Applying the Composition Principle to Verify a Hierarchy of Security Servers *

Mark R. Heckman
Department of Computer Science
University of California
One Shields Ave.
Davis, CA 95616
heckman@cs.ucdavis.edu

Karl N. Levitt
Department of Computer Science
University of California
One Shields Ave.
Davis, CA 95616
levitt@cs.ucdavis.edu

Abstract

This paper describes how the composition principle of Abadi and Lamport can be applied to specify and compose systems where access control policies are distributed among a hierarchy of agents. Examples of such systems are layered secure operating systems, where the mandatory access control policy is enforced by the lowest system layer and discretionary and application-specific policies are implemented by outer layers, and microkernel operating systems, where the access control policy may be distributed among a hierarchy of server processes.

We specifically consider the case of a microkernel operating system type architecture, in which resource management policies are enforced by server processes outside of the kernel, and where the system access control policy is a composition of the distinct policies implemented by the servers. As an example, we have specified a two-server system, including both safety and progress properties. We formally verified the composition of the two server processes using the HOL theorem proving system.

1 Introduction

Formal specification and proof are requirements for systems that must maintain the highest degree of security. In general, however, it is difficult to formally reason about such system due to their complexity. The difficulty of verifying a complex, secure system can be reduced by decomposing a system into smaller subsystems, independently verifying the implementation of the subsystems, and then proving that the composition of the subsystem specifications satisfies the overall system specification.

In some secure system architectures, moreover, this type of compositional proof is not only desirable, but a necessity. For example, consider an extensible trusted computing base built using TCB subsets [13, 11]. In such systems the access control policy is hierarchically distributed among agents in several logically—or even physically—independent layers, and the overall system access control policy is a composition of the policies enforced by each layer.

In secure, micro-kernel operating systems such as TMach [4], the mandatory access control policy is implemented by a combination of the kernel and system tasks called "servers." The discretionary access control policy, furthermore, is completely implemented by servers. Systems like Synergy [10] and the UC Davis Silo [14] take this idea a step further by implementing even the mandatory access control policy by servers alone, outside of the kernel. The servers, furthermore, run concurrently, introducing an element of nondeterminism into the system specification. Reasoning about the access control policy enforced by the system as a whole requires reasoning about the composition of the servers, and the reasoning method must be able to handle concurrency.

Abadi and Lamport devised a general composition principle and proof rule for composing modular and concurrent specifications that consist of both safety and progress properties [2]. Their composition method is based on the *transition-axiom* specification method [7] and a refinement mapping method of proving that one specification implements another [1].

Hemenway and Fellows applied the composition principle to the integration of a secure system from existing components [6], but used only safety properties in their specifications. McLean argued that the composition principle cannot be applied to Noninterference and other "possibilistic" security properties [8]. Our work, unlike that of Hemenway and Fellows, includes progress properties in the specifications and proof. Like their work, however, we focus on access control policies, which are safety properties and can therefore be addressed under the framework.

An access control policy is a type of "resource management" policy, similar to other resource management policies concerned with, for example, memory management or mutual exclusion. The possibilistic properties with which McLean's work is concerned are information flow control policies. Put very simply, an information flow control policy is a policy on the flow of information that is stored in the system, while an access control

*This work was sponsored by DARPA under contract USN N00014-93-1-1322 with the Office of Naval Research and by the National Security Agency's URP Program.

1060-3425/98 $10.00 © 1998 IEEE
Proc. 31st Annual Hawaii International Conference on System Sciences

policy is a policy on the use of the system objects that are used to store the information.

Our work with the composition principle is a part of our larger effort to formally verify an entire secure and distributed system [14]. The ability to verify that a hierarchy of security servers satisfies an access control policy is necessary, but not sufficient, to prove the overall security of an operating system. That larger proof requires us to show that the kernel and servers that make up the operating system together satisfy an information flow control policy. The work presented in this paper is one step toward that long-term goal.

In a previous paper we described our translation of Abadi and Lamport's specification, composition, and refinement mapping methods into HOL [5]. In this paper we give an overview of an example system specification and the composition proof, and describe how the techniques used in the example are applicable to the specification and proof of a system that consists of a hierarchy of security servers.

Section 2 explains in more detail the notion of a hierarchy of security servers, and discusses how Silo uses the hierarchy to implement the system's access control policy. Section 3 gives an overview of our translation of Abadi and Lamport's specification and proof method. Section 4 describes the example system and its specifications. Section 5 describes the formal proof of the system.

2 A Hierarchy of Security Servers

An important design goal of Silo is to keep the kernel as small as possible. In micro-kernel operating systems this goal is generally achieved by putting the responsibility for resource management in the server processes outside the kernel. Silo treats the access control policy on the objects that it manages (mailboxes) as a resource management policy that is the proper responsibility of servers.

Another Silo design goal is to support incremental verification and reuse of previously verified components. Ideally, as modules that implement new features are added to the operating system, the previously verified system can be treated as a module to be composed with the new modules. For example, servers that implement additional access control policies can be added on top of the original system access control policy—e.g., adding discretionary access control (DAC) to an existing mandatory access control (MAC) policy—and the new servers can be composed with the existing, previously verified, system. Separating the access control policy management from the kernel allows a separately verified kernel to be composed with different access control servers, so that a new access control policy does not require reverification of the entire system.

These design goals led to the use of a hierarchy of security servers. A hierarchy allows for layering, to allow a secure system to be extended to include a richer set of security properties, and the security properties can be modularized in the form of servers to support composition and reuse of previously verified modules.

2.1 Different Hierarchies

The usual idea of a hierarchy in the context of a secure operating system is that each layer serves as an abstract machine, creating the abstraction of subjects and objects and enforcing an access control policy for the layer above it. A possible implementation of this type of hierarchical system using servers would be to have a server in each abstract layer that enforces that layer's access control policy, then calls the next-lower layer. At the end of this sequential chain of calls is a call (or calls) to the kernel. This type of sequential hierarchy is depicted in figure 1.

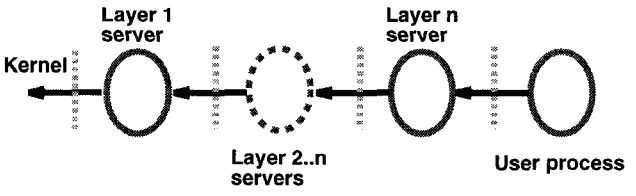

Figure 1: Sequential hierarchy of servers

The sequential hierarchy assumes that access control decisions are made as a result of requests from higher layers. Silo, however, is a distributed operating system and the access control mechanism must be able to handle accesses across a network. While the operating system is distributed, the knowledge of object classifications and subject clearances is not, so each access across the network results in a "bubbling-up" of activity from the lower layers to higher layers.

As shown in figure 2, in this alternative type of hierarchy a network server receives an access request from the network and passes the request to a resource manager. The resource manager consults the security server for its layer, which passes a request to the next higher layer, and so on, until a request propagates to the highest layer of the system. The decision (to allow access or not) is passed back down the hierarchy.

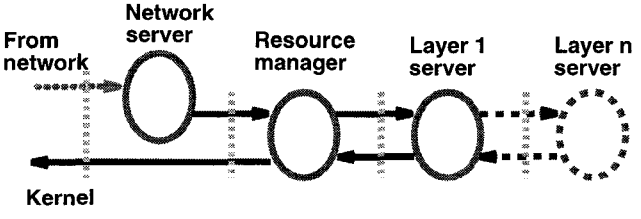

Figure 2: Alternative hierarchy of servers

Each security server compares the security attributes (classification) of the object that it manages with the security attributes (clearance) of the subject that is requesting access, and grants or denies the request according to the access control policy that it implements. If the security servers all approve, the resource manager performs the service requested by the subject.

A security server implements a function of type $classification \rightarrow clearance \rightarrow boolean$. The security server can consist of a single server process or of a

hierarchy of server processes, as in the alternative hierarchy. The function implemented by a hierarchy of such servers would be the conjunction of the boolean results of each of the component functions. For example, the layer 1 server in the hierarchy could implement a mandatory access control policy, while the layer 2 server could implement a discretionary access control policy. Access would be granted only if both servers returned "TRUE."

Another reason for this alternative hierarchy is to facilitate the evaluation of a system under a trusted computer system evaluation criteria, such as that of the United States Department of Defense [9], which call for a strict separation between security-critical and non-security critical system components. In DTMach, for example [12], the designers separated the non-security-critical network protocol software from the trusted software of the secure operating system. In the DTMach design, the network server that is within the TCB passes the data to be transmitted to a protocol server that is above the TCB. The protocol server, in turn, passes the processed data to another network server that is within the TCB.

3 Overview of Specification and Proof Methods

In this section we briefly and informally define some of the main concepts and terms of the transition-axiom, refinement mapping, and composition methods of Abadi and Lamport, and of our adaptation of these methods.

A state is an assignment of values to a set of state variables. A system specification describes all possible infinite sequences of atomic state transitions (finite sequences are represented by sequences where the state reaches a fixpoint), which we call *traces*. We define a trace as a function from natural numbers to system state, where *trace* 0 represents the initial state.

State transitions are due to the actions of *agents*. A specification generally includes a description of all the allowable state transitions that may be caused by a particular agent or set of agents.

A *property* is a predicate that defines a set of traces. For example, a property could be stated as

```
(∀ trace . example_property trace =
  (∀ i . x(trace i) = 5))
```

which is a property that is true for all traces where the state variable x has the value 5 at all times.

There are two types of properties: safety properties and progress properties. Intuitively, safety properties of a system define the acceptable initial states and the allowable state transitions. Progress properties assert that specific state transitions eventually occur.

An initial state safety property looks like this:

```
(∀ trace . initial_property trace = I(trace 0))
```

which is true for all traces where the initial state satisfies a predicate I.

Each state transition has this basic specification format:

```
(∀ ss : system_state . transition_n ss =
  ¬(precondition ss) → ss | (next_state_n ss))
```

where ss is the set of state variables of type system_state, and next_state_n ss is the new state function. This specification says that if the transition's precondition is not satisfied then the system state is unchanged by the transition. (Abadi and Lamport call this a "stuttering step"), but if the precondition is satisfied then the new system state is as defined by next_state_n ss.

A state transition safety property is the disjunction of all allowable state transitions, and looks something like this:

```
(∀ trace. transition_property trace =
 (∀ i. let ss1 = (trace i) and ss2 = (trace(i+1))
  in (ss2 = transition_1 ss1) ∨ ... ∨
     (ss2 = transition_n ss1)))
```

A progress property for a transition asserts that the transition will happen infinitely often, and has this form:

```
(∀ trace . progress_property_n trace =
  (∀ i . ∃ j .  i≤j ∧
    (get_system_state(trace(j+1)) =
      transition_n(get_system_state(trace j)))))
```

This says that, for all valid traces, at any point i in the trace there is another point j that occurs after i where the transition will happen. In temporal logic, this statement is represented as "always eventually transition_n", or $\Box\Diamond transition\_n$.

The specification of a system consists of the conjunction of various safety and progress properties. If, for example, I is an initial state safety property, T is a state transition safety property, and L is the conjunction of progress properties for the valid state transitions defined by T, the specification for a system that satisfied all these properties could be defined as the property $I \cap T \cap L$.

The inputs to a system come from the system's *environment*. A system is specified assuming that its inputs satisfy some definition of correctness, which is simpler than specifying the behavior of a system for all possible conditions. In other words, the system specification includes assumptions about the behavior of the environment. A specification of a system M and the constraints on its environment E is the property $E \Longrightarrow M$, which includes all behaviors where the system satisfies its specification or the environment doesn't. The specifications E and M are written to apply to disjoint sets

of agents: the set of agents that are part of the system μ and those agents outside of the system $\neg\mu$.

Informally, this is Abadi and Lamport's composition proof rule: Given three specifications $E \Longrightarrow M$ (the overall system specification), $E_1 \Longrightarrow M_1$ and $E_2 \Longrightarrow M_2$ (two component specifications), if

$$E \wedge \overline{M_1} \wedge \overline{M_2} \Longrightarrow E_1 \wedge E_2,$$

then

$$E \wedge (E_1 \Longrightarrow M_1) \wedge (E_2 \Longrightarrow M_2) \Longrightarrow (E \Longrightarrow M_1 \wedge M_2),$$

where $\overline{M}$ is the smallest safety property that contains M.

This means that, if we prove that the conjunction of the overall system environment E and the safety properties of the two component systems implies the environments of the two component systems, then we can compose the two component system specifications into the form $E \Longrightarrow M_1 \wedge M_2$.

A refinement mapping is a function from states in one specification to states in another specification. Once component specifications M_1 and M_2 have been composed using the composition proof rule, the remaining step to complete the proof that the composition of M_1 and M_2 implements M is to find a refinement mapping from $M_1 \wedge M_2$ to M. In other words, the composition rule allows us to derive $E \Longrightarrow M_1 \wedge M_2$ and the refinement step allows us to show that $M_1 \wedge M_2 \Longrightarrow M$.

4 Specifications

The system that we formally verified consists of two server processes, which together implement two system calls. In this section we give an overview of our example system specification.

Our system specification in this example assumes the existence of a kernel, similar to the formally verified KIT kernel [3], that implements process separation and process scheduling, and that provides processes with the basic operations for sending and receiving messages through mailboxes.

In the KIT kernel every process can write to every other process's mailbox. In Silo, however, mailboxes are the objects to which access is controlled. In order to mediate access to mailboxes, Silo uses descriptors: A process can only send a message to a mailbox to which it has a descriptor. Processes obtain a descriptor to a mailbox through a system call to a mailbox server.

In this example we do not yet assume the extensions to the kernel that are necessary for the Silo system to implement an access control policy on mailboxes; our intention is simply to demonstrate that the specification and proof method can be used to prove the composition of two servers, and of the functions that they implement, in a message-passing system.

4.1 System Calls and Server Processes

In our example, each process has its own mailbox from which it alone can read messages, but processes can send messages to any other process's mailbox. For simplicity in the specification, mailboxes have unlimited length.

A user process makes a system call by sending a "request" message to one of the mailboxes of the server processes. A server process responds to a system call by sending a "response" message to the mailbox of the process that sent the "request" message. The value returned in a response message—i.e., the value returned by the system call—is a function of the value passed in the corresponding request message.

The overall system specification specifies two system calls, `F` and `FG`. System call `F` accepts an input value y in a request message and returns the value $f(y)$. System call `FG` accepts an input value x in a request message and returns the value $f(g(x))$. These system calls are depicted in the top half of figure 3.

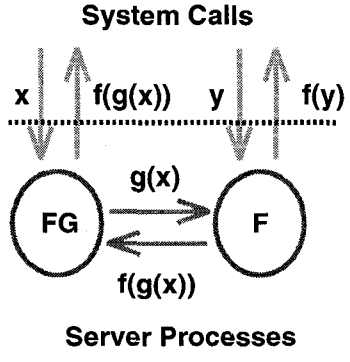

Figure 3: System calls and server processes

System call `F` is implemented by a single server process, also called `F`, that applies the function f to the input value. System call `FG` is implemented by a server process, also called `FG`, that accepts an input value x in a request message and then sends the value $g(x)$ in a message to the `F` server. The `F` server returns the value $f(g(x))$ to the `FG` server, which forwards the result to the process that originally sent x. The implementation of the system calls is depicted in the bottom half of figure 3.

The environment for our system is all the other processes that send messages to, and receive them from, the two server processes.

The two server processes are a simplified example of the alternative hierarchy of servers that was depicted in figure 2. In this example, the `FG` server represents the server in layer 1 of the system, while the `F` server represents the server at layer n. The management policies for some resource are represented by the functions calculated by each server. The overall resource management policy for the resource is the composition of the two functions, and is specified by system call `FG`.

4.2 General Form of the Specifications

Each system call, and each corresponding server process, performs two atomic actions: transferring a single request message from its mailbox to an internal queue, and sending a response message for a single queued request message, as shown in figure 4. A progress property for each atomic action ensures that all request messages are buffered and eventually generate responses.

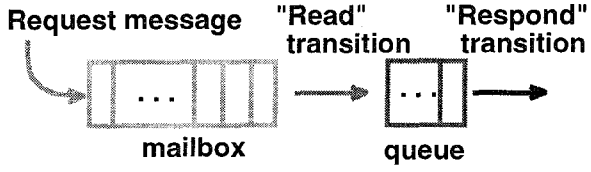

Figure 4: General form of specifications for system calls and server processes

4.2.1 The F and FG System Calls

The read and respond state transitions of the F and FG system calls are similar. The precondition of a read transition is that there is at least one message in the system call's mailbox. If so, the transition transfers the message from the head of the mailbox to the tail of the internal queue.

This is the specification for the F system call read transition:

```
(∀ ss .  syscall_F_reads ss =
 let mbxs = (get_mbxs ss)
 and Fs = (get_Fs ss) in
 let Fmbx = (get_mbxs_mbx mbxs F_ID) in
% If no messages then return state unchanged. %
  ((~mbx_is_unread_msg Fmbx) → ss |
% Otherwise, read message... %
  (let (msg, new_Fmbx) = (read_mbx_msg Fmbx) in
% put back the modified mailbox %
  let new_mbxs = (put_mbxs_mbx mbxs F_ID new_Fmbx)
% put msg in internal queue %
  and new_Fs = (put_Fq_msg Fs msg) in
   (put_mbxs (put_Fs ss new_Fs) new_mbxs))))
```

where ss is the system state, Fs is the internal state for the F server, and Fmbx is the F server (and system call) mailbox.

The progress property for this state transition is

```
(∀ trace .  syscall_F_reads_progress trace =
      (∀ (i : num) .  (∃ (j : num) .
          (i <= j) ∧
          (get_state(trace(j+1)) =
           (syscall_F_reads (get_state(trace j)))))))
```

The respond transitions are as simple as the read transitions. Their precondition is that there is at least one message in the internal queue. If so, the transition removes the message from the head of the internal queue and puts a response message in the mailbox of the request message sender.

This is the specification for the F system call response transition function:

```
(∀ ss .  syscall_F_responds ss =
 let mbxs = (get_mbxs ss)
 and Fs = (get_Fs ss) in
% If no messages then return state unchanged. %
  ((~(Fq_pending Fs)) → ss |
% Otherwise, read message from head of queue... %
  (let (rqst, new_Fs) = (Fq_get_msg Fs) in
% src is the ID of the request message sender. %
  let src = (get_msg_sndr rqst)
% mdata is the request data value. %
  and mdata = (get_msg_data rqst) in
% Return f(mdata) in response message. %
  let response = (cons_msg F_ID (F_func mdata)) in
% put response in destination mailbox %
  let new_mbxs = (put_mbxs_msg mbxs src response)
  in
    (put_mbxs (put_Fs ss new_Fs) new_mbxs))))
```

4.2.2 The F Server Process

Unlike the F system call, the F server process has only a single-length internal buffer. For this reason, the precondition of the F server read transition not only requires that there be at least one message in the F mailbox, but also that the internal buffer is empty. Otherwise, it is the same as that for the F system call read transition.

The precondition of the F server response transition, as for the F system call response transition, is simply that there is a message in the internal buffer. If so, the F server takes the message from the internal buffer and sends a response message.

4.2.3 The FG Server Process

The FG server process sends messages to the F server and must wait until the F server responds before it can send a response message. For this reason, the FG server has an unlimited length internal queue as well as a single-length buffer.

The read transition for the FG server, similar to that of the F server, has the precondition that there is a message in the FG mailbox and that the single-length internal buffer is empty.

The respond transition for the FG server, however, is more complicated than that of the F server. Because processes have only a single mailbox, response messages from the F server arrive in the FG mailbox along with new request messages from other processes. The respond transition must determine the origin of the message and take different actions depending on whether the message came from the F server or not.

```
(∀ ss . server_FG_responds ss =
 let mbxs = (get_mbxs ss)
 and FGs = (get_FGs ss) in
% If no message in buffer then no state change. %
  ((~(FG_msg_in_buf FGs)) → ss |
% Otherwise, check who sent the message %
  (let bufmsg = (FG_read_buf FGs) in
  let bufsrc = (get_msg_sndr bufmsg) in
% If sender is not F then put request message %
% in the queue and send a message to F %
   ((~(bufsrc = F_ID)) → (FG_push_request ss) |
% Otherwise, remove rqst message from queue %
% and send response. %
    (FG_send_response ss)))))
```

If the message is not from the F server—i.e., the message is a new request message—the message is added to the end of the internal queue and the FG server sends a request message to the F server.

```
(∀ ss . FG_push_request ss =
 let mbxs = (get_mbxs ss)
 and FGs = (get_FGs ss) in
 let (buf, FGs1) = (FG_get_buf FGs) in
% (FGs1 has cleared buffer) %
% put request message on queue %
 let new_FGs = (FG_put_queue_msg FGs1 buf) in
% create message to send to F %
 let bufmdata = (get_msg_data buf) in
 let msg = (cons_msg FG_ID (G_func bufmdata)) in
% send message to F %
 let new_mbxs = (put_mbxs_msg mbxs F_ID msg) in
  (put_mbxs (put_FGs ss new_FGs) new_mbxs))
```

For some request message containing the data value `x`, the corresponding request message sent to the F server will contain the value `g(x)`.

When a message is from the F server, it is a response message to a previous request message from the FG server. The message from the F server corresponds to the request message at the head of the internal queue (we proved this correspondence as an invariant). For a message at the head of the internal queue that contains the data value `x`, the corresponding response message from the F server will contain the value `f(g(x))`. The FG server uses the value from the F server in its response message to the sender of the original request message.

```
(∀ ss . FG_send_response ss =
 let mbxs = (get_mbxs ss)
 and FGs = (get_FGs ss) in
 let (buf, FGs1) = (FG_get_buf FGs) in
% (FGs1 has cleared buffer) %
 let bufmdata = (get_msg_data bufmsg) in
% Get request message from head of queue. %
 let (rqst, new_FGs) = (FG_get_queue_msg FGs1) in
% rsrc is ID of the request message sender. %
 let rsrc = (get_msg_sndr rqst) in
% Create response message; use data from F server. %
 let response = (cons_msg FG_ID bufmdata) in
% send response message %
 let new_mbxs = (put_mbxs_msg mbxs rsrc response) in
  (put_mbxs (put_FGs ss new_FGs) new_mbxs))
```

The data path between the F and FG server processes is shown in figure 5, where the solid arrows show the path up until the response message from the F server, and the dashed lines after that point.

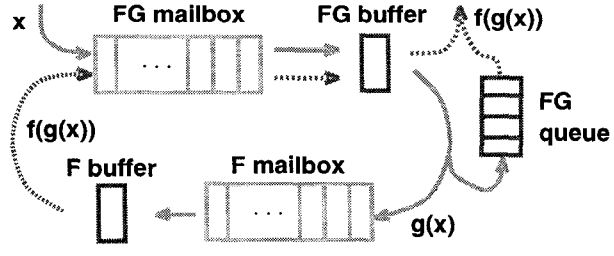

Figure 5: Data path for FG responses

5 Proof

The compositional proof of the two-server system consisted of two steps: the composition step and the refinement step.

5.1 Proof Effort

We found during the course of the proof that the composition proof step was a relatively small part of the overall proof effort. Using "lines of HOL text" as a rough gauge of proof effort, the composition proof step, at roughly 1600 lines, was only about 7% of the total proof (about 23,000 lines). Of the remaining 93%, refinement of the liveness properties took three times more work than the refinement of the safety properties.

5.2 Composition

The antecedent of the compositional proof rule for our example system becomes the following HOL goal:

```
(∀ trace .
  ((env_init trace) ∧ (env_transitions trace) ∧
   (F_init trace) ∧ (F_transitions trace) ∧
   (FG_init trace) ∧ (FG_transitions trace)) ⟹
  ((F_env_init trace) ∧ (F_env_transitions trace) ∧
   (FG_env_init trace) ∧
      (FG_env_transitions trace)))
```

where env_init and env_transitions are the overall environment initial state and state transition properties and F_init and F_transitions are the initial internal state and state transition safety properties for the F server (similarly for the FG server).

This proof goal can be broken down into three subgoals, which makes the proof simpler than it might first appear:

1. That the initial state properties on the left side of the implication imply the initial state properties on the right side. The state transitions cannot affect the initial state properties, and so play no part is this subgoal.
2. That the environment and FG state transitions implement the F environment state transitions. The agent set in the F specification is disjoint from the agent set in the environment for the F specification, and so the F state transitions play no part in this subgoal.
3. That the environment and F state transitions implement the FG environment state transitions.

5.3 Refinement

A refinement mapping is a function from states in one specification to states in another. Because we are mapping states that are in traces, not just states in general, we must show that the state transitions of the first specification map to state transitions or stuttering steps in the second, and that all progress properties are satisfied.

In this section we describe our refinement mapping from the composed server process specification to the system call specification.

5.3.1 Mapping the States

The mapping between the composed server processes and the system call specifications has three parts:

1. Mapping the mailboxes.
2. Mapping the F server internal buffer to the F system call internal queue.
3. Mapping the FG server internal buffer and queue to the FG system call internal queue.

In the composed server process specification the FG server sends messages to, and receives them from, the F server. These messages appear in the F and FG mailboxes and in the servers' internal buffers. In the system call specification, however, there are no server processes, only the F and FG system calls, and there are no state transitions that could account for the appearance in the system call mailboxes of the messages passed between the server processes. For this reason, our refinement mapping hides the messages that are passed between the two servers, leaving all other messages and their ordering alone.

Mapping the F server internal buffer to the the F system call internal queue is just as straightforward; any message from the FG server is hidden, which means that the single-length buffer either maps to an empty F system call internal queue or to a queue with only one message in it.

The mapping of the FG server internal buffer and queue to the FG system call internal queue is only slightly more complicated. Recall that the FG server saves each message in a queue until it receives a response message from the F server. The concatenation of the FG server buffer (the messages from the F server are hidden) with the queue contains the same messages in the same order as in the FG system call internal queue.

5.3.2 Mapping the F State Transitions

To prove the refinement of the two atomic state transitions of the F server requires us to show that the transitions always map to either a valid transition of the system call specification or to a stuttering step, for all possible cases.

The *F_reads* transition has three cases:

1. When the mailbox is empty or there is already something in the buffer. In this case the *F_reads* transition leaves the state unchanged, which maps to a stuttering step.
2. When the buffer is empty and the message at the head of the mailbox is from the FG server. The message from the FG server is filtered by the mapping function, so this transition also maps to a stuttering step.
3. When the buffer is empty and the message at the head of the mailbox is not from the FG server. This case is the only one where the F server read transition maps to an F system call read transition that does not stutter.

The *F_responds* transition has three cases:

1. When the buffer is empty. In this case the *F_responds* transition leaves the state unchanged, which maps to a stuttering step.
2. When the message in the buffer is from the FG server. The message from the FG server is filtered by the mapping function, so this transition also maps to a stuttering step.
3. When the message in the buffer is not from the FG server. This case is the only one where a *F_responds* transition maps to a non-stuttering step.

5.3.3 Mapping the FG State Transitions

The two atomic state transitions of the FG server specification correspond to the two transitions of the FG system call or to stuttering steps.

The cases of the *FG_reads* transition are the same as that of the *F_reads* transition, described above.

The *FG_responds* transition has three cases:

1. When the buffer is empty. In this case the transition leaves the state unchanged, which maps to a stuttering step.

2. When the message in the buffer is not from the F server. As shown in figure 5, only messages from the F server result in a response message being sent. All other messages are request messages and are transferred to the internal queue. The mapping function concatenates the queue and the buffer, so no change to the mapped state occurs.

3. When the message in the buffer is not from the F server. This case is the only one where a *FG_responds* transition maps to a non-stuttering step.

5.3.4 Mapping the Progress Properties

The proof that the mapping satisfies the progress properties requires a much more complex proof than that of the state transitions.

Our specifications have simple progress properties that guarantee the eventual occurrence of each transition. The eventual occurrence of a server transition, however, does not always guarantee the eventual occurrence of a system call transition under the refinement mapping.

For example, if there is at least one message in the F server mailbox that is not from the FG server, but there is also a message in the F server buffer, the server specification's *F_reads* transition will implement a stuttering step because the precondition for the transition requires the buffer to be empty. The corresponding transition in the system call specification, on the other hand, is enabled because there is a message in the mailbox. This is shown in figure 6.

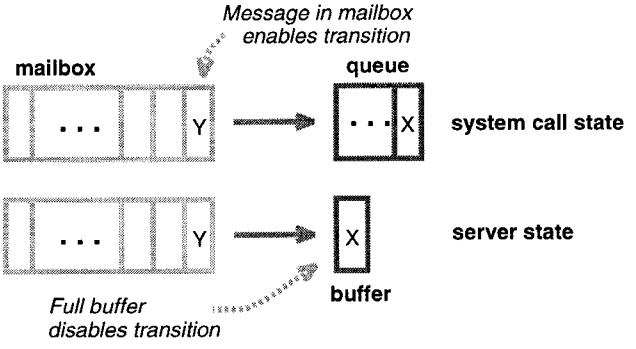

Figure 6: Non-mapping of F_reads Transition, Case 1

If, on the other hand, the F server buffer is empty, but the only messages in the mailbox are from the FG server, then the server transition is enabled but the system call transition is not, as shown in figure 7.

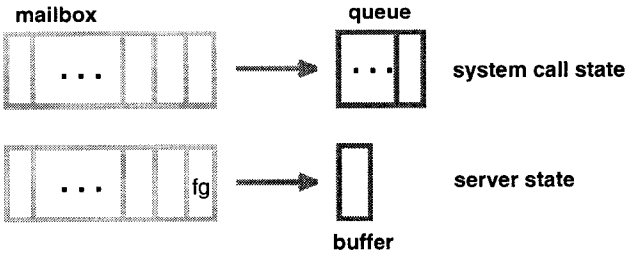

Figure 7: Non-mapping of F_reads Transition, Case 2

In a third case, where the F server buffer is empty, but there is at least one message in the F mailbox that is not from the FG server and the message at the head of the mailbox is from the FG server, then the preconditions are satisfied for both the server transition and the system call transition, but the server transition, nevertheless, does not map to the system state transition. This is because the refinement mapping filters out the messages from the FG server so the message at the head of the mailbox in the system call state is not the same message as at the head of the mailbox in the server state. This situation is depicted in figure 8.

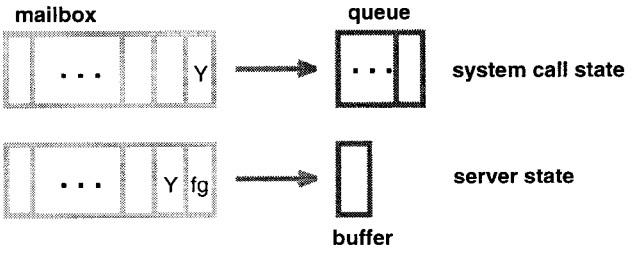

Figure 8: Non-mapping of F_reads Transition, Case 3

The only conditions where the server's *F_reads* transition maps to the corresponding system call transition are when both the transitions are disabled (viz., when the mailbox is empty) or when both are enabled and the message at the head of the mailbox in the server state is not from the FG server. If we call these conditions ρ and denote the server transition and system call transition as t_L and t_H, respectively, then to prove $\Box\Diamond t_H$ we need to prove $\Box\Diamond(\rho \wedge t_L)$.

The progress properties on each transition guarantee that the transitions will occur, but not that the preconditions will ever be true. It is possible that the preconditions could be true infinitely often, but never true when the transition occurs. Only if the preconditions become true and remain true until the transition occurs can we show that a transition will ever implement anything other than a stuttering step. In order to obtain $\Box\Diamond(\rho \wedge t_L)$, therefore, we need the following three conditions:

1. That t_L occurs infinitely often, i.e., $\Box\Diamond t_L$. This is precisely the progress property on t_L.

2. That ρ always eventually holds, i.e., $\Box\Diamond\rho$.

3. That ρ, once true, remains true unless the server transition occurs, i.e., $\Box(\rho \mathcal{W} t_L)$.

To show that ρ, once true, remains true until the next server *F_reads* transition, we prove that each of the other transitions leaves the F server buffer and head of the F mailbox unchanged.

To show that ρ always eventually holds we first find a termination condition and prove that it is reached. In this case, $\neg\rho$ implies that there is an unread message in the mailbox that is not from the FG server. We prove that any message in the F mailbox eventually advances to the head of the mailbox, which is the termination condition. We also prove that, if the F server buffer is non-empty, then it will eventually be emptied. Together, these intermediate results lead to the desired result that, at any point in the trace, either ρ holds or else it eventually holds at some future point in the trace.

We used the same method described above for the *F_reads* transition to prove that the progress property for the *FG_reads* transition is satisfied. The *FG_reads* transition reads from the FG mailbox and receives messages from the F server, but otherwise is the same as the *F_reads* transition.

5.3.5 Mapping the *F_responds* Transition Progress Property

The *F_responds* transition is unique among the four transitions in our specifications in that its mapping condition is always true. There are three possible cases:

1. The F server buffer is empty, which maps to an empty F system call queue. In this case both transitions are disabled.
2. The F server buffer contains a message from the FG server, which maps to an empty F system call queue. Because the result of the server *F_responds* transition is an empty buffer that maps to an empty system call queue, at the system call level the transition is disabled and it implements a stuttering step.
3. The F server buffer contains a message that is not from the FG server, which maps to a F system call queue that contains a single message. The empty buffer after the server transition maps to an empty queue, which is the result of the system call responding to the message.

5.3.6 Mapping the *FG_responds* Transition Progress Property

The termination condition for the *FG_responds* transition progress property is somewhat more complex than that of the *F_reads* and *FG_reads* transitions. As shown in figure 5, a response message is generated only after the FG server sends a message to the F server, and the F server sends a message back to the FG server. The termination condition is that a message from the F server must eventually arrive in the FG buffer. Furthermore, if the data value in the message at the head of the internal FG queue is x, then the data value in the message from the F server must have the value $f(g(x))$.

To prove that this termination condition is achieved, we abstractly define a queue of messages from the F and FG servers that is parallel to the internal FG queue. The abstract queue is constructed from the concatenation of the following components, beginning from the head of the abstract queue and working toward the tail:

1. The message in the FG buffer, if it exists and if it is from the F server.
2. All messages in the FG mailbox from the F server, in order.
3. The message in the F buffer, if it exists and if it is from the FG server.
4. All messages in the F mailbox from the FG server, in order.

As an invariant, we prove that the abstract queue is the same length as the FG queue. Furthermore, the invariant shows a correspondence between the data values in the abstract queue and the FG queue. For all data values x in the FG queue, the corresponding data values in the messages in the abstract queue for components 1 and 2 are $f(g(x))$. The corresponding data values in the messages in the abstract queue for components 3 and 4 are $g(x)$.

The refinement mapping between the FG server state and the FG system call state appends any message in the FG buffer that is not from the F server to the contents of the server queue, in order to form the system call message queue. Such messages are not in the abstract queue, but always enter the abstract and FG server queue upon the next server *FG_responds* transition. We must prove that a message in any component of the abstract queue, or a message in the FG buffer that is not from the F server, will eventually advance through the abstract queue and arrive back in the FG buffer with the correct value.

To do this, we use the other server transition progress properties to prove that any message in one of the components will advance to the next component. For example, the progress property on the *FG_responds* transition guarantees that a non-F message in the FG buffer that contains the data value x will eventually cause the FG server to send a message to the F mailbox that contains the data value $g(x)$. We can use the result proved for the server *F_reads* transition, that any message in the F mailbox eventually advances to the head of the mailbox, and the progress property on the server *F_reads* transition to prove that any message from the FG server in the F mailbox eventually advances to the F buffer. We use the other progress properties in a similar manner to show that eventually there is a message from the F server, containing the value $f(g(x))$, in the FG buffer. This gives us $\Box\Diamond\rho$ for the *FG_responds* transition. It is a comparatively simpler task to show that

ρ persists until the next server *FG_responds* transition. Together with the progress property on the server *FG_responds* transition, we use these results to prove that the composition of the servers satisfies the system call *FG_responds* progress property.

6 Conclusion

In this paper we've presented an overview of the specification and proof of the composition of two server processes, using the composition method of Abadi and Lamport. Our example system includes both safety and progress properties, and the proof required both a composition step and a refinement step. The composition step took a very small part of the overall proof effort, and the majority of the work was spent in refining the progress properties.

We have also described how the composition method can be applied to the composition of a hierarchy of security servers that enforce an access control policy. The method presented here can be used to reason about the access control policy on the storage objects that are used to store sensitive information, and is a step toward our eventual goal of proving the security of an entire distributed operating system.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[2] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[3] Willam R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, November 1989.

[4] Martha Branstad, Homayoon Tajalli, Frank Mayer, and David Dalva. Access mediation in a message passing kernel. In *Proceedings of the 1989 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 66–72, Oakland, California, May 1989.

[5] Mark R. Heckman, Cui Zhang, Brian R. Becker, David Peticolas, Karl N. Levitt, and Ron A. Olsson. Towards applying the composition principle to verify a microkernel operating system. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, number 1125 in Lecture Notes in Computer Science, pages 235–250, Turku, Finland, August 1996. Springer-Verlag.

[6] Judith A. Hemenway and Dr. Jonathan Fellows. Applying the Abadi-Lamport composition theorem in real-world secure system integration environments. In *Proceedings of the Tenth Annual Computer Security Applications Conference*, pages 44–53, December 1994.

[7] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[8] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the Symposium on Research in Security and Privacy*. IEEE, 1994.

[9] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985. DOD-5200.28-STD.

[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, NSA INFOSEC Research and Technology, November 1993.

[11] Marvin Schaefer and Roger R. Schell. Towards an understanding of extensible architectures for evaluated trusted computer system products. In *Proceedings of the 1984 Symposium on Security and Privacy*, Oakland, California, April 1984. Technical Committee on Security and Privacy, IEEE Computer Society.

[12] E. John Sebes and Richard J. Feiertag. Trusted distributed computing: Using untrusted network software. In *Proceedings of the 14th National Computer Security Conference*, pages 608–618, October 1991.

[13] W. R. Shockley and R. R. Schell. TCB subsets for incremental evaluation. In *Proceedings of the 3rd Aerospace Computer Security Conference*, pages 131–139, Orlando, Florida, December 1987.

[14] C. Zhang, R. Shaw, M. R. Heckman, G. D. Benson, M. Archer, K. Levitt, and R. A. Olsson. Towards a formal verification of a secure distributed system and its applications. In *Proceedings of the 17th National Computer Security Conference*, Baltimore, October 1994.