Part Number 00-0930897A001 Rev C
Version Date 17 February 1999

Assurance in the Fluke Microkernel Formal Top Level Specification

CONTRACT NO. MDA904-97-C-3047
CDRL SEQUENCE NO. A004

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 ______________________

Distribution limited to U.S. Government Agencies Only. This document contains NSA information (17 February 1999). Request for the document must be referred to the Director, NSA.

Not releasable to the Defense Technical Information Center per DOD Instruction 3200.12.

Assurance in the Fluke Microkernel Formal Top Level Specification

Secure Computing Corporation

Abstract

This document contains formal specifications of the access vector cache and prototype security server for the Flask system.

Part Number	00-0930897A001 Rev C
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/ftls/RCS/ftls.vdd,v 1.8` `17 February 1999`

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

Contents

List of Figures

List of Tables

List of PVS Theories

Section 1
Scope

1.1 Identification

This document contains the Formal Top Level Specification, CDRL A004, 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 [15] and Synergy [10] 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

As described in the Formal Security Policy Model [17], the Flask architecture provides a sharp logical separation between security policy decisions and the enforcement of those decisions. Security policy enforcement is performed within object managers. Security policy decisions are made by a security server and may be stored by object managers within an access vector cache (AVC).

This document describes the interfaces to the security server and AVC. These interfaces are independent of the particular object manager and should therefore be readily adaptable to systems other than Fluke. The document also describes the policy implemented by the Flask prototype security server, a combination of multilevel security and type enforcement. Finally, it discusses the implementation of dynamic security policies and how this affects the security server and AVC interfaces.

The specification is formalized in the PVS specification language [8]. This document is also produced in a version which does not contain the PVS specification [18]. The two versions are otherwise identical and are generated from the same source text.

Editorial Note:
Since the Flask system continues to evolve, the descriptions in this document must be interpreted as a snapshot of the system at the time each was written.

The document is structured as follows:

- Section 1, **Scope**, defines the scope and this overview of the document.
- Section 2, **System Overview**, provides an overview of the Flask access control system and describes the motivation for the design of the AVC and security server interfaces.
- Section 3, **Access Vector Cache**, contains a model of an access vector cache.
- Section 4, **Security Server**, contains a model of the prototype Flask security server.
- Section 5, **Alternate Multilevel Security Policy Design**, describes a design of a multilevel security policy server which differs from the Flask prototype and which completely separates the multilevel security policy rules from the type enforcement rules.
- Section 6, **Dynamic Security Policy Study**, presents the results of a study intended to provide a framework for analyzing the support for dynamic security policies in Flask and other similar systems.
- Appendix A, **Bibliography**, provides citations for all referenced documents.
- Appendix B contains the formal PVS specifications for the Dynamic Security Policy Study.

Section 2
System Overview

The Flask architecture provides a sharp logical separation between security policy decisions and security policy enforcement. However, since policy enforcement is often part of the processing of every request made to an object manager, performance requirements dictate that security decisions can usually be retrieved with little overhead. At a minimum, it is unacceptable for context switching to be required to retrieve most security decisions.

A simple solution to this requirement would be to directly link a security decision making component into each object manager to execute within the same address space. This approach would work well if there is no need for coordination among the policy decisions made for each object manager. Unfortunately, coordination is typically required when changes are made to the security policy, to ensure that all object managers are responding to the same overall security policy.

One possible solution to this problem would be to define a protocol among the decision making components linked into each object manager to ensure consistency among them. Such a protocol could become quite complex, as it would need to recognize the different levels of trust granted to each object manager. A much simpler solution is to add a centralized security server which coordinates individually with the decision making component linked into each object manager.

This separation can also provide the traditional assurance benefits of a reference monitor. Verification of the security server is simplified by separation from the object managers, and failure of a single object manager only affects the policy enforcement within that object manager.

The performance concerns identified above must now be addressed by minimizing the amount of communication that is required between each object manager and the security server. There are a range of options which can be explored, with tradeoffs between policy flexibility, simplicity and performance. These options are discussed in more detail in section 3.4.4 of the DTOS Lessons Learned Report [15].

The Flask system includes an implementation of a cache for security decisions (known as the access vector cache, or AVC) within the object manager while including the logic for making and coordinating security decisions entirely within the security server. The goal of this implementation is to find the proper compromise between the goals of simplicity and performance while causing no loss of policy flexibility beyond the limitations already imposed by the basic interface between the enforcement and decision making components.

Figure 1 identifies the interfaces provided by the AVC and the security server and the typical clients of those interfaces. The remainder of this section discusses the use of these interfaces, organized according to the clients shown in the figure. Sections 3 and 4 contain detailed descriptions of the services provided by the AVC and the security server through these interfaces.

2.1 Object Manager Labeling to Security Server Interface

Security decisions are made based upon security identifier (SID) labels associated with objects being accessed. There are three requests that an object manager may make to a security server to support labeling.

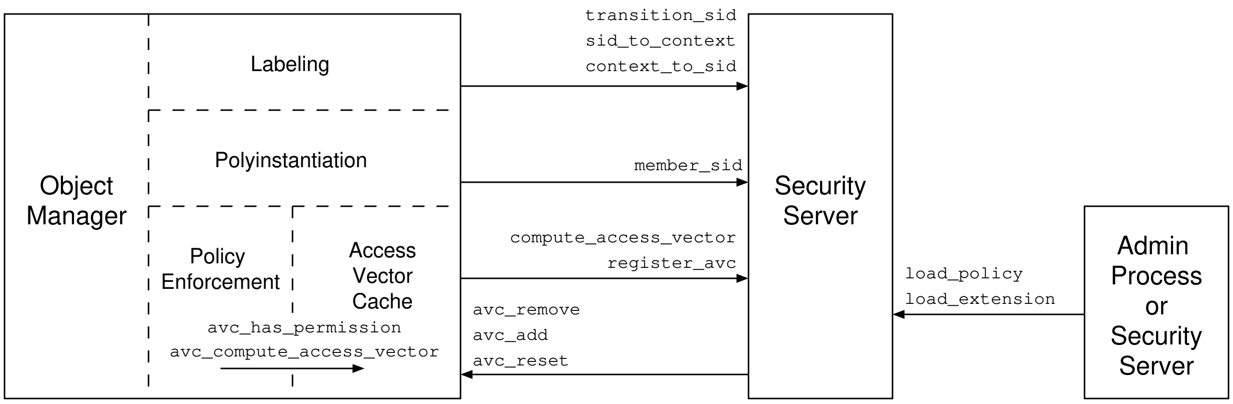

Figure 1: AVC and Security Server Interfaces

- `transition_sid` — Section 2.2 of the Formal Security Policy Model (FSPM) [17] discusses various models for assigning a SID to a newly created object. If the SID of the new object is copied from a related object in the object manager, or from the SID of the client requesting object creation, then no interaction with the security server is necessary.

  However, in some cases the SID is not simply copied but needs to be computed by the security server. As an example, the SID of a newly created file is dependent upon both the SID of the client requesting creation of the file and the SID of the containing directory. Since SIDs have no interpretation independent of the security policy, the new SID in these cases can only be calculated by the security server. The `transition_sid` interface is used for this purpose.[1]

- `sid_to_context` — This request is called whenever an object manager needs a readable string value (security context) for a SID. This may be needed by a policy aware client of the object manager or for labeling of persistent objects by the object manager.[2]

- `context_to_sid` — This request is called whenever an object manager needs to translate a readable string into a SID. This may be needed to translate a string provided by a policy aware client into a SID which can be used by the object manager. It may also be needed to translate a label stored with a persistent object.

2.2 Object Manager Polyinstantiation to Security Server Interface

One of the difficulties in designing a system which can satisfy security policies controlling the flow of information is the existence of certain resources which are intended to be shared among all processes in the system. The most obvious example is a system's hardware. However, there are important examples in software as well, including the file system name space and the port name space on a network interface.

Satisfying confinement in the presence of shared resources is done by either partitioning or polyinstantiating the resource. Polyinstantiation is clearly difficult for hardware resources, unless the maximum number of required instantiations is quite small. However it can be a

[1] The name `transition_sid` reflects the history of this request. It was originally provided to return the correct SID for a process that "transitions" by exec'ing a file.

[2] SIDs are not intended to be persistent, i.e., there is no guarantee that a SID has the same meaning across reboots of the system.

practical approach for software resources. In the Flask servers, polyinstantiation has been used for (configurable) portions of the file system name space and port name space.

When a particular process needs to access a polyinstantiated resource, the manager of that resource must be able to determine which instance of the resource to access. Since polyinstantiation is driven by the security policy, in the Flask architecture, each instance of a polyinstantiated resource is identified by a SID. The `member_sid` request is called by the object manager to determine the correct instance for a particular process.

Note that if the response to a `member_sid` call identifies an instance of a resource that has not been created, it may result in creation and labeling of that resource. Therefore the call may also be considered part of the labeling interface.

2.3 Object Manager Policy Enforcement to AVC Interface

The four requests previously described between the object manager and the security server are not expected to have a significant impact on performance, and therefore responses are not cached in the object manager. However, because policy enforcement is often required for every object manager request, to improve performance security server responses may be cached within the object manager. Therefore, the policy enforcement element of the object manager receives security decisions through an interface with the AVC, not directly from the security server.

There are two requests across this internal interface. In the absence of an AVC, these requests would be made directly to the security server through its `compute_access_vector` interface.

- `avc_has_permission` — As described in Section 2.2 of the FSPM [17], all security decisions in Flask are reduced to requiring one or more permissions between a source SID (typically of the client making a request) and a target SID (typically of the object which the request acts upon). This request is used to ask whether a set of one or more permissions are allowed according to the AVC.
- `avc_compute_access_vector` — This request is a generalization of the previous as it returns individual decisions for a set of permissions rather than a single decision over the set. This is useful when storing the individual decisions for future reference outside of the AVC. For example, this request is used to determine the permissions which are allowed to a page of memory when those permissions are to be stored in page tables.

2.4 AVC to Security Server Interface

The simplest interface that could be provided between the AVC and the security server would simply propagate all `avc_has_permission` or `avc_compute_access_vector` requests which cannot be answered by the current state of the AVC. Upon receiving the response from the security server, the AVC would pass the result back to enforcement component of the object manager while adding the result to the AVC so that any future identical request can be answered without communication with the security server.

To improve the cache hit rate for `avc_has_permission` and `avc_compute_access_vector` requests, the AVC does not simply ask the security server for a single permission between a source SID and a target SID, but for a class of permissions between the two SIDs. This class of permissions is known as an *access vector*. An access vector typically includes all permissions for the same types of objects as the requested permission.

Unfortunately, asking for permissions which are not immediately needed complicates the ability to support *stingy* security policies (this terminology was introduced in the DTOS Generalized Security Policy Specification [14]). A stingy policy is one in which there are constraints among the permissions which can be granted so that the act of granting a permission actually causes another permission to be denied. Examples of such policies include Dynamic N-Person, Chinese Wall and ORCON.

As a simple example, consider a requirement that a purchase order and the payment for the purchase order must be authorized by different individuals. A particular individual may have the authority to authorize either the purchase order or the payment, however, when either authorization is executed, the individual loses the authority to execute the other.

Now consider an automated system enforcing this requirement, in which an individual has three potential permissions controlling access to a purchase order, *read_order*, *authorize_order*, and *authorize_payment*. If the security server grants either *authorize_order* or *authorize_payment* permission, it cannot grant the other permission. However, either permission can be granted initially.

Suppose an individual requests to read the purchase order. When the AVC asks the security server if *read_order* permission is granted to the user, the security server is asked to return all permissions in the access vector which contains the other two permissions. However, since neither of these permissions has been specifically requested, the security server should not be required to decide whether the individual has *authorize_order* or *authorize_payment* permission.

To address this, when asked an access vector, the security server specifies which permissions are granted but can also refuse to provide decisions on permissions which were not specifically requested.

There are two requests across this interface:

- `compute_access_vector` — This request is called by the AVC when it is unable to service an `avc_has_permission` or `avc_compute_access_vector` request by the object manager. The response from the security server indicates whether the requested permissions are allowed and also provides decisions for other permissions in the access vector whenever possible.
- `register_avc` — This request is used to identify an AVC to a security server so that the security server can make requests back to the AVC. See Section 2.5.

2.5 Security Server to AVC Interface

The security server has an interface to the AVC so that it can explicitly request changes to the AVC as a response to security policy changes.

- `avc_remove` — If the security policy changes so that a previously granted permission is no longer allowed, the security server can call this request to remove the permission from the AVC.
- `avc_add` — Until now, the AVC has been described entirely as a performance enhancement. However, for some object managers at least a minimal AVC is required to address deadlock concerns. As an example, if the object manager is an operating system kernel the reliance on an external security server can cause deadlocks similar to those caused by reliance upon external memory sources. Therefore just as certain pages must be wired into an operating system kernel, so might certain permissions.

The primary purpose of the avc_add request is to allow the security server to add wired permissions to the AVC. For instance, this may be necessary before a new application is started.

The request could also be used to improve efficiency. If the security server is aware of typical access patterns, it may be able to provide an access vector to an AVC before it is requested.

- avc_reset — This request allows the security server to reset an AVC to its initial state (which is typically a minimal state required to avoid deadlocks). During significant policy changes, this may be more efficient than using avc_remove. The current Flask security server makes this request to all registered AVCs whenever either the load_policy or load_extension request is made to the security server.

2.6 Other Requests to the Security Server

Updates or additions to the security policy may be identified by the security server itself but may also be identified by external subjects. There are two requests which can be called to request these changes, corresponding to the two elements of the security policy, the *root policy* and the *extension policy*. These elements are discussed in detail in Section 4.

- load_policy — This request is called to load a new root policy into the security server. This would typically occur only if there is an administrative interface separate from the security server which can be used to request basic changes to the security policy. Because this request loads a new root policy, the granularity of the changes which can be made by this request and the specific format of the request is dependent upon the general kind of policy being enforced.
- load_extension — This request is provided primarily to support a hierarchy of security servers within the Fluke recursive virtual machine model, as described in Section 2.2 of the FSPM [17]. However, it could also be used by administrative subjects to load new portions of a security policy, for instance when a new application is added.

Section 3
Access Vector Cache

This section describes the design of the Flask access vector cache (AVC), which stores the results of security server decisions for later use by an object manager. This eliminates the need to query the security server for every access control decision, thus mitigating the most significant performance impact of Flask permission checking. The Flask AVC is based upon the microkernel AVC developed as part of the DTOS program. The DTOS AVC is described at a high level in Section 3.4 of [15].

The description of the AVC is divided into three sections:

- Section 3.1 provides a model of the state of the AVC, describing key data elements and the purpose of each.
- Section 3.2 describes the processing of requests made to the AVC and responses to requests made by the AVC to the security server.
- Section 3.3 describes some elements which were present in previous AVC designs and the reasons why they were removed.

Editorial Note:
The current Flask implementation contains two distinct AVC implementations, one for the microkernel and one for all user-space object servers. In general, the differences between the two implementation are in details below the level of the specifications in this section. However, we have added notes wherever the specification differs from one (or both) of the implementations.

3.1 Access Vector Cache State Model

The state model presented here is essentially the same as the model in the FSPM [17].

The AVC consists of a collection of nodes, each one of which describes security policy decisions for a particular (security class, SID, security class, SID) 4-tuple. Specifically, a node contains the following information:

- *sclass* — The source security class.
- *ssid* — The source sid.
- *tclass* — The target security class.
- *tsid* — The target sid.
- *decided* — An access vector indicating the set of all permissions for which this node provides a decision, i.e., which it identifies as either allowed or disallowed.
- *allowed* — An access vector indicating 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.

- *expiration* — The time at which the decisions in the node becomes invalid, or a value of 0 to indicate that the node has no expiration time. The usefulness of this is discussed below.

- *wired* — A bit indicating whether the node is wired into the AVC and cannot be removed except by a specific request. As discussed in Section 2.5, it may be necessary to wire certain permissions to avoid deadlock. Note that to avoid deadlocks, the node's *expiration* time must be zero and specific permissions required to avoid deadlock must be marked as both *decided* and *allowed*.[3]

Security classes are analogous (and often the same as) object types, for example, there are separate security classes for processes, files and directories. They are necessary to interpret permissions and access vectors, for instance to distinguish "read" permissions between a process and a file and between a process and a directory.

To summarize, a particular permission is allowed by a node if the following conditions hold:

- The permission is in the *decided* access vector.

- The permission is in the *allowed* access vector.

- The node is not expired, i.e., the *expiration* time of the node is zero (indicating no expiration time) or the *expiration* time is the same or later than the current time.

We conclude this section with a discussion of the usefulness of the *expiration* field in a node.

While the *decided* and *wired* fields have very specific purposes, the *expiration* field is probably not required to support any particular class of security policies. Rather it is provided for a combination of less critical reasons.

- In the original DTOS AVC design, an important expected use of the *expiration* time was to allow the security server to force an object manager to periodically communicate with the security server. This can be used to minimize denial of service attacks where the communication between security server and object manager is blocked, hence preventing the security server from responding to a change in policy. This is especially important if the security server resides on a separate node in a distributed system. Since this is no longer considered a practical implementation option, this use of the *expiration* value is not as critical as originally expected.

- Some security policies are dependent upon the time of day. For instance, permissions may vary depending upon whether a business is open or closed or depending upon whether a particular individual is currently on duty. The *expiration* time provides a convenient mechanism for enforcing such a policy.

  However, these policies also tend to have the property that slight delays in the policy change are acceptable.[4] Therefore, the policies could be enforced by having the security server, at the appropriate times, request that affected nodes be removed from the AVC.[5]

[3] The *wired* bit is the one element of the current model that is not included in the FSPM. It is unnecessary in the FSPM since it is irrelevant to security decisions, but it is necessary here because it is an important element of the interface.

[4] In fact, expiration times are not absolute either because "migrated" permissions are not revoked when a node expires.

[5] Assuming that the security server is not subject to denial of service attacks, and there are probably better ways to address that concern.

- A security server supporting a dynamic security policy may need to keep track of all accesses which have been previously granted, so that appropriate nodes can be removed from the AVC during a policy change. Providing an *expiration* time on a node allows the security server to only keep track of those accesses which have been granted and which are not yet expired.

In summary, the *expiration* time is primarily a convenience for these purposes and is not absolutely essential for supporting any particular known security policies, given the other features that are available.

Editorial Note:
The current implementation of the server AVC does not support expiration times, i.e., the expiration time must be zero for a permission to be granted. This was done because an object server currently requires a CP call to obtain the current time, though it is expected that this impediment will be removed.

We need to determine the correct way to deal with the expiration time. There seem to be three choices:

- Remove it altogether.
- Leave it in, and require all AVCs to support it.
- Leave it in, but allow some AVCs to ignore the expiration time, like the current server AVC. In this case, it is important that security server developers are aware that some AVCs will treat expiring access vectors as always being expired.

Also, if the expiration time is retained anywhere, then we need to discuss what impact an expiration time should have on a migrated permission.

Based upon the discussion here and above, SCC sees no reason to require that the implementation support an expiration time. However, we also don't see any significant assurance burden added by the expiration time, since it is not necessary (and not possible) to demonstrate that expiration times are absolutely obeyed.

Therefore this decision should be dependent upon implementation and especially performance concerns. If the expiration time causes problems in these areas, then it should be removed.

THEORY *shared_data_types*

```
shared_data_types : THEORY

BEGIN

% This theory defines data types shared between the AVC and its client,
% which is usually a security server, and between the security and its
% clients, which are usually object managers with an AVC or perhaps
% administrative clients.

  SID : TYPE+
  SECURITY_CLASS : TYPE+
  PERMISSION : TYPE+
  ACCESS_VECTOR : TYPE = setof[PERMISSION]
  NAME : TYPE+
  STATUS : TYPE = {Error, Ok}

  Subject_Class, Avc_Class, Process_Class, File_Class, Dir_Class, SS_Class : SECURITY_CLASS

  Avc_Add_Perm, Avc_Remove_Perm, Avc_Reset_Perms,
  SS_Compute_Access_Vector_Perm, SS_Sid_To_Context_Perm,
  SS_Context_To_Sid_Perm, SS_Transition_Sid_Perm, SS_Member_Sid_Perm,
  SS_Register_Avc_Perm, SS_Load_Policy_Perm, SS_Load_Extension_Perm : PERMISSION
```

```
% Wildcard values
  Wild_Class : SECURITY_CLASS
  Wild_Sid : SID

END shared_data_types
```

THEORY *AVC*

```
avc : THEORY

BEGIN

  IMPORTING shared_data_types

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

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

  node_allows(node, scls, ssid, tcls, tsid, perm, current_time) : bool =
        sclass(node) = scls
        AND ssid(node) = ssid
        AND tclass(node) = tcls
        AND tsid(node) = tsid
        AND (expiration(node) = 0 OR expiration(node) >= current_time)
        AND member(perm, decided(node))
        AND member(perm, allowed(node))

  AVC_NODE_ADDRESS : TYPE+

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

  avc : VAR AVC
  node_addr : VAR AVC_NODE_ADDRESS

  avc_allows(avc, scls, ssid, tcls, tsid, perm, current_time) : bool =
    EXISTS node_addr :
      member(node_addr, existing_nodes(avc))
      AND
      node_allows(node_data(avc)(node_addr), scls, ssid, tcls, tsid, perm, current_time)

  END avc
```

3.2 Access Vector Cache Transitions

This section describes the AVC requests, `avc_add`, `avc_remove`, `avc_reset`, `avc_has_permission` and `avc_compute_access_vector`.

The descriptions of these transitions are intended to capture only the fundamental impacts on the state of the AVC, they are not intended to be complete descriptions of the processing of the requests. This allows us to ignore unnecessary detail (such as any limits on the size of the AVC) and to focus instead on the processing which must be performed during each transition. It may be best to view these descriptions as requirements on the implementation rather than as specifications of any particular implementation.

Some specific consequences of this approach are the following:

- Specific error cases are not considered. We only address the basic requirements on the new state of the AVC when an error is encountered.

- For successful cases, preconditions on the state required for success are defined, without specifying the particular logic that chooses between multiple successful cases if such choices are possible. It is extremely important to recognize that these are only the necessary preconditions for success, and there is no guarantee of success even if those conditions are met. Again, this is necessary to allow us to ignore unnecessary detail such as restrictions on the size of the AVC.

- The final state of the AVC after a transition is not completely defined, rather it is restricted by defining the minimal and maximal sets of nodes which may exist in the AVC. An actual implementation can be tuned to provide maximum performance (or for any other characteristic) as long as it ensures that the AVC satisfies these minimum and maximum requirements.

- The base requirements on the final state of the AVC are the following:

  – All previously wired nodes must be retained (this is the minimal set of nodes). Note that unwired nodes can be removed at any time.

  – All nodes in the final state must have been in the AVC in the initial state (this is the maximal set of nodes).

  These two requirements must be met by any error case, and the interesting processing in other cases is identified as exceptions to these requirements.

- The model of the state of the AVC presents no consistency requirements among nodes, e.g., it allows for the possibility of separate nodes appearing for the same 4-tuple with different contents. The descriptions of these transitions are intended to address consistency.

  Specifically, the formal specifications are intended to maintain a state invariant that any nodes for the same 4-tuple contain identical data, though this has not been verified. For an implementation we would expect this requirement to be met by requiring there to be only one node for each 4-tuple, but the consistency requirement is easier to formalize and ultimately it is the key property.

3.2.1 avc_add

The `avc_add` request is called by a client attempting to add a node to the AVC or to replace an existing node. This request is the only way to add new wired nodes to the AVC, which may be necessary if a new application is added to an operational system.

3.2.1.1 Input Parameters

There are nine input parameters to the request. The first eight of the parameters simply contain the values associated with the node to be added. The final parameter is a flag, `replace`, which indicates what the request should do if there is an existing node with the same 4-tuple, as discussed in more detail below.

```
sclass       /* source class */
ssid         /* source sid */
tclass       /* target class */
tsid         /* target sid */
allowed      /* allowed vector */
decided      /* decided vector */
expiration   /* expiration time */
wired        /* wired flag */
replace      /* should an existing node be replaced? */
```

3.2.1.2 Output Parameters

```
status   /* error status from the request */
```

3.2.1.3 Description

There are two successful cases in processing this request, i.e., cases in which the requested node is actually added to the AVC. In both cases, the client making the request must have *Avc_Add* permission to the object manager in which the AVC is used. It is important to emphasize also that unless the *wired* bit is set for the requested node, even if it is added there is no guarantee that it will remain in the AVC for any length of time.

- If the AVC does not contain a node for the input 4-tuple, then the `replace` flag is irrelevant and a non-error status code indicates that the requested node has been added. The requirements on the final state of the AVC are that all previously wired nodes have been retained and only the requested node has been added.
- If the AVC does contain a node for the input 4-tuple and the `replace` flag is set, then a non-error status code indicates that the requested node has been added to replace the old node. The requirements on the final state of the AVC are that all previously wired nodes have been retained with the possible exception of the node being replaced and that only the requested node has been added.

Editorial Note:
It may actually be appropriate to allow a third successful case. Since a non-wired node can be removed from the AVC at any time, perhaps it makes sense to allow for replacement of an existing unwired node even if the replace flag is not set, so that replace is actually interpreted as "replacewired"? Note that the current implementation only replaces a node if the replace flag is set.

This is not a very important issue to resolve because the interpretation of *replace* is likely to be changed in the suggestions resulting from the dynamic policy study.

THEORY *avc_common*

```
avc_common : THEORY

BEGIN

  IMPORTING avc

  avc, old_avc, new_avc : VAR AVC
  sclass, tclass : VAR SECURITY_CLASS
  ssid, tsid : VAR SID
  allowed, decided : VAR ACCESS_VECTOR
  expiration : VAR nat
  wired, replace : VAR bool
  node_addr : VAR AVC_NODE_ADDRESS


% Routines that return some attribute about the contents of an AVC
%    node_matches returns TRUE if the given node matches the given 4-tuple
%    node_matches_wild returns TRUE if the given node matches the given
%       4-tuple, allowing wildcarding
%    node_matches_contents returns TRUE if the given node matches all
%       of the given contents
%    node_found returns TRUE if there is a node in the AVC for the given 4-tuple
%    no_node_found_wild returns TRUE if there are no nodes in the AVC
%       for the given 4-tuple, allowing wildcarding
%    no_unwired_node_found_wild returns TRUE if there are no unwired
%       nodes in the AVC for the given 4-tuple, allowing wildcarding


  node_matches(avc, (node_addr : (existing_nodes(avc))), sclass, ssid, tclass, tsid) : bool =
    sclass(node_data(avc)(node_addr)) = sclass
    AND ssid(node_data(avc)(node_addr)) = ssid
    AND tclass(node_data(avc)(node_addr)) = tclass
    AND tsid(node_data(avc)(node_addr)) = tsid

  node_matches_wild(avc, (node_addr : (existing_nodes(avc))), sclass, ssid, tclass, tsid) : bool =
    (sclass(node_data(avc)(node_addr)) = sclass OR sclass = Wild_Class)
    AND (ssid(node_data(avc)(node_addr)) = ssid   OR ssid = Wild_Sid)
    AND (tclass(node_data(avc)(node_addr)) = tclass OR tclass = Wild_Class)
    AND (tsid(node_data(avc)(node_addr)) = tsid   OR tsid = Wild_Sid)

  node_matches_contents(avc, (node_addr : (existing_nodes(avc))),
                  sclass, ssid, tclass, tsid, decided, allowed, expiration, wired) : bool =
    node_matches(avc, node_addr, sclass, ssid, tclass, tsid)
    AND decided(node_data(avc)(node_addr)) = decided
    AND allowed(node_data(avc)(node_addr)) = allowed
    AND expiration(node_data(avc)(node_addr)) = expiration
    AND wired(node_data(avc)(node_addr)) = wired

  node_found(avc, sclass, ssid, tclass, tsid) : bool =
    EXISTS (node_addr : (existing_nodes(avc))) :
      node_matches(avc, node_addr, sclass, ssid, tclass, tsid)

  no_node_found_wild(avc, sclass, ssid, tclass, tsid) : bool =
    FORALL (node_addr : (existing_nodes(avc))) :
      NOT node_matches_wild(avc, node_addr, sclass, ssid, tclass, tsid)
```

```
no_unwired_node_found_wild(avc, sclass, ssid, tclass, tsid) : bool =
  FORALL (node_addr : (existing_nodes(avc))) :
    node_matches_wild(avc, node_addr, sclass, ssid, tclass, tsid)
      IMPLIES wired(node_data(avc)(node_addr))



% Routines which define a relationship between an old_avc and a new_avc
%   node_unchanged returns TRUE if the contents of the given node are the same
%     in the old_avc and the new_avc
%   all_wired_nodes_retained returns TRUE if all wired nodes in old_avc are
%     unchanged in new_avc
%   all_wired_nodes_retained_except returns TRUE if all wired nodes in
%     old_avc are unchanged in new_avc, with one identified exception
%   all_wired_nodes_retained_except_wild returns TRUE if all wired
%     nodes in old_avc are unchanged in new_avc, with identified
%     exceptions with possible wildcarding
%   no_nodes_added returns TRUE if all of the nodes in the new_avc are
%     unchanged from the old_avc
%   one_node_added returns TRUE if all of the nodes in the new_avc are
%     in the old_avc, with the possible exception of the given node.
%     It requires that the given node is added and that any node for
%     that 4-tuple have the same values.

node_unchanged(old_avc, new_avc, node_addr) : bool =
  member(node_addr, existing_nodes(old_avc))
  AND member(node_addr, existing_nodes(new_avc))
  AND node_data(old_avc)(node_addr) = node_data(new_avc)(node_addr)

all_wired_nodes_retained(old_avc, new_avc) : bool =
  FORALL (node_addr : (existing_nodes(old_avc))) :
    wired(node_data(old_avc)(node_addr))
      IMPLIES node_unchanged(old_avc, new_avc, node_addr)

all_wired_nodes_retained_except(old_avc, new_avc, sclass, ssid, tclass, tsid) : bool =
  FORALL (node_addr : (existing_nodes(old_avc))) :
    (wired(node_data(old_avc)(node_addr))
     AND NOT node_matches(old_avc, node_addr, sclass, ssid, tclass, tsid)
    )
      IMPLIES node_unchanged(old_avc, new_avc, node_addr)

all_wired_nodes_retained_except_wild(old_avc, new_avc, sclass, ssid, tclass, tsid) : bool =
  FORALL (node_addr : (existing_nodes(old_avc))) :
    (wired(node_data(old_avc)(node_addr))
     AND NOT node_matches_wild(old_avc, node_addr, sclass, ssid, tclass, tsid)
    )
      IMPLIES node_unchanged(old_avc, new_avc, node_addr)

no_nodes_added(old_avc, new_avc) : bool =
  FORALL (node_addr : (existing_nodes(new_avc))) :
    node_unchanged(old_avc, new_avc, node_addr)

one_node_added(old_avc, new_avc, ssid, sclass, tsid, tclass, decided, allowed, expiration, wired) : bool =
  node_found(new_avc, sclass, ssid, tclass, tsid)
  AND FORALL (node_addr : (existing_nodes(new_avc))) :
    ((NOT node_matches(new_avc, node_addr, sclass, ssid, tclass, tsid)
      AND node_unchanged(old_avc, new_avc, node_addr))
     OR node_matches_contents(new_avc, node_addr, sclass, ssid, tclass, tsid,
                              decided, allowed, expiration, wired))

END avc_common
```

THEORY *avc_add*

```
avc_add : THEORY

  BEGIN

  IMPORTING avc_common

  client_sid, om_sid : VAR SID
  current_time : VAR nat
  old_avc, new_avc : VAR AVC
  sclass, tclass : VAR SECURITY_CLASS
  ssid, tsid : VAR SID
  allowed, decided : VAR ACCESS_VECTOR
  expiration : VAR nat
  wired, replace : VAR bool
  status : VAR STATUS

  avc_add_bool(client_sid, om_sid, current_time, old_avc, new_avc,
               sclass, ssid, tclass, tsid, allowed, decided, expiration, wired, replace, status) : bool =
       (status = Error
        AND all_wired_nodes_retained(old_avc, new_avc)
        AND no_nodes_added(old_avc, new_avc))
    OR
       (status = Ok
        AND avc_allows(old_avc, Subject_Class, client_sid, Avc_Class, om_sid, Avc_Add_Perm, current_time)
        AND NOT node_found(old_avc, sclass, ssid, tclass, tsid)
        AND all_wired_nodes_retained(old_avc, new_avc)
        AND one_node_added(old_avc, new_avc, ssid, sclass, tsid, tclass, decided, allowed, expiration, wired))
    OR
       (status = Ok
        AND avc_allows(old_avc, Subject_Class, client_sid, Avc_Class, om_sid, Avc_Add_Perm, current_time)
        AND node_found(old_avc, sclass, ssid, tclass, tsid)
        AND replace
        AND all_wired_nodes_retained_except(old_avc, new_avc, sclass, ssid, tclass, tsid)
        AND one_node_added(old_avc, new_avc, ssid, sclass, tsid, tclass, decided, allowed, expiration, wired))

  END avc_add
```

3.2.2 avc_remove

The `avc_remove` request is called by a client attempting to force one or more identified nodes to be removed from the AVC. This may be necessary due to a change in security policy or simply to remove wired nodes which are no longer necessary.

3.2.2.1 Input Parameters

There are five input parameters to the request. The first four of the parameters identify the node or nodes to be removed. These parameters contain specific security class or SID values, or any of the four can contain a wild card value to allow multiple nodes to match. The final parameter is a flag, `removewired`, which indicates whether the request should remove matching nodes even if they are wired.

```
sclass        /* source class */
ssid          /* source sid */
tclass        /* target class */
tsid          /* target sid */
removewired   /* remove even if node is wired? */
```

3.2.2.2 Output Parameters

```
status   /* error status from the request */
```

3.2.2.3 Description There are two successful cases in processing this request, i.e., cases in which the requested nodes are actually removed from the AVC (or weren't in the AVC to begin with). In both cases, the client making the request must have *Avc_Remove* permission to the object manager in which the AVC is used.

- If `removewired` is set, then any nodes matching the input parameters are removed from the AVC. The requirements on the final state of the AVC are that all previously wired nodes are retained except the input nodes, no new nodes are added and that the AVC contains no nodes matching the input parameters.
- If `removewired` is not set, then any unwired nodes matching the input parameters are removed from the AVC. The requirements on the final state of the AVC are that all previously wired nodes are retained, no new nodes are added and that the AVC contains no unwired nodes matching the input parameters.

Editorial Note:
The basic requirement that no unwired nodes are removed in the error case is probably too strong here, because of wild carding. In particular, we may want to allow an error case in which removewired is set and some but not all of the requested nodes are removed. However, it does not appear that this scenario can occur in the current implementation.

THEORY *avc_remove*

avc_remove : **THEORY**

BEGIN

IMPORTING *avc_common*

client_sid, *om_sid* : **VAR** *SID*
current_time : **VAR** *nat*
avc, *old_avc*, *new_avc* : **VAR** *AVC*
sclass, *tclass* : **VAR** *SECURITY_CLASS*
ssid, *tsid* : **VAR** *SID*
removewired : **VAR** *bool*
node_addr : **VAR** *AVC_NODE_ADDRESS*
status : **VAR** *STATUS*

avc_remove_bool(*client_sid*, *om_sid*, *current_time*, *old_avc*, *new_avc*,
sclass, *ssid*, *tclass*, *tsid*, *removewired*, *status*) : *bool* =

(*status* = *Error*
AND *all_wired_nodes_retained*(*old_avc*, *new_avc*)
AND *no_nodes_added*(*old_avc*, *new_avc*))
OR
(*status* = *Ok*
AND *avc_allows*(*old_avc*, *Subject_Class*, *client_sid*, *Avc_Class*, *om_sid*, *Avc_Remove_Perm*, *current_time*)
AND *removewired*
AND *all_wired_nodes_retained_except_wild*(*old_avc*, *new_avc*, *sclass*, *ssid*, *tclass*, *tsid*)
AND *no_nodes_added*(*old_avc*, *new_avc*)
AND *no_node_found_wild*(*new_avc*, *sclass*, *ssid*, *tclass*, *tsid*))
OR
(*status* = *Ok*
AND *avc_allows*(*old_avc*, *Subject_Class*, *client_sid*, *Avc_Class*, *om_sid*, *Avc_Remove_Perm*, *current_time*)
AND **NOT** *removewired*
AND *all_wired_nodes_retained*(*old_avc*, *new_avc*)
AND *no_nodes_added*(*old_avc*, *new_avc*)
AND *no_unwired_node_found_wild*(*new_avc*, *sclass*, *ssid*, *tclass*, *tsid*))

END *avc_remove*

3.2.3 avc_reset

The `avc_reset` request is called by a client to force the AVC to revert to its initial state. This initial state is typically empty or contains the minimal number of wired entries necessary to allow the object manager and its security server to initiate communications, at which time the security server can load any other necessary wired entries.

There are no parameters to this request, and it is always successful.

No formal specification is provided for this request because it is so simple and since it would require adding some kind of initial AVC contents to the model which at this time isn't useful anywhere else.

3.2.4 avc_has_permission

The `avc_has_permission` request is called by the policy enforcement element of the object manager when it requires a security decision.

3.2.4.1 Input Parameters

The input parameters identify the a set of permissions being requested between two SIDs.

```
sclass      /* source class */
ssid        /* source sid */
tclass      /* target class */
tsid        /* target sid */
requested   /* set of permissions required */
```

3.2.4.2 Output Parameters

```
status   /* error status from the request */
```

3.2.4.3 Description This request returns a non-error status only if the AVC allows all of the requested permissions.

If there is no node in the AVC which identifies all of the requested permissions as *decided*, this request will trigger a security fault to make a `compute_access_vector` call to the security server. In the microkernel AVC, the security fault is triggered by altering the state of the client thread and returning an error code from this request. In the server AVC, the security fault triggers an immediate `compute_access_vector` request to the security server before returning from the `avc_has_permission` request. When the request finally returns, a non-error status indicates that the AVC now allows all of the requested permissions.

Security faults are discussed further in the next section.

Editorial Note:
An additional *nowait* input flag may be added to this request to avoid the blocking behavior of the server AVC. If this flag is set, the routine will immediately return with an error indicating that a security fault occured. Meanwhile, the server AVC will process the security fault so that a future `avc_has_permission` request may be successful.

This flag would be set when `avc_has_permission` is called from a routine which is expected to return immediately.

THEORY *avc_has_permission*

```
avc_has_permission : THEORY

BEGIN

  IMPORTING avc
  IMPORTING avc_common

  current_time : VAR nat
  avc : VAR AVC
  sclass, tclass : VAR SECURITY_CLASS
  ssid, tsid : VAR SID
  requested : VAR ACCESS_VECTOR
  status : VAR STATUS

% This function only models the return value from the request. State
% changes may also occur if a security fault is triggered.

  avc_has_permission_return_bool(current_time, avc, sclass, ssid, tclass, tsid, requested, status) : bool =
      (status = Error)
    OR
      (status = Ok
       AND FORALL (perm : (requested)) : avc_allows(avc, sclass, ssid, tclass, tsid, perm, current_time))

  END avc_has_permission
```

3.2.4.4 Security Fault Processing In the current implementation, AVC changes can also result from processing of security faults by the object manager. This section discusses those changes and discusses some issues with the current implementation.

A security fault occurs when an object manager is performing an operation which requires some permissions that cannot be located in the AVC. This occurs in three situations:

- There is no node in the AVC for the relevant 4-tuple.
- There is a node, but it has expired.
- There is an unexpired node, but the required permissions are not listed as *decided*.

After the security fault is encountered, the object manager makes a `compute_access_vector` call to the security server. This request is described in Section 4.2.2. The response to this request either indicates that an error occurred, in which case no changes are made to the AVC, or the response provides *allowed*, *decided* and *expiration* values for the 4-tuple.

This response is therefore similar to an `avc_add` request, except that it does not contain the parameters *wired* or *replace*. Since the main purpose for wiring nodes in the AVC is to avoid deadlock situations, a request for a wired node may result in deadlock so wired nodes should always be added to the AVC using `avc_add`. Therefore a response to a `compute_access_vector` request is always treated as an unwired node.

Editorial Note:
The other missing parameter is the `replace` flag. The current implementation behaves as if the `replace` flag were set. However, it is likely that the `replace` flag will itself be replaced as a result of ongoing study of dynamic policies.

3.2.5 avc_compute_access_vector

The `avc_compute_access_vector` request is called by the policy enforcement element of the object manager when it requires a complete (or partial) access vector.

Editorial Note:
This request is currently only implemented for the microkernel AVC.

3.2.5.1 Input Parameters

The input parameters identify the a set of permissions being requested between two SIDs.

```
sclass      /* source class */
ssid        /* source sid */
tclass      /* target class */
tsid        /* target sid */
requested   /* set of permissions required */
```

3.2.5.2 Output Parameters

```
allowed   /* the set of permissions allowed by the AVC */
status    /* error status from the request */
```

3.2.5.3 Description The processing of this request is similar to the processing of `avc_has_permission`. If there is no node in the AVC which identifies all of the requested permissions as *decided*, this request will trigger a security fault. If there is no security fault, then this request is a simple lookup within the AVC.

However, the output from this request contains more information than the output from the `avc_has_permission` request. The *allowed* parameter returns all permissions which are both *allowed* and *decided* according to a node which contains a decision for all of the permissions in *requested*. The effect of this is that while `avc_has_permission` returns a single decision over the entire set of *requested* permissions, `avc_compute_access_vector` returns individual decisions over each of the *requested* permissions. Moreover, `avc_compute_access_vector` may also return other permissions which are granted by the AVC.

Editorial Note:
If this request is implemented in the server AVC, then (like `avc_has_permission`) it may be necessary to add an input parameter to indicate that the request should immediately return if a security fault occurs.

THEORY *avc_compute_access_vector*

```
avc_compute_access_vector : THEORY

BEGIN

  IMPORTING avc
  IMPORTING avc_common

  current_time : VAR nat
  avc : VAR AVC
  sclass, tclass : VAR SECURITY_CLASS
  ssid, tsid : VAR SID
  requested, allowed : VAR ACCESS_VECTOR
  status : VAR STATUS

% This function only models the return value from the request. State
% changes may also occur if a security fault is triggered.

  avc_compute_access_vector_return_bool(current_time, avc, sclass, ssid, tclass, tsid, requested, allowed, status) : bool =
      (status = Error)
    OR
      (status = Ok
       AND EXISTS (node_addr : (existing_nodes(avc))) :
            LET node_expiration = expiration(node_data(avc)(node_addr)),
                node_decided = decided(node_data(avc)(node_addr)),
                node_allowed = allowed(node_data(avc)(node_addr)) IN
            (node_matches(avc, node_addr, sclass, ssid, tclass, tsid)
             AND (node_expiration = 0 OR node_expiration >= current_time)
             AND subset?(requested, node_decided)
             AND allowed = intersection(node_allowed, node_decided)))

% The current implementation does not actually intersect node_allowed
% and node_decided. In effect, it requires that the security server
% doesn't identify any permissions as allowed if not decided. This is
% a reasonable requirement, though intersecting the two vectors
% eliminates that requirement altogether.

  END avc_compute_access_vector
```

3.3 Obsolete Access Vector Cache Elements

This section describes two major elements in previous AVC designs which are no longer present in the current design, and the rationale for removing each.

3.3.1 Single Use Permissions

In the DTOS AVC, the implementation of the *cache control vector*, which most directly corresponds to the *decided* vector, allowed for the implementation of security policies which would allow a permission to be granted for a single use in the object manager. A subsequent need for the permission would generate another security fault. In the original Flask AVC design, this feature was retained though it was implemented in a more direct manner.

It was later decided that there were no significant uses for this feature, and since there were complications in both implementations, the feature was dropped. This section describes why no significant uses were seen for the feature.

We are able to identify only three potential uses for this feature:

- Some security policies are most easily supported if the security server can assume that certain kinds of subjects or objects are uniquely identified by their SIDs (see [14]). One way to guarantee unique SIDs would be to grant the "create" permission only once for each SID. However, this still requires the object manager to identify a unique SID for each object. A better way to provide unique SIDs is for the object manager to call the `transition_sid` interface to the security server to have a SID assigned when creating a new object, so that the security server takes the responsibility of determining the unique SID (if it is necessary) and the object manager simply labels the object accordingly.

- The security server may want to avoid a permission being used "a lot", for instance if that may indicate a covert channel. By granting permissions one at a time, the security server can keep count of the maximum number of times that the permission is used. However, we know of no specific policy which requires this feature, and moreover, good audit facilities combined with the use of the `avc_remove` request can probably accomplish the same thing.

- If the security server expects that a permission may be revoked "soon", and is not confident about its ability to complete a flush, giving a permission out for a single use minimizes the damage if the flush cannot be completed. However, the expiration time can be used for essentially the same purpose. Moreover, in the Flask implementation, there is no guarantee that the permission will be used anytime soon, and the AVC could probably be manipulated indirectly to ensure that the permission lives on for awhile.

It is probably fair to say that when the cache control vector was originally envisioned, it was expected that a minimal time lag in changing policies would generally be acceptable, and therefore marking a permission as non-decided was just a way to avoid doing a flush later. But as part of the DTOS program, it was determined [14] that most policies actually rely upon complete and explicit revocation, and a time lag is not acceptable. Moreover, for those policies where a time lag is acceptable, use of the expiration time is a more appropriate solution.

3.3.2 Notification Vector

Each node of the DTOS AVC contained a *notification vector*. If a permission was included in the notification vector, then the object manager was responsible for notifying the security server whenever that permission was looked up in the AVC. This was later generalized so that the notification could be sent to an audit server instead.

The original purpose of this vector was to provide support for those policies which may change as a result of permissions which are previously granted. However, as the DTOS program progressed it was determined that such policies are generally stingy [14], and that the feature necessary to support stingy policies is to allow the security server not to provide decisions on all permissions until specifically requested.

The notification vector was then later used to provide an audit mechanism. As an audit mechanism it is quite weak, since it only identifies the SIDs involved in a permission check, not the service being provided, and it only identifies when the permission is actually accessed in the AVC. Audit mechanisms have not been considered further on the Fluke program, so the notification vector has not been retained.

Section 4
Security Server

This section describes state maintained by the prototype Flask security server and the interfaces it provides.

The security policy determined by a Flask security server has two distinct elements, a *root policy* and an *extension policy*. Ultimately, the distinction between these policies is defined by the way in which they are implemented in the security server, but the general distinction is that the root policy consists of that portion of the policy which is interpreted by some software logic within the security server while the extension policy consists of a collection of simple lookup tables provided by clients of the security server. Extension policies were initially introduced to provide support for a hierarchy of security servers in the Fluke recursive virtual machine architecture, as discussed in Section 2.2 of the FSPM [17].

Because extension policies consist only of lookup tables, all data and processing related to extension policies are independent of the root policy. The descriptions in this section are intended to isolate as much as possible the aspects of the security server which are dependent upon the particular root policy.

4.1 Security Server State Model

This section presents a model of the state of the prototype security server.

4.1.1 Internal Data Types

This section describes some basic types which are internal to the security server but which are independent of any particular security policy.

The most important of these types is the security context, of which there are two kinds. A root security context is a security context defined by the security server's root policy. An extension context is defined by the extension policies loaded into the security server. The following describes six fields which define a security context.

- *isroot* — This is a flag which is set if the security context is a root security context. This is the only field which is relevant to both kinds of security contexts.
- *mcontext*, *acontext* — A root security context is defined by two fields, a mandatory security context (*mcontext*) and an authentication context (*acontext*).
- *parent*, *name* — An extension context is defined by two fields, the *parent* SID and a *name*.

A security server also maintains two pieces of information about registered access vector caches:

- *avc_ref* — A reference used by the security server to identify the destination for a message to be sent to the AVC.

- *client_sid* — The SID of the client which registered the AVC with the security server. When making requests of the registered AVC, the security server impersonates this client.[6]

THEORY *internal_data_types*

```
internal_data_types[MANDATORY_CONTEXT : TYPE, AUTH_CONTEXT : TYPE] : THEORY

BEGIN

  IMPORTING shared_data_types

  SECURITY_CONTEXT : TYPE = [#
        isroot            : boolean,
        mcontext          : MANDATORY_CONTEXT,
        acontext          : AUTH_CONTEXT,
        parent            : SID,
        name              : NAME,
        depth             : nat
        #]

% "depth" is not mentioned in the text and ideally depth would not be
% part of the model of a security context,
% as it is certainly not a fundamental concept and since it should be
% computable from the current state of the security server. It is only
% included because it is used in the specification of
% compute_access_vector. This is discussed more in the
% extension_policy_compute theory.

  Null_Parent : SID
  Null_Name : NAME

  AVC_REF : TYPE+
  AVC_INFO : TYPE = [#
        avc_ref           : AVC_REF,
        client_sid        : SID
        #]

END internal_data_types
```

4.1.2 Mandatory Contexts

Mandatory contexts are specific to a particular security policy. This section describes the mandatory contexts for the prototype Flask security server supporting a combination of multi-level security (MLS) and type enforcement (TE).

An *MLS_LEVEL* consists of two elements:

- A sensitivity level, *sens*, which indicates the hierarchical level, e.g., Unclassified or Secret.
- A set of categories, *cats*, which indicates the set of categories, e.g., NATO or Nuclear.

There are four possible relationships between two levels, *level1* and *level2*:

- *EQUAL* — The *sens* level and *cats* of *level1* and *level2* are identical.

[6] Impersonation in IPC is discussed in the Security Architecture section of the FSPM [17].

- *DOM* — *level1* dominates *level2*, i.e., the *sens* for *level1* is greater than or equal to the *sens* for *level2* and the set of *cats* for *level1* contain (or are equal to) the set of *cats* for *level2*, and *level1* is not equal to *level2*.
- *DOMBY* — *level1* is dominated by *level2*.
- *INCOMP* — The levels are incomparable, i.e., none of the previous three cases hold.

In the Flask MLS/TE security server, mandatory contexts indicate a range of levels with a *low* and a *high* value. To be valid, the *high* value must equal or dominate the *low* value.

Editorial Note:
There is a practical implementation problem with defining a level range when each individual level is expressed as a (*sens*, *cats*) pair. To illustrate the problem, suppose *SECRET* is a sensitivity level and *A* and *B* are categories. Further, suppose that we want to specify a security context that includes the three levels *SECRET{A,B}*, *SECRET{A}* and *SECRET{B}*. This set of levels is not directly representable as a range of levels, because there is no unique low level for the range.

One way to solve this problem is to define a "fake" category *F* which has no real meaning but which is always included in the category set of a level whenever either *A* or *B* is included. Then the desired set of levels can be represented through the range where *low* = *SECRET{F}* and *high* = *SECRET{A,B,F}*.

While there are many possible relationships between ranges, the only one of direct interest is whether one range contains another. We say that *range1* contains *range2* if

- the *high* level of *range1* is equal to or dominates the *high* level of *range2*, and
- the *low* level of *range1* is equal to or dominated by the *low* level of *range2*.

We will also commonly compare a range of levels to a single level, again using *EQUAL*, *DOM*, *DOMBY* and *INCOMP* with the following meanings:

- *EQUAL* — *level* is contained in *range*.
- *DOM* — The *high* level of *range* is equal to or dominates *level*, but *EQUAL* does not hold.
- *DOMBY* — The *low* level of *range* is equal to or dominated by *level*, but *EQUAL* does not hold.
- *INCOMP* — None of the previous hold.

Because most of the interesting relationships between levels can be reduced to these four classes, it is convenient to store access vectors according to these four classes. Therefore we define *MLS_ACCESS_VECTORS* to be a collection of four access vectors denoted by *equal*, *dom*, *domby* and *incomp*.

The type enforcement element of a mandatory context is represented by an identifier *typ*.[7] Unlike MLS, there are no general comparisons which are made between type enforcement attributes.

THEORY *mandatory_context*

---

[7] We use *typ* rather than *type* because *type* is a reserved word in the PVS specification language.

```
mandatory_context : THEORY

BEGIN

  IMPORTING shared_data_types

  SENSITIVITY : TYPE = int
  CATEGORY : TYPE+

  MLS_LEVEL : TYPE = [#
          sens      : SENSITIVITY,
          cats      : setof[CATEGORY]
          #]

  LEVEL_RELATION : TYPE = {
          EQUAL,
          DOM,
          DOMBY,
          INCOMP
          }

  level, level1, level2 : VAR MLS_LEVEL

  mls_level_relation(level1, level2) : LEVEL_RELATION =
    LET sens1 =  sens(level1),
        sens2  = sens(level2),
        cats1 =  cats(level1),
        cats2  = cats(level2) IN
    IF     ((sens1  = sens2) AND (cats1 = cats2)) THEN EQUAL
    ELSIF ((sens1 >= sens2) AND (subset?(cats2, cats1))) THEN DOM
    ELSIF ((sens1 <= sens2) AND (subset?(cats1, cats2))) THEN DOMBY
    ELSE INCOMP
    ENDIF


  MLS_RANGE : TYPE = [#
          low       :  MLS_LEVEL,
          high      :  MLS_LEVEL
          #]

  range, range1, range2 : VAR MLS_RANGE

  mls_range_valid(range) : bool =
        mls_level_relation(high(range), low(range)) = EQUAL
    OR mls_level_relation(high(range), low(range)) = DOM

  mls_range_contains(range1, range2) : bool =
    LET low1 = low(range1),
        low2 = low(range2),
        high1 = high(range1),
        high2 = high(range2) IN
      (    mls_level_relation(high1, high2) = EQUAL
       OR mls_level_relation(high1, high2) = DOM)
    AND
      (    mls_level_relation(low1, low2) = EQUAL
       OR mls_level_relation(low1, low2) = DOMBY)

  mls_range_relation(range, level) : LEVEL_RELATION =
    LET low = low(range),
        high = high(range) IN
    IF
           (    mls_level_relation(high,level) = EQUAL
            OR mls_level_relation(high,level) = DOM)
       AND
```

```
        (   mls_level_relation(low,level) = EQUAL
         OR mls_level_relation(low,level) = DOMBY)
    THEN EQUAL
    ELSIF
        (   mls_level_relation(high,level) = EQUAL
         OR mls_level_relation(high,level) = DOM)
    THEN DOM
    ELSIF  (   mls_level_relation(low,level) = EQUAL
         OR mls_level_relation(low,level) = DOMBY)
    THEN DOMBY
    ELSE INCOMP
    ENDIF


  MLS_ACCESS_VECTORS : TYPE = [#
        equal     : ACCESS_VECTOR,
        dom       : ACCESS_VECTOR,
        domby     : ACCESS_VECTOR,
        incomp    : ACCESS_VECTOR
        #]

  TYPE_ENFORCEMENT : TYPE+

  MANDATORY_CONTEXT : TYPE = [#
        typ       : TYPE_ENFORCEMENT,
        range     : MLS_RANGE
        #]

END mandatory_context
```

4.1.3 Authentication Contexts

This section describes the authentication contexts used by the Flask prototype security server. An authentication context is comprised of the following:

- An effective user, *euser*
- An effective group, *egroup*
- A set of groups, *groups*
- A real user, *ruser*
- A real group, *rgroup*
- A saved user, *svuser*
- A saved group, *svgroup*

We say that an authentication context *auth1* contains another, *auth2*, if the following relationships hold:

- The *euser* for *auth2* is the same as either the *euser*, *ruser* or *svuser* for *auth1*.
- The *egroup* for *auth2* is the same as either the *egroup*, *rgroup* or *svgroup* for *auth1*.
- The *ruser* for *auth2* is the same as either the *euser* or *ruser* for *auth1*.
- The *rgroup* for *auth2* is the same as either the *egroup* or *rgroup* for *auth1*.

- The *svuser* for *auth2* is the same as the *svuser* for *auth1*.
- The *svgroup* for *auth2* is the same as the *svgroup* for *auth1*.
- The set of *groups* for *auth2* is a subset of the set of *groups* for *auth1*.

In the policy, each user is associated with *USER_POLICY_DATA* that defines a set of groups, a set of types, and a set of ranges that the user can be associated with. Note that by controlling a user's access to sets of ranges rather than sets of levels, the policy can specify that a user can create certain single level subjects but not necessarily multilevel subjects across the same levels.

THEORY *auth_context*

```
auth_context : THEORY

BEGIN

  IMPORTING mandatory_context

  USER : TYPE+
  GROUP : TYPE+

  AUTH_CONTEXT : TYPE = [#
        euser    : USER,
        egroup   : GROUP,
        groups   : setof[GROUP],
        ruser    : USER,
        rgroup   : GROUP,
        svuser   : USER,
        svgroup  : GROUP
        #]

  auth1, auth2 : VAR AUTH_CONTEXT

  acontext_contains(auth1, auth2) : bool =
        (      euser(auth1) =   euser(auth2)
         OR    ruser(auth1) =   euser(auth2)
         OR   svuser(auth1) =   euser(auth2)
        )
    AND
        (     egroup(auth1) =  egroup(auth2)
         OR   rgroup(auth1) =  egroup(auth2)
         OR  svgroup(auth1) =  egroup(auth2)
        )
    AND
        (      euser(auth1) =   ruser(auth2)
         OR    ruser(auth1) =   ruser(auth2)
        )
    AND
        (     egroup(auth1) =  rgroup(auth2)
         OR   rgroup(auth1) =  rgroup(auth2)
        )
    AND       svuser(auth1) =  svuser(auth2)
    AND      svgroup(auth1) = svgroup(auth2)
    AND subset?(groups(auth2), groups(auth1))

  USER_POLICY_DATA : TYPE = [#
        groups   : setof[GROUP],
        types    : setof[TYPE_ENFORCEMENT],
```

levels : *setof*[*MLS_RANGE*]
#]

END *auth_context*

4.1.4 Security Server State Data

In this section we describe a model of the state maintained by the security server. The state can be divided into three categories.

The first category is the basic state maintained by any Flask security server, regardless of the particular security policy defined by the server.

- *existing_sids* — This is the set of all SIDs currently known by the security server.
- *context* — This is a function which returns the security context associated with any existing SID.
- *registered_avcs* — The set of AVCs which have been registered with the security server.
- *ss_avc* — An access vector cache from which the security server looks up permissions to control its own services.

  While an AVC is certainly not necessary in a security server, it is included in the security server model to demonstrate how to incorporate the AVC model into the formal specification of an object manager. Note that because the AVC is included in the specification of the security server, the AVC transitions from Section 3.2 are also extended to be transitions of the security server which affect only *ss_avc*.
- *ss_sid* — The SID of the security server, to be used in permission checking on security server services.

The second category is the state maintained by the Flask prototype security server for the specific purpose of defining the root MLS/TE security policy.

- *existing_users* — This is the set of all users currently known by the security server.
- *user_data* — This is a function which returns the *USER_POLICY_DATA* associated with any existing user.
- *valid_levels* – A level is a combination of a sensitivity and a category, and not all combinations may be allowable. As an example, the very existence of a category may be classified information, so it should not be possible to combine such a category with the UNCLASS sensitivity. The *valid_levels* set indicates which combinations are allowed.
- *mls_vectors_defined* — This is the set of all (source class, source type, target class, target type) 4-tuples for which a specific set of *MLS_ACCESS_VECTORS* are defined. If undefined, the policy allows no permissions for the 4-tuple.
- *mls_vectors* — This is a function which returns the *MLS_ACCESS_VECTORS* for any 4-tuple in the set *mls_vectors_defined*.

- *priv_auth_perms* — There are certain permissions for which the four relationships reflected in *mls_vectors* are insufficient for making a policy decision, and additional requirements involving authentication contexts must be considered. The function *priv_auth_perms* returns, for any pair of security classes, the set of such permissions for that pair. Details of these additional requirements are provided in the discussion of the `compute_access_vector` request in Section 4.2.2.
- *priv_mls_perms* — Similar to the previous, this is a function which returns, for any pair of security classes, the set of permissions for that pair which are only granted if additional requirements involving the level ranges must be considered. Details of these additional requirements are provided in the discussion of the `compute_access_vector` request in Section 4.2.2.
- *priv_auth_types* — This is a set of types for which the additional requirements for the permissions in *priv_auth_perms* may be overridden. Note that this set is not specific to any particular permission but allows all of the additional authentication context requirements to be overridden.
- *priv_mls_types* — This is a set of types for which the additional requirements for the permissions in *priv_mls_perms* may be partially overridden, as described in Section 4.2.2. Note that this set is also not specific to any particular permission.
- *root_transition_defined* — This is a set of (source type, target type, security class) triples for which transitions (see the description of the `transition_sid` request in Section 4.2.5) are defined. Any triples not in this set are subject to default transition rules.
- *root_transition* — This is a function which returns a new transition type for any triple in the set *root_transition_defined*.

The third category is the state required to support extension policies. This state is maintained regardless of the particular root policy being enforced.

- *ext_perms_defined* — This is a set of (source class, source sid, target class, target sid) 4-tuples for which explicit permissions have been defined by an extension policy. Any 4-tuples not contained in this set are subject to default rules described in the discussion of the `compute_access_vector` request in Section 4.2.2.
- *ext_perms* — This is a function which returns an access vector for any 4-tuple in *ext_perms_defined*
- *ext_transition_defined* — This is a set of (sid, sid, security class) triples for which a transition SID has been explicitly defined by an extension policy. Any triples not in this set are subject to default transition rules.
- *ext_transition* — This is a function which returns a transition SID for any triple in the set *ext_transition_defined*.

THEORY *ss_state*

ss_state : **THEORY**

BEGIN

```
  IMPORTING mandatory_context
  IMPORTING auth_context
  IMPORTING internal_data_types[MANDATORY_CONTEXT, AUTH_CONTEXT]
  IMPORTING avc

  SS_STATE : TYPE = [#

% generic security server state
  % We do not seem to need an existing_classes function, since
  % compute_access_vector treats an unknown class as a denied
  % permission, transition_sid treats an unknown class as if no
  % transitions are defined, and member_sid ignores the class

        existing_sids        : setof[SID],
        context              : [(existing_sids) -> SECURITY_CONTEXT],
        registered_avcs      : setof[AVC_INFO],
        ss_avc               : AVC,
        ss_sid               : SID,

% root policy state
  % We could add existing_groups, existing_sensitivities,
  % existing_categories and existing_types. The only reason we do not
  % is simplicity in the model, and the fact that there is not really
  % any use for these functions since we do not model any changes to
  % them.
  %
  % The model also ignores the fact that certain types are explicitly
  % identified as domains in the implementation, and effectively
  % treats all types as domains.

        existing_users       : setof[USER],
        user_data            : [(existing_users) -> USER_POLICY_DATA],
        valid_levels         : setof[[CATEGORY, SENSITIVITY]],
        mls_vectors_defined  : setof[[SECURITY_CLASS, TYPE_ENFORCEMENT,
                                       SECURITY_CLASS, TYPE_ENFORCEMENT]],
        mls_vectors          : [(mls_vectors_defined) -> MLS_ACCESS_VECTORS],
        priv_auth_perms      : [[SECURITY_CLASS, SECURITY_CLASS] -> ACCESS_VECTOR],
        priv_mls_perms       : [[SECURITY_CLASS, SECURITY_CLASS] -> ACCESS_VECTOR],
        priv_auth_types      : setof[TYPE_ENFORCEMENT],
        priv_mls_types       : setof[TYPE_ENFORCEMENT],
        root_transition_defined    : setof[[TYPE_ENFORCEMENT, TYPE_ENFORCEMENT, SECURITY_CLASS]],
        root_transition            : [(root_transition_defined) -> TYPE_ENFORCEMENT],

% extension policy state

        ext_perms_defined          : setof[[SECURITY_CLASS, SID, SECURITY_CLASS, SID]],
        ext_perms                  : [(ext_perms_defined) -> ACCESS_VECTOR],
        ext_transition_defined     : setof[[SID, SID, SECURITY_CLASS]],
        ext_transition             : [(ext_transition_defined) -> SID]
        #]

END ss_state
```

4.2 Security Server Transitions

This section describes all requests that can be made of the security server. For the prototype security server, all transitions are caused by receipt of a request. For other policies, there may be additional transitions which are generated internal to the security server, for instance, if the policy changes based upon the time of day, or simply for garbage collection.

Like the descriptions of the AVC transitions in Section 3.2, the descriptions here focus on successful processing. The possible error cases are not distinguished and return values are not specified when an error occurs. However, unlike the AVC descriptions, the state of the security server after each transition is completely defined.

4.2.1 Common Processing Routines

This section describes some properties of security contexts which are dependent upon not only the basic definition of security contexts but upon the current state of the security server.

The following properties are defined for root security contexts.

- A mandatory context is *valid* in some state em state of the security server if the following hold:
  - The level range is a valid range as defined in Section 4.1.2.
  - The low level is valid in *state*, i.e., for all categories in the low level the category and the low sensitivity are in the set *valid_levels*.
  - The high level is similarly valid in *state*.
- An authentication context is *valid* in some state *state* of the security server if the following hold:
  - The user from the context is in the set of *existing_users* in *state*.
  - The *egroup* from the context is in the set of allowed groups for the user (according to *user_data*) in *state*.
  - All groups in the set of *groups* from the context are in the allowed groups for the user in *state*.
- A root security context is *valid* in some state *state* if the mandatory and authentication contexts are each valid in *state* and the following holds for the combination:
  - The level range from the mandatory context is in the allowed set of level ranges for the user from the authentication context (according to *user_data*) in *state*.
  - The type from the mandatory context is in the allowed set of types for the user from the authentication context (according to *user_data*) in *state*.

  The validity of a root context is verified when it is created and when a `load_policy` request is serviced.

THEORY *root_policy_routines*

---

root_policy_routines : **THEORY**

BEGIN

  IMPORTING *ss_state*

  state : **VAR** *SS_STATE*
  mcontext : **VAR** *MANDATORY_CONTEXT*
  acontext : **VAR** *AUTH_CONTEXT*
  old_state : **VAR** *SS_STATE*

```
  sid, new_sid : VAR SID

  valid_mandatory_context(state, mcontext) : bool =
    LET range = range(mcontext) IN
    LET low = low(range),
        high = high(range) IN
    mls_range_valid(range)
    AND (FORALL (category : (cats(low))) : member((category, sens(low)), valid_levels(state)))
    AND (FORALL (category : (cats(high))) : member((category, sens(high)), valid_levels(state)))

  valid_authentication_context(state, acontext) : bool =
    LET user = euser(acontext) IN
    member(user, existing_users(state))
    AND LET user_data = user_data(state)(user) IN
        member(egroup(acontext), groups(user_data))
    AND subset?(groups(acontext), groups(user_data))

  valid_root_context(state, mcontext, acontext) : bool =
    valid_mandatory_context(state, mcontext)
    AND valid_authentication_context(state, acontext)
    AND LET user_data = user_data(state)(euser(acontext)) IN
        member(range(mcontext), levels(user_data))
    AND member(typ(mcontext), types(user_data))

  root_sid_context_match(state, sid, mcontext, acontext) : bool =
    member(sid, existing_sids(state))
    AND isroot(context(state)(sid))
    AND mcontext = mcontext(context(state)(sid))
    AND acontext = acontext(context(state)(sid))

% The following function should only be called if the combination of
% mcontext and acontext is actually a valid context, and if new_sid is
% not an existing_sid

  create_root_sid(old_state, mcontext, acontext, new_sid) : SS_STATE =
    old_state
      WITH [existing_sids := add(new_sid, existing_sids(old_state)),
            context := context(old_state)
              WITH [new_sid |-> (# isroot := TRUE,
                                   mcontext := mcontext, acontext := acontext,
                                   depth :=0, parent := Null_Parent, name := Null_Name #)]]

END root_policy_routines
```

The following properties are defined for SIDs which may correspond to extension security contexts.

- An ancestor sequence for a SID *sid* in a state *state* is a non-empty sequence of SIDs from *existing_sids* in *state* such that:
  - The sequence begins with a SID corresponding to a root context in *state*,
  - No other SIDs in the sequence correspond to a root context in *state*,
  - The sequence ends with *sid*, and
  - The *parent* from the security context associated in *state* with each SID in the sequence points to the previous SID in the sequence.
- A SID is *valid* in a state it it has an ancestor sequence.
- Every valid SID in a state has a *root* SID which is the first SID in its ancestor sequence.

- A security context is identified across the security server interface by three elements, a mandatory context, an authentication context, and a sequence of names. A SID matches a security context in a state if:
  - The SID is valid,
  - The security context associated with its root SID matches the mandatory and authentication contexts, and
  - The given sequence of names is identical to the sequence of names from the security contexts associated with each SID of the ancestor sequence (after the root SID).

THEORY *extension_policy_routines*

```
extension_policy_routines : THEORY

BEGIN

  IMPORTING ss_state
  IMPORTING root_policy_routines

  state : VAR SS_STATE
  sid, root_sid : VAR SID
  name_sequence : VAR finite_sequence[NAME]
  mcontext : VAR MANDATORY_CONTEXT
  acontext : VAR AUTH_CONTEXT

  sid_ancestor_sequence(state, (sid : (existing_sids(state))), (sid_sequence : finite_sequence[(existing_sids(state))])) : bool =
    LET length = length(sid_sequence),
        seq = seq(sid_sequence) IN
    length > 0
    AND isroot(context(state)(seq(0)))
    AND sid = seq(length − 1)
    AND (length > 1 IMPLIES
          ((FORALL (i : {i : nat | i < length−1}) : parent(context(state)(seq(i+1))) = seq(i))
           AND (FORALL (i : {i : nat | 0 < i AND i < length−1}) : NOT isroot(context(state)(seq(i))))))

  valid_sid(state, sid) : bool =
    member(sid,existing_sids(state))
    AND EXISTS (sid_sequence : finite_sequence[(existing_sids(state))]) :
        sid_ancestor_sequence(state, sid, sid_sequence)

  sid_to_root(state, (sid : (existing_sids(state))), root_sid) : bool =
    EXISTS (sid_sequence : finite_sequence[(existing_sids(state))]) :
      sid_ancestor_sequence(state, sid, sid_sequence)
      AND root_sid = seq(sid_sequence)(0)

  sid_context_match(state, (sid : (existing_sids(state))), mcontext, acontext, name_sequence) : bool =
    EXISTS (sid_sequence : finite_sequence[(existing_sids(state))]) :
      sid_ancestor_sequence(state, sid, sid_sequence)
      AND root_sid_context_match(state, seq(sid_sequence)(0), mcontext, acontext)
      AND length(sid_sequence) = length(name_sequence)+1
      AND FORALL (i : {i : nat | 0 < i AND i < length(sid_sequence)}) :
          name(context(state)(seq(sid_sequence)(i))) = seq(name_sequence)(i−1)

END extension_policy_routines
```

4.2.2 compute_access_vector

This request is called by a client (typically an object manager) requesting some set of permissions for a (source class, source sid, target class, target sid) 4-tuple. The security server responds with an access vector indicating some set of allowed permissions, another access vector indicating those permissions for which the security server has made a decision (the *decided* vector), an expiration time and a request status.

4.2.2.1 Input Parameters

```
sclass     /* source class */
ssid       /* source sid */
tclass     /* target class */
tsid       /* target sid */
requested  /* requested access vector */
```

4.2.2.2 Output Parameters

```
allowed     /* computed access vector */
decided     /* permissions for which allowed contains a decision */
expiration  /* expiration time for access vector */
status      /* error status from the request */
```

4.2.2.3 Description The processing of this request is highly dependent upon the security policy. Some simplifications are possible because of the MLS/TE policy supported by the Flask prototype:

- The *requested* vector is ignored and the returned *decided* vector always includes all permissions because the policy is not stingy.
- The *expiration* time is always set to zero so that security server decisions never expire.
- Computing the permissions is simply a lookup in the state of the security server and does not affect the state of the security server.

There are two successful cases in the processing of this request. In either case, the client must have `SS_Compute_Access_Vector` permission to both *ssid* and *tsid*.

- If both of the input SIDs represent root security contexts, then the root policy is consulted directly.
- If one or both of the input SIDs represent extension contexts, then the extension policy logic must be consulted, which may ultimately lead back to the root policy as well.

These cases are considered in more detail below.

THEORY *compute_access_vector*

```
compute_access_vector : THEORY

BEGIN

  IMPORTING root_policy_compute
  IMPORTING extension_policy_compute
  IMPORTING extension_policy_routines

  client_sid : VAR SID
  current_time : VAR nat
  state : VAR SS_STATE
%inputs
  sclass, tclass : VAR SECURITY_CLASS
  ssid, tsid : VAR SID
%outputs
  allowed : VAR ACCESS_VECTOR
  status : VAR STATUS

%unlike all of the other routines, we specify particular output values
%even in the error case. This is only because it makes the
%specification a little easier to write.

  Compute_Access_Vector(client_sid, current_time, state,
                        sclass, ssid, tclass, tsid, allowed, status) : bool =
      status = Error
    OR
      (status = Ok
       AND avc_allows(ss_avc(state), Subject_Class, client_sid, SS_Class, ssid,
                      SS_Compute_Access_Vector_Perm, current_time)
       AND avc_allows(ss_avc(state), Subject_Class, client_sid, SS_Class, tsid,
                      SS_Compute_Access_Vector_Perm, current_time)
       AND member(ssid, existing_sids(state))
       AND member(tsid, existing_sids(state))
       AND isroot(context(state)(ssid))
       AND isroot(context(state)(tsid))
       AND allowed = root_policy_compute(state, sclass, context(state)(ssid), tclass, context(state)(tsid)))
    OR
      (status = Ok
       AND avc_allows(ss_avc(state), Subject_Class, client_sid, SS_Class, ssid,
                      SS_Compute_Access_Vector_Perm, current_time)
       AND avc_allows(ss_avc(state), Subject_Class, client_sid, SS_Class, tsid,
                      SS_Compute_Access_Vector_Perm, current_time)
       AND valid_sid(state, ssid)
       AND valid_sid(state, tsid)
       AND allowed = extension_policy_compute(state, sclass, ssid, tclass, tsid))

  END compute_access_vector
```

4.2.2.3.1 Computing a Root Policy Decision

There are four steps in making a root policy decision in the MLS/TE prototype security server:

- The (source class, source type, target class, target type) 4-tuple is looked up in the *mls_vectors_defined*. If no MLS vectors are defined for the 4-tuple, then an empty access vector is returned. This is not an error case, as there is simply no need to store empty access vectors.
- If MLS vectors are defined for the 4-tuple, then the initial access vector is chosen from this set according to the following rules:

- If the source class is *Subject_Class*, then the range of the source context is compared to the high level of the target context as described in Section 4.1.2.
- Otherwise the high level of the source is compared to the high level of the target.

The permissions allowed by this initial access vector may however be reduced in the following two exceptional cases.

- Additional requirements for any permissions in *priv_auth_perms* must be considered. These permissions are only allowed if at least one of the following conditions hold:
  - The source authentication context contains the target authentication context, as described in Section 4.1.3.
  - The type from the source context is privileged, i.e., is in the set *priv_auth_types*.
- Additional requirements for any permissions in *priv_mls_perms* must be considered. These permissions are only allowed if at least one of the following conditions hold:
  - The target context has a single level range.
  - The type from the source context is privileged, i.e., is in the set *priv_mls_types*, and the level range of the source context contains the level range of the target context.

In the current Fluke system, there are three kinds of services for which these additional requirements are defined, creating an object, relabeling an object, and impersonating another subject in reliable IPC.

THEORY *root_policy_compute*

```
root_policy_compute : THEORY

BEGIN

  IMPORTING ss_state

  state : VAR SS_STATE
  sclass, tclass : VAR SECURITY_CLASS
  scontext, tcontext : VAR SECURITY_CONTEXT

% note the calling function should verify that the input contexts are
% root contexts

  root_policy_compute(state, sclass, scontext, tclass, tcontext) : ACCESS_VECTOR =
    LET stype = typ(mcontext(scontext)),
        ttype = typ(mcontext(tcontext)) IN
    IF NOT member((sclass, stype, tclass, ttype), mls_vectors_defined(state))
    THEN emptyset
    ELSE
      LET mls_vectors = mls_vectors(state)(sclass, stype, tclass, ttype),
          srange = range(mcontext(scontext)),
          trange = range(mcontext(tcontext)) IN
      LET mls_relation = IF (sclass = Subject_Class)
                         THEN mls_range_relation(srange, high(trange))
                         ELSE mls_level_relation(high(srange), high(trange))
                         ENDIF IN
      LET initial_av = CASES mls_relation OF
                         EQUAL  :  equal(mls_vectors),
                         DOM    :    dom(mls_vectors),
                         DOMBY  :  domby(mls_vectors),
```

```
                          INCOMP : incomp(mls_vectors)
                        ENDCASES IN
    LET updated_av = IF acontext_contains(acontext(scontext), acontext(tcontext))
                          OR member(stype, priv_auth_types(state))
                        THEN initial_av
                        ELSE difference(initial_av, priv_auth_perms(state)(sclass, tclass))
                        ENDIF IN
    IF high(trange) = low(trange)
       OR (member(stype, priv_mls_types(state))
           AND mls_range_contains(srange, trange))
    THEN updated_av
    ELSE difference(updated_av, priv_mls_perms(state)(sclass, tclass))
    ENDIF
  ENDIF

END root_policy_compute
```

4.2.2.3.2 Computing an Extension Policy Decision Security decisions between SIDs when either or both are not associated with a root context can be rather complex. We begin by describing the default decision making process, and then discuss how this default process may be altered and some complications that arise.

To describe the default security decision process, we consider all root contexts to be siblings. This allows the process to be separated into three cases:

- If the two SIDs have the same parent SID, or one is the parent of the other, then the access vector is simply looked up in the *ext_perms* function. If this function is not defined for the two SIDs, then an empty access vector is returned.
- If one of the SIDs is a direct ancestor of the other, then the access vector is the intersection of access vectors along the path between the SIDs. For instance, suppose A is a child of B and B is a child of C (see Figure 2). Then the access vector between A and C is the intersection of the access vectors between A and B and between B and C, with *sclass* and *tclass* unchanged, according to *ext_perms*. If *ext_perms* is not defined for any pair of SIDs, an empty access vector is returned.

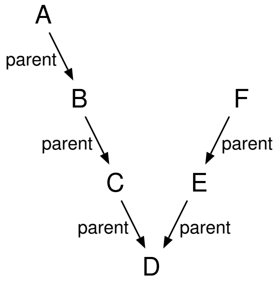

Figure 2: SID Parent Relationships

- If neither of the previous cases hold, then there are a unique pair of ancestors which are siblings (and not identical). A path exists from *ssid* to *tsid* through their parents and passing through this unique pair of ancestor siblings. For instance, suppose A is a child of B is a child of C is a child of D, and F is a child of E is a child of D, as in Figure 2. Then the path from A to F is A-B-C-E-F. As we will discuss below, it is essential that the path not include the common ancestor D.

The access vector between the two SIDs is the intersection of the access vectors along each step in this path, with the *sclass* and *tclass* remaining unchanged. Each pair is computed according to *ext_perms*, except for the case that the siblings are root contexts, in which case the root policy is consulted. If *ext_perms* is undefined for any other pair, an empty access vector is returned.

If we think of the hierarchy of SIDs as parallel to a process hierarchy, we see why it is essential that the path not include the common ancestor (D in the example). A process interacts in very different ways with its parent than with its siblings. There may well be operations that are allowed on siblings but not on the parent. Similarly, there may be communications allowed between a child and its parent which are not allowed by two siblings.

This default logic relies upon the *ext_perms* function to define the access vectors among siblings and parents. However, as discussed in the description of *load_extension*, an extension policy can be used to define access vectors between any pair of SIDs as long as one of the SIDs is not a root SID. Defining such access vectors can allow for shortcuts in the paths described in the default decision making logic. An important requirement is that such shortcuts cannot be defined unless the access vector contains no more permissions than would be allowed by following along each individual step in the path.

However, the use of shortcuts is not as straightforward as it may initially appear. Consider the example in Figure 2 again, and suppose that shortcuts are defined between B and E and also between C and F. In this case, there are three possible paths between A and F through which the permission checks may be made (A-B-C-E-F, A-B-E-F, and A-B-C-F), and none of these three paths incorporate both of the shortcuts.

This is a potentially serious problem: since one of the purposes of shortcuts is to further limit permissions, all possible shortcuts should be incorporated into the permission checking.

Editorial Note:
SCC has presented a proposal for expanding the flexibility of extension policies, and among other things this proposal would ensure that all shortcuts are considered. The proposal will be discussed in the Final Report [16].

THEORY *extension_policy_compute*

```
extension_policy_compute : THEORY

BEGIN

  IMPORTING root_policy_compute
  IMPORTING extension_policy_routines

% This theory is the only place where the "depth" element of a
% security context is used.
%
% While it would be possible to explicitly define the depth of a
% valid_sid from other elements of the model, there does not appear to
% be any simple way to do this using PVS. Since the problems with
% shortcuts are still to be resolved, this specification may yet
% change in such a way that depth could be removed. Therefore we
% haven't gone through the efforts that would be necessary to define
% depth directly.
```

```
  state : VAR SS_STATE
  context : VAR SECURITY_CONTEXT
  sclass, tclass : VAR SECURITY_CLASS

valid_child_context(state, context) : bool =
    NOT isroot(context)
    AND LET parent = parent(context) IN
        member(parent, existing_sids(state))
    AND depth(context(state)(parent)) < depth(context)

% The extension_policy_compute function presented here is recursive
% and therefore does not immediately map to the more direct
% computation described in the text.  While it would be possible
% to make a more direct model, that would require adding a
% considerable amount of machinery in order to define "paths".
% Instead, we have chosen to define extension_policy_compute as a
% relatively direct model of the implementation, and in particular, of
% the function compute_access_vector_helper.

  extension_policy_compute_measure(state, sclass, (ssid : (existing_sids(state))),
                                   tclass, (tsid : (existing_sids(state)))) : nat =
    depth(context(state)(ssid)) + depth(context(state)(tsid))


  extension_policy_compute(state, sclass, (ssid : (existing_sids(state))),
                           tclass, (tsid : (existing_sids(state)))) : RECURSIVE ACCESS_VECTOR =
    LET scontext = context(state)(ssid),
        tcontext = context(state)(tsid) IN
    LET sparent = parent(scontext),
        tparent = parent(tcontext) IN
    IF (isroot(scontext) AND isroot(tcontext))
      THEN root_policy_compute(state, sclass, scontext, tclass, tcontext)
    ELSIF member((sclass, ssid, tclass, tsid), ext_perms_defined(state))
      THEN ext_perms(state)(sclass, ssid, tclass, tsid)
    ELSIF (sparent = tparent OR sparent = tsid OR tparent = ssid)
      THEN emptyset
    ELSIF (valid_child_context(state, scontext)
           AND depth(scontext) >= depth(tcontext)
           AND member((sclass, ssid, tclass, sparent), ext_perms_defined(state)))
      THEN intersection(ext_perms(state)(sclass, ssid, tclass, sparent),
                        extension_policy_compute(state, sclass, sparent, tclass, tsid))
    ELSIF (valid_child_context(state, tcontext)
           AND depth(tcontext) >= depth(scontext)
           AND member((sclass, tparent, tclass, tsid), ext_perms_defined(state)))
      THEN intersection(ext_perms(state)(sclass, tparent, tclass, tsid),
                        extension_policy_compute(state, sclass, ssid, tclass, tparent))
      ELSE emptyset
    ENDIF
    MEASURE extension_policy_compute_measure

% There is really only one significant difference between
% extension_policy_compute and the implementation function
% compute_access_vector_helper.  By adding the
%       "AND member(...,ext_perms_defined...)"
% condition to the final two ELSIF, we avoid having to call
% extension_policy_compute to lookup a permission between a sid and
% its parent.  This simplifies the function
% extension_policy_compute_measure, which otherwise would need
% a separate definition for computing parent-child permissions.  This
% difference also explains the need for the final "ELSE emptyset"
% which has nothing corresponding in the implementation.

  END extension_policy_compute
```

4.2.3 sid_to_context

This request is called to return the security context associated with a particular SID.

4.2.3.1 Input Parameters

```
sid   /* sid to be translated */
```

4.2.3.2 Output Parameters

```
mcontext   /* mandatory context associated with the input SID */
acontext   /* authentication context associated with the input SID */
econtext   /* extension context associated with the input SID */
status     /* error status from the request */
```

4.2.3.3 Description This request observes the state of the security server but does not affect the state of the security server. For the request to be successful the client must have *SS_Sid_To_Context* permission to the input SID and the input SID must be valid. A successful request returns the security context associated with the input SID. The extension context is returned as the sequence of the names from all extension contexts on the path from a root SID to the input *sid*.

THEORY *sid_to_context*

```
sid_to_context : THEORY

BEGIN

  IMPORTING extension_policy_routines

  client_sid : VAR SID
  current_time : VAR nat
  state : VAR SS_STATE
%inputs
  sid : VAR SID
%outputs
  mcontext : VAR MANDATORY_CONTEXT
  acontext: VAR AUTH_CONTEXT
  ename : VAR finite_sequence[NAME]
  status : VAR STATUS

  Sid_To_Context(client_sid, current_time, state,
                 sid, mcontext, acontext, ename, status) : bool =
       status = Error
    OR
      (status = Ok
       AND avc_allows(ss_avc(state), Subject_Class, client_sid, SS_Class, sid,
                      SS_Sid_To_Context_Perm, current_time)
       AND member(sid, existing_sids(state))
       AND sid_context_match(state, sid, mcontext, acontext, ename))
```

END *sid_to_context*

4.2.4 context_to_sid

This request is called to return the SID associated with a particular security context. If no such SID exists, and the security context is a valid root context, an attempt will be made to create a new SID for the context. New SIDs for extension contexts can only be created as part of a `load_extension` request.

4.2.4.1 Input Parameters

```
mcontext   /* mandatory context */
acontext   /* authentication context */
econtext   /* extension context */
```

4.2.4.2 Output Parameters

```
out_sid   /* sid associated with the input context */
status    /* error status from the request */
```

4.2.4.3 Description There are two successful cases in processing this request. In both cases, the client making the request must have *SS_Context_To_Sid* permission.

- An existing SID was found which matched the requested security context. The state is not changed, the matching SID is returned for *out_sid*,
- The requested context is a valid root context, and a new SID is created to match that context. The state is changed by the addition of the new SID and the new SID is returned for *out_sid*.

THEORY *context_to_sid*

---

context_to_sid : **THEORY**

BEGIN

IMPORTING *root_policy_routines*
IMPORTING *extension_policy_routines*

client_sid : **VAR** *SID*
current_time : **VAR** *nat*
old_state, *new_state* : **VAR** *SS_STATE*
%*inputs*
mcontext : **VAR** *MANDATORY_CONTEXT*
acontext : **VAR** *AUTH_CONTEXT*
ename : **VAR** *finite_sequence*[*NAME*]
%*outputs*
out_sid : **VAR** *SID*

```
status : VAR STATUS

Context_To_Sid_bool(client_sid, current_time, old_state, new_state,
                    mcontext, acontext, ename, out_sid, status) : bool =
    (status = Error
     AND new_state = old_state)
  OR
    (status = Ok
     AND avc_allows(ss_avc(old_state), Subject_Class, client_sid, SS_Class, ss_sid(old_state),
                    SS_Context_To_Sid_Perm, current_time)
     AND new_state = old_state
     AND member(out_sid, existing_sids(old_state))
     AND sid_context_match(old_state, out_sid, mcontext, acontext, ename))
  OR
    (status = Ok
     AND avc_allows(ss_avc(old_state), Subject_Class, client_sid, SS_Class, ss_sid(old_state),
                    SS_Context_To_Sid_Perm, current_time)
     AND length(ename) = 0
     AND valid_root_context(old_state, mcontext, acontext)
     AND NOT member(out_sid, existing_sids(old_state))
     AND new_state = create_root_sid(old_state, mcontext, acontext, out_sid))

END context_to_sid
```

4.2.5 transition_sid

This request is called to return the SID computed from a transition rule. Transition rules are used to define the SID for newly created objects of certain classes where the new SID is not simply copied from an existing subject or object. As an example, when a file is executed by an existing process, the SID of the new process is computed using a transition rule on the SID of the existing process and the SID of the file.

4.2.5.1 Input Parameters

```
ssid    /* source sid */
tsid    /* target sid */
class   /* object class */
```

4.2.5.2 Output Parameters

```
out_sid   /* sid computed from the transition rule */
status    /* error status from the request */
```

4.2.5.3 Description

There are five successful cases in processing of this request. In all cases, the client making the request must have *SS_Transition_Sid* permission to both *ssid* and *tsid*, and *class* must be one of the classes for which transitions are defined.

- There are three successful cases when the two input SIDs are associated with root security contexts.

- If there is a transition defined for the types associated with the input SIDs, then a new context is constructed from the level range and authentication context of the source and the type indicated by the *root_transition* function. Like the `context_to_sid` routine, there are now two further cases:
  - ∗ An existing SID matching the new security context is returned for *out_sid*, and the state is unchanged.
  - ∗ The new security context is a valid root context, and a new SID is created to match that context. The state is changed by the addition of the new SID and the new SID is returned for *out_sid*.
- If there is no transition defined, then *out_sid* is chosen from *ssid* and *tsid* according to default rules. The current default rules are to choose *ssid* if *class* is *Process_Class* and *tsid* otherwise.

■ There are two successful cases when at least one of the two input SIDs is not associated with a root security context. In both of these cases, there is no change to the state.

- There is an *ext_transition* defined for the pair, which is returned as *out_sid*.
- There is no *ext_transition* defined for the pair, and *out_sid* is computed according to the default rules described above.

THEORY *transition_sid*

```
transition_sid : THEORY

BEGIN

  IMPORTING root_policy_routines
  IMPORTING extension_policy_routines

  client_sid : VAR SID
  current_time : VAR nat
  old_state, new_state : VAR SS_STATE
%inputs
  ssid, tsid : VAR SID
  class : VAR SECURITY_CLASS
%outputs
  out_sid : VAR SID
  status : VAR STATUS


  default_transition(ssid, tsid, class) : SID =
  IF class = Process_Class
  THEN ssid
  ELSE tsid
  ENDIF


% the calling function should verify that the input sids are actually
% root sids

  root_transition_sid_success(old_state, new_state, (ssid : (existing_sids(old_state))),
                                              (tsid : (existing_sids(old_state))), class, out_sid) : bool =
    LET scontext = context(old_state)(ssid),
          tcontext = context(old_state)(tsid) IN
    LET styp = typ(mcontext(scontext)),
          ttyp = typ(mcontext(tcontext)),
```

```
      new_acontext = acontext(scontext) IN

     (member((styp, ttyp, class), root_transition_defined(old_state))
      AND LET new_mcontext = (# range := range(mcontext(scontext)),
                                typ := root_transition(old_state)(styp, ttyp, class) #) IN
           (new_state = old_state
            AND member(out_sid, existing_sids(old_state))
            AND root_sid_context_match(old_state, out_sid, new_mcontext, new_acontext))
          OR
           (valid_root_context(old_state, new_mcontext, new_acontext)
            AND NOT member(out_sid, existing_sids(old_state))
            AND new_state = create_root_sid(old_state, new_mcontext, new_acontext, out_sid)))
   OR
     (NOT member((styp, ttyp, class), root_transition_defined(old_state))
      AND new_state = old_state
      AND out_sid = default_transition(ssid, tsid, class))


  ext_transition_sid_success(old_state, new_state, (ssid : (existing_sids(old_state))),
                             (tsid : (existing_sids(old_state))), class, out_sid) : bool =

    (member((ssid, tsid, class), ext_transition_defined(old_state))
     AND new_state = old_state
     AND out_sid = ext_transition(old_state)(ssid, tsid, class))
   OR
    (NOT member((ssid, tsid, class), ext_transition_defined(old_state))
     AND new_state = old_state
     AND out_sid = default_transition(ssid, tsid, class))


  Transition_Sid_bool(client_sid, current_time, old_state, new_state,
                      ssid, tsid, class, out_sid, status) : bool =
    (status = Error
     AND new_state = old_state)
   OR
    (status = Ok
     AND avc_allows(ss_avc(old_state), Subject_Class, client_sid, SS_Class, ssid,
                    SS_Transition_Sid_Perm, current_time)
     AND avc_allows(ss_avc(old_state), Subject_Class, client_sid, SS_Class, tsid,
                    SS_Transition_Sid_Perm, current_time)
     AND (class = Process_Class OR class = File_Class OR class = Dir_Class)
     AND member(ssid, existing_sids(old_state))
     AND member(tsid, existing_sids(old_state))
     AND
        (
           (isroot(context(old_state)(ssid))
            AND isroot(context(old_state)(tsid))
            AND root_transition_sid_success(old_state, new_state, ssid, tsid, class, out_sid))
          OR
           (NOT (isroot(context(old_state)(ssid))
                 AND isroot(context(old_state)(tsid)))
            AND ext_transition_sid_success(old_state, new_state, ssid, tsid, class, out_sid))
        )
    )

END transition_sid
```

4.2.6 member_sid

This request is called to return the appropriate SID for accessing a polyinstantiated object, i.e., for choosing the correct member of the collection of those objects.

4.2.6.1 Input Parameters

```
ssid    /* source sid */
tsid    /* target sid */
class   /* object class */
```

4.2.6.2 Output Parameters

```
out_sid   /* sid for the member object */
status    /* error status from the request */
```

4.2.6.3 Description There are two successful cases in processing of this request. In both cases, the client making the request must have *SS_Member_Sid* permission to both *ssid* and *tsid*, and *ssid* and *tsid* must be valid SIDs. Currently the request is independent of the *class* parameter.

The two successful cases are quite similar to the `context_to_sid` routine, once a member root context is identified. This root context comes from tracking each of the input SIDs back to a root SID, and combining the level range of the root context from *ssid* with the type and authentication context of the root context from *tsid*.

Then, either an existing SID matching this context is returned for *out_sid* and the state is unchanged, or if the new context is a valid root context, then a new SID may be created to match that context and returned as *out_sid*.

THEORY *member_sid*

```
member_sid : THEORY

BEGIN

  IMPORTING root_policy_routines
  IMPORTING extension_policy_routines

  client_sid : VAR SID
  current_time : VAR nat
  old_state, new_state : VAR SS_STATE
%inputs
  ssid, tsid : VAR SID
%outputs
  out_sid : VAR SID
  status : VAR STATUS

  Member_Sid_bool(client_sid, current_time, old_state, new_state,
                  ssid, tsid, out_sid, status) : bool =
       (status = Error
        AND new_state = old_state)
    OR
       (status = Ok
        AND avc_allows(ss_avc(old_state), Subject_Class, client_sid, SS_Class, ssid,
                       SS_Member_Sid_Perm, current_time)
        AND avc_allows(ss_avc(old_state), Subject_Class, client_sid, SS_Class, tsid,
                       SS_Member_Sid_Perm, current_time)
        AND valid_sid(old_state, ssid)
```

```
    AND valid_sid(old_state, tsid)
    AND EXISTS (sroot_sid, troot_sid : (existing_sids(old_state))) :
        (sid_to_root(old_state, ssid, sroot_sid)
         AND sid_to_root(old_state, tsid, troot_sid)
         AND LET new_mcontext = (# range := range(mcontext(context(old_state)(sroot_sid))),
                                   typ :=   typ(mcontext(context(old_state)(troot_sid))) #),
                 new_acontext = acontext(context(old_state)(troot_sid)) IN
              (
                 (new_state = old_state
                  AND member(out_sid, existing_sids(old_state))
                  AND root_sid_context_match(old_state, out_sid, new_mcontext, new_acontext))
               OR
                 (valid_root_context(old_state, new_mcontext, new_acontext)
                  AND NOT member(out_sid, existing_sids(old_state))
                  AND new_state = create_root_sid(old_state, new_mcontext, new_acontext, out_sid))
              )
         )
   )

END member_sid
```

4.2.7 register_avc

This request is called by a client to register an AVC with the security server, so that the security server can notify the AVC when policy changes occur (see the descriptions of `load_policy` and `load_extension`).

4.2.7.1 Input Parameters

`avc_ref`	/* a reference identifying the AVC to be registered */

4.2.7.2 Output Parameters

`status`	/* error status from the request */

4.2.7.3 Description For this request to be successful, the client must have *SS_Register_Avc* permission to the security server. If successful, a new entry is added to the set of *registered_avcs* for *avc_ref* and the SID of the client making the request.

THEORY *register_avc*

```
register_avc : THEORY

BEGIN

  IMPORTING ss_state

  client_sid : VAR SID
  current_time : VAR nat
  old_state, new_state : VAR SS_STATE
%input
```

```
  avc_ref : VAR AVC_REF
%output
  status : VAR STATUS

  Register_Avc_bool(client_sid, current_time, old_state, new_state, avc_ref, status) : bool =
    (status = Error
     AND new_state = old_state)
  OR
    (status = Ok
     AND avc_allows(ss_avc(old_state), Subject_Class, client_sid, SS_Class, ss_sid(old_state),
                    SS_Register_Avc_Perm, current_time)
     AND LET new_avc = (# avc_ref := avc_ref, client_sid := client_sid #) IN
         new_state = old_state WITH [registered_avcs := add(new_avc, registered_avcs(old_state))])

END register_avc
```

4.2.8 load_policy

This request is called by a client to load a new root policy into the security server.

4.2.8.1 Input Parameters

`path`	/* path to a file containing the new policy */

4.2.8.2 Output Parameters

`status`	/* error status from the request */

4.2.8.3 Description

For this request to be successful, the client must have *SS_Load_Policy* permission to the security server. If it is successful, a new root policy is loaded from the identified file.

Changes to the root policy must then be propagated to the other components of the security server state whenever possible. For each existing SID, the context must be verified. Each root context must be valid in the new policy, and each extension context must be traceable back to a valid root context. All invalid SIDs are removed. The extension policy is then updated to remove all permissions and transitions involving obsolete SIDs and reverifying that all permissions in the extension policy are allowable according to the new root policy (see the description of `load_extension`).

An `avc_reset` request is made on all registered AVCs.

> *Editorial Note:*
> We have not provided a formal specification of this request, primarily due to a desire to avoid the complications of the extension policy changes and recursive functions in PVS. There is also some concern that extension policies may not be defined correctly yet, as discussed in the description of `compute_access_vector`.

4.2.9 load_extension

This request is called by a client to add or remove from the current extension policy. The request includes a parent SID, and only those portions of the extension policy involving descendents of that SID can be changed.

4.2.9.1 Input Parameters

```
parent_sid   /* parent sid value */
path         /* path to a file containing the changes */
```

4.2.9.2 Output Parameters

```
status   /* error status from the request */
```

4.2.9.3 Description For this request to be successful, the client must have *SS_Load_Extension* permission to the parent SID, and the parent SID must be valid.

There are four specific kinds of changes that can be made by this request:

- New children of the parent SID can be added, by specifying the name for the extension context.
- Existing children of the parent SID can be removed. All permissions and transitions defined for the removed SID must also be removed. If the removed child SID is a parent of other SIDs, they must also be removed, and so on.
- Permissions can be added between children of the parent (either existing children or new children) and any other valid SID. These permissions must be verified according to the following rules:
  - If the permission is between two children of the same parent, or between the parent and child, then any permissions granted must be allowed between the parent SID and itself.
  - If the permission is between a child and any SID other than its parent or sibling, then a permission check is made by following the path between the two SIDs, as in the description of `compute_access_vector`. Permissions can be provided by the extension policy only to decrease the set of permissions which are already available.
- Transitions can be added between children of the parent (either existing children or new children) and any other valid SID. There are no restrictions on the definition of transitions, which is consistent with the intention that transitions are only "hints" by the security server, and are not a replacement for permission to create an object with the SID provided by the `transition_sid` request.

An `avc_reset` request is made on all registered AVCs.

Editorial Note:
Like `load_policy`, we do not provide a formal specification of this request. The reasons are basically the

same, to avoid recursive PVS definitions and to allow for further discussion of extension policies before making the effort required for a formal specification.

Section 5
Alternate Multilevel Security Policy Design

This section describes an alternate design of the root policy logic for an MLS/TE security server.

The basis for this design is an extension of the well-known Bell LaPadula (BLP) model [2] of multilevel security. This extension addresses two main problems that arise when attempting to apply the BLP model to the Flask system.

- The BLP model associates two degrees of trust with subjects. Either a subject is completely trusted and it is outside of the model or it is completely untrusted and its actions are constrained by all of the rules of the model. We extend the model to incorporate subjects which are trusted to handle data from multiple levels.
- The BLP model considers only operations between a subject and an object. We extend the model to incorporate IPC operations between subjects.

The most significant change to the design which this model facilitates is the decoupling of the MLS and TE policies. In the current design, these policies are linked because TE is used to identify those subjects which are trusted to violate the BLP rules.

This extended model and the alternate design are presented in the following sections:

- Section 5.1 discusses the labeling of subjects and objects in the extended model.
- Section 5.2 describes the set of system operations for the model, and the system requirements on each operation.
- Section 5.3 discusses an approach for performing a formal analysis of the model.
- Section 5.4 discusses the effects of failures of trusted subjects to satisfy their security requirements, and how these effects can be constrained by the system.
- Section 5.5 describes how the model allows the decoupling of type enforcement from MLS.
- Section 5.6 presents an informal specification for a security server incorporating the elements of this model. The specification is described by the differences between it and the specification of Section 4
- Section 5.7 compares the model presented here with other security models which have presented rules for subjects which handle data at multiple (but not necessary all) levels.

5.1 Labeling of Subjects and Objects

In the BLP model, all objects are labeled with a single level that identifies the maximum classification for data contained in the object and subjects are labeled with a single clearance level which indicates the highest classification of data which the subject can process. The level of a subject does not indicate any trust granted to the subject, as the system is required to constrain all subjects from reading higher level data or writing to lower level objects.

Our model labels a subject not with a single level but with a range of levels between a subject's *low* and *high* levels. A *singlelevel* subject, like a subject in the BLP model, is untrusted. However, a *multilevel* subject (having different low and high levels) is trusted to process data at all of its levels without mislabeling any data (see Section 5.4 for a discussion of how the effects of violations of this trust can be limited).

The fundamental motivation for introducing multilevel subjects is to allow for a finer control over the level of trust granted to a subject. The following two scenarios provide specific examples of the need for this flexibility.

- In a network of trusted computer systems, different systems may be cleared to handle data at different levels due to the environments in which the systems are placed (even if the systems are otherwise identical). Consider what happens when a trusted subject on a system cleared to SECRET attempts to interact with a system cleared to TOP SECRET, if all subjects are labeled with a single level, as in the BLP model. The TOP SECRET system is forced to treat the remote subject as either a singlelevel subject or a completely trusted subject. The first case may be unacceptable because a singlelevel label may not give the subject the permissions which it requires. The second case is unacceptable because it provides an attacker the opportunity to access TOP SECRET data by compromising a SECRET system.
- Practical limitations of using COTS (or legacy) applications on secure systems may lead to a decision to grant limited trust to such applications. For instance, in a system with more than a few levels, it may be impractical to polyinstantiate applications (or their configuration files) at every level. A simple example of such an application is an X server, which may be trusted to provide services to singlelevel clients having different (though related) categories in a single sensitivity level.[8]

Objects are similarly labeled with a range of levels. The primary example of a multilevel object is the private state of a multilevel subject, though a multilevel object may also be used to share data between multilevel subjects, and if necessary, the data format itself would identify the level of individual items in the object.

The level range of an object indicates not only the maximum classification of data stored in the object, but also indicates an integrity constraint. In particular, modification of a multilevel object may affect a multilevel subject's ability to meet its requirements.

5.2 System Operations and System Requirements

The BLP model defines two basic operations, read and write, which a subject can perform on an object. It identifies one system requirement on each operation. Though the operations are between a subject and an object, a subject may be treated as an object when it is being observed or altered by another subject.

We add two IPC operations to the model, *signal* and *connect*, which are performed directly from one subject to another. Subjects may also use impersonation when performing either of these operations.

This section describes each of these operations and identifies the system requirements which must be met for each operation to be performed.

[8] As is typical, a level is considered here to consist of a sensitivity level and a collection of categories.

5.2.1 Read Operation

The read operation has the same intuitive interpretation as in the BLP model. We only provide limits on read operations to prevent illegal downward flow of data. Unlike the Biba integrity model [4], we do not view the operation of a "higher" integrity subject (such as a multilevel subject) reading a "lower" integrity object (such as a single level object) as a prohibited operation. Instead, the multilevel subject is responsible for recognizing that data in a single level object is of unverified origin and for treating it appropriately.

Therefore the system requirement on a read operation is an immediate extension of the Simple Security Property of BLP:

> A subject shall be allowed to read data from an object only if the subject's high level dominates the object's high level.

5.2.2 Write Operation

A write operation is performed on an object anytime its contents are changed. We provide limits on write operations both to prevent downward flow of data and to prevent corruption of higher integrity objects.

Since singlelevel objects are low integrity objects, the requirement on writing to single level objects is an immediate extension of the *-property of BLP:

> A subject shall be allowed to write data to a singlelevel object only if the object's level dominates the subject's low level.

The integrity of multilevel objects is protected by restricting the write permission to subjects of higher trust. This integrity restriction is stronger than the *-property and is therefore also sufficient to address information flow.

> A subject shall be allowed to write data to a multilevel object only if the object's level range is a subset of the subject's level range.

5.2.3 Impersonation

As described in the Security Architecture section of the FSPM [17], a subject in Flask can use impersonation to alter its security identity as seen by another subject in an IPC operation.[9] In our model, impersonation can be used to limit the level range of a subject. Impersonation is not an operation in itself, but the ability to use impersonation is key in the derivation of the requirements on the signal and connect operations.

Impersonation can be performed by either subject involved in an IPC operation, though in very different ways. The subject initiating the IPC operation, the *client*, specifically requests impersonation when initiating the operation. The *server* subject is identified to the client by a port which represents the subject. The server achieves impersonation by creating the port with a particular level range.

[9] Since read and write operations on non-kernel objects are requested through IPC, impersonation is also indirectly used for these read and write operations.

The requirement on impersonation is actually a requirement on the IPC operation where the impersonation is used:[10]

> A subject can only impersonate a level range which is contained within the subject's own level range.

5.2.4 Signal Operation

A signal operation is one-way transfer of data directly from a client to a server, with no backward data flow or even receipt notification.

The requirement on this operation is derived by treating the signal operation as the combination of a write operation by the client to an interim object followed by a read operation by the server. At first glance, it appears that the operation could be allowed if there is any object label such that the client can write to the object and the server can read from the object. However, to meet their own requirements the client and server must be able to determine an unambiguous label for the data.

We therefore interpret the interim object as labeled by the client subject (or by another label chosen by the client using impersonation). Since the client can always write to an object with the same label, this leads to a requirement similar to the requirement on the read operation:

> A client subject shall be allowed to signal to a server subject only if the server subject's high level dominates the client subject's high level.

5.2.5 Connect Operation

A connect operation is similar to a signal operation except that a response is received by the client. The requirement can also be easily derived by considering the operation to be a series of read and write operations by the client and server to a single object labeled like the client. This leads to the following requirement:

> A client subject shall be allowed to connect to a server subject only if the client's level range is a subset of the server's level range.

By viewing a connect operation as a traditional client/server interaction, this requirement can be justified more directly by noting that it requires that a client only receive services from a server which is more trusted than the client. A client which wishes to request services from a less trusted server can do so, but only by using impersonation to limit its level range. Impersonation in this case can be considered to be an implicit acknowledgement that the server is less trusted.

5.3 Analysis of the Model

We would like to justify the model by demonstrating that downward flow of information is prevented if all of the requirements on a system and multilevel subjects are satisfied. It is natural to approach this using the framework of noninterference, by following Rushby's demonstration that a system satisfying the BLP model satisfies the noninterference condition [9].

[10] Note that in Flask, impersonation by the server must be checked as part of the IPC operation, not the operation of creating a port, since ports can be moved between subjects.

In attempting to follow this example, a significant difficulty arises: demonstrating that the unwinding conditions hold (the step consistency and local respect conditions in particular) is dependent upon requirements on multilevel subjects. In essence, what must be demonstrated is that the multilevel subjects themselves concurrently satisfy a noninterference property.

However, the specific statement of the property which must be satisfied by the multilevel subjects is unclear. This problem was identified in Section 7 of the DTOS Covert Channel Analysis Plan [12] in the context of a multiserver operating system. Indeed, this problem exists for any MLS system (even one using the simple Bell LaPadula model) having subjects which are trusted to violate one or more of the policy rules.

5.4 Ramifications of Subject Failure

As discussed above, multilevel subjects may be incorporated into a system for practical considerations even though it may be acknowledged that the subjects may not be completely trustworthy, i.e., they may not meet the requirements placed upon them. In this section we discuss how the effects of such failures are constrained.

Since the essential requirement on a multilevel subject is that it be able to properly segregate data at all of the levels, the ramification of a failure of a multilevel subject is limited to the "mixing" of data among the levels of the subject. Unfortunately, there is a well-known "cascade" problem that can occur as data cascades through a collection of multilevel subjects with overlapping levels. The cascade problem was first identified in networks (see e.g., pages 249-252 of the Trusted Network Interpretation or "Red Book" [7]).

Thus the cumulative effect of multiple failures is that the set of levels in the system which are effectively separated is reduced by each failure. Mathematically, this "effective" set of levels can be represented by a partition of the original level set into a collection of levels where the level range of each of the failed subjects is completely contained in one of the sets in the partition. Since each of the failed subjects is a singlelevel subject with respect to this effective set of levels, there are no requirements on the failed subjects and therefore the system satisfies the MLS policy for the effective set of levels.

In practice, this provides a means for risk mitigation. In the simplest case, an acceptable effective set of levels may be predetermined and used to limit the levels of an "untrustworthy" subject which it may be functionally necessary to execute as a multilevel subject. More generally, a combination of functional necessity and trustworthiness could be considered along with various possible effective sets of levels.

5.5 Relationship with Type Enforcement

In the current design, the connection between type enforcement and MLS is reflected in two elements of the state of the security server (see Section 4.1).

- Type enforcement is used to identify subjects which are trusted to violate the basic BLP rules, through the function $mls\_vectors$.
- Type enforcement is used to identify subjects which are "privileged" for certain permissions, through the set $priv\_mls\_types$.

The model presented here eliminates the need for both of these explicit couplings between MLS and type enforcement by incorporating all permissions and all "violations" of BLP directly into the model.

However, the decoupling of MLS and type enforcement does not eliminate the value of type enforcement. Type enforcement is still a valuable mechanism for many purposes, including:

- Least privilege among subjects and objects which would otherwise be allowed to interact according to the MLS rules. For instance, TE can be used to further constrain the damage caused by a failure of a multilevel subject to correctly segregate data of various levels.
- Type enforcement is a more general integrity mechanism. The MLS model presented here only addresses integrity of multilevel objects, and then only in the context of the multilevel information flow policy. As a specific example, type enforcement can be used to protect trusted subject executables.
- Type enforcement can be used as a mechanism for enforcing other information flow policies which may coexist with MLS, such as a trusted pipeline.

5.6 Security Server Specification

In this section we describe changes to the security server specification of Section 4 to accommodate the decoupling of MLS and type enforcement. The changes are limited to the following four areas:

- We add relationships among permissions, while there are no such relationships in the existing specification.
- The comparison of mandatory contexts in Section 4.1.2 is changed.
- The description of security server state in Section 4.1.4 is simplified.
- The computation of a root policy decision in Section 4.2.2 is changed.

We discuss each of these in a separate subsection. Note that the design presented here is intended to be simple, and it is likely that optimizations can be made by limiting the various combinations of permissions or by judiciously precomputing some elements of the policy decisions.

5.6.1 Permissions

The starting point for this implementation is to recall that the requirements on the operations described in Section 5.2 were derived by considering all operations to be a combination of read and write operations. In other words, the permissions required for each operation are a combination of read and write permissions, though the read and write permissions may be from the target to the source (e.g., signal and connect).

We therefore introduce four $BASE\_PERMISSIONS$, *Read*, *Write*, *Readby* and *Writeby*.[11] All other permissions are assumed to be some combination of these four base permissions, which is reflected by the function *base_permissions*. For the kernel permissions, one choice of these combinations is the following:

- $base\_permissions(READ\_PERM) = base\_permissions(EXEC\_PERM) = \{Read\}$

[11] It is not enough to simply consider *Read* and *Write*, because permissions are computed between an ordered pair of subjects or objects, so that *Read* and *Readby* are distinct.

- $base\_permissions(WRITE\_PERM) = base\_permissions(CREATE\_OB\_PERM) = \{Write\}$
- $base\_permissions(SPECIFY\_CLIENT\_PERM) = base\_permissions(SPECIFY\_SERVER\_PERM) = \{Read, Write\}$
- $base\_permissions(SEND\_PERM) = \{Readby\}$
- $base\_permissions(CONNECT\_PERM) = base\_permissions(CALL\_PERM) = base\_permissions(THREAD\_SCHEDULER\_PERM) = base\_permissions(TASK\_KEEPER\_PERM) = base\_permissions(MAP\_PERM) = \{Readby, Writeby\}$

These definitions are not absolute and may be site-specific. For instance, a site which is particularly concerned about covert channels may add *Writeby* to *SEND_PERM* since an `ipc_send` request returns an error if the destination port is not connected to a port set.

5.6.2 Mandatory Contexts

A mandatory context itself is no different in this specification. Two levels are compared in the same way, though incomparable (INCOMP) levels are of no interest. However, level ranges are compared in a different way in this design. In particular, we define four possible relationships between level ranges corresponding to the four base permissions, and define a boolean function to represent each of these relationships.

In the notation here, s and t are levels, while $=, >, <$ are used to indicate the $EQUAL$, DOM, and $DOMBY$ relationships between levels. $s.high$ indicates the high level of s, and so on. We always require that the low level of a level range is equal to or dominated by the high level of the range.

- $read(s, t) = True$ if $s.high \geq t.high$
- $write(s, t) = True$ if $s.low \leq t.low = t.high$ or if $s.low \leq t.low \leq t.high \leq s.high$
- $readby(s, t) = read(t, s)$
- $writeby(s, t) = write(t, s)$

5.6.3 Security Server State Data

The following changes, which separate type enforcement from MLS, are made to the security server state data:

- *mls_vectors* is replaced by *te_vector*, which returns a single access vector reflecting the type enforcement policy.
- The domain of the function *mls_vectors*, represented by *mls_vectors_defined*, is replaced with *te_vectors_defined*.
- *priv_mls_perms* and *priv_mls_types* are removed because the special MLS permissions are represented within the extended model.

5.6.4 Computing A Root Policy Decision

A root policy decision is computed as follows:

- The *ssid* and *tsid* are converted to types and level ranges, *stype*, *srange*, *ttype* and *trange*.
- The (*sclass*, *stype*, *tclass*, *ttype*) 4-tuple is looked up in *te_vectors_defined*. If no TE vector is defined for the 4-tuple, then an empty access vector is returned. Otherwise the defined vector is the initial access vector which will be input into the MLS logic.
- The four relationships between *srange* and *trange* are computed. For each relationship which does not hold, all permissions which require the relationship to hold (according to *base_permissions*) are removed from the initial access vector.
- Finally, additional requirements for any permissions in *priv_auth_perms* are considered as in the existing design.

5.7 Comparison With Other Models

Our model is primarily motivated by the two shortcomings of the classical Bell and LaPadula security model which were enumerated in the introduction to this section. The first of these shortcomings, the inability to specify a limited range of trust for subjects, has been addressed in other previous efforts which also provide for multilevel subjects. Here we compare our model to three published models which include multilevel subjects.

The GEMSOS security kernel [11] assigns a high and low level to each subject, and a single level to each object. There are effectively two operations which are controlled in the model, read and write. These operations are controlled by the same rules that were presented in Sections 5.2.1 and 5.2.2. Our model is an extension of the GEMSOS model in that it also allows a range of levels to be assigned to an object and it addresses subject-to-subject operations.

In applying the original Bell and LaPadula model to network operations, Bell [1] introduces a level range for a subject. Bell's condition for allowing a read or a write operation is slightly more restrictive than ours: in addition to our condition, the level of the object must be comparable to all levels in the range of the subject. For instance, suppose a subject's *high* level has a sensitivity level of SECRET and categories A and B, it's *low* level has a sensitivity level of SECRET and the single category A, and an object has a sensitivity level of SECRET and the single category B. In our model, the subject would be allowed to read the object, while in Bell's model it would not be allowed. It is not clear whether this example is realistic and if so, which model would be preferable.

Otherwise, Bell's model differs from ours in that it defines several subject-to-object operations which we do not consider, but it does not allow for multilevel objects and it does not define any subject-to-subject operations.

The Trusted Mach (TMach) system [20] extended the Bell and LaPadula model in similar ways, and was a specific motivation for our model. Like the previous two examples, TMach assigns a range of levels to a subject and a single level to an object. Read and write operations are allowed under the same conditions as in our model and as in the GEMSOS model.

TMach also considers an operation between subjects that is somewhat analogous to the connect operation described in Section 5.2.5, and is controlled by the TMach permission BLP_NONE. However, this operation is allowed as long as the ranges of the two subjects have a non-empty intersection, rather than requiring the range of the client to be a subset of the range of the server as in our model. As discussed in Section 5.2.5, we believe that our more restrictive requirement is appropriate, and the impersonation feature of Flask allows this stricter requirement to be enforced without limiting the functions of the system.

To summarize, our model extends these earlier efforts by allowing multilevel objects and by considering the signal and impersonate operations. The only differences when our model overlaps with these other models are the requirements for reading and writing in Bell's model and the requirements for the connect operation in TMach.

Section 6

Dynamic Security Policy Study

This section provides a model for reasoning about the ability of a computing system to support dynamic security policies, and in particular changes to the security policy which restrict operations which were previously allowed. The model is based upon the definition of a security policy as "a set of rules used to decide whether a particular service is allowed to be provided by a system." A dynamic security policy is a security policy in which the set of rules changes over time.

The essential difficulty of supporting a dynamic security policy is that security decisions (made according to the rules of the policy) are often retained within the system for use at a later time. A simple example of this which occurs in essentially all modern operating systems is the storage of memory permissions within page tables that are queried by the hardware when memory accesses are attempted. When the set of rules are changed to be more restrictive, all such retained decisions must be removed or recomputed.

The model is presented in the following sections:

- Section 6.1 defines the fundamental property that a system must satisfy if it is to support dynamic security policies. The essence of this property is that all services provided by the system must be allowable according to the security policy in force at the time.

- Section 6.2 defines the state model upon which all further analysis will be based. This state model is intended to be rich enough to provide useful insight while still being general enough to be applicable to a wide range of system security mechanisms.

- Section 6.3 refines the fundamental property into subproperties that reflect more naturally the relationships between the various elements of the state model. These properties are primarily stated as state invariants which must hold in the system.

- Section 6.4 further refines the subproperties into lower level requirements. The purpose of this refinement is to refine the state invariants into requirements on the state transitions made by the system.

- Section 6.5 provides a further refinement of those requirements which simultaneously control the security server and those object managers which enforce the decisions of the security server.

- Section 6.6 discusses a different approach to demonstrating the fundamental property based upon theories of database consistency. This approach actually allows for a stronger property to be proven with no additional work, though the approach has some other disadvantages.

- Section 6.7 considers the availability of the services that allow a change in policy to occur, in contrast to the previous sections which address correctness of policy changes.

All of the proofs in this section, with the exception of Section 6.6, have been specified and proven in PVS. These specifications are included in Appendix B.

6.1 The Fundamental Property

All of the effort of the dynamic security policy study is directed towards analyzing the ability of a system to satisfy the following fundamental property:

> **Fundamental_Property** — Any time a service is provided by the system, that service is allowed according to the current security policy.

The statement of this property implies that the security policy can be consulted for a yes/no decision directly when a service is provided. However, due to practical considerations (such as performance) this is an unrealistic assumption in most systems. The security policy is usually accessed indirectly through local copies, and it is the purpose of this study to provide a framework for reasoning about such systems and their ability to satisfy the fundamental property.

6.2 State Model

This section provides the state model upon which all further analysis will be based. The model is primarily influenced by our experiences with the DTOS and Flask systems, though it is also influenced by our understanding of the security mechanisms provided by other operating systems as will be described after presenting the model. While we hope that the model can also be applied to higher level services (such as database manipulations) not usually supplied by operating systems, we have not performed any analysis or looked at any specific examples to determine if this is actually possible.

Section 6.1 states that this study is intended to address systems which do not directly consult the security policy when performing every service. The model provides for this in three ways:

- The security policy is not required to be stated directly in terms of the objects involved in a service, but is instead allowed to be stated in terms of security contexts (e.g., label) associated with the objects. A level of indirection between an object and its security context is also allowed.
- The security policy is not required to be stated directly in terms of the service itself, but is instead allowed to be stated in terms of permissions that are associated with the service.
- Policy enforcement is not required to directly access the security policy but is instead allowed to access copies of the policy which are retained from previous direct accesses.

The model logically separates the overall state of the system into two components, a security server which makes security decisions and an object manager which provides services and enforces security decision based upon retained copies of the decisions. This separation is convenient when discussing the model and is also useful within some Flask specific analysis that will be performed later in this report (see Section 6.5). However, this logical separation need not parallel any physical separation within an actual system implementation, and therefore does not limit the range of applicability of the model.

Figure 3 presents the state of the security server, which includes the following elements:

- The security server includes the current security policy. As discussed above, the security policy need not be represented in terms of objects and services, but instead in terms of security contexts and permissions. Note that the function *current_perms* identifies those

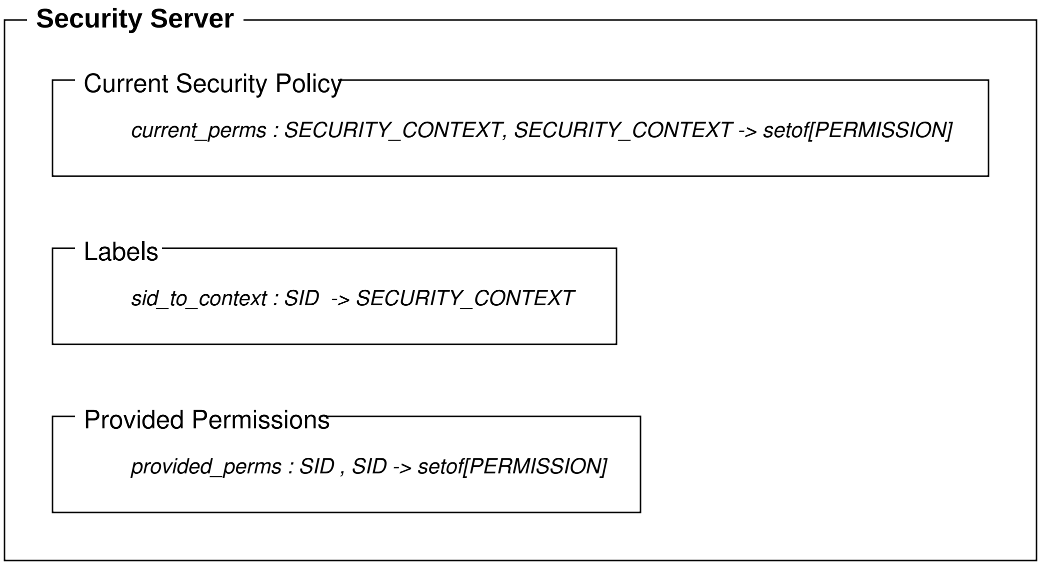

Figure 3: Security Server State Model

permissions currently allowed by the policy, but does not attempt to identify all of the variables that are used to compute the currently allowable permissions, as only the currently allowed permissions are relevant to the fundamental property.

- The security server includes one step of the indirection in labeling between an object and its security context. A security identifier (SID) is a policy-independent data type used by the object manager (see Figure 4). The security server maintains the mapping between each SID and a policy-dependent security context.
- The security server provides security decisions to the object manager in terms of SIDs rather than security contexts, since security contexts may be policy dependent. We include in the model a data structure, *provided_perms*, to allow the security server to record all permissions which it has provided to the object manager.

Figure 4 presents the state of the object manager, which includes the following elements:

- The object manager knows the SID for some collection of objects. Since the only state required is the SID, the set of objects is not limited to those objects which are actually “managed” by the object manager, but can include any object (such as clients of the object manager) for which the object manager has identified a SID.
- The object manager has a control policy that defines the permissions which are required for each service to be performed.
- The object manager accesses the security policy through various collections of retained permissions. We define three distinct categories of retained permissions:
  - *retained_perms_by_sid* — Permissions in this set are associated with a pair of SIDs, which is the way in which the object manager accesses the security policy from the security server. The access vector cache used in Flask is an example of this category of retained permission.

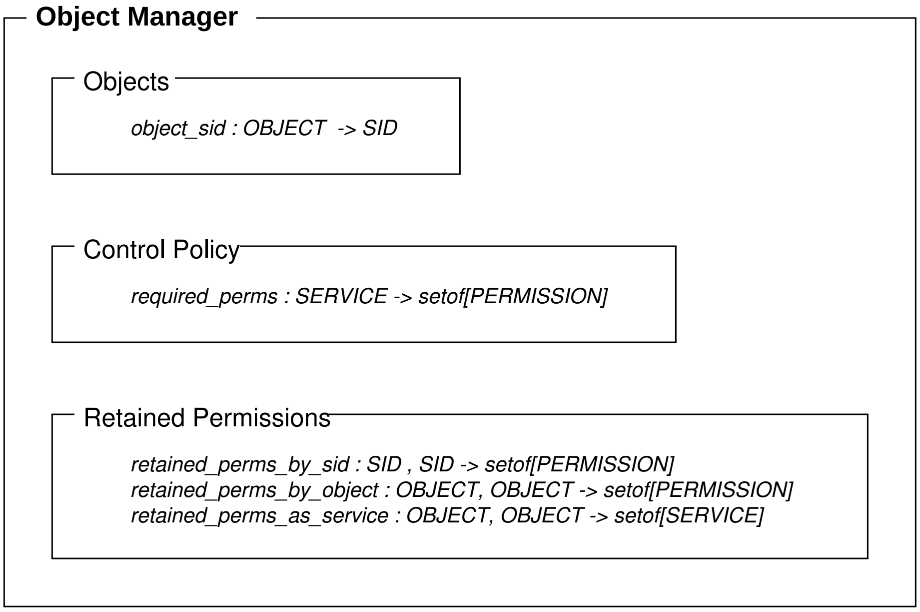

Figure 4: Object Manager State Model

- *retained_perms_by_object* — Permissions in this set are associated with a pair of objects, which is closer to the way in which the object manager enforces the security policy. The page table in an operating system kernel retains permissions between active subjects and memory objects and is an example of this category of retained permission.
- *retained_perms_as_service* — This set does not explicitly identify permissions which are retained but actual services which are allowed to proceed. A typical example is a system which performs permission checks when an access is requested but possibly before the service is actually provided.

Though we have tried to construct a general model, there are some potential limitations:

- The model includes only a single object manager. In a system with multiple distinct object managers, all could be considered as part of one larger object manager. However, this interpretation doesn't allow for consideration of coordination and consistency among the object managers. For example, in the model part of the state of an object manager can include the SID of a client, but that SID is actually maintained by the process manager.

  In the end, for the problem under consideration this extra complexity did not seem to be justified. Note however that Section 6.6 presents an analysis which does allow for multiple object managers without any additional complexity.

- The model includes only a single security server. If there are multiple security servers which partition the security policy among them, then they could be considered logically to be a single security server, so the model applies.

  However, systems with multiple security servers sharing portions of the policy are outside of the scope of this model.

- The security policy in this model computes permissions based upon a pair of SIDs. This does not allow for permissions depending upon more than two SIDs, or upon other parameters. It would not be difficult to add this capability to the model, but it would add notational complexity without providing any additional insight into the system's ability to satisfy the fundamental property.
- Though the model allows the control policy (i.e., *required_perms*) to change, it does not include a security server version of the control policy and therefore does not allow analysis of consistency between the security server and object manager versions of the control policy. We don't consider this to be a significant shortcoming since we know of no systems for which this is a practical issue. In fact, it may even be desirable for the control policy to be able to change independent of the security server.
- The model includes three kinds of retained permissions, though clearly these are not the only possibilities. For instance, permissions could be retained as a function of an object and a SID. As a more complex example, permissions could be partially retained as a service if some but not all of the permissions for that service have been checked.

  We believe that providing these three forms of retained permissions in the model, the analysis can provide sufficient guidance for how to manage other categories of retained permissions.

We conclude this section by describing the interpretation of this model for several security architectures other than DTOS and Flask, with the goal of further clarifying the model.

- In the simplest system in which the security policy is accessed directly for a yes/no decision on each service (as described in Section 6.1), the model applies trivially as follows:
  - There is no distinction in the system between the security server and object manager.
  - There is no distinction between the identity of an object, its SID and its security context. The functions *object_sid* and *sid_to_context* are the identity function.
  - There is no distinction between a service and the permission required to perform the service. The function *required_perms* is the identity function.
  - There is no distinction between the functions *current_perms*, *provided_perms*, *retained_perms_by_sid*, *retained_perms_by_object* and *retained_perms_as_service*.
- Consider a system which incorporates an access control list (ACL) with each object in the system, and a principal (e.g., a user) with each subject, with direct access to the ACL to make each security decision. This is very similar to the previous example, except that security decisions are not based directly upon the subject making an access but upon the principal associated with the subject. Since the interpretation of the model is that the policy is accessed for enforcement according to SIDs, the ACL should be interpreted as the SID (and security context) of an object and the principal as the SID (and security context) of a subject. Therefore:
  - There is no distinction in the system between the security server and object manager.
  - The function *object_sid* maps an object into its ACL and a subject into its principal.
  - There is no distinction between a SID and a security context, so the function *sid_to_context* is the identity function.
  - There may or may not be a distinction between a service and the permission required to perform the service.

- There is no distinction between the functions *current_perms*, *provided_perms*, and *retained_perms_by_sid*, which are all computed from the ACLs.
- The interpretation of the functions *retained_perms_by_object* and *retained_perms_as_service* is dependent upon the specific system. If there is no distinction between services and permissions, then these two functions are the same.

- A "pure" capability system is similar to the previous, except:
  - There is no distinction between an object, its SID and its security context.
  - The SID (and security context) of a subject is its capability list.

  If the system has some sort of security policy which controls initial distribution of capabilities, then this would be represented by *current_perms* while *provided_perms* and *retained_perms_by_sid* would represent the capabilities which have actually been granted.

  Note however that in some capability systems, such as Amoeba, it is impossible to identify all of the capabilities which are held by any subject. The model is of little use in such a system since it would have to assume that all subjects hold all possible capabilities.

- The Spring security model presents a combination of ACLs and capabilities.[12] Two interesting aspects of Spring is the support for revocation of permissions and the interpretation of accesses being made not by subjects but by front objects.
  - There is no distinction in the system between the security server and object manager.
  - Like an ACL system, the SID (and security context) of an object is its ACL. However, the SID (and security context) of a client subject is the principal associated with the front object through which it accesses an object.
  - The *required_perms* function is an explicit mapping in the object manager.
  - Like an ACL system, there is no distinction between *current_perms*, *provided_perms* and *retained_perms_by_sid*, which are all computed from the ACLs.
  - Caching in front objects is represented in the model by *retained_perms_by_object*.
  - There is likely other less explicit caching which would be represented by *retained_perms_by_object* and *retained_perms_as_service*.

- The Trusted Mach (TMach) system [20, 13] is a little more complex still, and in particular it provides separation between object manager and security server.
  - The TMach Root Name Server (RNS) and TMach Subject Server (TMSS) together function as the security server, and each TMach item manager as an object manager.
  - The SID of an object is its item control port, since this is used to identify the object across the interface between an item manager and the RNS, though the control port would probably be modeled as identical to the object since there is a one-to-one relationship between item control ports and objects. The security context of an object consists of a security level and ACL, maintained by the RNS.
  - The SID of a subject is its security ID ("security ID" is a TMach term that corresponds to a SID). The security context of a subject consists of a range of levels, a user and a set of groups, maintained by the TMSS.
  - The *required_perms* function is an explicit mapping in each item manager.

[12] The Spring security model has not been described in any detail in any published paper. The information used here comes from the report [19] .

- *current_perms* is a function of the security context of a subject and an object.
- *provided_perms* is the set of access rights for each agent associated with an object in the RNS (which are intended to have been passed to the item managers).
- *retained_perms_by_sid* is the set of access rights for each agent associated with an object in the item manager.
- There is likely other less explicit caching which would be represented by *retained_perms_by_object* and *retained_perms_as_service*.

6.3 Refinement into Subproperties

In this section we begin by restating the fundamental property in the context of the state model, and then proceed to define a collection of five subproperties which together imply the fundamental property.

Recall that the fundamental property reads:

> Any time a service is provided by the system, that service is allowed according to the current security policy.

To interpret the first clause of this property, we need to define what it means that a "service is provided by the system". The system is modeled as a state machine which undergoes a sequence of atomic transitions. Each transition performs zero or more services, and the services provided by a transition can be determined entirely by inspection of the initial and final states of the transition. In addition, a service is assumed to involve two objects (typically a subject accessing an object).

In the second clause, the phrase "allowed according to the current security policy" requires a careful interpretation in the state model. The difficulty is that services are expressed in terms of objects, while the current security policy is expressed by *current_perms* in terms of security contexts and permissions. The mapping from services to permissions is through the function *required_perms*, and the mapping from objects to security contexts is through the two functions *object_sid* and *sid_to_context*.

Finally, since we are using a state machine model, we are left to explicitly define whether "current security policy" refers to *current_perms* in the initial state of the (atomic) transition or the final state, or both. It is clear that the initial state of *current_perms* must be considered, since otherwise any subject could give itself permission to change *current_perms*. The more difficult decision is whether to consider only the initial state or both the initial and final states. One problem with considering the final state is that this would prevent a subject from performing the service of removing its own permission to change the security policy. Because of this fact, and for consistency with the control policy in [17], we have chosen to consider only the initial state of the transition.[13]

Therefore, the following is a more specific statement of the fundamental property:

> Any time a service is provided during a transition of the system, the service is allowed, according to the security policy, in the initial state of the transition. In

[13] Note also that if we were to include the final state of *current_perms* in the statement of the fundamental property, the only change that this would make to the rest of this report is that the subproperty *Services_Provided_Property* would require the permissions to be retained in both the initial and final state of the transition. No other changes would be necessary because this subproperty is not further refined and the other subproperties are state invariants which must hold in both the initial and final states.

particular, for a service *service* involving objects *object1* and *object2*, all of the permissions required for the service according to *required_perms* must be in the set *current_perms*(*sid_to_context*(*object_sid*(*object1*)), *sid_to_context*(*object_sid*(*object2*))).

The five subproperties are derived from this fundamental property based upon the interpretation of the various elements of the state model as discussed in Section 6.2. In particular, the indirection which the model allows is the motivator for the subproperties.

The first subproperty addresses the object manager's responsibility to verify that all services are allowed according to the retained permissions:

Services_Provided_Property — Any time a service is provided during a transition of the system, the service is allowed according to the retained permissions in the initial state of the transition. In particular, for a service *service* involving objects *object1* and *object2*, one of the following must hold:

- All of the permissions required for the service according to *required_perms* must be in the set *retained_perms_by_object*(*object1*,*object2*) or the set *retained_perms_by_sid*(*object_sid*(*object1*),*object_sid*(*object2*)), or
- The service must be in the set *retained_perms_as_service*(*object1*, *object2*).

The next three subproperties are state invariants which ensure that the three kinds of retained permissions are consistent with *provided_perms*.

Retained_Perms_By_Sid_Property — In any state of the system, a permission is in *retained_perms_by_sid* only if it is in *provided_perms*.

Retained_Perms_By_Object_Property — In any state of the system, a permission is in *retained_perms_by_object* for some pair of objects *object1*, *object2* only if it is in *provided_perms*(*object_sid*(*object1*),*object_sid*(*object2*)).

Retained_Perms_As_Service_Property — In any state of the system, a service is in *retained_perms_as_service* for some pair of objects *object1* and *object2* only if all of the permissions required for the service according to *required_perms* are in the set *provided_perms*(*object_sid*(*object1*),*object_sid*(*object2*)).

The final subproperty is a state invariant to ensure that *provided_perms* is consistent with *current_perms*.

Provided_Perms_Property — In any state of the system, a permission is in *provided_perms* for some pair of SIDs *sid1* and *sid2* only if it is in *current_perms*(*sid_to_context*(*sid1*),*sid_to_context*(*sid2*)).

Figure 5 indicates how the Fundamental property is a consequence of the five subproperties. This figure will be expanded as the subproperties are further refined.

6.4 State Transition Requirements

This section further refines the state invariant subproperties into requirements on transitions of the system, which is more useful for system design and analysis. We do not further refine *Services_Provided_Property* since it is already a requirement on transitions.

Fundamental_Property

- *Services_Provided_Property*
- *Retained_Perms_By_Sid_Property*
- *Retained_Perms_By_Object_Property*
- *Retained_Perms_As_Service_Property*
- *Provided_Perms_Property*

Figure 5: Proof of Fundamental Property from Subproperties

The requirements are derived by considering how a change to each element of the state model affects each of the state invariants. If a change to that element of the model can invalidate a previously valid state invariant, then a requirement is stated which ensures that this does not occur.

State Element		State Invariant Subproperty			
		R_P_By_Sid	*R_P_By_Object*	*R_P_As_Service*	*Provided_Perms*
current_perms	add	-	-	-	-
	remove	-	-	-	X
sid_to_context	change	-	-	-	X
provided_perms	add	-	-	-	X
	remove	X	X	X	-
object_sid	change	-	X	X	-
required_perms	add	-	-	X	-
	remove	-	-	-	-
retained_perms_by_sid	add	X	-	-	-
	remove	-	-	-	-
retained_perms_by_object	add	-	X	-	-
	remove	-	-	-	-
retained_perms_as_service	add	-	-	X	-
	remove	-	-	-	-

Table 1: State Elements and Subproperties

Table 1 summarizes the results of the refinement. Each row corresponds to a change to some element of the state model. For functions which returns a set (such as *current_perms* returning a set of permissions), we distinguish between adding an element to the set and removing an element from the set. Each column corresponds to one of the state invariants. An "X" in the table indicates that a particular change can invalidate that state invariant, so a transition requirement is needed.

For instance, the first line of the table indicates that no state invariants are invalidated by simply adding a permission to *current_perms*; the policy is always allowed to become more generous. However, the second line of the table indicates that the *Provided_Perms_Property* invariant can be invalidated by removing a permission from *current_perms*; revoking a permission from the policy requires knowing that the permission not be in *provided_perms*.

Table 1 indicates that there are nine transitions for which one or more requirements are necessary, and a total of 12 requirements. In the rest of this section we list the requirements individually for each transition and discuss the practical ramifications of the requirements. In some cases, we also identify alternate requirements which may be more practical to meet in a real system though they are actually stronger than necessary. Finally, the requirements are summarized in figures displaying how the fundamental property is proven as a consequence of the requirements.

Note that the requirements in this section are named according to the following rules:

- The "root" name for each requirement is the name of the state element which is changed during a transition.
- A prefix "add", "remove" or "change" is added to the root to indicate the type of change to the state element.
- When there is more than one requirement for a state change (i.e., more than one "X" in a row of Table 1), a suffix "by_sid", "by_object" or "as_service" is added to indicate the relevant subproperty.
- Requirements which are stated as an alternate localized version of another requirement have an added "local" prefix.

6.4.1 Removing a Permission from *current_perms*

Table 2 lists the requirements which must be met in the final state of a transition during which a permission is removed from *current_perms*.

Transition : Remove *permission* from *current_perms*(*context1*,*context2*)
Final State Requirements :
remove_current_perms :
permission $\notin$ *provided_perms*(*sid1*,*sid2*)
whenever *context1*=*sid_to_context*(*sid1*) and *context2*=*sid_to_context*(*sid2*)

Table 2: Transition Requirements

The effect of this requirement is that prior to (or concurrent with) revoking permissions in the security policy, those permissions must be removed from *provided_perms*. As we shall see in Section 6.4.4, this requires coordination with the object manager so that in practice there are several requirements that must be met for a permission to be revoked.

6.4.2 Changing *sid_to_context*

Table 3 lists the requirements which must be met in the final state of a transition during which the value of *sid_to_context* is changed for some SID.

Changing the value of *sid_to_context* relabels all objects having a particular SID. The effect of the requirement is that prior to (or concurrent with) the relabeling, any permissions which were granted under the old value of *sid_to_context* but are not granted under the new value must be removed from *provided_perms*, and hence from the object manager.

Transition : Change $sid\_to\_context(sid)$
Final State Requirements :
$change\_sid\_to\_context$:
$provided\_perms(sid1, sid2) \subseteq current\_perms(sid\_to\_context(sid1), sid\_to\_context(sid2))$
whenever $sid = sid1$ or $sid2$

Table 3: Transition Requirements

Note also that while not part of this model, a client of the security server (such as an object manager) may have stored the value of *sid_to_context*, so changing its value can have side effects on these subjects.[14]

6.4.3 Adding a Permission to *provided_perms*

Table 4 lists the requirements which must be met in the final state of a transition during which a permission is added to *provided_perms*.

Transition : Add $permission$ to $provided\_perms(sid1, sid2)$
Final State Requirements :
$add\_provided\_perms$:
$permission \in current\_perms(sid\_to\_context(sid1), sid\_to\_context(sid2))$

Table 4: Transition Requirements

Adding a permission to *provided_perms* corresponds to providing the permission to the object manager. This requirement simply states that the security server should provide a permission only if it is allowed by the current security policy.

6.4.4 Removing a Permission from *provided_perms*

Table 5 lists the requirements which must be met in the final state of a transition during which a permission is removed from *provided_perms*.

The effect of these requirements is that prior to (or concurrent with) removing a permission from *provided_perms*, the object manager must have removed all retained copies of the permission. In a system with a physical separation between object manager and security server, this requires some form of notification from the object manager to the security server. In all likelihood, the object manager would only provide such notification when requested by the security server, and the security server would only request such notification if it were attempting to remove the permission from *provided_perms* due to a requested change in *sid_to_context* or *current_perms*.

[14] The current implementation provides no means of notifying such clients when the *sid_to_context* map changes, so security servers should not be making such changes.

Transition : Remove *permission* from *provided_perms*(*sid1*,*sid2*)

Final State Requirements :

remove_provided_perms_by_sid :
permission $\notin$ *retained_perms_by_sid*(*sid1*,*sid2*)

remove_provided_perms_by_object :
permission $\notin$ *retained_perms_by_object*(*object1*,*object2*)
whenever *sid1*=*object_sid*(*object1*) and *sid2*=*object_sid*(*object2*)

remove_provided_perms_as_service :
permission $\notin$ *required_perms*(*service*)
whenever *service* $\in$ *retained_perms_as_service*(*object1*,*object2*),
sid1=*object_sid*(*object1*) and *sid2*=*object_sid*(*object2*)

Table 5: Transition Requirements

6.4.5 Changing *object_sid*

Table 6 lists the requirements which must be met in the final state of a transition during which the value of *object_sid* is changed for some object. As in all of the previous sections, the "Final State Requirements" listed in the table are necessary to satisfy the state invariants. However, these two requirements involve the state of the security server as well as the state of the object manager, and it is possible to state alternate requirements which are completely local to the object manager. For instance, rather than requiring direct coordination between the functions *retained_perms_by_object* and *provided_perms*, the object manager can coordinate *retained_perms_by_object* with *retained_perms_by_sid* and then coordinate *retained_perms_by_sid* with *provided_perms*. Though this alternate form of the requirement is no longer necessary to satisfy the state invariants, it may be easier to directly satisfy (and verify) in a system implementation.

Since changing the value of *object_sid* is a relabeling of the object, it is also interesting to compare these requirements with the requirements on transitions changing the value of *sid_to_context*. In either case, the requirements only apply if some previously granted (and retained) permission is no longer allowed with the new label. The biggest difference is that changing the value of *object_sid* can potentially be completed local to the object manager with no specific knowledge of the security server (other than possibly the object manager's request for certain permissions), while changing the value of *sid_to_context* can potentially be completed local to the security server with no specific knowledge of the object manager (other than possibly the security server's request to revoke certain permissions).

6.4.6 Adding a Permission to *required_perms*

Table 7 lists the requirements which must be met in the final state of a transition during which a permission is added to *required_perms*. As in the previous section, an alternate object manager local version of the requirement is stated.

Transition : Change $\mathit{object\_sid}(\mathit{object})$

Final State Requirements :

$\mathit{change\_object\_sid\_by\_object}$:
$\mathit{retained\_perms\_by\_object}(\mathit{object1},\mathit{object2}) \subseteq \mathit{provided\_perms}(\mathit{object\_sid}(\mathit{object1}),\mathit{object\_sid}(\mathit{object2}))$
whenever $\mathit{object} = \mathit{object1}$ or $\mathit{object2}$

$\mathit{change\_object\_sid\_as\_service}$:
$\mathit{required\_perms}(\mathit{service}) \subseteq \mathit{provided\_perms}(\mathit{object\_sid}(\mathit{object1}),\mathit{object\_sid}(\mathit{object2}))$
whenever $\mathit{service} \in \mathit{retained\_perms\_as\_service}(\mathit{object1},\mathit{object2})$ and $\mathit{object} = \mathit{object1}$ or $\mathit{object2}$

Alternate Object Manager Local Final State Requirements :

$\mathit{local\_change\_object\_sid\_by\_object}$:
$\mathit{retained\_perms\_by\_object}(\mathit{object1},\mathit{object2}) \subseteq \mathit{retained\_perms\_by\_sid}(\mathit{object\_sid}(\mathit{object1}),\mathit{object\_sid}(\mathit{object2}))$
whenever $\mathit{object} = \mathit{object1}$ or $\mathit{object2}$

$\mathit{local\_change\_object\_sid\_as\_service}$:
$\mathit{required\_perms}(\mathit{service}) \subseteq \mathit{retained\_perms\_by\_sid}(\mathit{object\_sid}(\mathit{object1}),\mathit{object\_sid}(\mathit{object2}))$
$\cup\ \mathit{retained\_perms\_by\_object}(\mathit{object1},\mathit{object2})$
whenever $\mathit{service} \in \mathit{retained\_perms\_as\_service}(\mathit{object1},\mathit{object2})$ and $\mathit{object} = \mathit{object1}$ or $\mathit{object2}$

Table 6: Transition Requirements

Transition : Add $\mathit{permission}$ to $\mathit{required\_perms}(\mathit{service})$

Final State Requirements :

$\mathit{add\_required\_perm}$:
$\mathit{permission} \in \mathit{provided\_perms}(\mathit{object\_sid}(\mathit{object1}),\mathit{object\_sid}(\mathit{object2}))$
whenever $\mathit{service} \in \mathit{retained\_perms\_as\_service}(\mathit{object1},\mathit{object2})$

Alternate Object Manager Local Final State Requirements :

$\mathit{local\_add\_required\_perm}$:
$\mathit{permission} \in \mathit{retained\_perms\_by\_sid}(\mathit{object\_sid}(\mathit{object1}),\mathit{object\_sid}(\mathit{object2}))$
$\cup\ \mathit{retained\_perms\_by\_object}(\mathit{object1},\mathit{object2})$
whenever $\mathit{service} \in \mathit{retained\_perms\_as\_service}(\mathit{object1},\mathit{object2})$

Table 7: Transition Requirements

6.4.7 Adding a Permission to *retained_perms_by_sid*

Table 8 lists the requirements which must be met in the final state of a transition during which a permission is added to *retained_perms_by_sid*.

Transition : Add *permission* to *retained_perms_by_sid*(*sid1*,*sid2*)
Final State Requirements :
add_retained_perm_by_sid :
permission ∈ *provided_perms*(*sid1*,*sid2*)

Table 8: Transition Requirements

6.4.8 Adding a Permission to *retained_perms_by_object*

Table 9 lists the requirements which must be met in the final state of a transition during which a permission is added to *retained_perms_by_object*. Once again, an alternate object manager local version of the requirement is stated.

Transition : Add *permission* to *retained_perms_by_object*(*object1*,*object2*)
Final State Requirements :
add_retained_perm_by_object :
permission ∈ *provided_perms*(*object_sid*(*object1*),*object_sid*(*object2*))
Alternate Object Manager Local Final State Requirements :
local_add_retained_perm_by_object :
permission ∈ *retained_perms_by_sid*(*object_sid*(*object1*),*object_sid*(*object2*))

Table 9: Transition Requirements

6.4.9 Adding a Service to *retained_perms_as_service*

Table 10 lists the requirements which must be met in the final state of a transition during which a service is added to *retained_perms_as_service*. Once again, an alternate object manager local version of the requirement is stated.

6.4.10 Summary

The basic set of requirements stated in this section are necessary to ensure that the state invariants are not invalidated by any transition. In conjunction with the subproperty *Services_Provided_Property* and requirements that the state invariants are satisfied in the initial state of the system, the requirements are also sufficient to prove the Fundamental Property, as outlined in Figure 6. These proofs have also been completed using PVS.[15]

[15] The PVS proofs are based upon the specifications included in Appendix B.

Transition : Add *service* to *retained_perms_as_service*(*object1*,*object2*)

Final State Requirements :

add_retained_perm_as_service :

$\mathit{required\_perms}(\mathit{service}) \subseteq \mathit{provided\_perms}(\mathit{object\_sid}(\mathit{object1}), \mathit{object\_sid}(\mathit{object2}))$

Alternate Object Manager Local Final State Requirements :

local_add_retained_perm_as_service :

$\mathit{required\_perms}(\mathit{service}) \subseteq \mathit{provided\_perms}(\mathit{object\_sid}(\mathit{object1}), \mathit{object\_sid}(\mathit{object2})) \cup \mathit{retained\_perms\_by\_object}(\mathit{object1}, \mathit{object2})$

Table 10: Transition Requirements

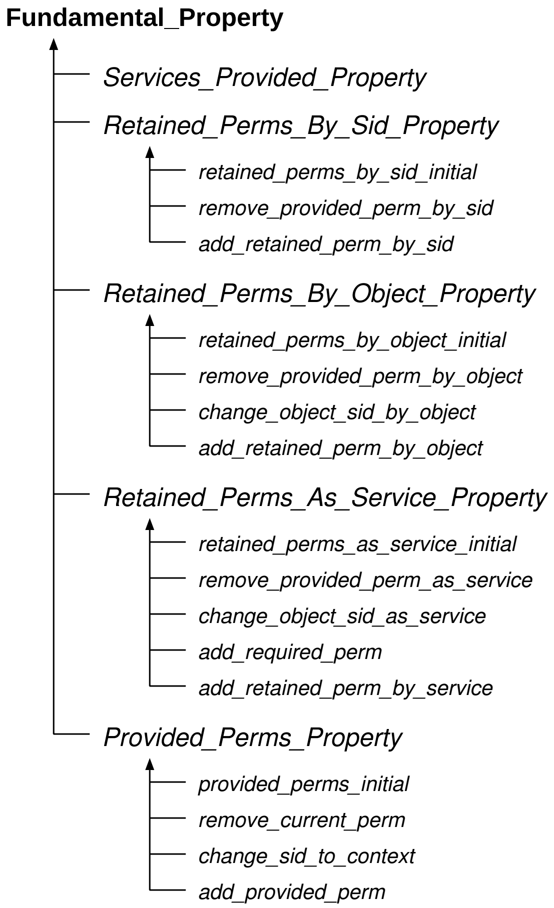

Figure 6: Proof of Fundamental Property from Requirements

Since one of our goals is to address systems in which the object manager and security server are separate components, it is desirable for each requirement to apply to one of the components but not both. Table 11 identifies the components to which each of the requirements used in the proof outlined in Figure 6 apply.

Transition Requirements	Component	Local Alternate
Services_Provided_Property	OM	-
remove_current_perms	SS	-
change_sid_to_context	SS	-
add_provided_perms	SS	-
remove_provided_perms_by_sid	Both	-
remove_provided_perms_by_object	Both	-
remove_provided_perms_as_service	Both	-
change_object_sid_by_object	Both	OM
change_object_sid_as_service	Both	OM
add_required_perm	Both	OM
add_retained_perm_by_sid	Both	-
add_retained_perm_by_object	Both	OM
add_retained_perm_as_service	Both	OM
Initial State Requirements		
retained_perms_by_sid_initial	Both	-
retained_perms_by_object_initial	Both	OM
retained_perms_as_service_initial	Both	OM
provided_perms_initial	SS	-

Table 11: State Transition and Initial State Requirements

The table also identifies those requirements for which we have stated an alternate version which applies only to the object manager. These requirements are not necessary to satisfy the state invariants but are sufficient, as depicted in the proof outlined in Figure 7

Note that it would be possible to state alternate versions of other requirements as well, including the initial state requirements *retained_perms_by_object_initial* and *retained_perms_as_service_initial*. However, the requirements *remove_provided_perms_by_sid* and *add_retained_perm_by_sid* are fundamentally dependent upon both the security server and object manager components. The following section will address how these requirements can be further refined into requirements on each of the components individually.

6.5 Synchronization Protocol

In this section, we describe a method of enforcing the Fundamental Property in systems with independent security server (SS) and object manager (OM). In a partitioned system, the knowledge one component has about the actions of the other is limited. Here, we assume that the SS and OM interact by sending messages through an unreliable medium. Specifically, we assume that messages may be lost, delivered more than once, or arrive out of order. We do not consider issues related to the secrecy, authenticity, or integrity of messages, as our main interest here is in centralized systems.

Table 11 shows a set of requirements that are sufficient to ensure the Fundamental

Fundamental_Property

- *Services_Provided_Property*
- *Retained_Perms_By_Sid_Property*
  - *retained_perms_by_sid_initial*
  - *remove_provided_perm_by_sid*
  - *add_retained_perm_by_sid*
- *Retained_Perms_By_Object_Property*
  - *retained_perms_by_object_initial*
    - *local_retained_perms_by_sid_initial*
    - *local_retained_perms_by_object_initial*
  - *remove_provided_perm_by_object*
  - *change_object_sid_by_object*
    - *Retained_Perms_By_Sid_Property*
    - *local_change_object_sid_by_object*
  - *add_retained_perm_by_object*
    - *Retained_Perms_By_Sid_Property*
    - *local_add_retained_perm_by_object*
- *Retained_Perms_As_Service_Property*
  - *retained_perms_as_service_initial*
    - *local_retained_perms_by_sid_initial*
    - *local_retained_perms_by_object_initial*
    - *local_retained_perms_as_service_initial*
  - *remove_provided_perm_as_service*
  - *change_object_sid_as_service*
    - *Retained_Perms_By_Sid_Property*
    - *Retained_Perms_By_Object_Property*
    - *local_change_object_sid_as_service*
  - *add_required_perm*
    - *Retained_Perms_By_Sid_Property*
    - *Retained_Perms_By_Object_Property*
    - *local_add_required_perm*
  - *add_retained_perm_by_service*
    - *Retained_Perms_By_Sid_Property*
    - *Retained_Perms_By_Object_Property*
    - *local_add_retained_perm_as_service*
- *Provided_Perms_Property*
  - *provided_perms_initial*
  - *remove_current_perm*
  - *change_sid_to_context*
  - *add_provided_perm*

Figure 7: Proof of Fundamental Property from Alternate Local Requirements

Property. With the exception of the initial state requirement *retained_perms_by_sid_initial* and the transition requirements *remove_provided_perms_by_sid*, *remove_provided_perms_by_object*, *remove_provided_perms_as_service* and *add_retained_perm_by_sid*, each requirement has a local alternate form that can be enforced by one of the components without assistance from the other. The initial state requirement can be satisfied through configuration of the initial state. In this section, we demonstrate how the remaining four transition requirements can be refined into requirements on each of the components by introducing a framework for a communication protocol between them. We focus only on the requirements *remove_provided_perms_by_sid* and *add_retained_perm_by_sid*, with the understanding that *remove_provided_perms_by_object* and *remove_provided_perms_as_service* can be handled similarly.

Our assumptions about the interaction between the SS and OM is shown in Figure 8. In the case of a security fault, the OM sends a query to the SS indicating a set of permissions. The response to that request indicates if the specified permission is in *current_perms*. If the permission is provided to the OM, the OM copies it into *retained_perms_by_sid* and the SS records the permission in *provided_perms*. Thus, *retained_perms_by_sid* serves as a permission cache that the OM uses to reduce communication between itself and the SS. The state component *provided_perms* serves as estimate for the SS of what permissions the OM has cached in *retained_perms_by_sid*. When the SS wants to revoke a permission, it sends a revocation request to the OM indicating the permission that should be removed. When the OM has finished removing the specified permission from *retained_perms_by_sid*, it sends an acknowledgment back to the SS.

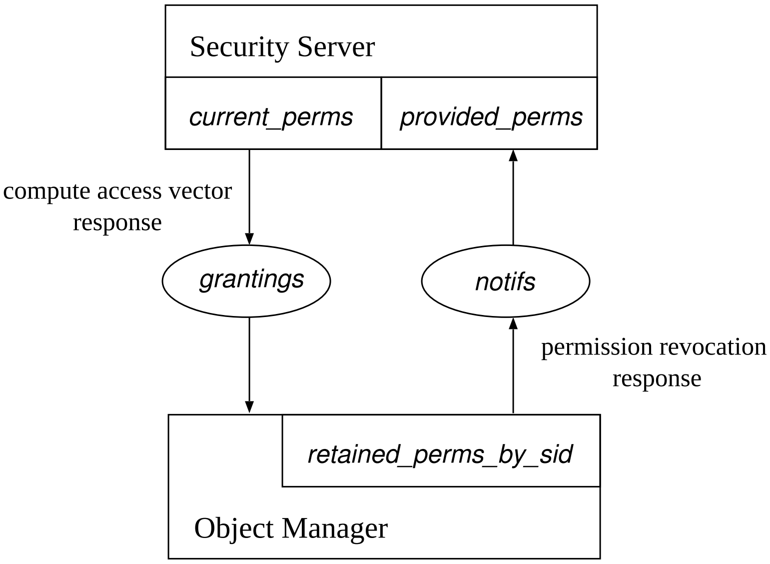

Figure 8: Interaction Between SS and OM

We will model requirements on the sending and processing of permissions and flush notifications, but we will not model the permission requests and flush requests themselves. Although these are obviously necessary in a practical protocol, they are not essential to correctness of the synchronization protocol. They serve as "hints" to the peer about permissions and flush notifications that should be sent.

In this context, to enforce *remove_provided_perms_by_sid*, we must ensure that if a permission is removed from *provided_perms* in the transition from st_1 to st_2, the permission is not in *retained_perms_by_sid* in state st_2. To enforce this requirement we ensure the following:

1. A permission is removed from *provided_perms* by the SS only after the SS has been notified of the permission's removal from *retained_perms_by_sid* by the OM.

2. A permission is not added to *retained_perms_by_sid* by the OM if the request originated at the SS before the previous revocation request.

The second condition requires that the OM be able to determine when an add permission request originated at the SS. This is accomplished by tagging messages with sequence numbers. In the case of messages from the SS, the sequence number can be used to determine the relative ordering of add permission and revoke requests at the SS.

The *add_retained_perm_by_sid* requirement states that if a permission is added to *retained_perms_by_sid* in the transition from state st_1 to st_2, that permission is in *provided_perms* in state st_2. The following conditions ensure this requirement.

1. A permission is added to *retained_perms_by_sid* by the OM only after the OM has been notified of the permission's addition to *provided_perms* by the SS.

2. A permission is not removed from *provided_perms* by the SS if the notification from the OM originated before the permission was last added by the OM.

In this case, we can use the sequence numbers on notifications from the OM to determine the relative order in which it added permissions and sent notifications.

In the remainder of this section, we expand on these sequence numbers and how they are used to enforce the *remove_provided_perms_by_sid* and *add_retained_perm_by_sid* requirements.

6.5.1 Protocol State

Here we introduce new data types and state information necessary to express the protocol requirements. The basic unit of communication is a quadruple (*sid1*, *sid2*, *permission*, *seqno1*). It is used both for security decisions sent by the SS to the OM and for notifications sent by the OM to the SS. In security decisions it signifies that the OM is allowed to add the permission to *retained_perms_by_sid* for (*sid1*, *sid2*). In addition, the SS promises to ignore any future notifs (*sid1*, *sid2*, *permission*, *seqno2*) where *seqno2* $\leq$ *seqno1*. The security server maintains a set of these permission grantings as shared state between the OM and SS.

$$grantings : setof[SID, SID, PERMISSION, SEQNO]$$

The set *grantings* represents the medium the SS uses to send messages to the OM. The SS adds elements to this set. The OM can use this set to justify adding a permission to *retained_perms_by_sid*. Our use of a set to model messages in transit is consistent with our assumption that messages can be delivered out of order. To model the fact that messages can be delivered multiple times, messages are not removed upon receipt. To capture our assumption that messages can be lost, we allow an element added in one state to disappear in some future state.

Our model is a generalization of the more familiar access vector. The requirements below will allow the SS to add an arbitrary number of quadruples (with no mandatory relationship between SID-pairs and sequence numbers) to grantings in any one transition. Sending an access vector (augmented with a sequence number) corresponds to the special case of adding a set of quadruples that share the same SID-pair and seqno. Given the data type for some of the state elements to be described later, this generalization makes some things simpler. Also, it makes clear that there is nothing in the analysis that hinges on an interface using access vectors.

For notifications a quadruple (*sid1*, *sid2*, *permission*, *seqno1*) indicates that the OM has removed all instances of (*sid1*, *sid2*, *permission*) in *retained_perms_by_sid* and promises to ignore any future grantings (*sid1*, *sid2*, *permission*, *seqno2*) where *seqno2* $\leq$ *seqno1*. The set of these notification quadruples constitute state shared between the OM and SS.

$$notifs : setof[SID, SID, PERM, SEQNO]$$

This represents the medium used to transfer messages from the OM to the SS. The OM adds elements to this set. The SS can use this set to justify reductions in *provided_perms*. Requirements on these operations are described below. As with *grantings* the SS may use a *notif* zero or more times. In addition, we allow a *notif* to disappear from this set at any time.

One piece of new private data is needed in the SS:

$$latest\_granting : [SID, SID, PERM- > SEQNO]$$

This is used to record the largest sequence number that has been used when placing a granting for (*sid1*, *sid2*, *permission*) in *grantings*.

We also need a new piece of private state in the OM:

$$latest\_notif : [SID, SID, PERM- > SEQNO]$$

This models the largest sequence number that has been used when placing a notification for (*sid1*, *sid2*, *permission*) in *notifs*.

6.5.2 Object Manager Requirements

The object manager must meet an initial state requirement and four transition requirements. The initial state requirement that applies to the OM is:

> *notifs_initial* — In every valid initial state st of the object manager,
>
> *notifs*(st) = $\emptyset$.

The requirement *latest_notif_mono* is a transition requirement that ensures that *latest_notif* is never decreased.

> *latest_notif_mono* — For all state transitions (*st1*, *st2*) of the object manager, and for all (*sid1*,*sid2*,*permission*),
>
> *latest_notif*(*st2*)(*sid1*,*sid2*,*permission*) $\geq$ *latest_notif*(*st1*)(*sid1*,*sid2*,*permission*)

The requirement follows from our intention that *latest_notif* represent the largest *seqno* of all *notifs* sent by the OM.

The next transition constraint applies when an element is added to *notifs*.

send_notif — If (*sid1*, *sid2*, *permission*, *seqno*) is added to *notifs* in the transition from *st1* to *st2* then

a) *latest_notif*(*st2*)(*sid1*, *sid2*, *permission*) $\geq$ *seqno*; and

b) (*sid1*, *sid2*, *permission*) $\notin$ *retained_perms_by_sid*(*st2*).

Part a) ensures that *latest_notif* for the permission is an upper bound on all notifications it has sent to the security server. Condition b) ensures that if a notification is sent, the permission is in fact removed from *retained_perms_by_sid*. Both conditions must be in force by state *st2*.

The final requirement applies when a permission is added to *retained_perms_by_sid*.

use_granting_by_sid — If (*sid1*, *sid2*, *permission*) is added to *retained_perms_by_sid* in the transition from *st1* to *st2*, then

(*sid1*, *sid2*, *permission*, *seqno*) $\in$ *grantings*(*st1*) where *seqno* > *latest_notif*(*st1*)(*sid1*,*sid2*,*permission*)

The constraint requires that if a permission is added to *retained_perms_by_sid*, then there is a *granting* for the permission with a *seqno* greater than *latest_notif*. This constraint must hold by state *st1*. This ensures that after the OM issues a *notif* for a permission *permission*, it only honor requests to add the permission again, if the *seqno* on the request is greater than that of the original *notif*.

6.5.3 Security Server Requirements

There are two initial state requirements that apply to the security server. The first pertains to the *grantings* state component.

grantings_initial — In every valid initial state st of the security server,

grantings(st) = $\emptyset$.

The second, constrains the *provided_perms* state component in such a way that the Fundamental Property holds in the initial state.

provided_perms_initial — In every valid initial state st of the security server,

If (*sid1*, *sid2*, *permission*) $\in$ *retained_perms_by_sid*(st), then (*sid1*, *sid2*, *permission*) $\in$ *provided_perms*(*st*).

In addition to these constraints, there are three transition constraints. The first requirement *latest_granting_mono* ensures that *latest_granting* is never reduced.

latest_granting_mono — For all state transitions (*st1*, *st2*) of the security server, and for all (*sid1*,*sid2*,*permission*),

latest_granting(*st2*)(*sid1*,*sid2*,*permission*) $\geq$ *latest_granting*(*st1*)(*sid1*,*sid2*,*permission*)

This requirement is similar in motivation to *latest_notif_mono*.

The *send_granting* requirement applies when the security server adds an entry to *grantings*.

> *send_granting* — If (*sid1*, *sid2*, *permission*, *seqno*) is added to *grantings* in the transition from *st1* to *st2* then
>
> **a)** *latest_granting*(*st2*)(*sid1*, *sid2*, *permission*) $\geq$ *seqno*; and
>
> **b)** (*sid1*, *sid2*, *permission*) $\in$ *provided_perms*(*st2*).

It ensures first of all that *latest_granting* is an upper bound on the *seqno* 's of all grantings it has sent to the OM. In addition, it ensures that if a granting is sent, the permission is in *provided_perms* by state *st2*.

The final requirement applies when a permission is added to *retained_perms_by_sid*.

> *use_notif* — If (*sid1*, *sid2*, *permission*) is removed from *provided_perms* in the transition from *st1* to *st2*, then
>
> (*sid1*, *sid2*, *permission*, *seqno*) $\in$ *notifs*(*st1*) where *seqno* > *latest_granting*(*st1*)(*sid1*,*sid2*,*permission*)

Here we require that if a permission is removed from *provided_perms*, then in state *st1*, there must be a *notif* for it with a *seqno* greater than the *latest_granting*. The purpose is to ensure that if the SS sends a *granting* to the OM for a permission, it only process *notifs* with a *seqno* greater than that of the original *granting*.

6.5.4 Properties of the Composition

In this section we make the argument that the requirements we have outlined are sufficient to ensure that the Fundamental Property holds in all reachable states. We begin by introducing some useful lemmas.

Lemma- *latest_notif_correct*
For all reachable states st, if (*sid1*, *sid2*, *permission*, *seqno*) $\in$ *notifs* then *latest_notif*(*sid1*, *sid2*, *permission*) $\geq$ *seqno*.

Proof- The proof follows directly from *send_notif* and *latest_notif_mono*. □

Lemma- *latest_granting_correct*
For all reachable states st, if (*sid1*, *sid2*, *permission*, *seqno*) $\in$ *grantings* then *latest_granting*(*sid1*, *sid2*, *permission*) $\geq$ *seqno*.

Proof- The proof follows directly from *send_granting* and *latest_granting_mono*. □

Before presenting the next lemmas, we need to define some terms to simplify their presentation.

Definition- A notification (*sid1*, *sid2*, *permission*, *seqno*) is a *usable_notif* if (*sid1*, *sid2*, *permission*, *seqno*) $\in$ *notifs* and *seqno* > *latest_granting*(*sid1*, *sid2*, *permission*).

Definition- A granting (*sid1*, *sid2*, *permission*, *seqno*) is a *usable_granting* if (*sid1*, *sid2*, *permission*, *seqno*) $\in$ *grantings* and *seqno* > *latest_notif*(*sid1*, *sid2*, *permission*).

By referring to the requirements on the SS and the OM, we see that a *usable_notif* is necessary to justify removing the corresponding permission from *provided_perms*. Similarly, a *usable_granting* is needed to justify adding a permission to *retained_perms_by_sid*.

The next lemma states that it is not possible to have a *usable_notif* and a *usable_granting* in the same state.

Lemma- *usable_notif_usable_granting_nand*
For all reachable states st, there cannot exist both a *usable_granting* and a *usable_notif* for (*sid1*, *sid2*, *permission*) in state st.

Proof- The proof is by contradiction and is based on the following observations:

1. The value of *latest_granting* (*latest_notif*) in state st is greater or equal to the *seqno* of every *granting* (*notif*) for the same permission in state st. (*latest_granting_correct*)

2. Having a *usable_notif* implies that a *notif* exists with a *seqno* sn_n that is strictly greater than *latest_granting* for the corresponding permission. (Definition of *usable_granting*)

3. A *usable_granting* implies that a *granting* exists with a *seqno* sn_g that is greater than *latest_notif*. [16] (Definition of *usable_notif*)

Combining these results, we are left with:

$$\begin{aligned} latest\_granting(prot(st))(sid_1, sid_2, perm) &\geq sn_g \\ &> latest\_notif(prot(st))(sid_1, sid_2, perm) \\ &\geq sn_n \\ &> latest\_granting(prot(st))(sid_1, sid_2, perm) \end{aligned}$$

This points out the contradiction we were seeking. □

The next lemma states that if a *usable_notif* exists, then there is no corresponding entry in *retained_perms_by_sid*.

Lemma- *usable_notif_not_in_retained_perm_by_sid*
In all reachable states st, if a *usable_notif* (*sid1*, *sid2*, *permission*, *seqno*) exists in st then (*sid1*, *sid2*, *permission*) $notin$ *retained_perms_by_sid*(st).

Proof- The proof of this is by induction over the execution trace. The property holds in the initial state because by *notifs_initial*, *notifs* is empty. Now, assume this is true for $st = st_1$ and that (st_1, st_2) satisfies the protocol requirements. We assume that a *usable_granting* with a sequence number of *sn* exists. We must show that the permission is not in *retained_perms_by_sid* in state st_2. Assume for the sake of a contradiction, that it is. In state st_1 *retained_perms_by_sid* is true by the induction hypothesis. Thus, one of three things must have been True in st_1.

1. There was no *notif*; or

2. The value of *latest_granting* is less than or equal to *sn*; or

3. The permission is not in *retained_perms_by_sid*.

In the first case, the condition *send_notif* implies that the permission is not retained in state st_2. But this contradicts our previous assumption.

The second case would require that *latest_granting* be smaller in st_2 than in st_1. This contradicts the *latest_notif_mono* requirement.

[16] The requirement imposed in the PVS model given in the appendix is less stringent only requiring that $sn_g \geq$ *latest_notif*

Finally, we consider the third case. From *use_granting_by_sid* we know that there must have existed a *usable_granting* for the permission in state st_1. This is contradicted by Lemma *usable_notif_usable_granting_nand*. In all three cases we reach a contradiction. Thus, we must assume that *usable_notif_not_in_retained_perm_by_sid* holds in st_2 as well, and by induction, it must hold in all states. □

The next lemma is similar to the previous one. It states that whenever there is a *usable_granting*, the corresponding permission must be in *provided_perms*.

Lemma- *usable_granting_in_provided_perm*
In all reachable states st, if a *usable_granting* (*sid1*, *sid2*, *permission*, *seqno*) exists in st then (*sid1*, *sid2*, *permission*) $\in$ *provided_perms*(st).

The proof of this lemma is similar to that for *usable_notif_not_in_retained_perm_by_sid* but makes use of the *send_granting*, *latest_granting_mono* and *use_notif* requirements. We will not present it here.

The next two theorems are the main goal of this section. We will show that the *remove_provided_perms_by_sid* and *add_retained_perm_by_sid* requirements hold for all state transitions.

Theorem- *remove_provided_perms_by_sid*
For all state transitions (st_1, st_2), if (*sid1*, *sid2*, *permission*) is removed from *provided_perms* during the transition, then (*sid1*, *sid2*, *permission*) $notin$ *retained_perms_by_sid*(st_2).

Proof- If a permission is removed from *provided_perms* during the transition from st_1 to st_2, a *usable_notif* must exist for the permission in state st_1. This follows because the machine satisfies the *use_notif* requirement. Lemma *usable_notif_not_in_retained_perm_by_sid* tells us that the permission is not in *retained_perms_by_sid* in st_1. Thus, if the permission is to be in *retained_perms_by_sid* in state st_2, it must be added during the transition. Assume for the sake of a contradiction that it is added during the transition. Then from *use_granting_by_sid* we know that a *usable_granting* for the permission must exist in state st_1. However, by Lemma *usable_notif_usable_granting_nand* we know that it is impossible for state st_1 to have both a *usable_notif* and a *usable_granting* for the same permission. Thus, we are forced to conclude that the permission is not in *retained_perms_by_sid* in state st_2. □

Now, we consider *add_retained_perm_by_sid*.

Theorem- *add_retained_perm_by_sid*
For all state transitions (st_1, st_2), if (*sid1*, *sid2*, *permission*) is added to *retained_perms_by_sid* during the transition, then (*sid1*, *sid2*, *permission*) $\in$ *provided_perms*(st_2).

Proof- The justification for this Theorem is similar to the argument made for *remove_provided_perms_by_sid*. From the condition *use_granting_by_sid* we know that a *usable_granting* must exist in st_1. The Lemma *usable_granting_in_provided_perm* implies that the permission is in *provided_perms* in state st_1. If the permission is removed from *provided_perms* during the transition to st_2, we know that there must also be a *usable_notif* (condition *use_notif*). But this possibility is contradicted by Lemma *usable_notif_usable_granting_nand*. Thus, the permission must remain in *provided_perms* in state st_2. □

The relationship between the principal theorems and lemmas discussed in this section is shown in Figure 9.

6.6 Transactional Consistency

The protocol described in Section 6.5 allows the security policy to be modified in a manner that provides all parties with a consistent view of the security policy. The consistency ensured is that the policy maintained by an Object Manager (OM) is at least as restrictive as the policy

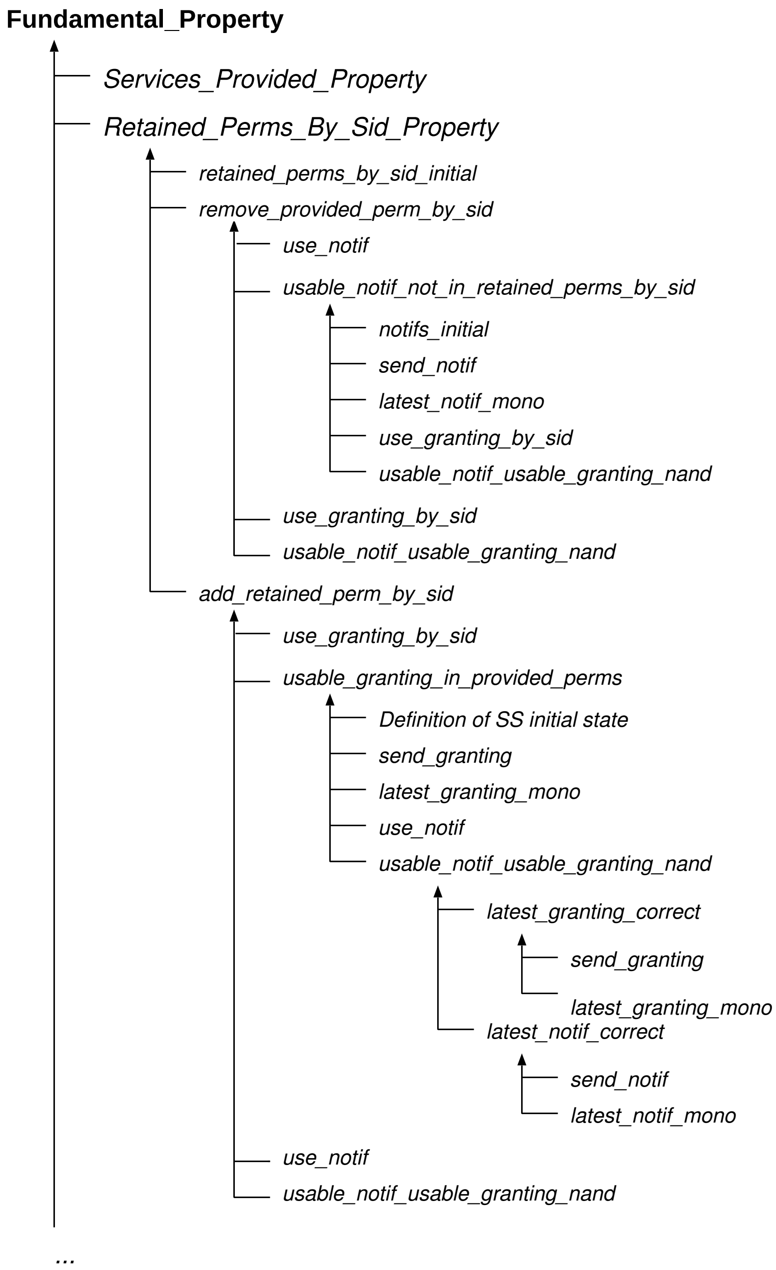

Figure 9: Structure of Proof from Protocol Requirements

maintained by the Security Server (SS). A policy is modeled as a set of permissions that may or may not be granted. It does not distinguish between permissions that have not yet been granted and those that have been specifically denied. We can model denied permissions to a degree, by always responding in the negative when the permission is requested. However, the denial is not cached at the OM. The next time the question arises, the OM will ask again.

The model we discuss in this section imposes a more complex model of consistency. In this model, a permission is not a boolean quantity (i.e., granted or denied) but instead we associate an abstract value with each potential permission. This allows us to model not only the presence or absence of a permission as in the original model, but other characteristics as well. This allows us to model "denials" explicitly. We can use the value of the permission to indicate that the permission is "granted", "denied" or "undecided". Thus, using this model it is possible for an OM to cache permission denials as well as grants.

The consistency model we propose here ensures that the effect of the caching is not apparent to the parties accessing the policy database. In addition, it appears to each party as if they are accessing the policy directly from the SS and that they have exclusive access to the SS.

The proposed consistency model also simplifies the construction of security servers supporting dynamic security policies. We allow policy modifications to be structured in the form of transactions allowing the policy designer to ignore the effects of concurrency and certain types of failures.

The work in this section represents an attempt to fit our requirements for synchronization into an existing model of consistency. In this way we hope to borrow the solutions from that model to help solve our problem. In this way we avoid having to develop these solutions from scratch. We examine Transactional consistency in this report, but solutions related to cache and distributed shared memory consistency may also serve as useful models.

6.6.1 Background

In this section we provide a brief background on transaction processing concepts. The coverage of the theoretical aspects of concurrency control is based on [3]. Our discussion of design and implementation issues is based on [6].

A transaction is a set of database access operations that are guaranteed (by the creator of the transaction) to preserve consistency. We model a database as a set of data elements, denoted x, y, etc. A transaction T_i is modeled as a sequence[17] of read operations, denoted $r_i[x]$, or write operations $w_i[x]$ followed by either c_i indicating the commitment of the transaction, or a_i denoting its abort. A history H is a sequence of operations from one or more transactions indicating the order in which the operations were executed. The committed projection of a history H, denoted $C(H)$ is H, with all operations from transactions T_i that are not committed in H (i.e., $c_i \in H$) removed.

When data elements can be replicated we need to change our model slightly. A logical data element x such as what a transaction may access is represented in the database as a collection of replicas x_1, x_2, ..., x_n maintained at sites S_1, S_2, ..., S_n respectively. In a replicated data history (RDH) accesses are to replicas (e.g., x_i) rather than data elements (e.g., x). Every $r_i[x]$ in T_i is translated to $r_i[x_j]$ and $w_i[x]$ is translated to $w_i[x_{i1}]$, $w_i[x_{i2}]$, ..., $w_i[x_{im}]$ in the replicated data history where the actual replicas read or written are determined by the replica control protocol. The most common protocol reads the local replica of an item and writes all available replicas.

[17] More precisely, a transaction need only be a partial order of operations. To simplify the presentation here we will assume that transactions and histories are totally ordered.

A history H is serial if none of the transactions are interleaved. A history H_1 is *view equivalent*[18] to H_2 if they both contain the same operations and every transaction T_i in H_1 reads-x-from T_j if and only if it does so in H_2. A history H is serializable if there exists a serial history H_S that is view equivalent to $C(H)$. A serial one-copy history is a serial RDH in which whenever a transaction T_i reads-x-from T_j, T_j is the last transaction to write any copy of x before T_i. We say a history is one-copy-serializable (1SR) if it is view equivalent to a serial one copy history. One-copy-serializability ensures that replication is transparent to the transactions accessing the database because the transaction reads the same values it would in a database with only one copy. The concurrency is transparent because the execution is equivalent to a serial execution, i.e., one in which the transaction executed in isolation.

A serialization graph is often used to argue formally about properties of histories. The serialization graph for a history H, denoted $SG(H)$, is a graph with nodes corresponding to the committed transactions in H. There is an edge from T_i to T_j in $SG(H)$ if an operation $o_i[x]$ precedes (in H) and conflicts with some other operation $o_j[x]$. For non-replicated databases, a history H is serializable if and only if $SG(H)$ is acyclic (see [3]).

A commonly used method for concurrency control is two-phase locking (2PL). The protocol requires that before an element is accessed, an appropriate lock on the data element is acquired on behalf of the transaction. This lock must be maintained at least until the access is complete. Once a transaction releases a lock, it must not acquire any more. Strict 2PL is a commonly used variation in which no lock is released until the transaction commits.

The terms lock and latch are used as in [6]. Both locks and latches come in several varieties including exclusive and shared. Latches provide no means for detecting deadlock as it is assumed that some policy for avoiding deadlocks will be used. Locks on the other hand do provide some mechanism for detecting deadlock (this could be as complex as constructing a waits-for graph and searching it for cycles, or as simple as a timeout mechanism).

6.6.2 Assumptions

Here we discuss the assumptions that led us to this design. By Fluke semantics[21], an operation may execute in one or more user-visible atomic stages. We require that all policy changes be made atomically with respect to these stages.

Transactions are used to access the security database. These transactions (both at the SS and OMs) must be synchronized to ensure one-copy serializability. Following a failure, the security database must be restored to either the last committed state of the database or at least to some acceptable consistent state.

Our design is based in part on the assumption that locking, deadlock detection, and atomic commitment protocols (e.g., Two Phase Commit) incur too much overhead for use in OM. However, the Security Server can support these protocols if necessary. For this reason the design recognizes several classes of transactions that access the policy database, including those dedicated to (1) policy modification, (2) copying information into caches in the OM, and (3) executing operations under the control of the cached version of the policy.

6.6.3 Interpretation

The SS and a collection of OMs constitute a distributed collection of database systems. The security policy stored in *current_perms* is the database we wish to ensure the consistency of. The

[18] This is not complete, but close enough for our purposes here.

elements of the database describe permissions. The SS maintains a replica of every element in the security policy database. Additional replicas may be retained by the OMs. Each OM can maintain several copies of the same element representing the same permission retained in multiple caches (e.g., *retained_perms_by_sid*, *retained_perms_by_object*, and *retained_perms_as_service*). Additionally, multiple OMs may cache the same element. In this case, the OMs manage objects with a common security identifier.

All accesses to the *current_perms* database is done in the form of a transaction. There are three general types of access transactions. The first type is executed at the SS and is used to modify *current_perms*. It updates the replica maintained at the SS and invalidates the others. The second type executes exclusively at an OM. It represents a thread executing a user-visible atomic stage of an operation. If one or more of the required permissions are not cached at the OM, the transaction aborts and is retried when after the permissions have been placed in a local cache. We refer to the third type of transaction as a *copier*. These transactions handle security faults. They create a local replica at the OM by reading the value of another replica, either the primary copy at the SS, or another copy of the same element at the OM.

6.6.4 Design

The goal here is to avoid adding complexity to the OMs. Most of the complexity and overhead will be borne by the SS. We assume that the SS will execute strict two-phase locking (2PL) for local transactions. Update transactions must acquire locks on all database (DB) elements they access and hold those locks until they commit.

The caching of database contents at the various OMs presents a special type of replication. It is simpler to deal with because of the fact that cached replicas are not required to ensure durability and so they can be destroyed at any time [5]. In addition, in our application, they are never updated.

6.6.4.1 Caching

The SS maintains the primary copy of every permission. We model this with the following state component:

current_perms: *SECURITY_CONTEXT*, *SECURITY_CONTEXT*, *PERMISSION* -> *VALUE*

We have modified the original *current_perms* state component. The original definition is equivalent to the definition above if *VALUE* is replaced with *BOOLEAN* (the *BOOLEAN* value indicates membership of the permission in *current_perms*). To manage the replicas that arise through caching, we maintain a directory at each SS:

provided_copies: *SECURITY_CONTEXT*, *SECURITY_CONTEXT*, *PERMISSION*, *OM* -> *BOOLEAN*

that indicates which OMs have potentially cached the permission denoted by a source context, a target context and a permission. This is similar in purpose to the *provided_perms* of the last section. The only significant difference is the fact that we accommodate caching at more than one OM. Other than that, we have simply chosen to represent the information as a boolean valued function rather than a set. We use this form simply to clarify the explanation of the design.

Editorial Note:
For simplicity, in this section we do not consider the indirection between security contexts and SIDs. Thus to compare this model to the earlier model, security contexts should be considered identical to SIDs.

Each OM has a directory of copies:

retained_copies: *SECURITY_CONTEXT*, *SECURITY_CONTEXT*, *PERMISSION* -> *COPIES*

indicating what copies are maintained locally. The diagram depicting the relationship between *current_perms*, *provided_copies*, and *retained_copies* is shown in Figure 10. The filled circles, crosses and stars represent three different replicas. The corresponding non-filled shapes represent the directory entry for that replica. There is one entry in *provided_copies* for every replica maintained at some OM. Each of these entries includes a reference to each OM that maintains at least one copy. At the OM, *retained_copies* maintains a directory of all the copies stored at the site.

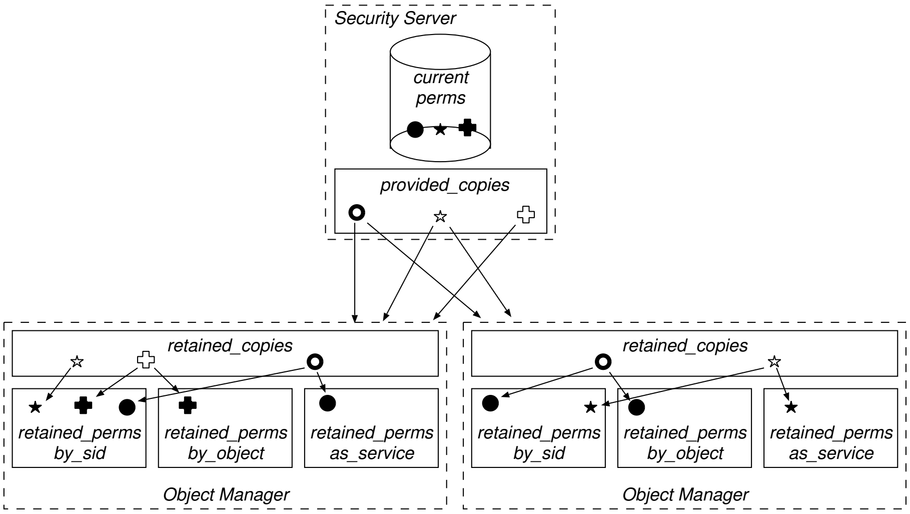

Figure 10: State Components

6.6.4.2 Concurrency Control at SS

A strict two-phase locking (rigorous 2PL) discipline is used at the SS when a data element is accessed (read or modified). Before accessing an element, an appropriate lock is acquired. If a transaction fails it must undo the modifications it made to the database and release all of its locks.

6.6.4.3 Replica Control Protocol

When a thread accesses a replica at an OM, *retained_copies* is accessed and and a shared lock on it is obtained. When updating an element of *current_perms* at the SS, the transaction invokes *avc_invalidate* at all OMs listed in *provided_copies* and updates only the local copy. The locking required is discussed further in Section 6.6.4.6.

6.6.4.4 Permission Migration

A permission can migrate from one cache to another at an OM. One method of handling a security fault is to check for the missing permission in some of the other caches at the OM. For example, if the desired permission is not found in *retained_perms_by_object*, a search can be initiated in *retained_perms_by_sid*. If the permission is found there, it can be copied into *retained_perms_by_object* and the thread making the original access can be restarted. We call the transaction that copies the permission from one cache to another, a *copier* transaction. We assume that whenever a migration occurs, a read-latch is acquired on the source entry and that this is held until the entry has been copied to the target. We also assume that the transfer to the target is atomic with respect to all other accesses to the target cache entry.

6.6.4.5 Compute Access Vector When an OM suffers an AVC miss, it requests the appropriate access vector using *compute_access_vector*.

SS Upon receiving the request, the SS acquires a read-lock on the appropriate element of *current_perms* and write-locks on the corresponding components of *provided_copies*. The value of *provided_copies* is then updated to reflect the new copy being cached. At this point, locks are released, and a response is returned. If successful, the requested access vector is returned. Otherwise, a negative acknowledgment is returned.

OM Before issuing a *compute_access_vector* request, a write-latch is acquired on the appropriate entry of *retained_copies*. This will be held until a response is returned. If the request is successful, i.e., the value returned is cached, and the directory *retained_copies* is updated to represent the new cache entry. If the request times-out, or a negative acknowledgment is received, the latch is released, and the operation will be retried.

6.6.4.6 Invalidate Cache Entry When the SS wants to modify *current_perms*, it may need to invalidate the contents of the AVC at one or more OMs. The protocol for this is described below.

SS The operation is invoked by the SS on a target OM. The parameters associated with the operation describe the entries to be invalidated. Before issuing the operation, the SS acquires write-locks on all corresponding components of *provided_copies*. After a positive acknowledgment from the OM, *provided_copies* is updated. If a negative acknowledgment is received, the transaction is aborted. In this case, the transaction must restore any data elements it modified at the SS, but no recovery at the OM is required.

OM The OM must remove the specified permission from all of its caches. The revocation must be atomic with respect to other cache accesses. To achieve atomicity with respect to copier transactions, a write latch is obtained on the corresponding entry of *retained_copies*. To achieve atomicity with respect to operations reading from the cache, either read-latches can be used, or we can stop all threads as is done currently (this assumes we are using the non-blocking model of Fluke[21]). This in effect aborts transactions with a lock conflict. The operation must then be restarted.

6.6.5 Consistency

Our standard for consistency is one-copy serializability. This requires that any execution of the system be equivalent to a serial execution operating on a one-copy database. This hides the effects of concurrency and replication from any transaction accessing the database. In this section, we will prove that the protocol we describe above ensures this property.

The first thing we note is that the histories produced have acyclic serialization graphs. The only transactions that do not satisfy two-phase locking rules, are transactions that revoke access at more than one OM. In this case, each OM will acquire locks on its own *retained_copies* but will only hold them until the *avc_invalidate* completes, not until the entire transaction completes as 2PL would require.

Lemma 1 *Let H be a history produced by the protocol given above. There exists a locked history H' that is constructed from $C(H)$ (i.e., the committed projection of H) by adding all required lock and unlock events such that:*

1. *All locks are honored (i.e., no conflicting locks are held by distinct transaction); and*

2. *A transaction never acquires a new lock after it has released one.*

Proof- We will consider each type of transaction in turn and explain how we can add the appropriate locks to H'. Copier transactions already acquire locks as required by (1) and (2). Thus, these can be introduced into H' directly. We will insist that the locking events be moved as close to the operation they protect as possible, consistent with (1) and (2).

Transactions representing stages of operations do not acquire locks explicitly. However, an operation runs to completion having sole access to permission caches it reads from. When interrupted, a stage either declares itself complete and commits, or aborts and is retried later. Thus, we can request all required locks at the beginning of the transaction and release them when it is complete. This will obviously satisfy both conditions (1) and (2).

Now, we consider policy update transactions that take place at the SS. These transactions acquire locks in accordance with (1) already. However, the locks held on *retained_copies* must be extended in duration so that they are held until immediately before the lock on *provided_copies* is released. We need to show that no other transaction in $C(H)$ will hold a conflicting lock on *provided_copies* during this period. From the time that *avc_invalidate* completes until the lock on *provided_copies* is released, the cache entry for the desired element will be empty. Thus, any copier transaction trying to copy this element, or any operation stage attempting to access this element will have aborted. Another revoking transaction will be blocked until the lock on *provided_copies* is released, as this is acquired first. An attempt to copy a new value into the cache element that was revoked, will require a sequence of one or more copier/*compute_access_vector* transactions. The first in the chain must acquire a write-latch on *retained_copies* and a read-latch on *provided_copies* before it makes its first access. Thus, its read of provided copies cannot occur in the period we are concerned with. However, it could have acquired its write-latch on *retained_copies* during this period. This is the reason we insist above that lock events be as close to the operation they protect as possible.

We have shown how to construct H' by adding locks to $C(H)$ in accordance with conditions (1) and (2). □

The next step in the proof is to show that the serialization graph of any history produced by the protocol is acyclic. This implies that if we consider only replicas and not the data elements they represent, the history is serializable.

Lemma 2 *For any history H produced by the protocol given above, $SG(H)$ is acyclic.*

Proof- By Lemma 1, we can construct a history H' with identical operations and ordering as H that satisfies two-phase locking rules. From Theorem 3.6 of [3] we know that $SG(H')$ is acyclic. Because the operations and ordering of H is identical to $SG(H')$, we know that $SG(H)$ is acyclic as well. □

The final step is to demonstrate that all transactions see the failure and creation of replicas as happening in the same order. This is sufficient to demonstrate that a history is 1SR. The proof of its sufficiency appears in [3]. The proof below is very similar to the proof in [3] showing that the available-copies protocol is 1SR.

Theorem 1 *The protocol given above produces only 1SR histories.*

Proof- We will base this proof on the Failure-Recovery Serialization Graph (FRSG) approach of Bernstein [3]. Informally, we need to show that all transaction see failures and recoveries occurring in the same order. Specifically, we need to show how to construct an acyclic FRSG for any history H that the protocol produces.

Let H be a history produced by the protocol given above. Construct the graph G with nodes N and edges E from H as follows:

$$
\begin{aligned}
N &= \{T_i | T_i \in H\} \cup \{create[x_a], fail[x_a] | x_a \text{is a copy of some data item } x\} \\
E &= \{T_i \rightarrow T_j | T_i \rightarrow T_j \in SG(H)\} \cup F_1 \cup F_2 \cup F_3 \cup F_4 \\
F_1 &= \{create[x_a] \rightarrow T_i | COPIER(x_a) << T_i \text{ in } SG(H)\} \\
F_2 &= \{T_i \rightarrow fail[x_a] | T_i << INVALIDATE(x_a) \text{ in } SG(H)\} \\
F_3 &= \{fail[x_a] \rightarrow T_i | INVALIDATE(x_a) << T_i \text{ in } SG(H) \text{ or } INVALIDATE(x_a) = T_i\} \\
F_4 &= \{T_i \rightarrow create[x_a] | T_i << COPIER(x_a) \text{ in } SG(H)\}
\end{aligned}
$$

In the definitions above, we use $<<$ to denote a path in $SG(H)$. The notation COPIER(x_a) refers to the transaction that creates the replica x_a. We assume that a replica is created, fails, and never recovers, so this is well defined. This transaction can be a copier transaction as discussed in the section on permission migration, an execution of *compute_access_vector*, or for the permanent replicas of the SS policy DB, an imaginary transaction, T_0, that executes to completion before all others begin. Similarly, INVALIDATE(x_a) is the transaction that invalidates the replica x_a. For cached replicas at the OM, this will be a transaction executed by the SS that performs an *avc_invalidate*. For replicas at the SS, this will be another imaginary transaction T_∞ that begins execution only after all others have completed.

We begin by showing that G is an FRSG of H. The definition requires that an FRSG include the following sets of edges.

$$
\begin{aligned}
E_1 &= \{create[x_a] \rightarrow T_i | T_i \text{ reads or writes[19]} x_a\} \\
E_2 &= \{T_i \rightarrow fail[x_a] | T_i \text{ reads } x_a\} \\
E_3 &= \{T_i \rightarrow create[x_a] \text{ or } fail[x_a] \rightarrow T_i | T_i \text{ writes some copy of } x \text{ but not } x_a\}
\end{aligned}
$$

We first argue that any transaction that accesses a replica x_a conflicts with COPIER(x_a) and INVALIDATE(x_a). Before accessing x_a, the transaction must first access the local directory for the replica, either *provided_copies* or *retained_copies* depending on where the replica is located. Because COPIER(x_a) and INVALIDATE(x_a) both update this directory, both will conflict with the transaction making the access to x_a. In addition, COPIER(x_a) and INVALIDATE(x_a) conflict with each other.

The set E_1 is a subset of F_1. This follows from the fact that a replica can only be accessed by a transaction if it appears in the local directory. To see the update made by COPIER(x_a), the accessing transaction must follow COPIER(x_a) in $SG(H)$. Thus, any transaction that reads or writes x_a must follow COPIER(x_a) in $SG(H)$.

As we argued above, any transaction that reads x_a, conflicts with INVALIDATE(x_a). If it follows INVALIDATE(x_a), the replica is marked as unavailable and so the accessing transaction will fail. Because INVALIDATE(x_a) does not read x_a itself, we can see that any transaction that reads x_a must precede INVALIDATE(x_a) in $SG(H)$. Thus, we see that E_2 is a subset of F_2.

Now, we must show that E_3 is a subset of $F_3 \cup F_4$. The only writes performed are by transactions at the SS. These transactions perform *avc_invalidate* of any copy listed in *provided_copies*. Thus, if a transaction T writes x, but not x_a, either COPIER(x_a) follows T in $SG(H)$, INVALIDATE(x_a) precedes T or INVALIDATE(x_a) = T. We know that $SG(H)$ orders COPIER(x_a), INVALIDATE(x_a) and T because they all conflict. Thus, we see that E_3 must be a subset of $F_3 \cup F_4$. Thus, we know that G is an FRSG of H.

Now, we need to show that the graph G, we have constructed is acyclic. Let f be a function mapping nodes of G to nodes in $SG(H)$ as follows:

$$
\begin{aligned}
f(T_i) &= T_i \\
f(create[x_a]) &= COPIER(x_a) \\
f(fail[x_a]) &= INVALIDATE(x_a)
\end{aligned}
$$

Assume for the sake of a contradiction, that G contains a cycle $p =< n_0, n_1, ..., n_m - 1, n_0 >$

[19] Note that we do not consider the initialization of a copy by the COPIER transaction to be a write.

where $m > 1$. Let $T_i \preceq T_j$ mean that there is a path from T_i to T_j in $SG(H)$ or that $T_i = T_j$. By inspection of the rules for constructing G, we see that $f(n_0) \preceq f(n_1) \preceq ... \preceq f(n_{m-1}) \preceq f(n_0)$. There must be at least one edge not from the set F_3. In this case, we have by transitivity $f(n_0) << f(n_0)$. But this contradicts the fact that $SG(H)$ is acyclic. Thus, we are forced to assume that G is acyclic as well. By Theorem 8.5 of [3], we have, H is one-copy serializable. □

6.6.6 Deadlock

Deadlock can arise in the protocol between a *compute_access_vector* and an *avc_invalidate* in the following way:

1. A *compute_access_vector* transaction (T_1) at the OM begins by locking the entry for permission P in *retained_copies*.
2. At the SS a *avc_invalidate* transaction (T_2) acquires a write lock on P in *provided_copies*.
3. The second part of T_1, now executing at the SS attempts to acquire a shared lock on P in *provided_copies* but is blocked due to the lock held by T_2.
4. Finally, the second part of T_2, executing at the OM, attempts to get a write lock on P in *retained_copies* but is delayed due to the lock already held by T_1.

Thus, both transactions are blocked waiting for the other to complete and release its locks. One way of addressing this problem is to insist that *compute_access_vector* not wait for locks. If they are not immediately available the operation is rejected and later retried. This will rule out deadlock, but live-lock may still be a problem.

6.6.7 System Failures

We assume here that all components of the system exhibit fail-fast behavior. In a distributed system we must deal with partial failures. Recovery of an OM is not difficult as it maintains no persistent state. It will be initialized with a set of fixed perms (e.g., the set of "fixed" perms). The SS may believe the OM has elements cached that were lost during an OM failure, but this will do no harm.

If the SS fails, we will need to take some corrective action. First, we need to restore the database to an acceptable state. If log based recovery is used, we simply recover the last committed state of the database (including *provided_copies*). This will restore consistency. If log based recovery is considered too expensive, then we must restore a predefined state for the security database. To bring *provided_copies* to a consistent value, we can *avc_reset* all of the OM and then empty *provided_copies*.

6.6.8 Evaluation

We have chosen to modify the policy by invalidating any information cached at an OM and then replenishing the cache using *compute_access_vector*. This approach allows replication of the permission database across multiple OMs without requiring a two-phase commit or rollback at the OM.

This approach has drawbacks as well. First, it cannot be used to modify "wired" permissions. Invalidating these permissions would by definition cause the system to fail. If changing wired permissions is required, we must either provide rollback and atomic commitment in the OMs or insist that a transaction only modify data values at one OM.

The *Fundamental_Property* from Section 6.5 required that whenever a permission is retained by SID, it is in *provided_perms*. The requirement we use in this approach is stronger in that it ensures equality of values not just the set inclusion requirement of the *Fundamental_Property*. However, our 1SR requirement is also weaker. The 1SR requirement ensures that there is some logical ordering of the transactions that can be used to justify the values read by the transactions. It does not guarantee that this order is consistent with the execution order of the transactions. For example, if the updates to *current_perms* were never propagated to the AVC, the result would satisfy the 1SR requirement. In the serialization order, all transactions executing at the OM would appear before any of the transactions that updated *current_perms*. This unsettling result is due in part to the fact that information flows only from *current_perms* to the AVC. Updates to the AVC are never propagated back. One way to strengthen our requirement is to insist, not only that there exist an equivalent one-copy serial history, but that the order of transactions in that history be compatible with the execution order of the transactions. In other words, if a transaction T_i completes before transaction T_j begins, then the equivalent serial history must order T_i before T_j. Transactions that overlap can be ordered either way. The protocol we propose here does satisfy this stronger requirement. It is not clear if this strengthened requirement is as strong as the *Fundamental_Property*.

Now, let us compare the efficiency of the two protocols. The one based on sequence numbers allows an *avc_invalidate* and a *compute_access_vector* to proceed concurrently. If there is a large number of concurrent requests, it is likely that many of these will be rejected. In the locking protocol of this section, only one of these requests will be allowed to execute. The others will be delayed. When a large number of concurrent requests are presented, locking approaches also do poorly. In this case, requests are rejected to resolve deadlocks.

In general, when the probability of conflict between transactions is low, timestamp based approaches tend to perform better. The overhead of locking each request in this case is largely wasted effort. Because locking approaches attempt to reorder requests submitted in the “wrong” order, they work better when there are higher levels of contention. Where this cross-over point occurs depends on the overhead imposed by locking, and the efficiency of roll-back.

One of our main motivations for using a lock-based protocol is that these protocols are well understood. While we believe it would be possible to use the sequence number approach to synchronize *provided_copies* and *retained_copies*, it would be more difficult to argue that the result is 1SR.

While the rate of conflict in the protocol requests is obviously dependent upon the security policy, we expect that permission revocations (and therefore conflicts) will be relatively uncommon. Moreover, the cost of permission revocation itself is likely to overwhelm the cost of resolving conflicts among requests. Thus in a relatively static security policy the cost of the locking protocol is greater than the cost of the sequence number protocol, while in a highly dynamic security policy the cost of revocations will render differences in the protocol irrelevant.

Ultimately, the final choice between the two proposed protocols, or variants of them, is largely dependent upon determining the complete set of properties which the protocol is required to satisfy. If the Fundamental Property is sufficient, then the simplicity of the sequence number protocol would make it the preferable choice. If it is determined that other properties are desirable, such as 1SR or consistency in both positive and negative policy decisions, then either the locking protocol should be adopted or modifications must be made to the sequence number protocol.

6.6.9 Future Work

The first issue to study, is the performance and feasibility of this design. It was designed to be compatible with the revocation framework already in place in Fluke, but the impact of this design should be evaluated.

This proposal does not support replication of the database across multiple SS. The potential availability would be quite valuable in some applications. To accomplish this, the replica control protocol used must be generalized. The difficulty comes from the fact that the protocol used for the SS replicas will likely be different from the protocol for OM replicas.

Another question to address, is the feasibility of adding some rollback and simple atomic commitment protocol to the OM. If it can be done efficiently and without adding too much complexity, it may prove beneficial. It would simplify the concurrency control and perhaps improve the efficiency of updates.

6.7 Timeliness for Policy Changes

The Fundamental Property ensures that in any state, the security policy enforced by the object manager is consistent with the policy recorded by the security server (in *current_perms*). It addresses the correctness of all processing of security policy changes, but it leaves open the availability of policy changes. Indeed, a system in which the object manager never responds to the security server's request to revoke a permission will satisfy the Fundamental Property even though it is incapable of supporting a dynamic security policy.

What is needed is a property to ensure that policy changes will also be completed. In this section we discuss a very informal statement of such a property and provide some direction on how this property could be refined into requirements on the various components of a system.

We propose the following as an informal statement of a property which guarantees that policy changes will occur:

> Whenever a change in policy is identified by the security server (either due to internal logic or a request from an administrator), any permission revoked by this change will be removed from the policy (i.e., from *current_perms*) in a timely manner.

The key word in this property is of course "timely". We cannot require that the change occur instantaneously, because a permission revocation requires action not just by the security server but by any object managers which may have retained the permission. Instead, our goal is to capture requirements to ensure that revocation cannot be arbitrarily delayed by a malicious subject, even though the very existence of a malicious subject may cause a (controllable) delay e.g., by its use of memory resources.

To define the necessary requirements, we identify the following five steps which are taken to complete a permission revocation:

- The security server generates a revocation request, identifies the destination and calls the IPC system to deliver the message.
- The message is delivered to its destination.
- The object manager receives the request, performs all necessary processing (i.e., to ensure that the revoked permission is no longer retained), and calls the IPC system to return a notification.

- The message is delivered back to the security server.
- The security server responds by removing the permission from *current_perms*.

For the first and last of these steps to be completed in a timely manner requires only internal processing by the security server, and there is no obvious impediment within the security server. The security server must rely upon the system to provide it with the necessary resources (e.g., CPU and memory paging), which is a basic requirement that would be placed upon any system. To satisfy this requirement, it may be desirable for the security server to be able to request a change in resource management during a policy change.[20]

Steps two and four require that IPC be performed in a timely manner, another requirement on the underlying system.

Step three is the most interesting, in particular, the ability of the object manager to remove the revoked permission from all versions of retained permissions.

Our experiences indicate that permissions can be retained in two distinct ways: within data structures (such as the Flask access vector cache) or as part of the state of requests which are currently in progress (assuming a multi-threaded processing model). In the first case, the object manager obviously can arbitrarily alter its data structures to remove retained permissions. However, it may need to respect locks held by other threads processing client requests, which effectively reverts to the second case.

Revocation of permissions which are retained as part of the state of a request in progress can be more difficult to address. There appear to be only two choices. The first is to undo the progress made by the request by reverting to a previous state before the permission was retained, in which case the request may fail when an attempt is made to reacquire the permission. The second is to process the request beyond the point at which the permission is retained.

The first of these choices is ideal since it satisfies the intent of revocation by preventing any forward progress which potentially disagrees with the new policy. However, since this is often difficult to accomplish, and since revocation cannot be immediate in any case because of required communication between security server and object manager, the second choice is allowable.[21] Either choice must meet the timeliness guarantee, which in particular means that the operation must be able to move forward (or backward) with no external dependencies controllable by a potentially malicious subject.

Another key requirement which arises out of step three is that the destination for any revocation request from the security server must be an object manager trusted to complete revocation in a timely manner. For instance, in Flask a subject can request to receive notification of poicy changes (`register_avc`), but this request is refused unless the subject making the request is granted the appropriate permission by the security policy. Without such a control, a malicious subject can prevent a policy change from being completed simply by registering to receive policy change requests and never responding to them.

[20] Note that when we refer to "the system", the security server and even object manager may be part of that underlying system. Therefore there is a potential for deadlock concerns which are very implementation dependent and beyond the scope of this discussion.

[21] In theory, immediate revocation could be possible if the object manager is notified of the policy change in advance. However, in time-based policies (including policies which use expiring certificates for granting permissions), small, controllable, delays appear to be acceptable.

Appendix *A*
Bibliography

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

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

[3] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading MA, 1987.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre Corp., Bedford, MA, 1977. Also available through Nat'l Technical Information Service, Springfield, Va., Report No. NTIS AD-A039324.

[5] Michael J. Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems.* Kluwer Academic, Boston MA, 1996.

[6] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[7] NCSC. Trusted network interpretation. Standard, NCSC-TG-005, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, July 1987.

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

[9] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI International, December 1992.

[10] 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.

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

[12] Secure Computing Corporation. DTOS Covert Channel Analysis Plan. DTOS CDRL A017, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, May 1997.

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

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

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

[16] 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.

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

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

[19] Lynn Te Winkel, Ray Spencer, and Todd Fine. Adding Security to Commercial Microkernel-Based Systems. Technical Report RL-TR-96-75, Rome Laboratory, Griffiss Air Force Base, NY, January 1996.

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

[21] Department of Computer Science University of Utah. Interface and execution models in the fluke kernel. Unpublished internal report.

Appendix B

PVS Specifications for Dynamic Security Policy Study

This section contains the PVS specifications corresponding to Sections 6.1 through 6.5.

Unlike the rest of the document, these specifications are not included in-line with the text. We chose to separate the specifications into this appendix because the specifications were written earlier than the text and by a different person than the authors of the text. Moreover, there was not sufficient time available to update the specifications, and therefore they do not always parallel the text.

This section is organized parallel to Sections 6.1 through 6.5, and simple comparisons between the text and the PVS are included. There is also a final section which presents some additions that were made to the composition framework in order to simplify some of the proofs.

B.1 The Fundamental Property

The PVS describing the fundamental property is deferred until Section B.3.

B.2 State Model

This section presents the formal state model.

The system as a whole is defined on type *common_state_t* which contains an *OM_STATE* field *om* and a *SS_STATE* field *ss*. Any additional constraints on the common state type can be expressed using the boolean function *common_state*. There are no additional constraints at this time.

Note that *PROT_STATE* will be defined in Section B.5.

THEORY *common_types*

```
common_types : THEORY

  BEGIN

  SID : TYPE+
  PERMISSION : TYPE+
  AGENT : TYPE+


  END common_types
```

THEORY *om_state*

```
om_state : THEORY

  BEGIN

  IMPORTING common_types

  OBJECT : TYPE+
  SERVICE : TYPE+

  ob, ob1, ob2: VAR OBJECT
  svc: VAR SERVICE
  sid, sid1, sid2: VAR SID

  OM_STATE: TYPE = [#
        object_sid : [OBJECT -> SID],
        required_perms : [SERVICE -> setof[PERMISSION]],
        retained_perms_by_sid : [SID, SID -> setof[PERMISSION]],
        retained_perms_by_object : [OBJECT, OBJECT -> setof[PERMISSION]],
        retained_perms_as_service : [OBJECT, OBJECT -> setof[SERVICE]]
        #]


  services_provided : [OM_STATE, OM_STATE -> setof[[OBJECT, OBJECT, SERVICE]]]


  END om_state
```

THEORY *ss_state*

```
ss_state : THEORY

  BEGIN

  IMPORTING common_types

  SECURITY_CONTEXT : TYPE+

  SS_STATE : TYPE = [#
        provided_perms : [SID, SID -> setof[PERMISSION]],
        sid_to_context : [SID -> SECURITY_CONTEXT],
        current_perms : [SECURITY_CONTEXT, SECURITY_CONTEXT -> setof[PERMISSION]]
        #]


  END ss_state
```

THEORY *common_state*

```
common_state : THEORY

  BEGIN

  IMPORTING om_state
  IMPORTING ss_state
  IMPORTING prot_state

  COMMON_STATE : TYPE+ = [#
```

```
    om: OM_STATE,
    ss: SS_STATE,
    prot: PROT_STATE
    #]

cs: VAR COMMON_STATE


common_state(cs): bool = TRUE

common_state_t : TYPE+ = (common_state)

END common_state
```

B.3 Refinement into Subproperties

This section presents the formal specifications of the Fundamental Property and the five subproperties. It also defines the theorem that the five subproperties are sufficient to ensure that the Fundamental Property holds. Note that the names of the subproperties are slightly different here than in Section 6.3.

THEORY *fund_prop*

```
fund_prop : THEORY

  BEGIN

  IMPORTING common_state
  IMPORTING framework_help[common_state_t, AGENT]
%   IMPORTING set_ops

  st, st1, st2: VAR common_state_t
  ag : VAR AGENT
  ob1, ob2: VAR OBJECT
  service: VAR SERVICE
  sid1, sid2: VAR SID
  perm: VAR PERMISSION
  perm_set: VAR setof[PERMISSION]
  t: VAR trace_t
  sid_perms: VAR [SID, SID -> setof[PERMISSION]]


  provided_perms_objs(st, ob1, ob2): setof[PERMISSION] =
    provided_perms(ss(st))(object_sid(om(st))(ob1),
                           object_sid(om(st))(ob2))

  retained_perms(st, ob1, ob2): setof[PERMISSION] =
    union(retained_perms_by_sid(om(st))(object_sid(om(st))(ob1),
                                         object_sid(om(st))(ob2)),
          retained_perms_by_object(om(st))(ob1,ob2))

  required_perms_check(st, service, perm_set): bool =
    subset?(required_perms(om(st))(service), perm_set)

  required_perms_retained(st, service, ob1, ob2): bool =
    required_perms_check(st, service, retained_perms(st, ob1, ob2))
```

```
services_provided_good(st1, st2, ag): bool =
  FORALL ob1, ob2, service:
    services_provided(om(st1), om(st2))(ob1, ob2, service)
    IMPLIES
          required_perms_retained(st1, service, ob1, ob2)
        OR
          retained_perms_as_service(om(st1))(ob1,ob2)(service)


by_sid_good(st): bool =
  FORALL sid1, sid2:
    subset?(retained_perms_by_sid(om(st))(sid1, sid2),
              provided_perms(ss(st))(sid1, sid2))

by_object_good_help(ob1, ob2, st, sid_perms): bool =
  subset?(retained_perms_by_object(om(st))(ob1, ob2),
          sid_perms(object_sid(om(st))(ob1),
                      object_sid(om(st))(ob2)))

by_object_good(st): bool =
  FORALL ob1, ob2:
    by_object_good_help(ob1, ob2, st, provided_perms(ss(st)))

as_service_good_help(ob1, ob2, st, perm_set): bool =
  FORALL service:
      member(service, retained_perms_as_service(om(st))(ob1, ob2))
  IMPLIES
      required_perms_check(st, service, perm_set)

as_service_good(st): bool =
  FORALL ob1, ob2:
      as_service_good_help(ob1, ob2, st,
                                provided_perms_objs(st, ob1, ob2))


current_perms_sid(st, sid1, sid2): setof[PERMISSION] =
  current_perms(ss(st))(sid_to_context(ss(st))(sid1),
                        sid_to_context(ss(st))(sid2))

provided_perms_good_help(sid1, sid2, st): bool =
  FORALL perm:
    member(perm, provided_perms(ss(st))(sid1, sid2))
    IMPLIES member(perm, current_perms_sid(st, sid1, sid2))


provided_perms_good(st): bool =
  FORALL sid1, sid2:
      provided_perms_good_help(sid1, sid2, st)


fund_prop(st1, st2, ag): bool =
  FORALL ob1, ob2, service:
    services_provided(om(st1), om(st2))(ob1, ob2, service)
    IMPLIES
      subset?(required_perms(om(st1))(service),
              current_perms_sid(st1, object_sid(om(st1))(ob1),
                                     object_sid(om(st1))(ob2)))


fund_thm: THEOREM
  services_provided_good
      /\ by_sid_good
      /\ by_object_good
      /\ as_service_good
```

```
      /\ provided_perms_good
   |- fund_prop


 END fund_prop
```

B.4 State Transition Requirements

This sections presents the formal specifications of the state transition requirements, organized according to the state element which is changed. After the statement of the requirements, the proof of the fundamental theorem is presented. Once again, the names are generally changed somewhat from the names in Section 6.4.

THEORY *current_perms_rqmts*

```
current_perms_rqmts : THEORY

 BEGIN

 IMPORTING fund_prop

 st, st1, st2: VAR common_state_t
 ag : VAR AGENT
 sid1, sid2: VAR SID
 con1, con2: VAR SECURITY_CONTEXT
 perm: VAR PERMISSION


 rem_current_perms(st1, st2, ag): bool =
   FORALL con1, con2, sid1, sid2, perm:
       NOT member(perm, current_perms(ss(st2))(con1, con2))
       AND member(perm, current_perms(ss(st1))(con1, con2))
       AND sid_to_context(ss(st2))(sid1) = con1
       AND sid_to_context(ss(st2))(sid2) = con2
    IMPLIES
       NOT member(perm, provided_perms(ss(st2))(sid1, sid2))

 END current_perms_rqmts
```

THEORY *sid_to_context_rqmts*

```
sid_to_context_rqmts : THEORY

 BEGIN

 IMPORTING fund_prop

 st, st1, st2: VAR common_state_t
 ag : VAR AGENT
 sid1, sid2: VAR SID
 perm: VAR PERMISSION
```

```
change_sid_to_context(st1, st2, ag): bool =
  FORALL sid1, sid2, perm:
      sid_to_context(ss(st2))(sid1) /= sid_to_context(ss(st1))(sid1)
  IMPLIES
      provided_perms_good_help(sid1, sid2, st2)
      AND provided_perms_good_help(sid2, sid1, st2)

END sid_to_context_rqmts
```

THEORY *provided_perms_rqmts*

```
provided_perms_rqmts : THEORY

  BEGIN

  IMPORTING fund_prop

  st, st1, st2: VAR common_state_t
  ag : VAR AGENT
  sid1, sid2: VAR SID
  ob1, ob2: VAR OBJECT
  perm: VAR PERMISSION
  serv: VAR SERVICE


  rem_provided_perms_by_sid(st1, st2, ag): bool =
    FORALL sid1, sid2, perm:
        NOT member(perm, provided_perms(ss(st2))(sid1, sid2))
        AND member(perm, provided_perms(ss(st1))(sid1, sid2))
     IMPLIES
        NOT member(perm, retained_perms_by_sid(om(st2))(sid1, sid2))

  rem_provided_perms_by_object(st1, st2, ag): bool =
    FORALL sid1, sid2, ob1, ob2, perm:
        NOT member(perm, provided_perms(ss(st2))(sid1, sid2))
        AND member(perm, provided_perms(ss(st1))(sid1, sid2))
        AND object_sid(om(st2))(ob1) = sid1
        AND object_sid(om(st2))(ob2) = sid2
     IMPLIES
        NOT member(perm, retained_perms_by_object(om(st2))(ob1, ob2))

  rem_provided_perms_as_service(st1, st2, ag): bool =
    FORALL sid1, sid2, ob1, ob2, perm, serv:
        NOT member(perm, provided_perms(ss(st2))(sid1, sid2))
        AND member(perm, provided_perms(ss(st1))(sid1, sid2))
        AND member(perm, required_perms(om(st2))(serv))
        AND object_sid(om(st2))(ob1) = sid1
        AND object_sid(om(st2))(ob2) = sid2
     IMPLIES
        NOT member(serv, retained_perms_as_service(om(st2))(ob1, ob2))

  add_provided_perms(st1, st2, ag): bool =
    FORALL sid1, sid2, perm:
        NOT member(perm, provided_perms(ss(st1))(sid1, sid2))
        AND member(perm, provided_perms(ss(st2))(sid1, sid2))
     IMPLIES
        member(perm, current_perms_sid(st2, sid1, sid2))

  END provided_perms_rqmts
```

THEORY *object_sid_rqmts*

```
object_sid_rqmts : THEORY

  BEGIN

  IMPORTING fund_prop

  st, st1, st2: VAR common_state_t
  ag : VAR AGENT
  ob1, ob2: VAR OBJECT
  perm: VAR PERMISSION


  change_object_sid_by_object(st1, st2, ag): bool =
    FORALL ob1, ob2:
        object_sid(om(st1))(ob1) /= object_sid(om(st2))(ob1)
    IMPLIES
        by_object_good_help(ob1, ob2, st2, provided_perms(ss(st2)))
        AND by_object_good_help(ob2, ob1, st2, provided_perms(ss(st2)))


  change_object_sid_as_service(st1, st2, ag): bool =
    FORALL ob1, ob2:
        object_sid(om(st1))(ob1) /= object_sid(om(st2))(ob1)
    IMPLIES
        as_service_good_help(ob1, ob2, st2,
                             provided_perms_objs(st2, ob1, ob2))
        AND as_service_good_help(ob2, ob1, st2,
                                 provided_perms_objs(st2, ob2, ob1))


  %% The following alternate properties are useful in analyzing
  %% systems in which by_object and as_service can be rejustified
  %% on the basis of by_sid (and by_object in the case of as_service).

  change_object_sid_by_object_om(st1, st2, ag): bool =
    FORALL ob1, ob2:
        object_sid(om(st1))(ob1) /= object_sid(om(st2))(ob1)
    IMPLIES
        by_object_good_help(ob1, ob2, st2, retained_perms_by_sid(om(st2)))
        AND by_object_good_help(ob2, ob1, st2, retained_perms_by_sid(om(st2)))

  change_object_sid_by_object_om_thm: THEOREM
    next(by_sid_good)
        /\ change_object_sid_by_object_om
    |-
    change_object_sid_by_object

  change_object_sid_as_service_om(st1, st2, ag): bool =
    FORALL ob1, ob2:
        object_sid(om(st1))(ob1) /= object_sid(om(st2))(ob1)
    IMPLIES
        as_service_good_help(ob1, ob2, st2,
                             retained_perms(st2, ob1, ob2))
        AND as_service_good_help(ob2, ob1, st2,
                                 retained_perms(st2, ob2, ob1))

  change_object_sid_as_service_om_thm: THEOREM
    next(by_sid_good)
        /\ next(by_object_good)
        /\ change_object_sid_as_service_om
    |-
```

```
    change_object_sid_as_service

  END object_sid_rqmts
```

THEORY *required_perms_rqmts*

```
required_perms_rqmts : THEORY

  BEGIN

  IMPORTING fund_prop

  st, st1, st2: VAR common_state_t
  ag : VAR AGENT
  ob1, ob2: VAR OBJECT
  perm: VAR PERMISSION
  serv: VAR SERVICE


  add_required_perms(st1, st2, ag): bool =
    FORALL serv, perm, ob1, ob2:
        NOT member(perm, required_perms(om(st1))(serv))
        AND member(perm, required_perms(om(st2))(serv))
        AND member(serv, retained_perms_as_service(om(st2))(ob1, ob2))
     IMPLIES
        member(perm, provided_perms_objs(st2, ob1, ob2))

  add_required_perms_om(st1, st2, ag): bool =
    FORALL serv, perm, ob1, ob2:
        NOT member(perm, required_perms(om(st1))(serv))
        AND member(perm, required_perms(om(st2))(serv))
        AND member(serv, retained_perms_as_service(om(st2))(ob1, ob2))
     IMPLIES
        member(perm, retained_perms(st2, ob1, ob2))

  add_required_perms_om_thm: THEOREM
    next(by_sid_good)
        /\ next(by_object_good)
        /\ add_required_perms_om
   |-
    add_required_perms

  END required_perms_rqmts
```

THEORY *by_sid_rqmts*

```
by_sid_rqmts : THEORY

  BEGIN

  IMPORTING fund_prop

  st, st1, st2: VAR common_state_t
  ag : VAR AGENT
  sid1, sid2: VAR SID
  perm: VAR PERMISSION
```

```
add_rp_by_sid(st1, st2, ag): bool =
  FORALL sid1, sid2, perm:
      NOT member(perm, retained_perms_by_sid(om(st1))(sid1, sid2))
      AND member(perm, retained_perms_by_sid(om(st2))(sid1, sid2))
   IMPLIES
      member(perm, provided_perms(ss(st2))(sid1, sid2))

END by_sid_rqmts
```

THEORY *by_object_rqmts*

```
by_object_rqmts : THEORY

  BEGIN

  IMPORTING fund_prop

  st, st1, st2: VAR common_state_t
  ag : VAR AGENT
  ob1, ob2: VAR OBJECT
  perm: VAR PERMISSION


  add_rp_by_object(st1, st2, ag): bool =
    FORALL ob1, ob2, perm:
        NOT member(perm, retained_perms_by_object(om(st1))(ob1, ob2))
        AND member(perm, retained_perms_by_object(om(st2))(ob1, ob2))
     IMPLIES
        member(perm, provided_perms(ss(st2))(object_sid(om(st2))(ob1),
                                             object_sid(om(st2))(ob2)))

  %% The following alternate property is useful in analyzing systems
  %% in which by_object can be rejustified on the basis of by_sid.

  add_rp_by_object_om(st1, st2, ag): bool =
    FORALL ob1, ob2, perm:
        NOT member(perm, retained_perms_by_object(om(st1))(ob1, ob2))
        AND member(perm, retained_perms_by_object(om(st2))(ob1, ob2))
     IMPLIES
        member(perm, retained_perms_by_sid(om(st2))
                     (object_sid(om(st2))(ob1),
                      object_sid(om(st2))(ob2)))

  END by_object_rqmts
```

THEORY *as_service_rqmts*

```
as_service_rqmts : THEORY

  BEGIN

  IMPORTING fund_prop

  st, st1, st2: VAR common_state_t
  ag : VAR AGENT
  ob1, ob2: VAR OBJECT
```

```
  serv: VAR SERVICE


  add_rp_as_service(st1, st2, ag): bool =
    FORALL ob1, ob2, serv:
        NOT member(serv, retained_perms_as_service(om(st1))(ob1, ob2))
        AND member(serv, retained_perms_as_service(om(st2))(ob1, ob2))
     IMPLIES
        required_perms_check(st2, serv, provided_perms_objs(st2, ob1, ob2))

  add_rp_as_service_om(st1, st2, ag): bool =
    FORALL ob1, ob2, serv:
        NOT member(serv, retained_perms_as_service(om(st1))(ob1, ob2))
        AND member(serv, retained_perms_as_service(om(st2))(ob1, ob2))
     IMPLIES
        required_perms_check(st2, serv, retained_perms(st2, ob1, ob2))


  add_rp_as_service_om_thm: THEOREM
    next(by_sid_good)
        /\ next(by_object_good)
        /\ add_rp_as_service_om
    |-
    add_rp_as_service

  END as_service_rqmts
```

Theory *rqmts* shows that the requirements imply the stability of the state invariant subproperties.

THEORY *rqmts*

```
rqmts : THEORY

  BEGIN

  IMPORTING object_sid_rqmts
  IMPORTING required_perms_rqmts
  IMPORTING by_sid_rqmts
  IMPORTING by_object_rqmts
  IMPORTING as_service_rqmts
  IMPORTING provided_perms_rqmts
  IMPORTING sid_to_context_rqmts
  IMPORTING current_perms_rqmts



  by_sid_good_step: THEOREM
    add_rp_by_sid /\ rem_provided_perms_by_sid
    |-
    stable(by_sid_good)


  by_object_good_step: THEOREM
    change_object_sid_by_object
        /\ add_rp_by_object
        /\ rem_provided_perms_by_object
    |-
    stable(by_object_good)
```

```
as_service_good_step: THEOREM
  change_object_sid_as_service
      /\ add_required_perms
      /\ add_rp_as_service
      /\ rem_provided_perms_as_service
  |-
  stable(as_service_good)

provided_perms_good_step: THEOREM
  add_provided_perms
      /\ change_sid_to_context
      /\ rem_current_perms
  |-
  stable(provided_perms_good)


%% The following alternate theorems are useful in analyzing
%% systems in which by_object and as_service can be rejustified
%% on the basis of by_sid (and by_object in the case of as_service).

% Would be nice to redo this proof using lemma relating *_om
% predicates to their non-om counterparts.
by_object_good_om_step: THEOREM
  next(by_sid_good)
      /\ change_object_sid_by_object_om
      /\ add_rp_by_object_om
      /\ rem_provided_perms_by_object
  |-
  stable(by_object_good)


as_service_good_om_step: THEOREM
  next(by_sid_good)
      /\ next(by_object_good)
      /\ change_object_sid_as_service_om
      /\ add_required_perms_om
      /\ add_rp_as_service_om
      /\ rem_provided_perms_as_service
  |-
  stable(as_service_good)

END rqmts
```

Up to this point, the PVS has only expressed stability of the state invariants under each transition. This is not the same as expressing that the state invariants hold in all states, i.e., the induction proof has not yet been performed. This is the purpose of the following specification, which defines the following theorems. As elsewhere, all theorems have been proved.

- *cmp_services_provided_good_thm* — Shows that a component satisfies subproperty *services_provided_good* in all transitions if each of its steps satisfies *services_provided_good*.
- *cmp_by_sid_good_thm* — Shows that a component satisfies subproperty *by_sid_good* in all reachable states if all of its allowed initial states satisfy *by_sid_good* and all of its steps satisfy *add_rp_by_sid* and *rem_provided_perms_by_sid*.
- *cmp_by_object_good_thm* — Shows that a component satisfies subproperty *by_object_good* in all reachable states if all of its allowed initial states satisfy *by_object_good* and all of its steps satisfy *change_object_sid_by_object*, *add_rp_by_object* and *rem_provided_perms_by_object*.

- *cmp_by_object_good_om_thm* — Shows that a component satisfies subproperty *by_object_good* in all reachable states if all of its allowed initial states satisfy *by_object_good*, all of its steps satisfy *change_object_sid_by_object_om*, *add_rp_by_object_om* and *rem_provided_perms_by_object*, and it satisfies subproperty *by_sid_good* in all reachable states.
- *cmp_as_service_good_thm* — Shows that a component satisfies subproperty *as_service_good* in all reachable states if all of its allowed initial states satisfy *as_service_good* and all of its steps satisfy *change_object_sid_as_service*, *add_rp_as_service*, *add_required_perms* and *rem_provided_perms_as_service*.
- *cmp_as_service_good_om_thm* — Shows that a component satisfies subproperty *as_service_good* in all reachable states if all of its allowed initial states satisfy *as_service_good*, all of its steps satisfy *change_object_sid_as_service_om*, *add_rp_as_service_om*, *add_required_perms_om* and *rem_provided_perms_as_service*, and it satisfies subproperties *by_sid_good* and *by_object_good* in all reachable states.
- *cmp_provided_perms_good_thm* — Shows that a component satisfies subproperty *provided_perms_good* in all reachable states if all of its allowed initial states satisfy *provided_perms_good* and all of its steps satisfy *add_provided_perms*, *change_sid_to_context* and *rem_current_perms*.
- *fund_prop_thm* — Shows that a component satisfies the fundamental property in all transitions if it satisfies *services_provided_good* in all transitions and satisfies *by_sid_good*, *by_object_good*, *as_service_good* and *provided_perms_good* in all reachable states.

THEORY *prop_analysis*

```
prop_analysis : THEORY

  BEGIN

  IMPORTING rqmts
  IMPORTING framework_aux

  cmp: VAR (comp_t)


  cmp_services_provided_good_thm: LEMMA
    steps_satisfy_under_inv(cmp, services_provided_good)
    IMPLIES
    satisfies(cmp, always(services_provided_good))

  cmp_by_sid_good_thm: LEMMA
    init_satisfies(cmp, by_sid_good)
        AND steps_satisfy_under_inv(cmp, add_rp_by_sid)
        AND steps_satisfy_under_inv(cmp, rem_provided_perms_by_sid)
    IMPLIES
    satisfies(cmp, always(by_sid_good))

  cmp_by_object_good_thm: LEMMA
    init_satisfies(cmp, by_object_good)
        AND steps_satisfy_under_inv(cmp, change_object_sid_by_object)
        AND steps_satisfy_under_inv(cmp, add_rp_by_object)
        AND steps_satisfy_under_inv(cmp, rem_provided_perms_by_object)
    IMPLIES
    satisfies(cmp, always(by_object_good))
```

```
cmp_by_object_good_om_thm: LEMMA
  init_satisfies(cmp, by_object_good)
      AND steps_satisfy_under_inv(cmp, change_object_sid_by_object_om)
      AND steps_satisfy_under_inv(cmp, add_rp_by_object_om)
      AND steps_satisfy_under_inv(cmp, rem_provided_perms_by_object)
      AND satisfies(cmp, always(by_sid_good))
  IMPLIES
  satisfies(cmp, always(by_object_good))

cmp_as_service_good_thm: LEMMA
  init_satisfies(cmp, as_service_good)
      AND steps_satisfy_under_inv(cmp, change_object_sid_as_service)
      AND steps_satisfy_under_inv(cmp, add_rp_as_service)
      AND steps_satisfy_under_inv(cmp, add_required_perms)
      AND steps_satisfy_under_inv(cmp, rem_provided_perms_as_service)
  IMPLIES
  satisfies(cmp, always(as_service_good))

cmp_as_service_good_om_thm: LEMMA
  init_satisfies(cmp, as_service_good)
      AND steps_satisfy_under_inv(cmp, change_object_sid_as_service_om)
      AND steps_satisfy_under_inv(cmp, add_rp_as_service_om)
      AND steps_satisfy_under_inv(cmp, add_required_perms_om)
      AND steps_satisfy_under_inv(cmp, rem_provided_perms_as_service)
      AND satisfies(cmp, always(by_sid_good))
      AND satisfies(cmp, always(by_object_good))
  IMPLIES
  satisfies(cmp, always(as_service_good))

cmp_provided_perms_good_thm: LEMMA
  init_satisfies(cmp, provided_perms_good)
      AND steps_satisfy_under_inv(cmp, add_provided_perms)
      AND steps_satisfy_under_inv(cmp, change_sid_to_context)
      AND steps_satisfy_under_inv(cmp, rem_current_perms)
  IMPLIES
  satisfies(cmp, always(provided_perms_good))

cmp_fund_prop_thm: THEOREM
  satisfies(cmp, always(services_provided_good))
      AND satisfies(cmp, always(by_sid_good))
      AND satisfies(cmp, always(by_object_good))
      AND satisfies(cmp, always(as_service_good))
      AND satisfies(cmp, always(provided_perms_good))
  IMPLIES
  satisfies(cmp, always(fund_prop))


END prop_analysis
```

B.5 Synchronization Protocol

This section adds additional state to the common state which was defined in Section B.2.

THEORY *prot_state*

```
prot_state : THEORY

  BEGIN
```

```
  IMPORTING common_types

  PROT_STATE : TYPE = [#
        %%% Maintained by SS
        grantings: setof[[SID, SID, PERMISSION, nat]],
        latest_granting: [SID, SID, PERMISSION -> nat],

        %%% Maintained by OM
        notifs: setof[[SID, SID, PERMISSION, nat]],
        latest_notif: [SID, SID, PERMISSION -> nat]
        #]

  END prot_state
```

The following theory includes the specification of the requirements of Sections 6.5.2 and 6.5.3. The "Utility Functions" defined in the theory are used in Section 6.5.4

THEORY *prot_rqmts*

```
prot_rqmts : THEORY

  BEGIN

  IMPORTING fund_prop

  st, st1, st2: VAR common_state_t
  ag : VAR AGENT
  sid1, sid2: VAR SID
  perm: VAR PERMISSION
  sn: VAR nat


%%% UTILITY FUNCTIONS %%%
  usable_notif(st)(sid1, sid2, perm, sn): bool =
    notifs(prot(st))(sid1, sid2, perm, sn)
        AND sn > latest_granting(prot(st))(sid1, sid2, perm)

  usable_granting(st)(sid1, sid2, perm, sn): bool =
    grantings(prot(st))(sid1, sid2, perm, sn)
        AND sn >= latest_notif(prot(st))(sid1, sid2, perm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%% REQUIREMENTS %%%

%% These requirements are sufficient for demonstrating the
%% by_sid requirements. They would need to be extended to
%% demonstrate the by_object and as_service requirements.


  latest_notif_mono(st1, st2, ag): bool =
    FORALL sid1, sid2, perm:
        latest_notif(prot(st2))(sid1, sid2, perm)
        >= latest_notif(prot(st1))(sid1, sid2, perm)

  latest_granting_mono(st1, st2, ag): bool =
    FORALL sid1, sid2, perm:
        latest_granting(prot(st2))(sid1, sid2, perm)
```

```
        >= latest_granting(prot(st1))(sid1, sid2, perm)

  use_granting_by_sid(st1, st2, ag): bool =
    FORALL sid1, sid2, perm:
        retained_perms_by_sid(om(st2))(sid1, sid2)(perm)
          AND NOT retained_perms_by_sid(om(st1))(sid1, sid2)(perm)
      IMPLIES
        EXISTS sn: usable_granting(st1)(sid1, sid2, perm, sn)

  send_notif(st1, st2, ag): bool =
    FORALL sid1, sid2, perm, sn:
        notifs(prot(st2))(sid1, sid2, perm, sn)
          AND NOT notifs(prot(st1))(sid1, sid2, perm, sn)
      IMPLIES
        latest_notif(prot(st2))(sid1, sid2, perm) >= sn

  send_notif_by_sid(st1, st2, ag): bool =
    FORALL sid1, sid2, perm, sn:
        notifs(prot(st2))(sid1, sid2, perm, sn)
          AND NOT notifs(prot(st1))(sid1, sid2, perm, sn)
      IMPLIES
        NOT retained_perms_by_sid(om(st2))(sid1, sid2)(perm)

  use_notif(st1, st2, ag): bool =
    FORALL sid1, sid2, perm:
        provided_perms(ss(st1))(sid1, sid2)(perm)
          AND NOT provided_perms(ss(st2))(sid1, sid2)(perm)
      IMPLIES
        EXISTS sn: usable_notif(st1)(sid1, sid2, perm, sn)

  send_granting(st1, st2, ag): bool =
    FORALL sid1, sid2, perm, sn:
        grantings(prot(st2))(sid1, sid2, perm, sn)
          AND NOT grantings(prot(st1))(sid1, sid2, perm, sn)
      IMPLIES
        latest_granting(prot(st2))(sid1, sid2, perm) >= sn
        AND provided_perms(ss(st2))(sid1, sid2)(perm)


  prot_rqmts(st1, st2, ag): bool =
    latest_notif_mono(st1, st2, ag)
        AND latest_granting_mono(st1, st2, ag)
        AND use_granting_by_sid(st1, st2, ag)
        AND send_notif(st1, st2, ag)
        AND send_notif_by_sid(st1, st2, ag)
        AND use_notif(st1, st2, ag)
        AND send_granting(st1, st2, ag)

  END prot_rqmts
```

The formal proof of the theorem uses the composition framework, including the additions of Section B.6. The following four theories define the lemmas and theorems as stated in Section 6.5 and also formally define the object manager and security server components as used by the composition framework.

THEORY *prot_props*

```
prot_props : THEORY

  BEGIN
```

```
IMPORTING prot_rqmts

st, st1, st2: VAR common_state_t
ag : VAR AGENT
sid1, sid2: VAR SID
perm: VAR PERMISSION
sn: VAR nat

latest_granting_correct(st): bool =
  FORALL sid1, sid2, perm, sn:
      grantings(prot(st))(sid1, sid2, perm, sn)
    IMPLIES
      latest_granting(prot(st))(sid1, sid2, perm) >= sn

latest_notif_correct(st): bool =
  FORALL sid1, sid2, perm, sn:
      notifs(prot(st))(sid1, sid2, perm, sn)
    IMPLIES
      latest_notif(prot(st))(sid1, sid2, perm) >= sn

usable_notif_not_rp_by_sid(st): bool =
  FORALL sid1, sid2, perm, sn:
      usable_notif(st)(sid1, sid2, perm, sn)
    IMPLIES
      NOT retained_perms_by_sid(om(st))(sid1, sid2)(perm)

usable_granting_provided(st): bool =
  FORALL sid1, sid2, perm, sn:
      usable_granting(st)(sid1, sid2, perm, sn)
    IMPLIES
      provided_perms(ss(st))(sid1, sid2)(perm)

END prot_props
```

THEORY *om*

```
om : THEORY

  BEGIN

  IMPORTING prot_props

  st1, st2: VAR common_state_t
  ag: VAR AGENT


  om_agent: AGENT

  om_cags(ag) : bool =
    ag = om_agent

  om_init(st1): bool =
    notifs(prot(st1)) = emptyset

  om_guar(st1, st2, ag): bool =
    om_cags(ag)
        AND latest_notif_mono(st1, st2, ag)
        AND use_granting_by_sid(st1, st2, ag)
        AND send_notif(st1, st2, ag)
        AND send_notif_by_sid(st1, st2, ag)
```

```
%% Only SS−managed data may change
om_hidd(st1, st2, ag): bool =
  NOT om_cags(ag) AND
  st2 = st1 WITH
      [ ss := ss(st2),
        prot := prot(st1) WITH
              [ (grantings) := grantings(prot(st2)),
                (latest_granting) := latest_granting(prot(st2))
              ]
      ]

om_rely(st1, st2, ag): bool = NOT om_cags(ag)

om : (comp_t) =
   (# init := om_init,
      guar := om_guar,
      hidd := om_hidd,
      rely := om_rely,
      cags := om_cags,
      view := id_view[common_state_t],
      wfar := emptyset,
      sfar := emptyset
      #)

latest_notif_mono_thm: THEOREM
  steps_satisfy(om, latest_notif_mono)

use_granting_by_sid_thm: THEOREM
  steps_satisfy(om, use_granting_by_sid)

send_notif_thm: THEOREM
  steps_satisfy(om, send_notif)

send_notif_by_sid_thm: THEOREM
  steps_satisfy(om, send_notif_by_sid)

latest_notif_correct_thm: THEOREM
  satisfies(om, always(latest_notif_correct))

END om
```

THEORY *ss*

```
ss : THEORY

  BEGIN
  IMPORTING prot_props

  st1, st2: VAR common_state_t
  ag: VAR AGENT


  ss_agent: AGENT

  ss_cags(ag) : bool =
    ag = ss_agent

  ss_init(st1): bool =
    grantings(prot(st1)) = emptyset
```

```
        AND by_sid_good(st1)

  ss_guar(st1, st2, ag): bool =
    ss_cags(ag)
        AND latest_granting_mono(st1, st2, ag)
        AND use_notif(st1, st2, ag)
        AND send_granting(st1, st2, ag)


  %% Only OM−managed data may change
  ss_hidd(st1, st2, ag): bool =
    NOT ss_cags(ag) AND
    st2 = st1 WITH
        [ om := om(st2),
          prot := prot(st1) WITH
                [ (notifs) := notifs(prot(st2)),
                  (latest_notif) := latest_notif(prot(st2))
                ]
        ]

  ss_rely(st1, st2, ag): bool = NOT ss_cags(ag)

  ss : (comp_t) =
      (# init := ss_init,
        guar := ss_guar,
        hidd := ss_hidd,
        rely := ss_rely,
        cags := ss_cags,
        view := id_view[common_state_t],
        wfar := emptyset,
        sfar := emptyset
        #)

  latest_granting_mono_thm: THEOREM
    steps_satisfy(ss, latest_granting_mono)

  use_notif_thm: THEOREM
    steps_satisfy(ss, use_notif)

  send_granting_thm: THEOREM
    steps_satisfy(ss, send_granting)

  latest_granting_correct_thm: THEOREM
    satisfies(ss, always(latest_granting_correct))


  END ss
```

THEORY *sys*

```
sys : THEORY

  BEGIN

  IMPORTING om
  IMPORTING ss
  IMPORTING prop_analysis

  st: VAR common_state_t
  sid1, sid2: VAR SID
  perm: VAR PERMISSION
```

```
sn1, sn2: VAR nat


sys_comps : (nonempty?[(comp_t)]) =
  {c: (comp_t) | c = om OR c = ss}

sys_comps_composable: THEOREM
  composable(sys_comps)

sys : (comp_t) =
  compose(sys_comps)

sys_cmp_thm: THEOREM
  %% from OM
  steps_satisfy(sys, latest_notif_mono)
      AND steps_satisfy(sys, use_granting_by_sid)
      AND steps_satisfy(sys, send_notif)
      AND steps_satisfy(sys, send_notif_by_sid)
      AND satisfies(sys, always(latest_notif_correct))

  %% from SS
      AND steps_satisfy(sys, latest_granting_mono)
      AND steps_satisfy(sys, use_notif)
      AND steps_satisfy(sys, send_granting)
      AND satisfies(sys, always(latest_granting_correct))


usable_notif_granting_nand : LEMMA
  latest_granting_correct(st)
      AND latest_notif_correct(st)
  =>
  NOT (usable_notif(st)(sid1, sid2, perm, sn1)
      AND usable_granting(st)(sid1, sid2, perm, sn2))

usable_notif_not_rp_by_sid_lem : LEMMA
  satisfies(sys, always(usable_notif_not_rp_by_sid))

usable_granting_provided_lem : LEMMA
  satisfies(sys, always(usable_granting_provided))

steps_rem_provided_perms_by_sid : LEMMA
  steps_satisfy_under_inv(sys, rem_provided_perms_by_sid)

steps_add_rp_by_sid : LEMMA
  steps_satisfy_under_inv(sys, add_rp_by_sid)

sys_by_sid_good: THEOREM
  satisfies(sys, always(by_sid_good))

END sys
```

B.6 PVS Composition Framework Additions

The *framework_help* and *framework_aux* theories serve serveral purposes. First, they provide some syntactic notation (i.e., $/\backslash$ and $|-$) intended to make some of the subsequent theorems easier to write and read. Second, they supply some lemmas that relate this notation to concepts from the composition framework. Finally, they provide additional lemmas to make the subsequent analysis easier.[22]

[22] Many of these lemmas could justifiably be included in the framework proper and might be considered for incorporation into any future versions of the framework.

The binary infix operator /\ denotes the "conjunction" of one state or action predicate with another.[23] Note that this operator is polymorphic. When applied to two state predicates or two actions predicates, it returns their intersection (*sand* and *aand*, respectively). When applied to a state predicate and an action predicate it returns the action predicate consisting of all transitions satisfying the action predicate in which the initial state satisfies the state predicate.

The binary operator |– denotes "provability". This operator is also polymorphic, taking either a pair of state predicates or a pair of action predicates.[24] This operator returns true if its first operand is a subset of its second. The precedence of /\ is higher than that of |–, so *a* /\ *b* |– *c* /\ *d* is equal to (*a* /\ *b*) |– (*c* /\ *d*).

Finally, the *next* function takes a state predicate *sp1* as an argument and returns an action predicate denoting the set of all transitions that end in a state in *sp1*.

Lemma *steps_satisfy_implies* allows one to demonstrate that a component satisfies a property *ap2* by exhibiting a property *ap1* that the component satisfies and showing that *ap1* |– *ap2*. Lemma *steps_satisfy_and* allows one to show that a component satisfies *ap1* /\ *ap2* by showing that it satisfies both *ap1* and *ap2*. Lemma *steps_satisfy_aimplies* lets one show that a component satisfies a property *aimplies*(*ap1, ap2*) by exhibiting a property *ap3* satisfied by the component such that (*ap1* /\ *ap3* |– *ap2*). This allows an analyst to introduce extra facts (the *ap3*) about the component into the *aimplies* proof. Lemma *satisfies_always_next* shows that if a component satisfies *sp1* in all reachable states then the end state of every reachable transition for that component also satisfies *sp1*.

THEORY *framework_help*

```
framework_help[ST, AG: TYPE+] : THEORY

  BEGIN

  framework : LIBRARY = "framework"

  IMPORTING framework@cmp_preds[ST, AG]
  IMPORTING framework@preds_aux[ST, AG]

  sp, sp1, sp2: VAR STATE_PRED
  ap, ap1, ap2, ap3: VAR ACTION_PRED
  st, st1, st2: VAR ST
  ag : VAR AG
  cmp: VAR (comp_t)


  /\(sp1, sp2) : STATE_PRED =
    sand(sp1, sp2);

  /\(ap1, sp2) : ACTION_PRED =
    aandas(ap1, sp2);

  /\(sp1, ap2) : ACTION_PRED =
    aandas(ap2, sp1);
```

[23] The symbol /\ is recognized by the PVS parser as a binary infix operator. Thus, even though it is defined with the normal function syntax, it can also occur in expressions such as *a* /\ *b*. A state predicate is a set of states, and an action predicate is a set of transitions.

[24] The operand could also be defined on mixed-type operands, but we do not expect this to be very useful in general and have not supplied such a definition.

```
/\(ap1, ap2) : ACTION_PRED =
  aand(ap1, ap2);

|-(sp1, sp2): bool =
  FORALL st1:
      simplies(sp1, sp2)(st1);

|-(ap1, ap2): bool =
  FORALL st1, st2, ag:
      aimplies(ap1, ap2)(st1, st2, ag);


next(sp1): ACTION_PRED =
  {st1, st2, ag | sp1(st2)}

steps_satisfy_under_inv(cmp, ap1): bool =
  EXISTS sp1:
      satisfies(cmp, always(sp1))
      AND
      (FORALL st1, st2, ag:
          sp1(st1) AND sp1(st2) AND steps(cmp)(st1, st2, ag)
          => ap1(st1, st2, ag))

under_inv_lem: LEMMA
  steps_satisfy(cmp, ap1) => steps_satisfy_under_inv(cmp, ap1)

inv1_under_inv : THEOREM
  init_satisfies(cmp, sp) AND steps_satisfy_under_inv(cmp, stable(sp))
    IMPLIES satisfies(cmp, always(sp))

inv2_under_inv : THEOREM
  steps_satisfy_under_inv(cmp, ap) IMPLIES satisfies(cmp, always(ap))


steps_satisfy_implies: LEMMA
  (EXISTS ap1:
      steps_satisfy(cmp, ap1)
      AND (ap1 |- ap2))
  IMPLIES steps_satisfy(cmp, ap2)

steps_satisfy_and: LEMMA
  steps_satisfy(cmp, ap1) AND steps_satisfy(cmp, ap2)
  IMPLIES steps_satisfy(cmp, ap1 /\ ap2)

steps_satisfy_implies_under_inv: LEMMA
  (EXISTS ap1:
      steps_satisfy_under_inv(cmp, ap1)
      AND (ap1 |- ap2))
  IMPLIES steps_satisfy_under_inv(cmp, ap2)

steps_satisfy_and_under_inv: LEMMA
  steps_satisfy_under_inv(cmp, ap1)
      AND steps_satisfy_under_inv(cmp, ap2)
  IMPLIES steps_satisfy_under_inv(cmp, ap1 /\ ap2)


steps_satisfy_aimplies: LEMMA
  (EXISTS ap3:
      steps_satisfy(cmp, ap3)
      AND (ap1 /\ ap3 |- ap2))
  IMPLIES
  steps_satisfy(cmp, aimplies(ap1, ap2))

satisfies_always_next: THEOREM
```

```
    satisfies(cmp, always(sp1))
    IMPLIES satisfies(cmp, always(next(sp1)))

  satisfies_always_next_under_inv: THEOREM
    satisfies(cmp, always(sp1))
    IMPLIES steps_satisfy_under_inv(cmp, next(sp1))

  END framework_help
```

Theorem *compose_steps* shows that the set of transitions allowed by a composite system is equal to the set of transitions allowed by all components of the system. Theorem *satisfies_pimplies_always* allows one to show that a component satisfies *pimplies*(*always*(*sp1*), *always*(*sp2*)) (i.e., every trace allowed by the component that satisfies *sp1* in every state also satisfies *sp2* in every state) by showing that

- every initial state allowed by the component that satisfies *sp1* also satisfies *sp2*, and
- every transition allowed by the component that starts in a state in both *sp1* and *sp2* ends in a state in *sp2*.

Theorem *satisfies_pimplies_always_ap* is similar to *satisfies_pimplies_always* with the difference that it uses an action predicate *ap1* in place of the state predicate *sp1*. Theorem *satisfies_implies* is similar in purpose to *steps_satisfy_aimplies*. It allows one to incorporate known facts (property *p1*) about a component into the proof of a property *p1*. Theorem *pand_intersection* proves the equivalence of *pand* and *intersection*. This is useful when one wishes to use theorems about intersection shile proving something about an expression including *pand*. Theorem *satisfies_pand* reduces the proof that a component satisfies *pand*(*p1, p2*) to showing it satisfies *p1* and *p2* individually.

THEORY *framework_aux*

```
framework_aux[ ST, AG: TYPE+]  : THEORY

  BEGIN

  lib : LIBRARY = "framework"
  IMPORTING framework@cmp_preds[ ST, AG ]

  s: VAR (composable[ST,AG])
  c: VAR (comp_t)
  tr: VAR transition
  sp1, sp2: VAR STATE_PRED
  ap1, ap2: VAR ACTION_PRED
  st, st1, st2: VAR ST
  ag: VAR AG
  p1, p2: VAR prop_t

  compose_steps: THEOREM
    steps(compose(s))(tr)
      = (FORALL c: s(c) => steps(c)(tr))

  compose_steps_satisfy: THEOREM
    (EXISTS c: steps_satisfy(c, ap1) AND s(c))
    => steps_satisfy(compose(s), ap1)

  composite_satisfies: THEOREM
    (EXISTS c: satisfies(c, p1) AND s(c))
    => satisfies(compose(s), p1)
```

```
%% The next three definitions should probably be restructured
%% to look more like the "_ap" ones below.

init_satisfies_sp(c, sp1, sp2) : bool =
  FORALL st:
    init(c)(st) AND sp1(st)
    IMPLIES sp2(st)

steps_satisfy_sp(c, sp1, sp2) : bool =
  FORALL st1, st2, ag:
    aandas(steps(c), sp1)(st1, st2, ag)
    IMPLIES stable(sp2)(st1, st2, ag)

satisfies_pimplies_always: THEOREM
  init_satisfies_sp(c, sp1, sp2)
      AND steps_satisfy_sp(c, sp1, sp2)
  IMPLIES satisfies(c, pimplies(always(sp1), always(sp2)))

%% The purpose of the following two theorems is to support proofs in
%% which the component c includes steps that violate stable(sp2) but
%% which are not reachable based upon the initial state conditions
%% of c.  Thus, the satisfies goal may be true even though the steps
%% satisfy decomposition produces by inv1 is not.  Theorem
%% satisfies_implies allows one to introduce a constraining property
%% p1 that prevents the undesirable transitions.  Component c must
%% satisfy this property and it must satisfy pimplies(p1, p2).
%% Theorem satisfies_pimplies_always_sp is like inv1 except that it
%% implies a pimplies property and that we can ignore steps of c
%% that violate ap1 when proving the step stability.
%%

satisfies_pimplies_always_ap: THEOREM
  init_satisfies(c, sp2)
      AND steps_satisfy(c, aimplies(ap1, stable(sp2)))
  IMPLIES satisfies(c, pimplies(always(ap1), always(sp2)))

satisfies_implies: THEOREM
  (EXISTS p1:
      satisfies(c, p1)
      AND satisfies(c, pimplies(p1, p2)))
  IMPLIES
  satisfies(c, p2)

pand_intersection: THEOREM
  pand = intersection

satisfies_pand: THEOREM
  satisfies(c, p1) AND satisfies(c, p2)
  IMPLIES
  satisfies(c, pand(p1, p2))


END framework_aux
```