KERNEL_STATE*, *kernel_state* and *KERNEL_STATE* have no analogue in the implementation because they model the overall state of the system.
- According to comments in the file, it appears that the correct interpretation of *existing_objects* is the set of all sob objects whose `active` bit is set. Another possible requirement could be that the object exists in the `all_obs` table, but this appears to be equivalent to the `active` bit being set. This equivalence would not hold in an implementation where a thread may need to wait to lock the `all_obs` table (such as an SMP implementation).

  Other *existing_** functions are defined as the subset of *existing_objects* with the appropriate `type` field in the sob.
- The *_data* functions are operators which resolve a *SOB_ADDRESS* to the actual sob, analogous to resolving a pointer to a structure.

B.2 Microkernel State Visbility

For each of the elements of the state model, this section this section lists all of the kernel system calls which provide direct visibility of that element. These lists do not include requests which only export state indirectly, e.g., through error codes. This subsection is not included if there is no state associated with the object.

B.2.1 Common Object State Visibility

The following requests directly return values of the common object state.

- `fluke_ref_hash`
- `fluke_ref_type`
- `fluke_ref_sid`
- `fluke_*_create_hash` returns the hash value of a newly created object (which, according to the API, may be different than the requested hash)

B.2.2 Thread State Visibility

The following requests directly return elements from the state of a thread object.

- `fluke_thread_get_state`

- fluke_thread_get_client
- fluke_thread_get_server
- fluke_thread_get_state_secure
- fluke_thread_get_client_secure
- fluke_thread_get_server_secure
- fluke_thread_get_saved_state
- fluke_thread_self

B.2.3 Task State Visibility

The following requests directly return elements from the state of a task object.

- fluke_task_get_state

B.2.4 Region State Visibility

The following requests directly return elements from the state of a region object.

- fluke_region_get_state

B.2.5 Mapping State Visibility

The following requests directly return elements from the state of a mapping object.

- fluke_mapping_get_state

B.2.6 Port State Visibility

The following requests directly return elements from the state of a port object.

- fluke_port_get_state

B.2.7 Mutex State Visibility

The following requests directly return elements from the state of a mutex object.

- fluke_mutex_get_state
- fluke_mutex_trylock

B.2.8 Condition Variable State Visibility

The following requests directly return elements from the state of a condition variable object. Recall that *waiting_threads* is internal state and is not visible through any kernel request.

- fluke_cond_get_state

B.2.9 Security Data Visibility

No requests directly return values from the cache.

B.2.10 Memory State Visibility

None of the elements of the memory system (as defined in Section 3.1.15) are directly visible outside of the kernel. There is some amount of indirect visibility, for instance the SID of a virtual address can be found[26] as can the contents of a virtual address.

B.2.11 IPC State Visibility

There are no requests which directly return the value of the set *active_connections*. The `fluke_thread_get_*` requests will indicate whether a thread possibly has a connection with another thread, however, calling any of those requests results in the pickling of the connection so that it is not actually in the set of *active_connections*.

B.3 Request Mapping

For each of the transformation services defined in the control policy, this section lists all of the kernel system calls which can result in the service being provided.

B.3.1 Common Object Transformation Services

- *Object_Create*
  - `fluke_*_create`
- *Object_Destroy*
  - `fluke_*_destroy`
- *Object_Set_Hash*

  No requests perform this service.
- *Object_Set_Type*

  No requests perform this service.
- *Object_Set_Sid*

  No requests perform this service.
- *Object_Set_U_Pa*
  - `fluke_*_move`

[26]Though there is currently no direct call to determine the SID at a virtual address.

B.3.2 Thread Transformation Services

- *Thread_Set_Simple_State*
  - `fluke_thread_disable_exceptions`
  - `fluke_thread_enable_exceptions`
  - `fluke_thread_interrupt`(?)
  - `fluke_thread_return_from_exception`(?)
  - `fluke_thread_set_handlers`
  - `fluke_thread_set_client`
  - `fluke_thread_set_client_secure`
  - `fluke_thread_set_server`
  - `fluke_thread_set_server_secure`
  - `fluke_thread_set_saved_state`
  - `fluke_thread_set_state`
  - `fluke_thread_set_state_secure`
  - `fluke_thread_schedule`???
  - `fluke_mutex_lock` (can change the state of the client)
  - `fluke_mutex_trylock` (can change the state of the client)
  - `fluke_cond_broadcast` (can change the state of waiting threads)
  - `fluke_cond_signal` (can change the state of waiting threads)
  - `fluke_cond_wait` (can change the state of the client)
  - Lots of IPC requests change the state of the client or another thread involved in the IPC
  - `fluke_*_destroy` when waiting on the destroyed object, or when the destroyed object is the thread's task??
- *Thread_Set_Task*
  - `fluke_thread_set_state`
  - `fluke_thread_set_state_secure`
- *Thread_Set_Scheduler*
  - `fluke_thread_set_state`
  - `fluke_thread_set_state_secure`
- *Thread_Set_Client*
  - `fluke_thread_set_state`
  - `fluke_thread_set_client`
  - `fluke_thread_set_state_secure`
  - `fluke_thread_set_client_secure`
  - Some IPC requests (can change the state of the client or the other side of a connection)
- *Thread_Set_Server*
  - `fluke_thread_set_state`
  - `fluke_thread_set_server`
  - `fluke_thread_set_state_secure`
  - `fluke_thread_set_server_secure`
  - IPC requests ...
- *Thread_Associate_Eff_Client*

- fluke_thread_set_state_secure
- fluke_thread_set_server_secure
- fluke_ipc_client_connect_* (when a connection is actually established)

■ *Thread_Associate_Eff_Server*

- fluke_thread_set_state_secure
- fluke_thread_set_client_secure
- fluke_ipc_client_connect_* (when a connection is actually established)

■ *Thread_Associate_Task*

- fluke_thread_set_state
- fluke_thread_set_state_secure

■ *Thread_Associate_Scheduler*

- fluke_thread_set_state
- fluke_thread_set_state_secure

B.3.3 Task Transformation Services

■ *Task_Set_Keeper*

- fluke_task_set_state

■ *Task_Associate_Keeper*

- fluke_task_set_state

B.3.4 Region Transformation Services

■ *Region_Set_Simple_State*

- fluke_region_set_state
- fluke_region_protect

■ *Region_Set_Task*

- fluke_region_set_state

■ *Region_Set_Keeper*

- fluke_region_set_state

■ *Region_Associate_Task*

- fluke_region_set_state

■ *Region_Associate_Keeper*

- fluke_region_set_state

B.3.5 Mapping Transformation Services

- *Mapping_Set_Simple_State*
  - `fluke_mapping_set_state`
  - `fluke_mapping_protect`
- *Mapping_Set_Region*
  - `fluke_mapping_set_state`
- *Mapping_Set_Task*
  - `fluke_mapping_set_state`
- *Mapping_Associate_Region*
  - `fluke_mapping_set_state`
- *Mapping_Associate_Task*
  - `fluke_mapping_set_state`

B.3.6 Port Transformation Services

- *Port_Set_Simple_State*
  - `fluke_port_set_state`
- *Port_Set_Pset*
  - `fluke_port_set_state`
- *Port_Associate_Pset*
  - `fluke_port_set_state`

B.3.7 Mutex Transformation Services

- *Mutex_Set_Simple_State*
  - `fluke_mutex_set_state`
  - `fluke_mutex_lock`
  - `fluke_mutex_trylock`
  - `fluke_mutex_unlock`
  - `fluke_cond_wait`
- *Mutex_Set_Owner*
  - `fluke_mutex_set_state`
  - `fluke_mutex_lock`
  - `fluke_mutex_trylock`
  - `fluke_mutex_unlock`
  - `fluke_cond_wait`

B.3.8 Condition Variable Transformation Services

- *Cond_Set_Simple_State*
  - `fluke_cond_set_state`
- *Cond_Set_Inheritor*
  - `fluke_cond_set_state`
- *Cond_Set_Waiting_Threads*
  - `fluke_cond_wait`
  - `fluke_cond_broadcast`
  - `fluke_cond_signal`

B.3.9 Reference Transformation Services

- *Ref_Set_Link*
  - `fluke_*_reference`
  - `fluke_thread_get_client`
  - `fluke_thread_get_server`
  - `fluke_thread_get_state`
  - `fluke_task_get_state`
  - `fluke_region_get_state`
  - `fluke_mapping_get_state`
  - `fluke_port_get_state`
  - `fluke_thread_get_state`
  - `fluke_mutex_get_state`
  - `fluke_cond_get_state`
  - `fluke_ref_copy`
  - `fluke_ref_sid`
  - IPC requests which copy references
  - IPC "secure" requests which return SIDs

B.3.10 Security Data Transformation Services

- *Increase_Cache_Allows*
  - `ipc_reply` (from security outcalls)
  - `avc_add` (kserver interface)
  - `avc_reset` (kserver interface) — If an unwired vector in the initial state has been removed, it will be added back by this routine.

B.3.11 Memory Transformation Services

- *Memory_Add_Protection*

  Every fluke request can cause a page fault and lead to the performance of this service.
- *Memory_Set_Sid*

  Every fluke request can cause a page fault and lead to the performance of this service.

- *Memory_Write*

  Every fluke request that modifies user-space memory performs this service.

B.3.12 IPC Transformation Services

- *IPC_Connect*

  An initial connection is always established by a `fluke_ipc_client_connect_send` call (or a variation of this call). However, the connection can be pickled by any call which results in either thread blocking or being cancelled. When that condition is changed, the connection is unpickled which results in the connection being re-established, and hence this service being provided.

Appendix C
Requirements Summary

Transformation Service	Source	Target	Requirement
Object_Create	cur_thread	sob_addr	FLUKE_PROT_WRITE
	cur_thread	sob_addr	FLUKE_PROT_CREATE_OB
	—	—	FLASK: sid(sob_addr) = phys_addr_sid(u_pa)
Object_Destroy	cur_thread	sob_addr	FLUKE_PROT_WRITE
Object_Set_Hash	—	—	PROHIBITED
Object_Set_Type	—	—	PROHIBITED
Object_Set_Sid	—	—	FLASK: PROHIBITED
Object_Set_U_Pa	cur_thread	old_pa	FLUKE_PROT_WRITE
	cur_thread	new_pa	FLUKE_PROT_WRITE
	cur_thread	new_pa	FLUKE_PROT_CREATE_OB
	—	—	FLASK: phys_addr_sid(old_pa) = phys_addr_sid(new_pa)

Table 14: Requirements on Common Object Services

Transformation Service	Source	Target	Requirement
Thread_Set_Simple_State	cur_thread	thread_addr	FLUKE_PROT_WRITE OR cur_thread = thread_addr
Thread_Set_Task	cur_thread	thread_addr	FLUKE_PROT_WRITE
Thread_Set_Scheduler	cur_thread	thread_addr	FLUKE_PROT_WRITE
Thread_Set_Client	cur_thread	thread_addr	FLUKE_PROT_WRITE OR cur_thread = thread_addr
Thread_Set_Server	cur_thread	thread_addr	FLUKE_PROT_WRITE OR cur_thread = thread_addr
Thread_Associate_Eff_Client	thread_addr	esid	FLASK: SPECIFY_CLIENT_PERM OR FLASK: sid(thread_addr) = esid
Thread_Associate_Eff_Server	thread_addr	esid	FLASK: SPECIFY_SERVER_PERM OR FLASK: sid(thread_addr) = esid
Thread_Associate_Task	—	—	FLASK: sid(thread_addr) = sid(task_addr)
Thread_Associate_Scheduler	thread_addr	port_addr	FLASK: THREAD_SCHEDULER_PERM

Invocation Service	Source	Target	Requirement
fluke_thread_get_client(_secure)	cur_thread	thread	cur_thread = thread
fluke_thread_set_client(_secure)	cur_thread	client_ref	FLUKE_PROT_READ
fluke_thread_get_server(_secure)	cur_thread	thread	cur_thread = thread
fluke_thread_set_server(_secure)	cur_thread	server_ref	FLUKE_PROT_READ
fluke_thread_get_state(_secure)	cur_thread	thread	FLUKE_PROT_WRITE
fluke_thread_set_state(_secure)	cur_thread	task_ref	FLUKE_PROT_READ
	cur_thread	scheduler_ref	FLUKE_PROT_READ
	cur_thread	waiting_for_ref	FLUKE_PROT_READ
	cur_thread	client_ref	FLUKE_PROT_READ
	cur_thread	server_ref	FLUKE_PROT_READ
fluke_thread_get_saved_state	cur_thread	thread	cur_thread = thread

Table 15: Requirements on Thread Services

Transformation Service	Source	Target	Requirement
Task_Set_Keeper	cur_thread	task_addr	FLUKE_PROT_WRITE
Task_Associate_Keeper	task_addr	port_addr	FLASK: TASK_KEEPER_PERM
Invocation Service	**Source**	**Target**	**Requirement**
fluke_task_get_state	cur_thread	task	FLUKE_PROT_READ
fluke_task_set_state	cur_thread	keeper_ref	FLUKE_PROT_READ

Table 16: Requirements on Task Services

Transformation Service	Source	Target	Requirement
Region_Set_Simple_State	cur_thread	region_addr	FLUKE_PROT_WRITE
Region_Set_Task	cur_thread	region_addr	FLUKE_PROT_WRITE
Region_Set_Keeper	cur_thread	region_addr	FLUKE_PROT_WRITE
Region_Associate_Task	—	—	FLASK: sid(region_addr) = sid(task_addr)
Region_Associate_Keeper	—	—	FLASK: sid(region_addr) = sid(port_addr)
Invocation Service	**Source**	**Target**	**Requirement**
fluke_region_get_state	cur_thread	region	FLUKE_PROT_READ
fluke_region_set_state	cur_thread	task_ref	FLUKE_PROT_READ
	cur_thread	keeper_ref	FLUKE_PROT_READ
fluke_region_search	cur_thread	region	FLUKE_PROT_READ
			Also need FLUKE_PROT_READ to all of the memory actually searched

Table 17: Requirements on Region Services

Transformation Service	Source	Target	Requirement
Mapping_Set_Simple_State	cur_thread	mapping_addr	FLUKE_PROT_WRITE
Mapping_Set_Region	cur_thread	mapping_addr	FLUKE_PROT_WRITE
Mapping_Set_Task	cur_thread	mapping_addr	FLUKE_PROT_WRITE
Mapping_Associate_Region	mapping_addr	region_addr	FLASK: MAP_PERM
Mapping_Associate_Task	—	—	FLASK: sid(mapping_addr) = sid(task_addr)
Invocation Service	**Source**	**Target**	**Requirement**
fluke_mapping_get_state	cur_thread	mapping	FLUKE_PROT_READ
fluke_mapping_set_state	cur_thread	region_ref	FLUKE_PROT_READ
	cur_thread	task_ref	FLUKE_PROT_READ

Table 18: Requirements on Mapping Services

Transformation Service	Source	Target	Requirement
Port_Set_Simple_State	cur_thread	port_addr	FLUKE_PROT_WRITE
Port_Set_Pset	cur_thread	port_addr	FLUKE_PROT_WRITE
Port_Associate_Pset	—	—	FLASK: sid(port_addr) = sid(pset_addr)
Invocation Service	**Source**	**Target**	**Requirement**
fluke_port_get_state	cur_thread	port	FLUKE_PROT_READ
fluke_port_set_state	cur_thread	pset_ref	FLUKE_PROT_READ

Table 19: Requirements on Port Services

Invocation Service	Source	Target	Requirement
fluke_pset_get_state	cur_thread	pset	FLUKE_PROT_READ

Table 20: Requirements on Port Set Services

Transformation Service	**Source**	**Target**	**Requirement**
Mutex_Set_Simple_State	cur_thread	mutex_addr	FLUKE_PROT_WRITE
Mutex_Set_Owner	cur_thread	mutex_addr	FLUKE_PROT_WRITE
Invocation Service	**Source**	**Target**	**Requirement**
fluke_mutex_get_state	cur_thread	mutex	FLUKE_PROT_READ
fluke_mutex_set_state	cur_thread	owner_ref	FLUKE_PROT_READ
fluke_mutex_trylock	cur_thread	mutex	FLUKE_PROT_WRITE

Table 21: Requirements on Mutex Services

Transformation Service	**Source**	**Target**	**Requirement**
Cond_Set_Simple_State	cur_thread	cond_addr	FLUKE_PROT_WRITE
Cond_Set_Inheritor	cur_thread	cond_addr	FLUKE_PROT_WRITE
Cond_Set_Waiting_Threads	cur_thread	cond_addr	FLUKE_PROT_WRITE
Invocation Service	**Source**	**Target**	**Requirement**
fluke_cond_get_state	cur_thread	task	FLUKE_PROT_READ
fluke_cond_set_state	cur_thread	inheritor_ref	FLUKE_PROT_READ

Table 22: Requirements on Condition Variable Services

Transformation Service	**Source**	**Target**	**Requirement**
Ref_Set_Link	cur_thread	ref_addr	FLUKE_PROT_WRITE
Invocation Service	**Source**	**Target**	**Requirement**
fluke_ref_check	cur_thread	ref	FLUKE_PROT_READ
fluke_ref_compare	cur_thread	ref1	FLUKE_PROT_READ
	cur_thread	ref2	FLUKE_PROT_READ
fluke_ref_copy	cur_thread	src_ref	FLUKE_PROT_READ
fluke_ref_hash	cur_thread	ref	FLUKE_PROT_READ
fluke_ref_sid	cur_thread	ref	FLUKE_PROT_READ
fluke_ref_type	cur_thread	ref	FLUKE_PROT_READ
fluke_object_reference	cur_thread	object	FLUKE_PROT_WRITE

Table 23: Requirements on Reference Services

Transformation Service	**Source**	**Target**	**Requirement**
Memory_Add_Protection	task_addr	va	FLASK: prot_to_perm(prot)
Memory_Set_Sid	—	—	FLASK: No existing objects have this physical address
	—	—	FLASK: Nothing is paged in at this physical address
Memory_Write	cur_thread	pa	FLUKE_PROT_WRITE
Invocation Service	**Source**	**Target**	**Requirement**
read_op	cur_thread	va	FLUKE_PROT_READ
exec_op	cur_thread	va	FLUKE_PROT_EXEC

Table 24: Requirements on Memory Services

Transformation Service	Source	Target	Requirement
IPC_Add_Connection	—	—	client = client_link(server)
	—	—	server = server_link(client)
	eff_client_sid	eff_server_sid	FLASK: CONNECT_PERM
	FLASK: eff_client_sid(client) = client_eff_client_sid(server)		
	FLASK: eff_server_sid(server) = server_eff_server_sid(client)		
Invocation Service	**Source**	**Target**	**Requirement**
fluke_ipc_send	cur_thread	destination	FLUKE_PROT_READ
	cur_thread	port_addr	FLASK: SEND_PERM
fluke_ipc_send_secure	cur_thread	destination	FLUKE_PROT_READ
	cur_thread	client_sid	FLASK: SPECIFY_CLIENT_PERM
	client_sid	port_addr	FLASK: SEND_PERM
fluke_ipc_call	cur_thread	destination	FLUKE_PROT_READ
	cur_thread	port_addr	FLASK: CALL_PERM
fluke_ipc_call_secure	cur_thread	destination	FLUKE_PROT_READ
	cur_thread	client_sid	FLASK: SPECIFY_CLIENT_PERM
	client_sid	port_addr	FLASK: CALL_PERM
fluke_ipc_client_connect_send(_secure)	cur_thread	destination	FLUKE_PROT_READ
fluke_ipc_wait_receive(_secure)	cur_thread	pset	FLUKE_PROT_READ

Table 25: Requirements on IPC Services